BridgeVIEW™
and LabVIEW™

G Programming Reference Manual

Internet Support

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

Bulletin Board Support

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

Fax-on-Demand Support

512 418 1111

Telephone Support (USA)

Tel: 512 795 8248
Fax: 512 794 5678

International Offices

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

National Instruments Corporate Headquarters

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

Important Information

Warranty

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

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

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

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

Copyright

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

Trademarks

LabVIEW™ , BridgeVIEW™, National Instruments™, NI-DAQ™, natinst.com™, and CVI™ are trademarks of National Instruments Corporation.

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

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

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

Contents

About This Manual

Part I—Basic G Concepts

Chapter 1
Introduction to G Programming

Chapter 2
Editing VIs

Chapter 3
Using SubVIs

Chapter 4
Executing and Debugging VIs and SubVIs

Chapter 5
Printing and Documenting VIs

Chapter 6
Setting up VIs and SubVIs

Chapter 7
Customizing Your Environment

Part II—Front Panel Objects

Chapter 8
Introduction to Front Panel Objects

Chapter 9
Numeric Controls and Indicators

Chapter 10
Boolean Controls and Indicators

Chapter 11
String Controls and Indicators

Chapter 12
Path and Refnum Controls and Indicators

Chapter 13
List and Ring Controls and Indicators

Chapter 14
Array and Cluster Controls and Indicators

Chapter 15
Graph and Chart Controls and Indicators

Chapter 16
ActiveX Controls

Part III—Block Diagram Programming

Chapter 17
Introduction to the Block Diagram

Chapter 18
Wiring the Block Diagram

Chapter 19
Structures

Chapter 20
Formula Nodes

Chapter 21
VI Server

Chapter 22
Attribute Nodes

Chapter 23
Global and Local Variables

Part IV—Advanced Topics

Chapter 24
Custom Controls and Type Definitions

Chapter 25
Calling Code from Other Languages

Chapter 26
Understanding the G Execution System

Chapter 27
Managing Your Applications

Chapter 28
Performance Issues

Chapter 29
Portability and Localization Issues

Appendices, Glossary, and Index

Appendix A
Data Storage Formats

Appendix B
Common Questions about G

Appendix C
Customer Communication

Glossary

Figures and Tables

Figures

Tables

About This Manual

The *G Programming Reference Manual* describes how to create, edit, and execute virtual instruments (VIs) using the G-programming language. 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.

Organization of This Manual

This manual covers four subject areas. Chapters 1 through 7 introduce basic G concepts. Chapters 8 through 16 explain how to use front panel objects. Chapters 17 through 23 explain block diagram programming objects and techniques, and Chapters 24 through 27 cover advanced G topics.

Part I—Basic G Concepts

This section contains basic information about programming in G and creating, editing, and customizing virtual instruments (VIs).

Part I, *Basic G Concepts*, contains the following chapters.

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

- Chapter 2, *Editing VIs*, describes the basic features of building and using VIs, including information about palettes and menus. It also describes such basic tasks as how to create objects, how to change tools, and how to open, run, and save VIs.

- Chapter 3, *Using SubVIs*, describes the concept of hierarchical design in G applications and explains two methods of creating subVIs. The chapter also describes two utilities—the Hierarchy window, which displays the hierarchy of your VIs, and the Find utility, which finds occurrences of subVIs, as well as other objects or strings of text that you indicate.

- Chapter 4, *Executing and Debugging VIs and SubVIs,* describes operating and debugging VIs, and explains the setup of VIs and subVIs for special execution modes.

- Chapter 5, *Printing and Documenting VIs,* describes a variety of issues involving printing and VI documentation.

- Chapter 6, *Setting up VIs and SubVIs*, describes how you customize VI behavior using the **VI Setup** and **SubVI Node Setup** dialog boxes.

- Chapter 7, *Customizing Your Environment*, explains how to customize your environment by setting editor preferences for features such as printing, display, and **Undo**, and by changing the contents of the **Controls** and **Functions** palettes.

Part II—Front Panel Objects

This section contains information about front panel objects, controls and indicators, and ActiveX controls.

Part II, *Front Panel Objects*, contains the following chapters.

- Chapter 8, *Introduction to Front Panel Objects*, introduces the front panel and its two components—controls and indicators. It also explains how to import graphics from other programs to use in your controls.

- Chapter 9, *Numeric Controls and Indicators,* describes how to create, operate, and configure the various styles of numeric controls and indicators.

- Chapter 10, *Boolean Controls and Indicators*, describes how to create, operate, and configure Boolean controls and indicators.

- Chapter 11, *String Controls and Indicators*, describes how to use string controls and indicators, and the table. You can access these objects through the **Controls»String & Table** palette.

- Chapter 12, *Path and Refnum Controls and Indicators*, describes how to use file path controls and refnums, which are available from the **Controls»Path & Refnum** palette.

- Chapter 13, *List and Ring Controls and Indicators*, describes the listbox and ring controls and indicators, which are available from the **Controls»List & Ring** palette.

- Chapter 14, *Array and Cluster Controls and Indicators*, describes how to use arrays and clusters. You access arrays and clusters from the **Controls»Array & Cluster** palette.

- Chapter 15, *Graph and Chart Controls and Indicators*, describes how to create and use the graph and chart indicators in the **Controls»Graph** palette.

- Chapter 16, *ActiveX Controls,* describes the ActiveX Container capability, which enhances the interactions between G-based software and other applications.

Part III—Block Diagram Programming

This section contains information about the components required to build and manipulate a block diagram in G.

Part III, *Block Diagram Programming*, contains the following chapters.

- Chapter 17, *Introduction to the Block Diagram*, describes terminals and nodes—two of the three elements you use to build a block diagram. The third element, wiring, is covered in Chapter 18, *Wiring the Block Diagram*.

- Chapter 18, *Wiring the Block Diagram*, explains how to connect terminals on the block diagram by wiring them together.

- Chapter 19, *Structures*, describes how to use the For Loop, While Loop, Case Structure, and Sequence Structure. These structures are in the **Functions»Structures** palette.

- Chapter 20, *Formula Nodes*, describes how to use the Formula Node to execute mathematical formulas on the block diagram. The Formula Node is available from the **Functions»Structures** palette.

- Chapter 21, *VI Server*, describes the mechanism for controlling VIs and applications programmatically. This chapter also describes how to control VIs or applications remotely.

- Chapter 22, *Attribute Nodes*, describes how to use attribute nodes to set and read attributes of front panel controls programmatically. Some useful attributes include display colors, control visibility, menu strings for a ring control, graph or chart plot colors, and graph cursors.

- Chapter 23, *Global and Local Variables*, describes how to define and use global and local variables. Use global variables to access a particular set of values easily from multiple VIs. Local variables serve a similar purpose within a single VI.

Part IV—Advanced Topics

This section contains information about advanced G features and techniques you can use to create virtual instruments.

Part IV, *Advanced Topics*, contains the following chapters.

- Chapter 24, *Custom Controls and Type Definitions*, introduces custom controls and type definitions.

- Chapter 25, *Calling Code from Other Languages*, describes various methods of calling code written in other languages.

- Chapter 26, *Understanding the G Execution System*, describes VI multitasking and execution.

- Chapter 27, *Managing Your Applications*, describes how to manage files in your G applications.

- Chapter 28, *Performance Issues*, is in three sections. The first section describes the Performance Profiler, a feature that shows you data about execution time of your VIs and monitors single-threaded, multithreaded, and multiprocessor applications. The second section describes factors affecting run-time speed. The third section describes factors affecting memory usage.

- Chapter 29, *Portability and Localization Issues*, describes issues related to transporting VIs between platforms and VI localization.

Appendices, Glossary, and Index

- Appendix A, *Data Storage Formats*, describes the formats in which you can save data.

- Appendix B, *Common Questions about G*, provides answers to some of the questions commonly asked by G users.

- Appendix C, *Customer Communication*, contains forms to help you gather the information necessary to help National Instruments solve your technical problems and a form you can use to comment on the product documentation.

- 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:

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

[]   Square brackets enclose optional items—for example, [response].

-	A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys—for example, <Control-Alt-Delete>.
»	The **»** symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options»Substitute Fonts** directs you to pull down the **File** menu, select the **Page Setup** item, select **Options**, and finally select the **Substitute Fonts** options from the last dialog box.
♦	The ♦ symbol indicates that the text following it applies only to a specific product, a specific operating system, or a specific software version.
bold	Bold text denotes the names of menus, menu items, parameters, dialog boxes, dialog box buttons or options, icons, windows, Windows 95 tabs, or LEDs.
bold italic	Bold italic text denotes an activity objective, note, caution, or warning.
<Control>	Key names are capitalized.
italic	Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text from which you supply the appropriate word or value, as in Windows 3.*x*.
`monospace`	Text in this font denotes text or characters that you should literally enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and for statements and comments taken from programs.
`monospace bold`	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.
`monospace italic`	Italic text in this font denotes that you must enter the appropriate words or values in the place of these items.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, folders, and files.
Platform	Text in this font denotes information related to a specific platform.
☞	This icon to the left of bold italicized text denotes a note, which alerts you to important information.

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

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

Related Documentation

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

*   *LabVIEW User Manual*
*   *BridgeVIEW User Manual*
*   *LabVIEW DAQ Basics Manual*
*   *LabVIEW Function and VI Reference Manual*
*   *LabVIEW Quick Start Guide*
*   LabVIEW or BridgeVIEW *Online Reference*, available by selecting **Help»Online Reference**
*   LabVIEW *Online Tutorial* **(Windows only)** which you launch from the LabVIEW dialog box
*   *LabVIEW Getting Started* card
*   *G Quick Reference* card
*   *VXI VI Reference Manual*
*   *LabVIEW Code Interface Reference Manual*, available in portable document format (PDF) on your software program disks or CD.

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.

Part I

Basic G Concepts

This section contains basic information about programming in G and creating, editing, and customizing virtual instruments (VIs).

Part I, *Basic G Concepts*, contains the following chapters.

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

- Chapter 2, *Editing VIs*, describes the basic features of building and using VIs, including information about palettes and menus. It also describes such basic tasks as how to create objects, how to change tools, and how to open, run, and save VIs.

- Chapter 3, *Using SubVIs*, describes the concept of hierarchical design in G applications and explains two methods of creating subVIs. The chapter also describes two utilities—the Hierarchy window, which displays the hierarchy of your VIs, and the Find utility, which finds occurrences of subVIs, as well as other objects or strings of text that you indicate.

- Chapter 4, *Executing and Debugging VIs and SubVIs,* describes operating and debugging VIs, and explains the setup of VIs and subVIs for special execution modes.

- Chapter 5, *Printing and Documenting VIs,* describes a variety of issues involving printing and VI documentation.

- Chapter 6, *Setting up VIs and SubVIs*, describes how you customize VI behavior using the **VI Setup** and **SubVI Node Setup** dialog boxes.

- Chapter 7, *Customizing Your Environment*, explains how to customize your environment by setting editor preferences for features such as printing, display, and **Undo**, and by changing the contents of the **Controls** and **Functions** palettes.

1

Introduction to G Programming

This chapter describes the unique G approach to programming. It also explains how to start using G to develop programs.

What Is G?

G is the programming language at the heart of LabVIEW. It is also integral to the National Instruments application development environment BridgeVIEW.

G, like C or BASIC, is a general-purpose programming language with extensive libraries of functions for any programming task. G includes libraries for data acquisition, GPIB and serial instrument control, data analysis, data presentation, and data storage. G also includes conventional program debugging 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.

G differs from those programming languages in one important respect. Other programming languages are *text-based* while G is *graphical*.

Components of a VI

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

A VI consists of an interactive user interface, a dataflow diagram that serves as the source code, and icon connections that set up the VI so that

it can be called from higher level VIs. More specifically, VIs are structured as follows.

- The interactive user interface of a VI is called the *front panel*, because it simulates the panel of a physical instrument. The front panel can contain knobs, pushbuttons, graphs, and other controls and indicators. You enter data using a mouse and keyboard, and then view the results on the computer screen.

- The VI receives instructions from a *block diagram*, which you construct in G. The block diagram is a pictorial solution to a programming problem. The block diagram is also the source code for the VI.

- VIs are hierarchical and modular. You can use them as top-level programs, or as subprograms within other programs or subprograms. A VI, when used 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, G makes the best use of 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, separate from the rest of the application, debugging is much easier. Furthermore, many low-level subVIs often perform tasks common to several applications, so you can develop a specialized set of subVIs well suited to future applications.

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

Note   *To view examples of different features that G uses for your LabVIEW or BridgeVIEW application, refer to the* examples *directory.*

Front Panel

The user interface of a VI is like the user interface of a physical
instrument—the front panel. A front panel of a VI might look like the
following illustration.
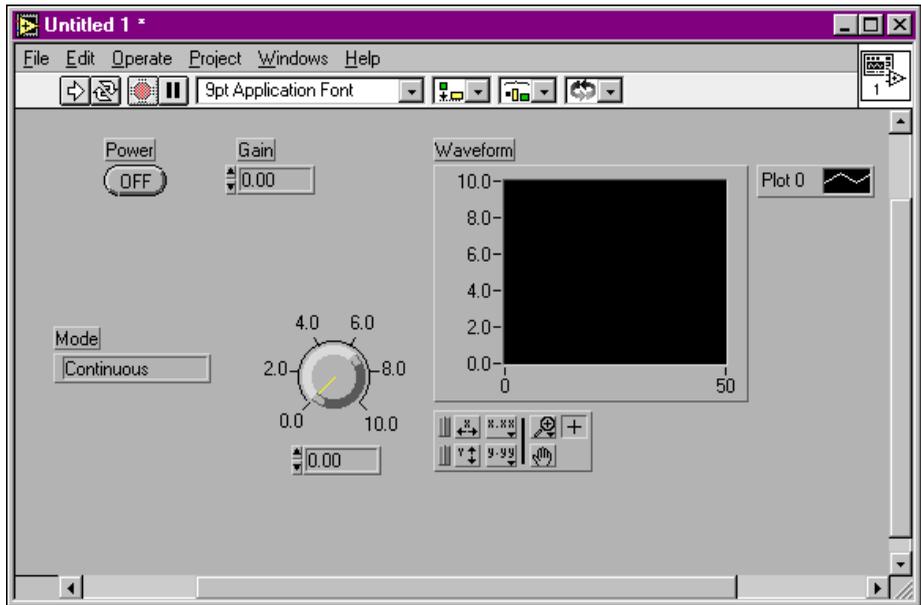
The front panel of a VI is primarily a combination of *controls* and
indicators. Controls simulate instrument input devices and supply data to
the block diagram of the VI. Indicators simulate instrument output devices
that display data acquired or generated by the block diagram of the VI.

You add controls and indicators to the front panel by selecting them from the floating **Controls** palette shown in the following illustration.

You can change the size, shape, and position of a control or indicator. In addition, each control or indicator has a pop-up menu you can use to change various attributes or select different menu items. You access this pop-up menu by doing the following.

- **(Windows and UNIX)** right-clicking the object with the mouse button.
- **(Macintosh)** `<command>`-clicking the mouse button.

See Chapter 2, *Editing VIs*, and Part II, *Front Panel Objects*, of this manual for information about how to build a front panel.

Block Diagram

You can switch from the front panel to the block diagram by selecting **Windows»Show Diagram** from the menu.

The Diagram window holds the block diagram of the VI, which is the graphical source code of a 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*.
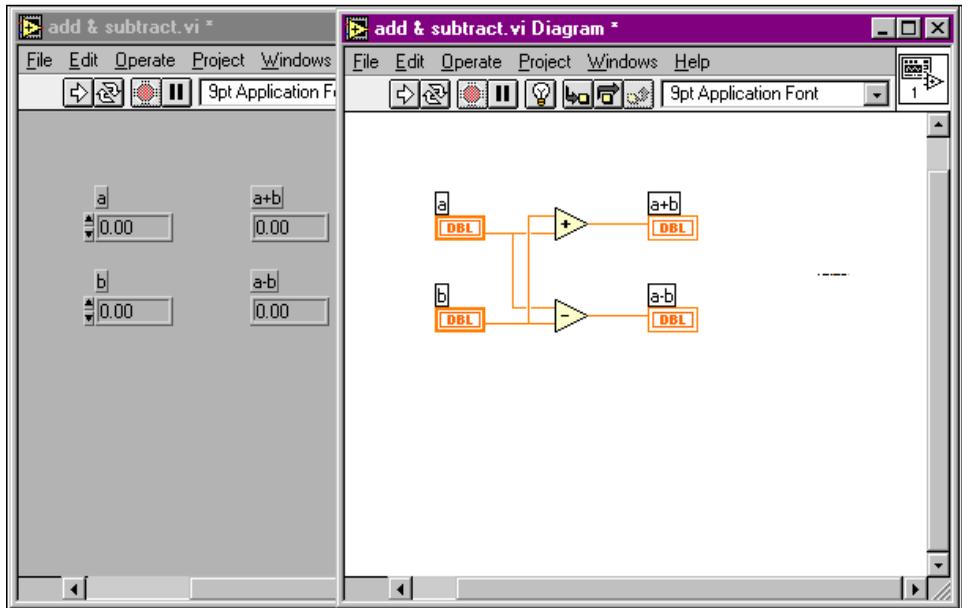
When you place a control or indicator on the front panel, a corresponding terminal appears on the block diagram. You cannot delete a terminal that belongs to a control or indicator. The terminal disappears only when you delete its control or indicator on the front panel.

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. Terminals that produce data are referred to as *data source terminals* and terminals that receive data are *data sink terminals*.

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. G has an extensive library of functions for math, comparison, conversion, I/O, and more.

Another type of node is a *structure*. Structures are graphical representations of the loops and case statements of traditional programming languages. The G-programming language 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, nor can you wire a sink terminal to another sink. However, you can wire one source to several sinks. Each wire is a different style or color, depending on the value that flows through the wire. The previous illustration shows the wire style for a numeric scalar value—a thin, solid line.

Data flow is the principle that governs G program execution. Stated simply, a node only executes when all data inputs arrive; the node supplies data to all of its output terminals when it finishes executing; the data immediately passes from source to sink terminals. Dataflow programming 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 Part III, *Block Diagram Programming,* of this manual for in-depth information about using block diagram objects to build a program.

Icon and Connector

When an icon of a VI is placed on the diagram of another VI, it becomes a *subVI*, the G equivalent of a subroutine. The controls and indicators of a subVI receive data from and return data to the diagram of the calling VI.

The *connector* is a set of terminals that correspond to the subVI controls and indicators. The *icon* is either the pictorial representation of the purpose of the VI, or a textual description of the VI or its terminals.

The connector is much like the parameter list of a function call; the connector terminals act like parameters. Each terminal corresponds to a particular control or indicator on the front panel. A connector receives data at its input terminals and passes the data to the subVI code through the subVI controls, or receives the results at its output terminals from the subVI indicators.

Every VI has a default icon displayed in the icon pane in the upper right corner of the front panel and block diagram windows. This VI icon is shown in the following illustration.
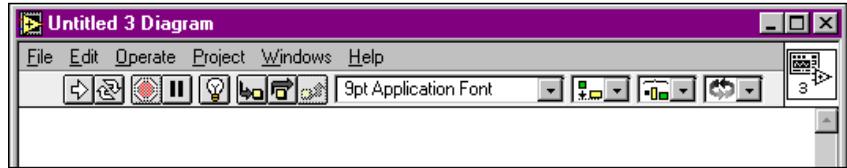
Every VI also has a connector, which you access by choosing **Show Connector** from the icon pane pop-up menu on the front panel. When you bring up the connector for the first time, you see a connector pattern. You can select a different pattern if you want. The connector generally has one terminal for each control or indicator on the front panel. You can assign up to 28 terminals. If you anticipate future changes to the VI that might require a new input or output, leave some extra terminals unconnected.

For more information, see Chapter 3, *Using SubVIs*.

Getting Help

The Help window contains help information for functions, constants, subVIs, controls and indicators, and dialog box menu items. To display the window, choose **Show Help** from the **Help** menu or press <Ctrl-h> **(Windows)**; <command-h> **(Macintosh)**; <meta-h> **(Sun)**; or <Alt-h> **(HP-UX)**. If your keyboard has a <Help> key, press that instead. Move your cursor over the icon of a function, a subVI node, or a VI icon (including the icon of the VI you have open, shown at the top right of the VI window) to see the help information.

 Selecting **Help»Lock Help** or clicking the lock icon at the bottom of the window locks the current contents of the Help window. When you lock it, moving over another function or icon does not change the display in the Help window. Select **Lock Help** or click the lock icon again to turn this menu item off.

Front Panel Help

When you move the cursor over a control or indicator, the Help window displays the description for that particular control or indicator. It is a good idea to enter descriptions for all controls and indicators when you create a VI. See the *Creating Object Descriptions* section of Chapter 2, *Editing VIs*, for information.

If you hold the cursor on a VI icon in the top right of a front panel for a few moments, the Help window displays help for that VI.

Block Diagram Help

For functions and subVI nodes, the Help Window displays the icon, inputs and outputs, as well as a description of the functionality of the node.

You can display the Help window as either a simple or a detailed view by pressing the second button at the bottom of the Help window or by toggling the **Help»Simple Help** menu item.

The simple view emphasizes the important connections and de-emphasizes the other less critical connections. In this view, labels of required connections are in bold text and recommended connections are in plain text. Optional connections do not display their name and only draw a small wire stub to indicate more options exist if the user requires them. The following illustration shows the simple view of a data acquisition VI.
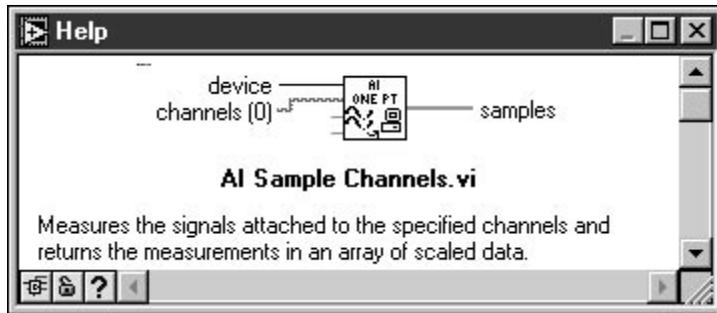
In the detailed view, the Help window shows all inputs with wires pointing to the left, and outputs with wires pointing to the right. Labels of optional inputs are in gray text. The following illustration shows the detailed view of the same data acquisition VI.
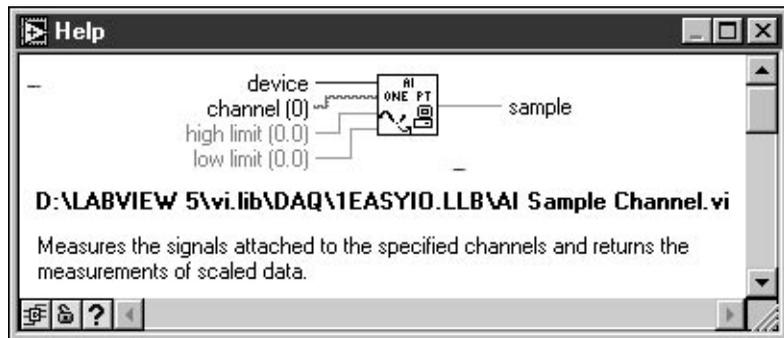
The disk location of the subVI displays beneath the icon in detailed mode. In simple mode, only the subVI name displays.

If a function input does not require wiring, the default value usually is indicated next to the name in parentheses. If the function can accept multiple data types, the Help window shows the most common type.

The terminal names for subVI nodes are the labels of the corresponding front panel controls and indicators. The default values of subVI inputs do not appear in the help window automatically. It is a good idea to include the default value in parentheses in the name of controls when you create a subVI.

When you place the Wiring tool over a wire, the Help window displays the data type of that wire. Also, when you move the Wiring tool over different areas of the VI icon, the corresponding connector terminal is highlighted in the Help window.

Attribute Help

If you do not know the meaning of a particular attribute, you can use the Help window to find out about the meaning of the attribute, its data type, and acceptable values. If the attribute is a more complicated type, such as a cluster, the Help window displays a hierarchical description of the data structure. The following illustration shows an attribute node and the corresponding help information displayed as you move the cursor over different attribute names
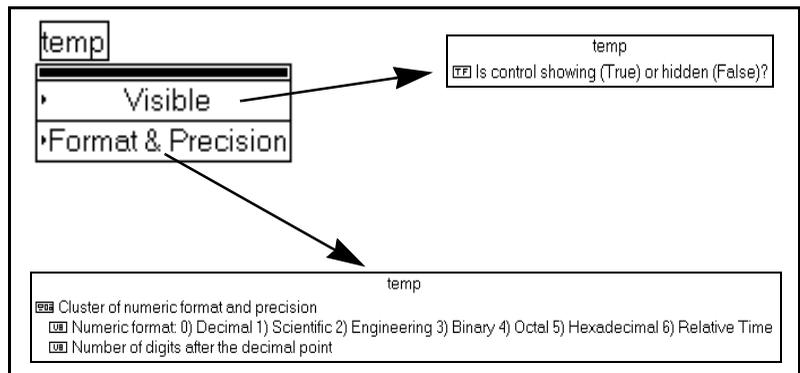
Online Reference

The Help window displays the most commonly required help information in a condensed format. To access more extensive online information, select **Help»Online Reference**. For most block diagram objects, you can select **Online Reference** from the pop-up menu of the object. You also can access this information by pressing the question mark button shown to the left, located at the bottom of the Help window.

See Chapter 5, *Printing and Documenting VIs,* for information about creating your own online help files.

Editing VIs

This chapter describes the basic features of building and using VIs, including information about palettes and menus. It also describes such basic tasks as how to create objects, how to change tools, and how to open, run, and save VIs.

G Environment

You use palettes and menus to build VIs. You can choose objects from palettes to place on the front panel or the block diagram. These objects can be moved or modified using the tools or menus.

☞ **Note** *If you are a BridgeVIEW user, make sure your palettes are changed to Basic G. Select* **Edit»Select Palette Set***. Then, choose the* **Basic G** *palette from the palette set menu.*

You create objects on the front panel and block diagram by selecting them from the floating **Controls** and **Functions** palettes. For example, if you

want to create a knob on a front panel, select it from the **Numeric** palette of the **Controls** palette, as shown in the following illustration.
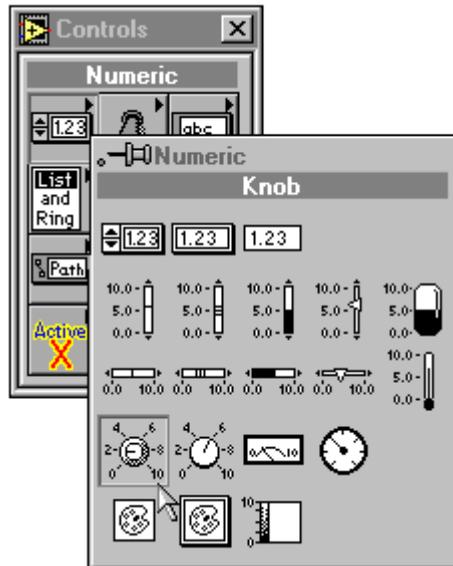
As you move the selection arrow over an object on the palette, the name of the object appears at the top of the palette. In the preceding illustration, the knob is selected. At this point, if you click and move the mouse over the front panel, a knob appears at the point where you release the mouse. Alternatively, you can click and release over the object on the palette. When you move over the front panel you see an outline of the control, which you can reposition and click to place. You can resize an object at the same time you create it by clicking and dragging as you place it.

☞ **Note**   *If you need several functions from the same palette, you can keep it open. To keep a palette open, select the pushpin in the top left corner of the palette. When you have pinned the palette open, it has a titlebar so you can move it around easily. When you quit your G development environment, the locations of the palettes are saved and they open in the same locations next time you launch the application.*

If you pop up in an empty space in a front panel or block diagram window, a temporary copy of the **Controls** *and* **Functions** *palettes appear. If you are using a small monitor, you can close the floating* **Controls** *and* **Functions** *palettes and create objects with these pop-up palettes instead.*

You can position the front panel and block diagram side-by-side or above and below one another by selecting the **Tile** *menu items from the* **Windows** *menu.*

When you create front panel objects, they appear with a blank label ready for you to enter the name of the new object. If you want to give the object a name, type in the name. When you finish entering the name, end text entry by pressing the <Enter> key on the *numeric* keypad. If you do not have a keypad, you can click the **Enter** button in the toolbar, or click anywhere outside the label.

☞ **Note**   *If you do not enter text into the label of a control when you first create it, the label disappears when you click elsewhere. You can show the label again by popping up on the control and selecting* **Show»Label***.*

The following illustration shows an example of the result of popping up on the control and selecting **Show»Label**.
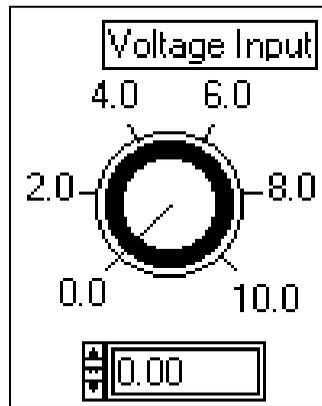
When you create an object on the front panel, a corresponding terminal is created on the block diagram. You use this terminal if you want to read data from a control or to send data to an indicator.

If you select **Windows»Show Diagram**, you can see the corresponding block diagram for the front panel. The block diagram contains terminals for all front panel controls and indicators.

front panel   block diagram
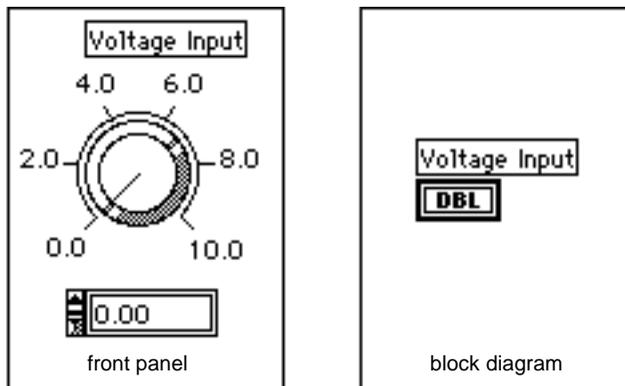
Tools Palette

A *tool* is a special operating mode of the mouse cursor. You use tools to define the operation taking place when subsequent mouse clicks are made on the panel or diagram.

Many of the tools contained in the floating **Tools** palette are shown in the following illustration. You can move this palette anywhere you want, or you can close it temporarily by clicking the close box (the small box with an ✕ in the top right corner of palettes). Once closed, you can access it again by selecting **Windows»Show Tools Palette**. You also can bring up a temporary version of the **Tools** palette at the location of your cursor by clicking while pressing <Ctrl-Shift> **(Windows)**; <command-Shift> **(Macintosh)**; <meta-Shift> **(Sun)**; or <Alt-Shift> **(HP-UX)**.

The tools and their functions/descriptions in the **Tools** palette are as follows.

Operating tool—Changes the value of a control or selects the text within a control.

Positioning tool—Positions, resizes, and selects objects.

Labeling tool—Edits text and creates free labels.

Wiring tool—Wires objects together in the block diagram.

Object Pop-up Menu tool—Opens the pop-up menu of an object.

Scroll tool—Scrolls the window without using the scroll bars.

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

Probe tool—Creates probes on wires.

Color Copy tool—Copies colors for pasting with the Color tool.

Color tool—Sets foreground and background colors.

Change from one tool to another by doing any of the following while in edit mode.

- Click the tool you want in the **Tools** palette.
- Use the <Tab> key to move through the most commonly used tools in sequence.
- Press the spacebar to toggle between the Operating tool and the Positioning tool when the front panel is active, and between the Wiring tool and the Positioning tool when the block diagram is active.

Using Menus

The *menu bar* at the top of a VI window (or at the top of the Macintosh screen) contains several *pull-down menus*. You can access pull-down menus from a menu bar by clicking a menu bar item. The pull-down menus are usually general in nature and contain items common to other applications, such as **Open**, **Save**, **Copy**, and **Paste**, and many others specific to the editor. Some menus also list shortcut key combinations.

The most often used menu is the object *pop-up menu*. Almost every G object, as well as empty front panel and block diagram space, has a context-sensitive pop-up menu with menu items and commands. When possible it is best to select a command or menu item from an object pop-up menu.

Pop-Up Menus

As you learned in the previous section, all G objects have associated pop-up menus. You pop up on an object by clicking the object with the right mouse button (on the Macintosh, hold down the <command> key while you click the object with the mouse). You can change the look or behavior of the object with the menu items in the resulting menu.
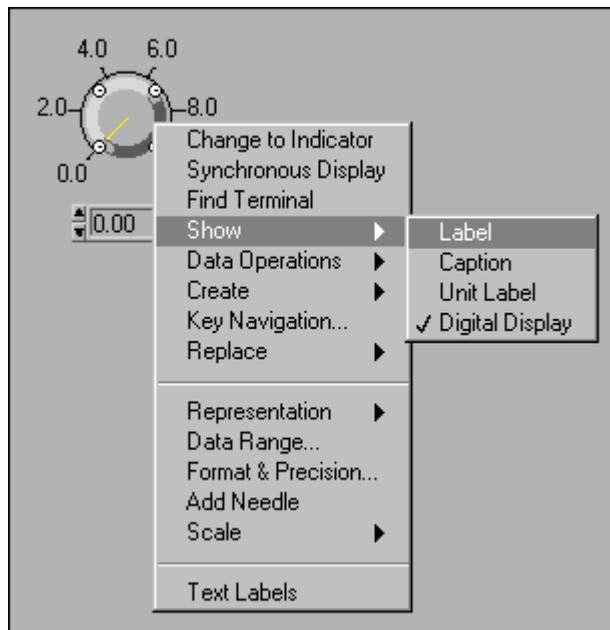
Editing VIs

After you are familiar with editing techniques, see Chapter 8, *Introduction to Front Panel Objects,* for information on building front panels, and Chapter 17, *Introduction to the Block Diagram*, for help in building block diagrams.

Selecting Objects

For many editing operations, such as moving, copying, and deleting, you must *select* an object.To select an object, make sure you are using the positioning tool, and click the mouse button when the point of the arrow is on the object. Once you select an object, it is surrounded with a moving dashed outline called a *marquee*.

Positioning
Tool
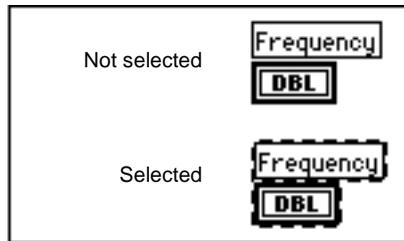
To select more than one object, `<Shift>`-click each additional object. You also can deselect a selected object by `<Shift>`-clicking it. Another way to select one or more objects is to drag a selection rectangle around the object(s), as shown on the right of the following illustration.
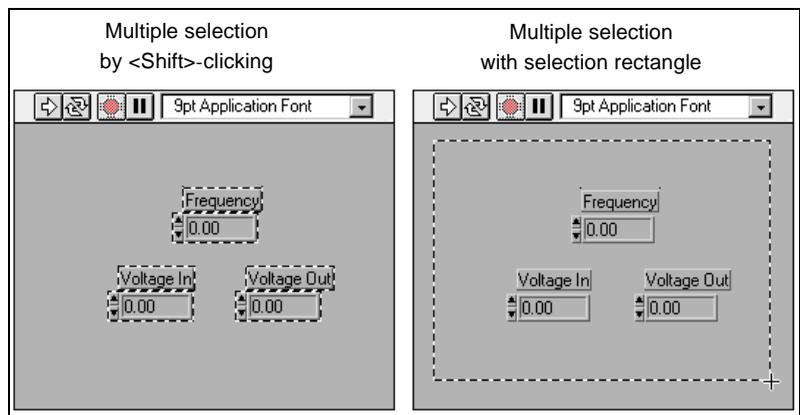
Hollow objects typically must be enclosed completely by the rectangle to be selected. Most objects are selected if any part of them is within the rectangle. By holding down the `<Shift>` key while dragging the selection rectangle, you can select additional objects or deselect an object enclosed by the rectangle.

Clicking an unselected object or clicking an open area deselects everything currently selected. 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.

Dragging and Dropping VIs, Pictures, and Text

You can drag VIs from the file system to a block diagram to create a subVI call to that VI. This feature also works for custom controls, type definitions, and globals. In addition, you can drag text and pictures from other applications to copy them to front panels and block diagrams.

In Windows 3.1, external drag support is limited to dragging VIs and controls from the File Manager. In Windows 95 and Windows NT, which have 32-bit object linking and embedding (OLE) support, you can drag text and pictures from OLE-supporting applications, in addition to dragging from the File Manager/Explorer.

On the Macintosh, your system must have the Drag Manager to use external Drag and Drop. The Drag Manager is built into Systems 7.5 and later. It is also available for System 7.0 up to System 7.5 as an extension from Apple.

You can use the following Drag and Drop capabilities internally.

- Drag front panel controls and indicators to block diagrams to create block diagram constants and vice versa.

- Drag a subVI to a path control or constant to drop the full pathname of the VI inside the control or constant.

You can use the following Drag and Drop capabilities externally (from other applications).

- Drag a file into a path control or constant to drop its full pathname.

- Drag a datalog file into a Data Log File Refnum to create a cluster containing the data structure in the datalog file.

- Drag a graphics file into a Pict Ring, front panel, or block diagram to drop the picture it contains.

- Drag a VI file into the block diagram of a VI to drop it as a subVI.

- Drag a custom control file into the front panel to drop the custom control stored in that file.

- **(Windows 95/NT)** Drag Text from an OLE source to drop its text into a string control or constant, front panel, block diagram, or label.

- **(Windows 95/NT)** Drag Image from an OLE source to drop its picture into a picture ring, front panel, or block diagram.

- **(Macintosh)** Drag a Text Clipping or text selected from any application to drop that text into a string control or constant, front panel, block diagram, or label.

- **(Macintosh)** Drag an Image clipping or an image selected from another application to drop it into a picture ring, front panel, or block diagram.

☞ **Note**   *If you can drop an object successfully, the destination is highlighted.*

For more information on Drag and Drop features, see your Windows or Macintosh system manual.

Positioning Objects

You can position an object by clicking it with the **Positioning** tool and then dragging it to the location you want.

If you hold down the <Shift> key and then drag an object, the action restricts the direction of movement horizontally or vertically, depending on which direction you first move the object.

You can move selected objects in small, precise increments by pressing the appropriate arrow key on the keyboard once for each pixel you want the objects to move. Hold down an arrow key to repeat the action. Hold down the <Shift> key along with an arrow key to move the object more rapidly.

If you change your mind about moving an object while you are dragging, continue to drag until the cursor is outside all open windows and the dotted outline disappears, then release the mouse button. This cancels the move operation, and the object remains where it was. If your screen is cluttered, the menu bar is a convenient and safe place to release the mouse button when you want to cancel a move.

☞ **Note**   *The positioning tool also enlarges objects. Use caution when clicking to make sure this does not happen. Move an object from the center; enlarge it from the corners.*

Reorder

You also can place objects on top of other objects. There are several commands to move them relative to each other in **Reorder**, on the far right of the tool bar across the top of the Front Panel. 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.
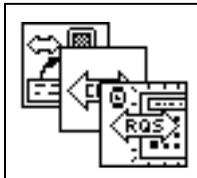
Move To Front moves the selected object in the stack to the top. If object 1 is selected, object 1 moves to the top of the stack, with object 3 under it, and object 2 on the bottom.
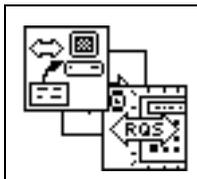
If object 2 is selected next, **Move To Front** changes the order to object 2 on top, object 1 under it, and object 3 on the bottom.
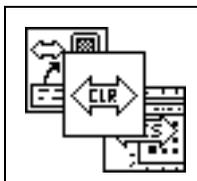
Move Forward moves the selected object one position higher in the stack. Starting with the original order of object 1 on the bottom of the stack and object 3 on top, selecting this item for object 1 puts object 2 on the bottom, object 3 on the top, and object 1 in the middle.
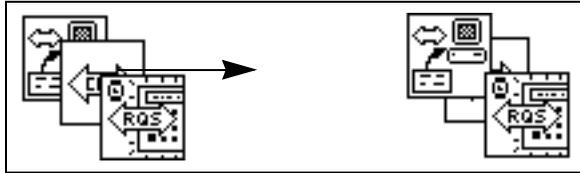
Move To Back and **Move Backward** work similarly to **Move To Front** and **Move Forward** except they move items down the stack rather than up.

Aligning Objects

Select the objects you want to align and choose the axis along which you want to align them from the **Align Objects** ring, as shown at left. The following illustration shows you how to select objects for alignment and the results of the selection.
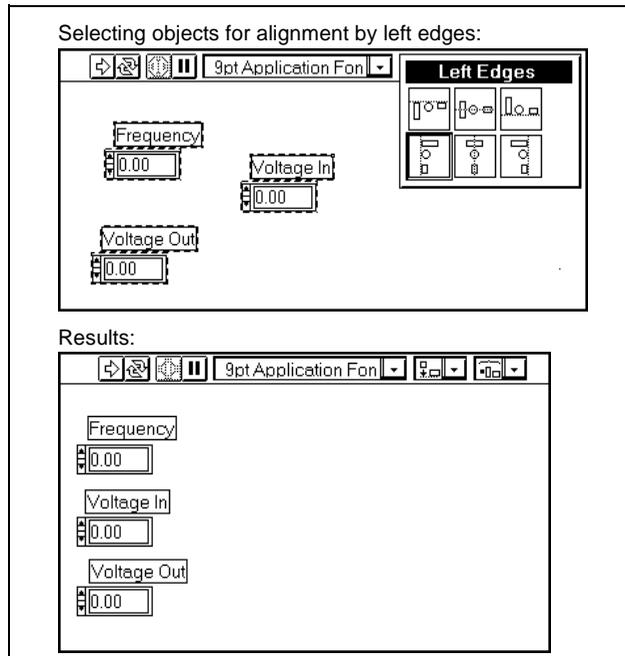
You can align an object along the vertical axis using its left, center, or right edge. You also can align an object along a horizontal axis using its top, center, or bottom edge. Your selection becomes the current alignment option, indicated by a dark border around the item in the palette. On the Macintosh platform, <command-A> repeats this alignment selection without using the menu.

Distributing Objects

To space objects evenly, or distribute them, select the objects you want to distribute and choose how you want to distribute them from the **Distribute Objects** ring. In addition to distributing selected objects by making their edges or centers equidistant, four menu items at the right side of the ring let you add or delete gaps between the objects, horizontally or vertically. On the Macintosh platform, `<command-D>` repeats the distribution on any selection.

Duplicating Objects

There are three basic methods for duplicating an object—by copying and pasting, by cloning, and by dragging and dropping.

To copy and paste within a VI or between VIs, select the object with the Positioning tool and choose **Edit»Cut** or **Edit»Copy**. Then, click the area in which you want to place the duplicate and choose **Edit»Paste**. You can duplicate several objects at the same time by dragging a selection marquee around the objects before duplicating them. You also can copy text or pictures from other applications and paste them into your front panel or block diagram.

To clone an object, click the Positioning tool on it while pressing the `<Ctrl>` **(Windows)**; `<option>` **(Macintosh)**; `<meta>` **(Sun)**; or `<Alt>` **(HP-UX)** key, and drag the copy to its new location. In UNIX, you also can clone objects by clicking and dragging the middle mouse button without modifier keys.

When you clone or copy objects that have labels, G-language software designates the copies with the same name as the original object with the word *copy* appended (or *copy 1*, *copy 2*, etc. for copies of copies).

If you copy a front panel control, a new terminal is created on the diagram. Notice that local variables and attribute nodes for the original control are not copied. When you use **Edit»Copy** or **Edit»Paste** on a local variable, the front panel object is copied as well. If you clone a local variable, a new reference to the original control is created. For more information see Chapter 22, *Attribute Nodes*, and Chapter 23, *Global and Local Variables*, in this manual.

You can copy, clone, or drag and drop front-panel controls to any diagram to make a corresponding constant. In the same way, you can drag user-defined constants from the block diagram to the front panel to create controls.

Deleting Objects

To delete an object, select the object and choose **Edit»Clear** or press the `<Backspace>` key or the `<Delete>` key. You only can delete block diagram terminals by deleting the corresponding front panel controls or indicators.

Although you can delete most objects, you cannot delete parts of a control or indicator such as labels or digital displays. However, you can hide these components by deselecting **Show»Label** or **Show»Digital Display** from the pop-up menu of the object.

If you delete a structure such as a While Loop, the contents are deleted as well. If you want to delete only the structure but preserve its contents, pop up on the edge of the structure and select **Remove While Loop** (or other structure name). This action deletes the structure but not its contents, and automatically reconnects any wires crossing the border of the structure.

Labeling Objects

Labels are blocks of text that annotate components of front panels and block diagrams. There are two kinds of labels—*owned* labels and *free* labels. Owned labels belong to and move with a particular object and annotate that object only. You can hide these labels, but you cannot copy or delete them independently of their owners. Free labels are not attached to any object, and you can create, move, or dispose of them independently. Use them to annotate your front panels and block diagrams. You use the Labeling tool, whose cursor is the I-beam and box cursor, to create free labels or to edit either type of label.

Front Panel Caption Labels

Every front panel object has a name associated with it. This can be used to distinguish it from other objects. The name also is used on the terminal, local variables, and attribute nodes of the object.

If the object is a control, the text of the label names the wire to which the control is connected on the block diagram. If the object is an indicator and it is connected to the connector pane, the label names wires connected to its connector-pane terminal on block diagrams of the VI callers. This means that when the text of the name label changes, LabVIEW must recompile the VI and possibly recompile the VI callers as well.

Front panel objects also can have caption labels. The caption does not affect the name of the object and you can use it as a more descriptive object name. You also can show, hide, and change the caption programmatically through attribute nodes. Changes to the caption do not cause the VI or its callers to be recompiled.
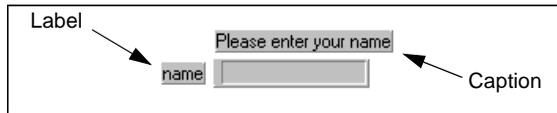
Figure 2-1. Front Panel Label Dialog Box

The label is displayed when you drop a front panel object. To access the caption while you are editing the object, pop up on the object and select **Show»Caption** as shown in the following illustration.
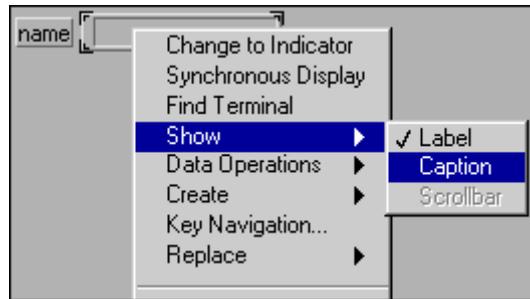
Figure 2-2. Pop-Up Menu in a Label

If you want to show, hide, or change the caption programmatically, use an attribute node with the attributes shown in the figure below.
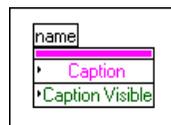
Figure 2-3. Attribute Node Showing Caption Attributes

When you view the help window for the object (by selecting **Help»Show Help**), the front panel object caption appears in a box and its label appears below in brackets, as shown in the following illustration.

Figure 2-4. Context-Sensitive Help Dialog Box

Free Labels

To create a free label, select the Labeling tool from the **Tools** palette and click any open area.
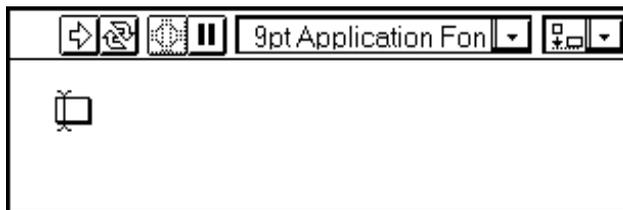
A small box appears with a text cursor at the left margin ready to accept typed input. Type the text you want to appear in the label. Use the **Enter** button on the toolbar or numeric keypad to complete the edit operation.
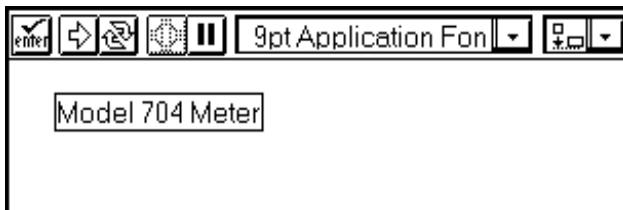
You also can end text entry by clicking anywhere outside the label. If you do not type any text in the label, the label disappears as soon as you click any other area.

☞ **Note**   *If you place a label or any other object over (partially covering) a control or indicator, it slows down screen updates and might make the control or indicator flicker. To avoid this problem, do not overlap a front panel object with a label or other object.*

You can copy the text of a label by selecting it with the Labeling tool. Double-click the text with the Labeling tool to select by words. Triple-click the text to highlight the entire label.When the text is selected, choose **Edit»Copy** to copy the text onto the Clipboard. Now you can highlight the text of a second label. Select **Edit»Paste** to replace the highlighted text in the second label with the text from the Clipboard. To create a new label with the text from the Clipboard, click the screen with the Labeling tool where you want the new label positioned. Then, select **Edit»Paste**.

When you create a control or indicator on the front panel, a blank, owned label accompanies it, waiting for you to type the name of the new control or indicator. The label disappears if you do not enter text into it before clicking another area with the mouse. You can show the label again by popping up on the control or indicator and selecting **Show»Label**.

Structures and functions come with a default label, which is hidden until you show it.

To display a hidden label, pop up on the object and select **Show»Label** from the pop-up menu, as shown in the following illustration.
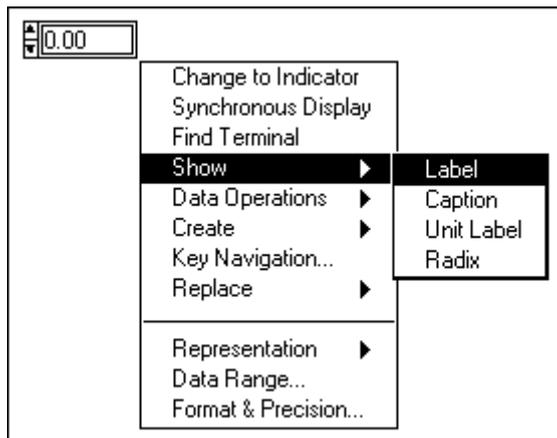
The label appears, waiting for typed input. The label disappears if you do not enter text into it before clicking outside the label.

☞ **Note** *On the block diagram, subVIs do not have labels you can edit. The label of a subVI always shows its name. Function labels, on the other hand, can be edited to reflect the use of the function in the block diagram. For example, you can use the label of an Add function to document what quantities are being added, or why they are being added at that point in the block diagram. See the note in the* Creating VI Descriptions *section in this chapter for related information.*

Text Characteristics

You can change text attributes by using the menu items from the **Font** ring on the toolbar, as shown in the following illustration. If you select objects or text and make a selection from this ring, the changes apply to everything selected. If nothing is selected, the changes apply to the default font, meaning new labels use the new default font. Changing the default font does not change the font of existing labels; it only affects labels created from that point on.

Font Dialog...	⌘0
✓ Application Font	⌘1
System Font	⌘2
Dialog Font	⌘3
✓ Current Font	⌘4
Size	▶
Style	▶
Justify	▷
Color	▶
AGaramond	
AGaramond Bold	
AGaramond BoldItalic	
AGaramond Italic	
AGaramond Semibold	
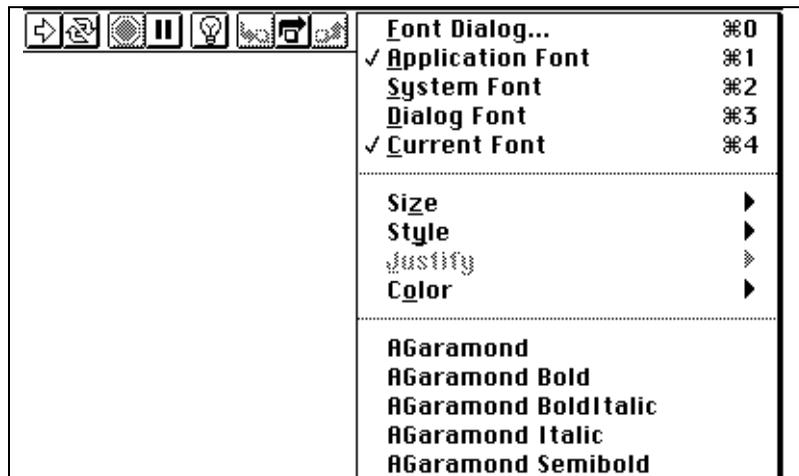
If you select **Font Dialog…** from the font ring while a front panel is active, the dialog box shown in the following illustration appears. If a block diagram is active instead, the dialog box differs only in the selection of **Diagram Default** instead of **Panel Default** in the checkboxes at the bottom of the dialog box.
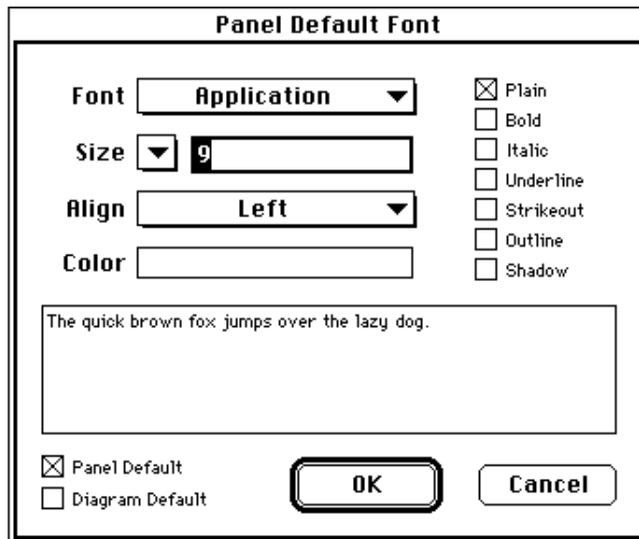
With either the **Panel Default** or **Diagram Default** checkbox selected, the other selections you make in this dialog box are used with new labels on the front panel or block diagram.

In the previous illustration, the word **Application** appears in the Font ring. This ring also contains the **System Font**, **Dialog Font**, and **Current Font** menu items. The last item in the ring, **Current Font**, refers to the last font style selected.

The Application, System, and Dialog fonts are used for specific portions of the interface. These fonts are predefined so that they map best between platforms. When you take a VI that contains one of these fonts to another platform, the fonts map as closely as possible.

The predefined fonts are as follows.

• Application font is the most commonly used *default* font. It is used for the **Controls** palette, the **Functions** palette, and text in new controls.

• System font is the font used for menus.

• Dialog font is the font used for text in dialog boxes.

If you click the **Panel Default** and **Diagram Default** checkboxes, the selected font becomes the current font for the front panel, the block diagram, or both. The current font is used on new labels. You can use the checkboxes to set different fonts for the front panel and block diagram. For example, you can have a small font on the block diagram and a large one on the front panel.

For information on portability issues regarding the three types of fonts, see the *Resolution and Font Differences section* in Chapter 29, *Portability and Localization Issues*. For information on changing the defaults for the three categories of fonts, see the *Font Preferences* section in Chapter 7, *Customizing Your Environment*.

The Font ring also has **Size**, **Style**, **Justify**, and **Color** menu items, as shown in the following illustration.
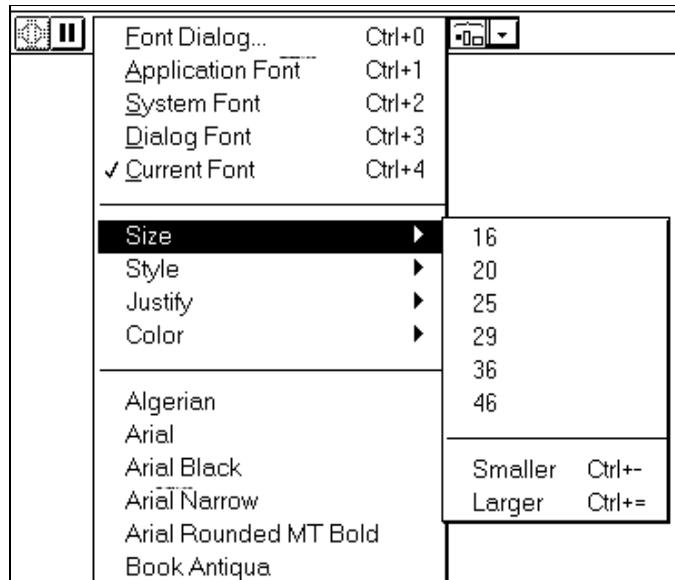
Font selections made from any of these submenus apply to objects you select. For example, if you select a new font while you have a knob and a graph selected, the labels, scales, and digital displays all change to the new

font. The following illustration shows a numeric and a Boolean control being changed from the current font to the System font.
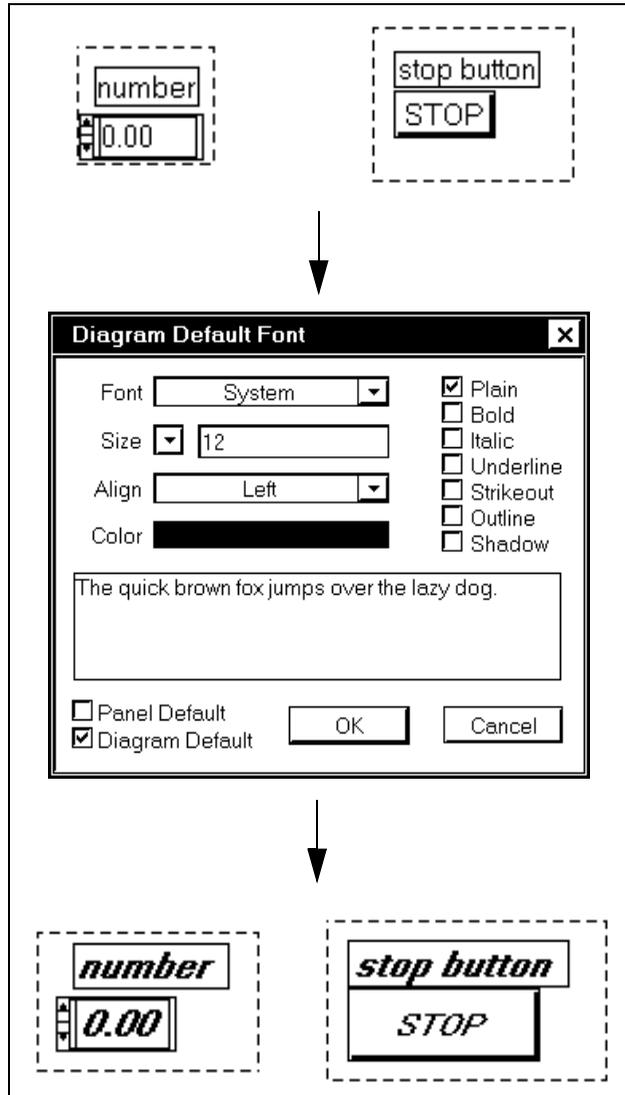
G preserves as many font attributes as possible when you make a change. If you change several objects to Courier font, the objects retain their size and styles if possible. In the same way, changing the size of multiple text selections does not cause the selections to have the same font. These rules do not apply if you select one of the predefined fonts, the current font, or when you use the Font dialog box, which changes the selected objects to the selected font and set of attributes.

When working with objects like slides, which have multiple pieces of text, remember that text selections affect the objects or currently selected text. For example, if you select the entire slide while selecting **Bold**, the scale, digital display, and label all change to a bold font. If you select only the label while selecting **Bold**, only the label changes to bold. If you select text

from a scale marker while selecting **Bold**, all the markers change to bold. These three types of changes are shown in the following illustration.
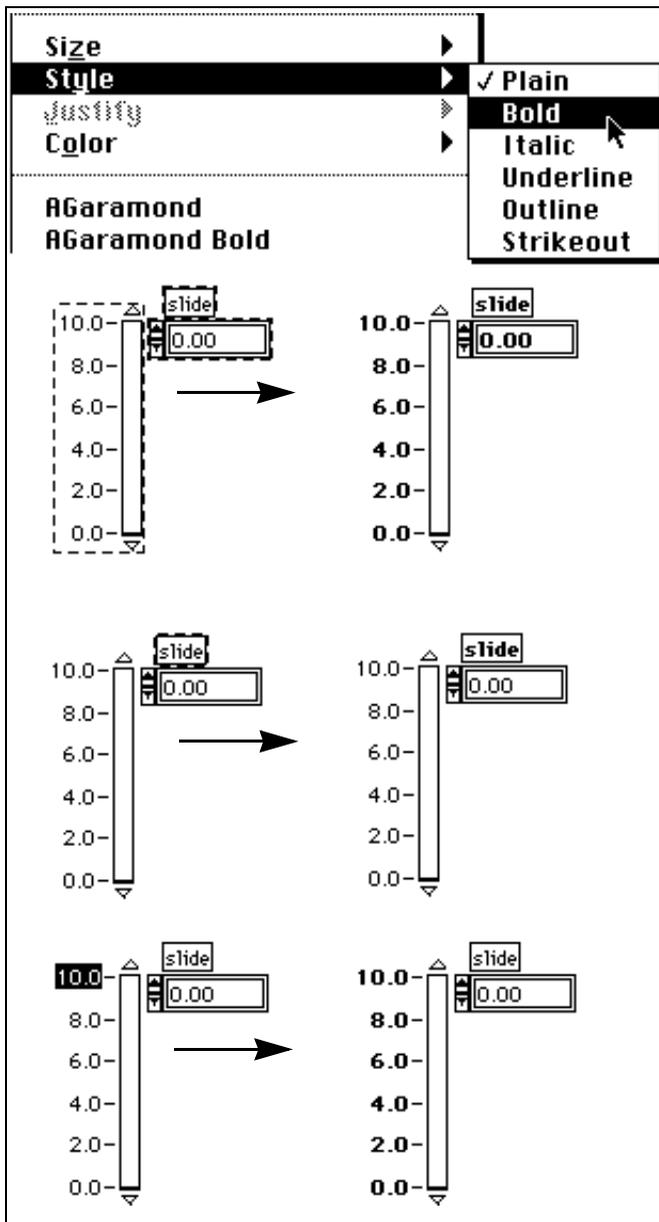
Resizing Objects

You can change the size of most objects. When you move the Positioning tool over a resizable object, resizing handles appear at the corners of a rectangular object, and resizing circles appear on a circular object, as shown in the following illustration.

Resize a circular object by pulling the circles with the cursor. Similarly, you resize a rectangular object by pulling the handles on the corners.

When you pass the tool over a resizing handle, the Positioning tool changes to the *Resizing cursor*. Click and drag this cursor until the dashed border outlines the size you want, as shown in the following illustration.
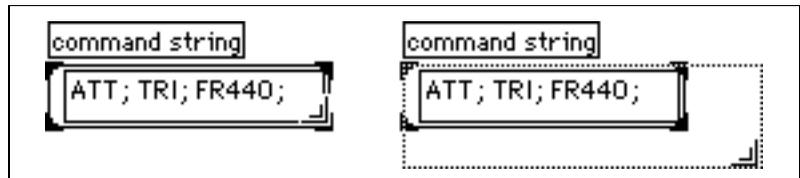
To cancel a resizing operation, continue dragging the frame corner outside the window until the dotted frame disappears. Then release the mouse button. The object maintains its original size.

Some objects only change size horizontally or vertically, or keep the same proportions when you resize it, such as a knob. The resizing cursor appears the same but the dotted resize outline moves in only one direction. To restrict the growth of any object vertically or horizontally, or to maintain the current proportions of the object, hold down the <Shift> key as you click the object and drag it.

Labels

You resize labels as you do other objects, using the resizing cursor. Labels normally *autosize*, which means 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 toggle autosizing on or off.

Adding Work Space

To insert additional work space to your front panel or block diagram, first `<Ctrl-click>` **(Windows)**; `<option-click>` **(Macintosh)**; `<meta-click>` **(Sun)**; `<Alt-click>` **(HP-UX)**, and then drag out a region with the Positioning tool. On the UNIX platform, you can use the middle mouse button to drag out a region without pressing any modifier keys. After clicking, you see a rectangle marked by a dotted line, which defines your new space.

Coloring Objects

Your G development environment 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 G 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.

To change the color of an object or the background of a window, pop up on it with the Color tool from the Tools palette, as shown at the left. The following palette appears in color.
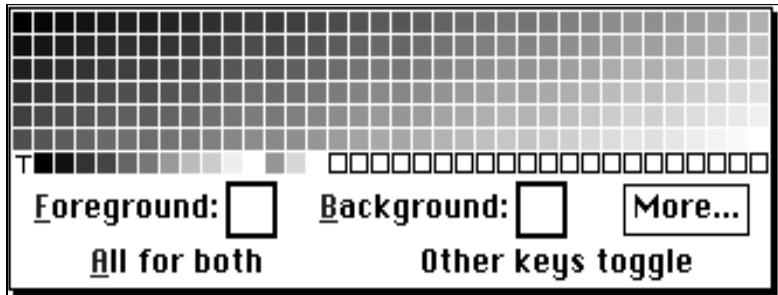
As you move through the palette while pressing the mouse button, the object or background you are coloring redraws with the color the cursor currently is touching. This gives you a preview of the object in the new color. If you release the mouse button on a color, the object retains the selected color. 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, the object becomes transparent. With this feature, you can layer objects. For example, you can place invisible controls on top of indicators, or you can create numeric controls without the standard three-dimensional container. Transparency affects only the appearance of an object. The object responds to mouse and key operations as usual.

Some objects have both a foreground and a background 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 if you are coloring the foreground, the background, or both. A black border around the color indicator for foreground or background indicates what is selected. 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.

Selecting the **More…** menu item from the Color palette accesses a dialog box with which you can customize the colors. The **More Colors** dialog box is shown in the following illustration.
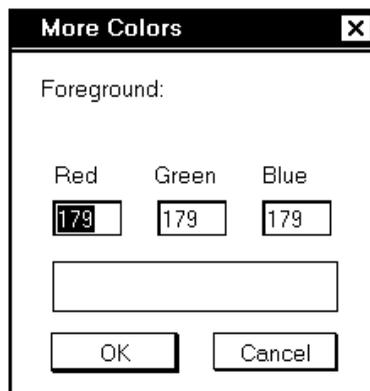
Each of the three color components, red, green, and blue, describes eight bits of a 24-bit color. Therefore, each component has a range of 0 to 255. To change the value of a color component, you can double-click the display of that color and enter the new value. To alter one of the base colors, click 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 an object with the Color tool sets that object to the current color.

You also can copy the color of one object and transfer it to a second object without using the **Color** palette. Click <Ctrl-click> **(Windows)**; <command-click> **(Macintosh)**; <meta-click> **(Sun)**; or <Alt-click> **(HP-UX)** with the Color tool on the object whose color you want to duplicate. The Color tool appears as a dropper and takes on the color of the selected object. Now, you can click another object with the Color tool, and that object takes on the color you just chose.

Undo

In addition to <Ctrl-Z> (or <Cmd-Z> on the Macintosh) for **Undo** and <Ctrl-Shift-Z> (or <Cmd-Shift-Z> on the Macintosh) for **Redo**, you also can set the number of actions to be undone or redone, as well as automatically update VI callers when the VI changes interface. Refer to the *Undo* section in Chapter 7, *Customizing Your Environment,* for more information.

Creating Object Descriptions

If you want to enter a description of a G object, such as a control or indicator, choose **Data Operations**»**Description…** from the object's pop-up menu. You must be in edit mode to edit a description. Enter the description in the dialog box, shown in the following illustration, and click **OK** to save it. This description appears when you subsequently choose **Description…** from the pop-up menu for the object.
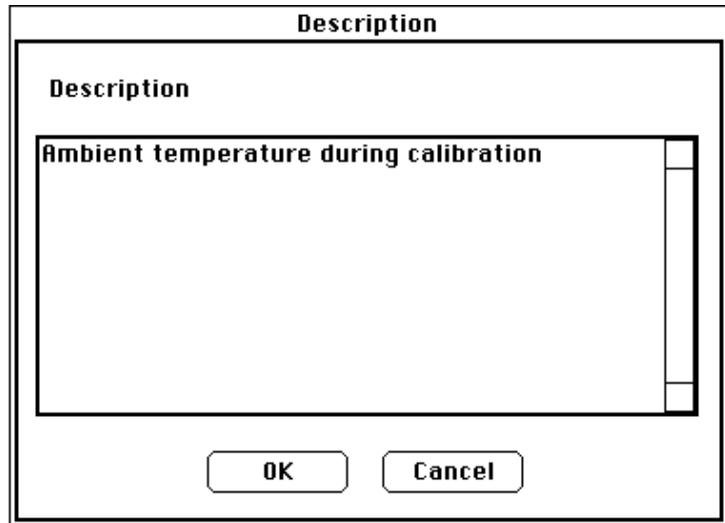
The description of the object also appears in the Help window when you place the cursor over the object. The best way to set up help for the VIs you create is to enter descriptions for all your controls and indicators.

Note   *You cannot edit subVI descriptions from the calling VI diagram. You can edit a VI description through the* **Windows»Show VI Info…** *menu item when the front panel of the VI is open.*

Creating VI Descriptions

Selecting **Windows»Show VI Info…** displays the information dialog box for the current VI. You can use the information dialog box to perform the following functions, as shown in the following illustration.
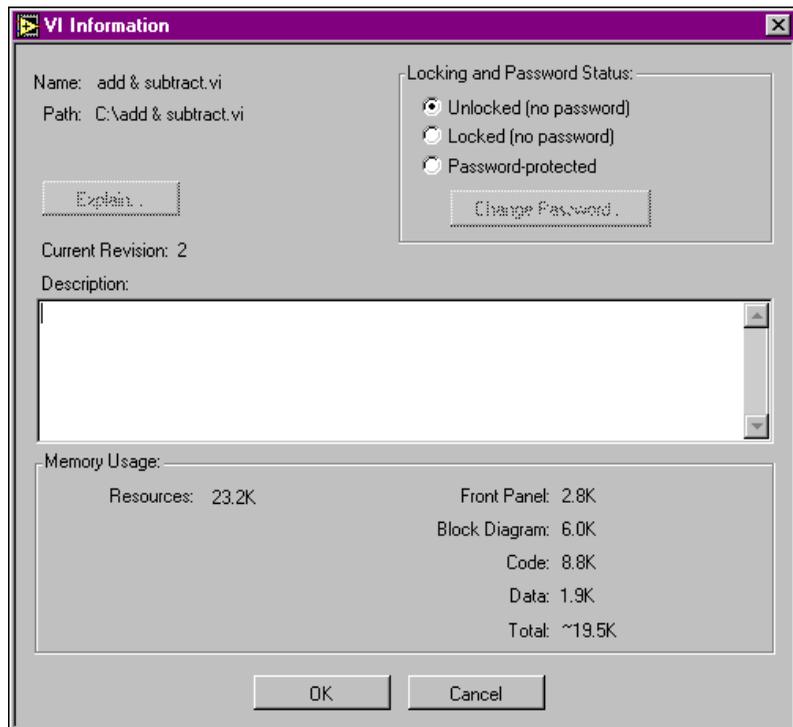
- Enter a description of the VI. The description window has a scrollbar for editing or viewing lengthy descriptions.

- Lock or unlock the VI. You can execute but not edit a locked VI.

- Password protect the VI. See Chapter 27, *Managing Your Applications*, for more information on password protection.

- View the current revision number.

- View the path of the VI.

- View how much memory the VI uses. The **Memory Usage** portion of the information box displays the disk and system memory used by the VI. (This figure only applies to the amount of memory the VI is using and does not reflect the memory used by any of its subVIs.)

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

Saving VIs

You can save VIs as individual files, or you can group several VIs together and save them in a VI library.

Individual VI Files

Five items in the **File** menu concern saving your VIs as individual files.

```
Save              Ctrl+S
Save As...
Save A Copy As...
Save with Options...
Revert...
```
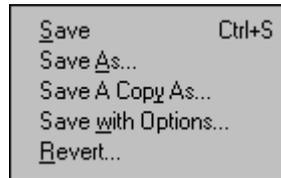
Select the **Save** item to save changes to an existing VI in its current disk location. If the VI is new, a file dialog box appears for you to name the VI and specify its location.

If you want to save a VI with a new name, you can use **Save As…**, **Save a Copy As…**, or **Save with Options…** from the **File** menu.

When you select the **Save As…** menu item, a copy of the VI in memory is saved to disk with the name you choose. After the save is finished, the VI in memory points to the new version. In addition, all callers to the old VI in memory now refer to the new VI. If you enter a new name for the VI, the disk version of the original VI is not overwritten or deleted.

When you select **Save A Copy As…**, a copy of the VI in memory is saved to disk with the name you enter. This does not affect the name of the VI in memory.

Save with Options… brings up a dialog box in which you can choose to save an entire VI hierarchy to disk, optionally saving VIs with passwords or without their block diagrams. This is useful when you are distributing VIs or making backup copies.
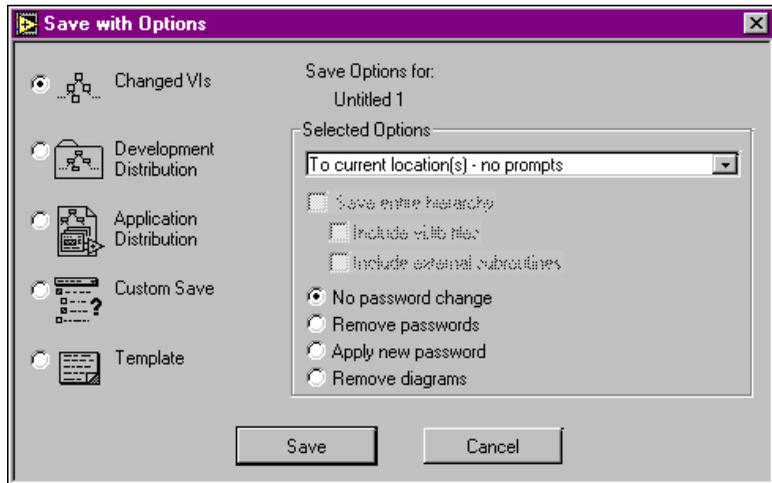
See Chapter 27, *Managing Your Applications*, for information on using this menu item.

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

You can use the **File»Revert…** menu item to return to the last saved version of the VI you are working on. A dialog box appears to confirm whether to discard all changes made to the VI.

VIs modified since they were last saved are marked with an asterisk in their titlebars and in the list of open VIs displayed in the **Windows** menu. When you save a VI, the asterisk disappears until you make a new change.

See Chapter 27, *Managing Your Applications*, for information on saving backup copies.

⚠️ **Caution**   *Do not save your VIs in the* vi.lib *directory found in your* LabVIEW *or* BridgeVIEW *directory. National Instruments updates this directory as needed during new version releases of LabVIEW and BridgeVIEW. Placing VIs in* vi.lib *risks creating a conflict during future installations. If you want your VIs to show up in the* **Functions** *palette, see the Customizing the Controls and Functions Palettes section in* Chapter 7*, Customizing Your Environment, for more information.*

VI Libraries (.LLBs)

You can group several VIs together and save them as a VI library. However, unless you have a good reason for saving them as libraries, it is preferable to save them as individual files, organized in directories. Read the following lists before deciding to save VIs in a library.

Reasons for you to save VIs as libraries.

*   If you are using Microsoft Windows 3.1 or plan to transfer your files to Windows 3.1, saving VIs in libraries (.llb files) enables you to use up to 255 characters to name your files. Other operating systems have long filename support (31 characters on the Macintosh, 255 characters on Windows 95/NT, and UNIX systems.)

*   If you are going to transfer VIs to other platforms, transferring a VI library might be easier than transferring multiple individual VIs.

*   If disk space is an important issue, you can save your files within VI libraries, because they are compressed, slightly reducing disk space requirements for VIs.

Reasons for you to save VIs as individual files.

*   If you store your VIs as individual files, you can use the file system to manage them (copying, moving, renaming, backing up, managing source code).

*   You cannot have hierarchy within a VI library — VI libraries cannot contain subdirectories or folders.

*   Loading and saving files are faster from the file system than from VI libraries. Less disk space is required for temporary files during the load and save processes.

*   Storing VIs and controls in individual files is more robust than storing your entire project in the same file.

*   VI Libraries are not compatible with the source code controls in the Professional Developer Toolkit.

☞ **Note**   *Many of the VIs shipped with G-language software (LabVIEW and BridgeVIEW) are stored in VI libraries. This makes for consistent storage locations on all platforms.*

Creating VI Libraries

To create a VI library, take the following steps.

1. Select **Save as…**, or **Save a Copy As…** from the **File** menu.

2. **(Macintosh)** If you configure through **Preferences»Miscellaneous** to use native file dialog boxes, click **Use LLBs** in the dialog box that appears.

3. In the dialog box appearing after step 1 or step 2, click the **New…** or the **New VI Library** button.

4. In the dialog box that appears, enter the name of the new library and click the **VI Library** button. G-language software adds an .llb extension to the name of the library automatically, if you do not.

5. A dialog box appears, ready for you to name your VI if you wish, and save it in the new library.

Saving in Existing VI Libraries

You can save new VIs in a VI library just as if it were a directory. After selecting one of the Save menu items from the **File** menu, the name of your library file, with the .llb extension, appears in the file dialog box. (Macintosh users using the native file dialog box must press the **Use LLBs** button first, before selecting a VI library.) When you double-click the name of the library file or press **Open**, the dialog box presents an item so you can save the file in the library.

Editing the Contents of Libraries

You can remove a file from a VI library using the **File»Edit VI Library…** menu item. When the Edit VI Library dialog box appears, you also can mark files as **Top Level**. This has two uses. First, when you create an application using the application builder libraries, the **Top Level** setting indicates which VIs open automatically when you run the application. Second, if you double-click a specific library from the file system on Windows or Macintosh, or if you launch your G development environment with a library name specified on a command line under Windows or UNIX, G opens all top level VIs automatically.

Top-level VIs are also separated by a dividing line from all other VIs in the library when viewing the .llb contents in a file dialog.

3

Using SubVIs

This chapter describes the concept of hierarchical design in G applications and explains two methods of creating subVIs. The chapter also describes two utilities—the Hierarchy window, which displays the hierarchy of your VIs, and the Find utility, which finds occurrences of subVIs, as well as other objects or strings of text that you indicate.

Hierarchical Design

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

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

You can create a subVI from a VI, or you can create a subVI from a selection (portion) of a VI. Both methods are described in this chapter.

Creating SubVIs from VIs

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

Creating the Icon

To create an icon, either double-click the icon in the top-right corner of the front panel, or pop up on the icon and select **Edit Icon…**, as shown in the following illustration.
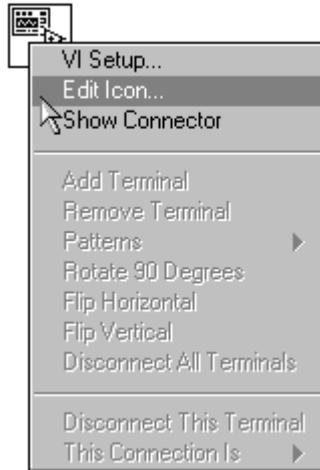
After you select **Edit Icon…**, the Icon Editor appears.
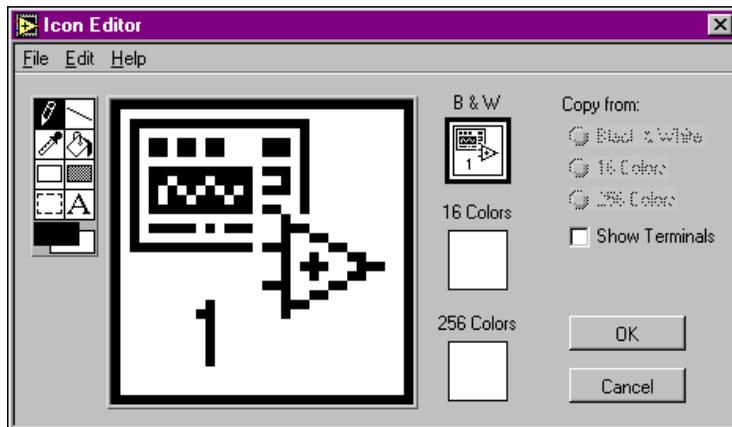
Use the tools to the left of the window, shown in the preceding illustration, to create the icon design in the fat-pixel editing area. The normal-size image of the icon appears in the appropriate box to the right of the editing area.

Depending on the type of monitor you are using, you can design a separate icon for display in monochrome, 16-color, and 256-color mode. You design and save each icon version separately. The editor defaults to **Black & White**, but you can click one of the other color menu items 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** menu items at the right of the Icon Editor.

☞ **Note**   *It is best to create at least a black-and-white icon. If you design a color icon only, it does not show up if you add it to the* **Functions** *palette. Also, it cannot be printed and it does not show up on a black-and-white monitor. In these cases, if a VI does not have a black-and-white icon, the software uses the blank icon. For more information, see the* Customizing the Controls and Functions Palettes *section in* Chapter 7*,* Customizing Your Environment*.*

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 a color to be the foreground color from an element in the icon. Use the <Shift> key to select the background color with the dropper.

fill bucket—Fills an outlined area with the foreground color.

rectangle—Draws a rectangular border in the foreground color. Double-click this tool to frame the icon in the foreground color. Use the <Shift> key to constrain the rectangle to be a square.

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. Use the <Shift> key to constrain the rectangle to a square.

select—Selects an area of the icon for moving, copying, or deleting. Double-click to select the entire icon. Pressing the <Delete> key erases the selected portion of the icon. Use the <Shift> key to constrain the rectangle to a square.

text—Enters text into the icon. Double-click this tool icon to select a different font. Use the cursor keys to move text around the icon.

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

Holding down the <Ctrl> **(Windows)**; <option> **(Macintosh)**; <meta> **(Sun)**; or <Alt> **(HP-UX)** key temporarily changes all of the tools to the dropper.

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

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

Cancel—Returns to the front panel without saving any changes.

☞ **Note**   *You can use the* **Edit** *menu to cut, copy and paste images from and to the icon. When you paste an image and a portion of the icon is selected, the image is resized to fit into the selection.*

Defining Connector Terminal Patterns

You send data to and receive data from a subVI through the terminals in its connector pane. You define connections by choosing the number of terminals you want for the VI and by assigning a front panel control or indicator to each of those terminals. Only the controls and indicators you use programmatically by wiring to the subVI require terminals on the connector pane.

If the connector for your VI is not displayed already in the upper right corner of the front panel, choose **Show Connector** from the icon pane pop-up menu, as shown in the following illustration. The block diagram does not have a connector pane.
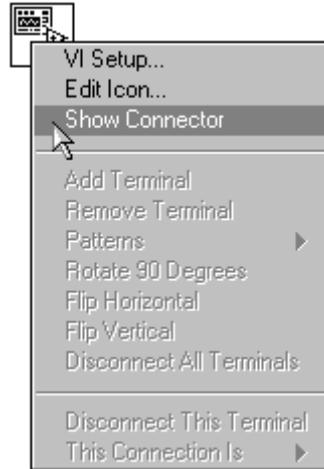
The connector replaces the icon in the upper right corner of the front panel. The initial selection is a terminal pattern with as many terminals on the left of the connector pane as controls on the front panel, and as many terminals on the right of the connector pane as indicators on the front panel. If this is not possible, the software selects the closest match.

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.
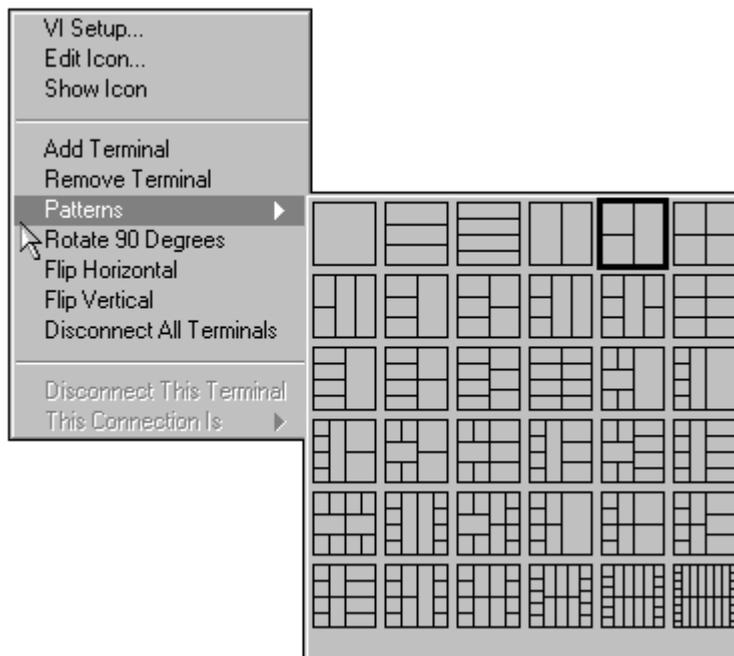
A solid border highlights the pattern currently associated with your icon. To change the pattern, choose a new one.

If you want to add a terminal to the pattern, place the cursor where the terminal is to be added, pop up, and select **Add Terminal**.

If you want to remove an existing terminal from the pattern, pop up on the terminal and select **Remove Terminal**.

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**.

Assigning Terminals to Controls and Indicators

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

1. Click a terminal of the connector. The tool automatically changes to the Wiring tool. The terminal turns black.
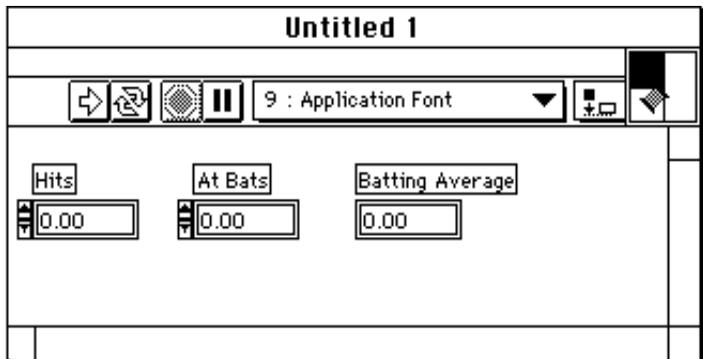
2. Click the front panel control or indicator you want to assign to the selected terminal. A marquee frames the selected control.
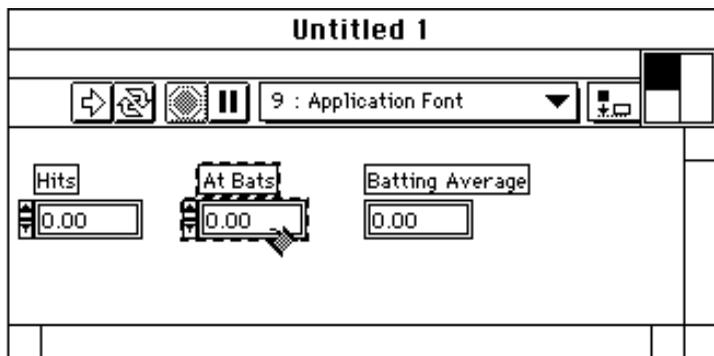
3. Position the cursor in an open area of the front panel and click it. The marquee disappears and the selected terminal takes on the data color of the connected control, indicating the terminal is assigned.
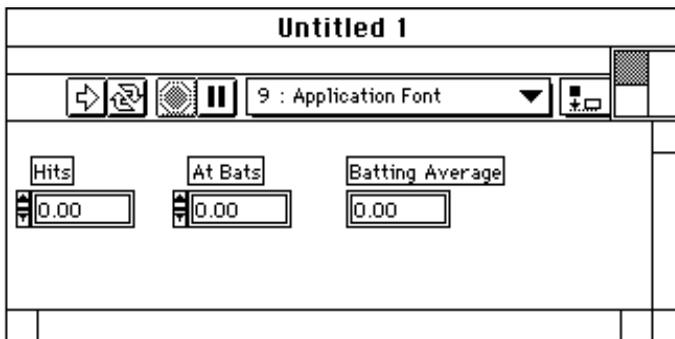
☞ Note   *If the connector terminal turns white, a connection was not made. Repeat steps 1 through 3 until the connector terminal takes on the proper data color.*

Although you use the Wiring tool to assign terminals on the connector to front panel controls and indicators, no wires are drawn between the connector and these controls and indicators.

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

You also can select the control or indicator first, and then select the terminal. You can choose a pattern with more terminals than you need. You can leave some extra terminals unconnected if you anticipate making changes to the VI in the future that require a new input or output. Having the extra connections available means that the new input or output does not affect other VIs that already are using this VI as a subVI. Unassigned terminals do not affect the operation of the VI. You also can have more front panel controls than terminals.

The connector pane has, at most, 28 terminals. If your front panel contains more than 28 controls and indicators that you want to use programmatically, group some of them into a cluster and assign the cluster to a terminal on the connector. For more information see the *Clusters* section in Chapter 14, *Array and Cluster Controls and Indicators*.

Required, Recommended, and Optional Connections for SubVIs

G has a feature that can keep you from forgetting to wire subVI connections—indications of required, recommended, and optional connections in the connector pane and the same indications in the Help window.

Inputs and outputs of VIs that come in `vi.lib` are premarked as **Required**, **Recommended**, or **Optional**. G sets inputs and outputs of VIs you create to **Recommended** by default.

When an input is marked as **Required**, you cannot run the VI as a subVI without wiring it correctly. When an input or output is marked as recommended, you can run the VI, but the Error List window lists a warning if warnings are enabled.

To see whether connections are **Required**, **Recommended**, or **Optional**, or to mark them as one of these states, click a terminal in the connector pane and select a menu item from the **This Connection Is** submenu. A checkmark indicates its status, as shown in the following illustration.
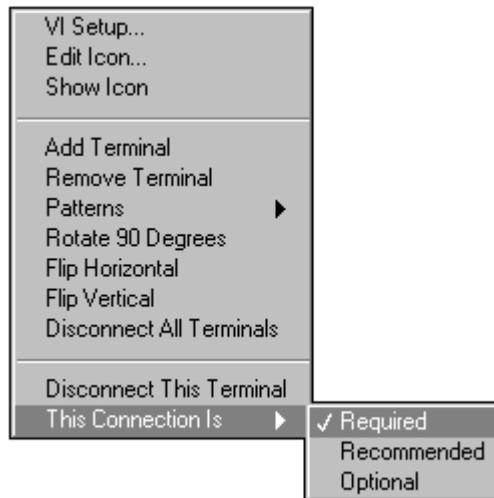
In the Help window, **Required** connections appear in bold, **Recommended** connections are in plain text, and **Optional** connections are in grayed-out text.

Deleting Terminal Connections

You can delete connections between terminals and their corresponding controls or indicators individually or all at once. To delete a particular connection, pop up on the terminal you want to disconnect on the connector and choose **Disconnect This Terminal** from the pop-up menu, as shown in the following illustration.

```
VI Setup...
Edit Icon...
Show Icon

Add Terminal
Remove Terminal
Patterns                    ▶
Rotate 90 Degrees
Flip Horizontal
Flip Vertical
Disconnect All Terminals

Disconnect This Terminal
This Connection Is          ▶
```
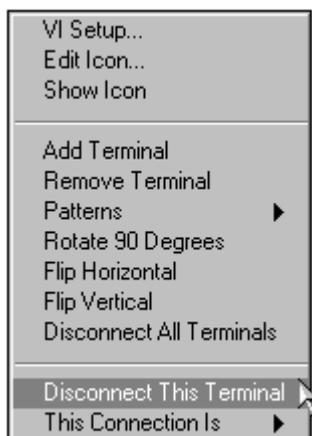
The terminal turns white, indicating the selected connection no longer exists. To delete all connections on the connector, choose **Disconnect All Terminals** from the pop-up menu anywhere on the connector. Notice that this is different from **Remove Terminal** in that **Disconnect this Terminal** does not remove the terminal from the pattern.

Confirming Terminal Connections

To see which control or indicator is assigned to a particular terminal, click a control, indicator, or terminal with the Wiring tool when the connector pane is visible. G selects the corresponding assigned object.

Creating SubVIs from VI Selections

You can convert a portion of a VI into a subVI to be called from another VI. You select a section of a VI, select **Edit»Create SubVI**, and the section becomes a subVI. Controls and indicators are created automatically for the new subVI, the subVI is wired automatically to the existing wires, and an icon of the subVI replaces the selected section of the block diagram in your original VI.

Creating a subVI from a selection is the same as removing the selected objects and replacing them with a subVI, except for the following behaviors.

- None of the front panel terminals included in the selection are removed from the caller VI. Instead, the front panel terminals are retained on the caller VI and are wired to the subVI.

- All attributes included in the selection are retained on the caller VI and are wired to the subVI. The selected attributes are replaced by front panel terminals in the subVI which act as channels for transferring the value of an attribute, in or out of the subVI.

- When a selection includes a local variable, it is replaced by a front panel terminal in the subVI. The local variable is retained on the caller VI and is wired to the subVI. When more than one instance of the same local variable is selected, the first instance becomes a front panel terminal and the rest remain as local variables in the subVI. Only one instance of the local variable is retained on the caller VI; the rest of the selected instances are removed from the caller and moved to the subVI.

- Because local variables can either read or write from a front panel terminal, when instances of both read and write local variables are selected, two front panel objects are created in the subVI—one to pass the value of the local variable into the subVI and another to pass the value out of the subVI. The front panel terminal created to represent the read local variables has the suffix `read` added to its name, and the front panel terminal created to represent the write locals has the suffix `write` added to its name.

Rules and Recommendations

The ability to create a subVI from a selection can be a great convenience but might not be as simple as it seems. Careful planning still is required to create a logical hierarchy of VIs. You must also consider which objects to include in the selection and thereby avoid situations where the functionality of a resulting VI is changed. In cases where a problem occurs, a subVI is not created out of the selection, but a dialog box appears explaining why the operation failed. In cases where there is only a potential problem, a dialog box appears with an explanation of the potential problem. You then must decide whether to continue.

☞ **Note**   *If there are no local variables or attribute nodes in a selection, you might want to use a Sequence Structure to encapsulate the code you want sent to a subVI and preview the results.*

The following rules and recommendations can help you use this feature effectively.

Number of Connections

Do not make very large selections that create a subVI with more than 28 inputs and outputs, the maximum number of connections on a connector pane.

Keep in mind each front panel terminal, each attribute, and certain local variables included in the selection require a slot in the connector pane; a selection with a large number of these items might run out of connector pane slots.

To avoid exceeding the maximum terminals, select a smaller section of the diagram, or group data into arrays and clusters before selecting a region of the diagram to convert.

Cycles

Avoid making selections that create cycles in the diagram. Cycles occur if a data flow originates from an output of the subVI and terminates as an input to the subVI.

Identifying cycles while making selections is difficult, but G detects them for you. When G detects a cycle, it displays a dialog box prompting you either to create a new VI from the selection or to cancel. If you choose to create a new VI, a new Untitled VI is created from your selection. The selected items in the original diagram are left untouched.

Attribute Nodes within Loops

Do not include attribute nodes within a loop in your selection.

Because an attribute node is retained on the caller VI and wired to the subVI, execution of the subVI does not update the value of the attribute on every iteration of the loop. You cannot make a subVI in such cases.

Illogical Selections

Do not convert selections into a subVI that do not make sense. For example, it does not make sense to convert a selection consisting of one object inside a Sequence Structure and another object outside of the Sequence Structure without including the Sequence Structure itself.

Such selections invoke a display with an explanation of the problem.

Locals and Front Panel Terminals within Loops

Try to avoid including local variables or front panel terminals inside a loop in your selection.

Because selected front panel terminals or locals are retained on the caller VI and wired to the subVI, execution of the subVI does not update the value of these items on every iteration of the loop. This can cause a change in the functionality of the caller VI.

In such cases a warning display prompts you to continue or cancel.

Case Structures Containing Attribute Nodes, Locals, or Front Panel Terminals

Try to avoid including a Case Structure containing an attribute node, front panel terminal, or local variable to which a value is written.

In such cases, when the subVI is executed, a specific value always is written to the object because you wire it to the subVI. Although it exists in the original block diagram, a value can be written only when either the TRUE or FALSE part of the Case Structure executes. This can change the functionality of the caller VI.

In such cases a warning display prompts you to continue or cancel. If you continue, you first must edit the subVI to supply a value in all cases.

Consider the following illustration as an example.
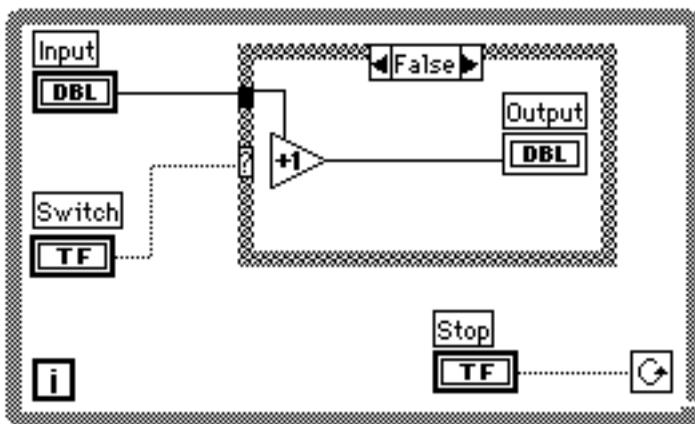
The Case Structure is selected for conversion into a subVI. The front panel terminal Output is left out in the original diagram and connected to the subVI through a connector as shown in the following illustration.
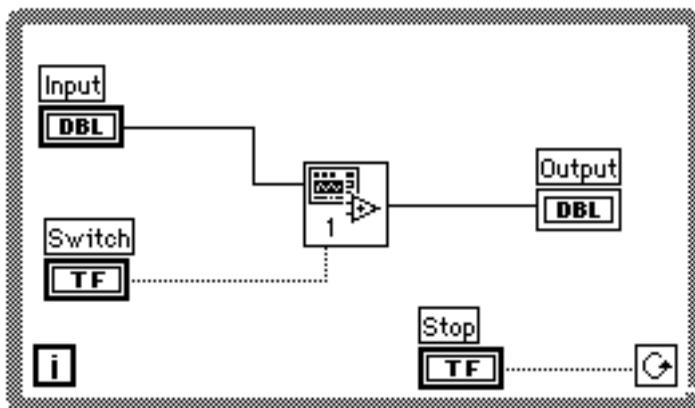
In the original block diagram, G writes a value to Output only when the FALSE part of the case executed. However, when the selection turns into a subVI, G always writes a value to Output regardless of whether the TRUE or FALSE part of the case was executed within the subVI. You need to edit the subVI to supply data for the extra cases.

Using the Hierarchy Window

The Hierarchy window displays a graphical representation of the calling hierarchy for all VIs in memory, including type definitions and global variables.

You can configure many aspects of the display in the Hierarchy window. For example, you can display the layout horizontally or vertically, and include or exclude VIs in `vi.lib`, global variables, or type definitions.

Other useful features of the Hierarchy window include the ability to access the **VI Setup…**, **Edit Icon…**, **Show VI Info…**, and **Print Documentation…** menu items, the ability to drag or copy and paste hierarchy nodes to a block diagram of another VI as a subVI, and the ability to search for hierarchy nodes by name.

Opening the Hierarchy Window

You can open the Hierarchy window in any of the following ways.

- Select **Project»Show VI Hierarchy**. The Hierarchy window is shown with the VI of the current active window as the focus node.

- Under the pop-up menus of subVIs, global variables, or type definitions, select **Show VI Hierarchy**. The Hierarchy window appears with the selected subVI, global variable, or type definition as the focus node.

- If the Hierarchy window is already open, you can bring it to the front by selecting it from the list of open windows under the **Windows** menu.

Switch the Hierarchy window between horizontal and vertical display through a menu item in the **View** menu or by pressing the **Horizontal Layout** or **Vertical Layout** button at the top of the window.

In a horizontal display, subVIs are shown to the right of their calling VIs; in a vertical display, they are shown below their calling VIs. SubVIs always

are connected with lines to their calling VIs. The window shown in the following illustration is displayed horizontally.
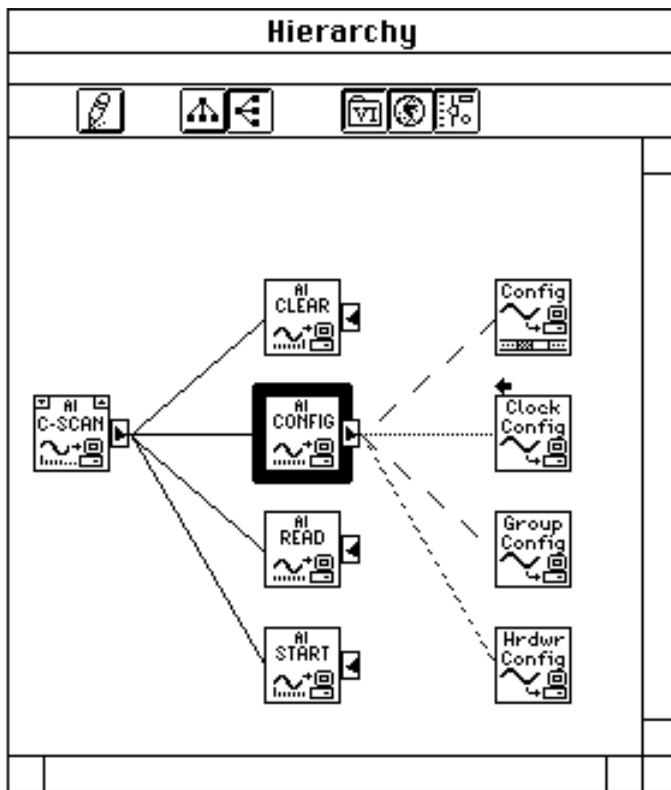
Arrow buttons and arrows beside nodes indicate what is displayed and what is hidden, as follows.

- • A red arrow button pointing towards the node indicates some or all subVIs are hidden. Clicking the button shows the immediate subVIs of the node.

- • A black arrow button pointing towards the subVIs of the node indicates all immediate subVIs are shown.

- • A blue arrow pointing towards the callers of the node indicates the node has additional callers not shown.

If a node has no subVIs, no red or black arrow buttons are shown.

A node becomes the focus node, surrounded by a thick red border, when an operation is performed on it. It is defocused when you perform an action on another node.

As you move your cursor over objects in the Hierarchy window and your cursor becomes idle over a node, G displays the name of the node below the icon of the node. If you prefer, you can use a menu item in the **View** menu to show the full path instead of the name.

You can double-click any icon in the Hierarchy window to open the associated VI.

If you show the Help window and move the cursor over an icon, G displays the Help window information for that VI.

Hierarchy Window Options

Several options let you control the display of the Hierarchy window; others let you perform actions on nodes displayed in the window. Options are available in the **View** menu, in buttons near the top of the Hierarchy window, in the pop-up menu of a node, and through mouse clicks. You also can access the **View** menu by popping up on an empty space in the Hierarchy window.

View Menu Options

The **View** menu contains the following menu items related to the display of the Hierarchy window. Many of the menu items are also available from buttons near the top of the window.

```
┌─────────────────────────────┐
│ View│                       │
├─────────────────────────────┤
│    Redraw              ⌘D    │
│    Show All VIs        ⌘A    │
├ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┤
│ ✓ Vertical Hierarchy        │
│    Horizontal Hierarchy     │
├ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┤
│    Include VIs in vi.lib    │
│ ✓ Include Globals           │
│    Include Type Defs        │
├ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┤
│    Full VI Path in Label    │
└─────────────────────────────┘
```
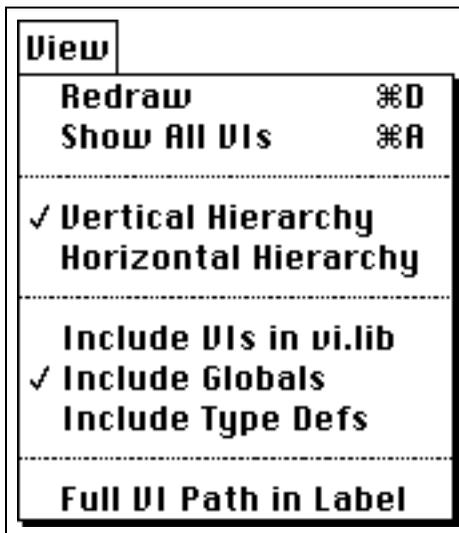
- **Redraw**—Redraws the window layout to minimize line crossings and maximize symmetry, a useful option after successive operations on hierarchy nodes. If a focus node exists, the window scrolls to show that node. If no focus node exists, the window scrolls to show the first root of subVIs.

- **Show All VIs**—Displays all hidden VIs. There is no focus node after this menu item is selected. This does not affect other settings in the **View** menu. In other words, VIs in vi.lib, global variables, and type definitions not included in the hierarchy remain hidden.

- **Vertical Hierarchy**—Arranges the nodes from top-to-bottom with the calling VIs above their subVIs.

- **Horizontal Hierarchy**—Arranges the nodes from left-to-right with the calling VIs to the left of their subVIs.

- **Include VIs in vi.lib**—Toggles the Hierarchy window to include or exclude VIs in vi.lib.

- **Include Globals**—Toggles the Hierarchy window to include or exclude globals.

- **Include Type Defs**—Toggles the Hierarchy window to include or exclude type definitions.

- **Full VI Path in Label**—Toggles the Hierarchy window to display either the full VI path or simply the name of the VI in the Tip strip for each hierarchy node.

Hierarchy Toolbar Buttons

A toolbar in the Hierarchy window contains buttons affecting the display of the window.

The buttons, which perform some of the same actions as items in the **View** menu, are described as follows.

Redo Layout—Redraws the window layout to minimize line crossings and maximize symmetry, which is useful after successive operations on hierarchy nodes. If a focus node exists, the window scrolls to show that node. If no focus node exists, the window scrolls to show the first root of subVIs.

Vertical Layout—Arranges the nodes from top to bottom with the calling VIs above their subVIs.

Horizontal Layout—Arranges the nodes from left to right with the calling VIs to the left of their subVIs.

Include VI Lib—Toggles the Hierarchy window to include or exclude VIs in `vi.lib`.

Include Globals—Toggles the Hierarchy window to include or exclude globals.

Include Type Definitions—Toggles the Hierarchy window to include or exclude type definitions.

Hierarchy Node Pop-Up Menu

If you pop up on a node (subVI, global variable, or type definition) displayed in the Hierarchy window, a menu appears with items for controlling the display or carrying out commands related to the selected node.
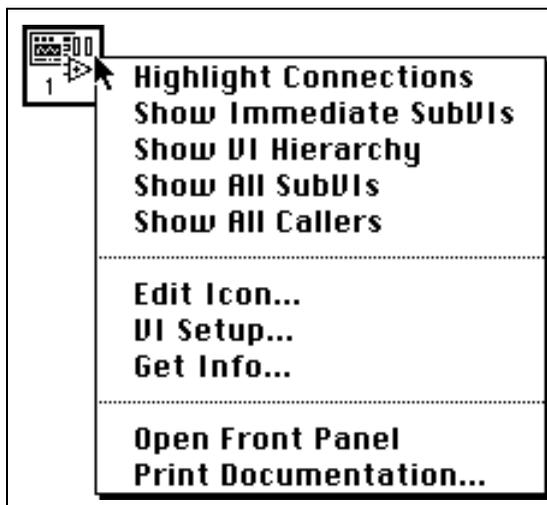
The menu items are described as follows.

- **Highlight Connections**—Makes the selected node the focus node and highlights in red the edges connecting its immediate callers and subVIs.

- **Show Immediate SubVIs**—If the node is hiding all or some subVIs, this menu item expands the node to show all of its immediate subVIs. The edges connecting the node to its subVIs are highlighted in red. The menu item is toggled with **Hide All SubVIs**.

- **Hide All SubVIs**—If a node is showing all immediate subVIs, this menu item collapses the node to hide its entire subVI chain. This is toggled with **Show Immediate SubVIs**.

- **Show VI Hierarchy**—Makes the selected node the focus node and displays the nodes belonging to its call chain and subVI chain. Unrelated roots are also visible, but all of their subVIs are hidden. The edges connecting the call chain of a node and subVI chain are highlighted in red.

- **Show All SubVIs**—Makes the selected node the focus node and expands its entire subVI chain. The edges connecting the subVI chain of the node are highlighted in red.

- **Show All Callers**—Makes the selected node the focus node and expands its entire call chain. The edges connecting the call chain are highlighted in red.

- **Edit Icon…**—Displays the Icon Editor for editing of the icon of the node.

- **VI Setup…**—Displays the VI Setup dialog box for the node.

- **Get Info…**—Displays the Get Info dialog box for the node.

- **Open Front Panel**—Opens the front panel of the VI, global, or type definition.

- **Print Documentation…**—Brings up the Print Documentation dialog box, from which you can choose the portion of the VI you want to print. For more information about this dialog box, see the *Printing Documentation* section of Chapter 5, *Printing and Documenting VIs*.

Hierarchy Node Mouse-Click Sequences

Mouse-click sequences are available for selecting nodes, for copying or dragging nodes, and for shortcuts to certain pop-up menu items. Where applicable, the sequences are performed with the Positioning tool selected.

The shortcut mouse clicks are as follows. For a more complete description of items available in the pop-up menu, see the section *Hierarchy Node Pop-Up Menu* in this chapter.

- Clicking the red arrow button executes the Show Immediate SubVIs action.

- <Shift>-clicking the red or black arrow button executes the Show All SubVIs action.

- Pressing <Ctrl-click> **(Windows)**; <option-click> **(Macintosh)**; <meta-click> **(Sun)**; and <Alt-click> **(HP-UX)**; on the node executes the Show VI Hierarchy action.

- Double-clicking the node executes the Open Front Panel action.

- Single-clicking the node selects it for dragging to a block diagram or copying to the Clipboard to use as a subVI.

- By <Shift>-clicking and holding the mouse button down while selecting, you can select multiple nodes for copying to other block diagrams or front panels. You also can make multiple selections by dragging a rectangle over the objects you want to select.

- Pressing the <Tab> key toggles between the Positioning and the Scroll tool of the **Tools** palette.

Finding VIs in the Hierarchy Window

You can initiate the search by simply typing in the name of the node. A small Search window appears displaying the text you typed. The search takes place immediately and highlights a matching node by displaying a Tip strip of its name.

In the following illustration the Find Hierarchy Node mechanism has found the node named `drawJoint.vi`.
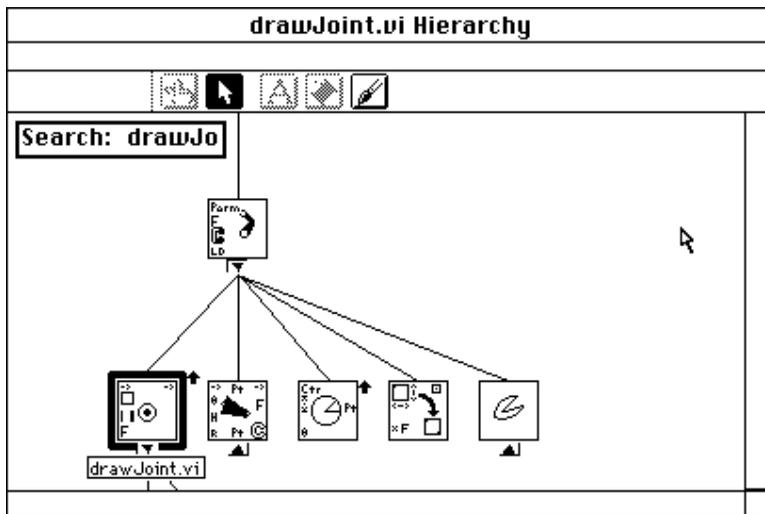
A search is performed as you type; thus, when no node matches the characters currently displayed in the Search window, the system beeps and no more characters can be typed. You then can use the `<Backspace>` or `<Delete>` key to delete one or more characters, and resume typing.

The Search window disappears automatically if no keys are pressed for a certain amount of time. You can press the `<Esc>` key to remove the Search window immediately.

When a match is made on a node, you can use the right or down arrow key, or the `<Enter>` on Windows and Sun or the `<Return>` key on Macintosh and HP-UX, to find the next node matching the search string. To find the previous matching node, you can press the left or up arrow key, or `<Shift-Enter>` on Windows and Sun or `<Shift-Return>` on Macintosh and HP-UX.

Finding VIs, Objects, and Text

When developing an application consisting of multiple subVIs or even a single large VI, you might want to find occurrences of a particular object or string of text. The **Project»Find** command can help you find all instances of the following objects with the names you indicate.

- VIs
- Built-in functions
- Type definitions
- Global variables
- Local variables
- Attribute nodes
- Breakpoints
- Front panel terminals
- Text

For information on type definitions, see the *Type Definitions* section in Chapter 24, *Custom Controls and Type Definitions*. For information on local and global variables, see Chapter 23, *Global and Local Variables*. For information on attribute nodes, see Chapter 22, *Attribute Nodes*. For information on breakpoints, see the *Placing Breakpoint Tools* section of Chapter 4, *Executing and Debugging VIs and SubVIs*.

In addition to the **Find** command, many objects have pop-up menu items to help you quickly find related objects. For example, if you pop up on a control, a menu item helps you find the corresponding terminal as well as any local variables or attribute nodes associated with it.

Find Dialog Box

To bring up the Find dialog box, select **Project»Find…**, or press `<Ctrl-f>` **(Windows)**; `<command-f>` **(Macintosh)**; `<meta-f>` **(Sun)**; or `<Alt-f>` **(HP-UX)**.

As a shortcut, you can select a piece of text or an object before accessing the dialog box. The dialog box appears, with the text or object preselected for the search.

Finding VIs and Other Objects

To search for VIs or other objects, click the **Objects** button in the
Find dialog box.
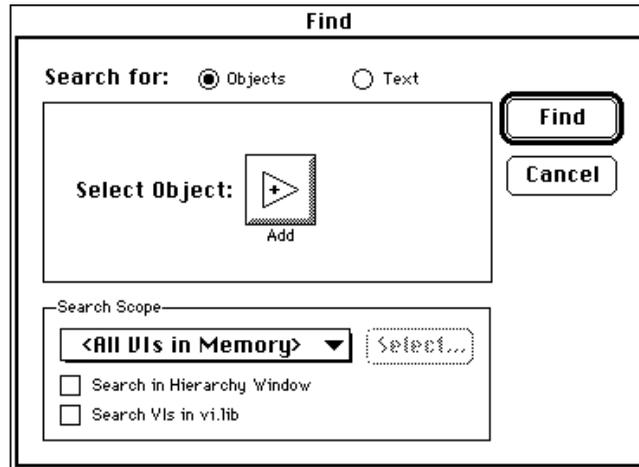
To select the object you want to search for, click the button after the words
**Select Object**. The **Select Objects** menu appears.
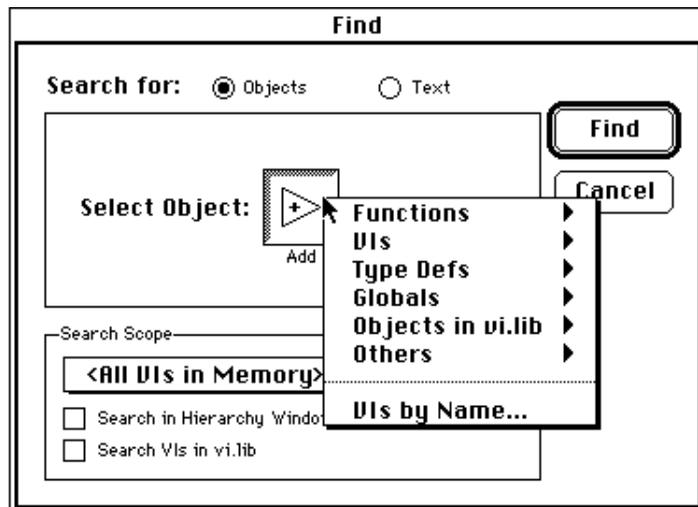
The items available in this menu are as follows.

•   **Functions**—Brings up the **Functions** palette so you can select built-in
     functions, VIs, or any objects you customized to appear in the
     Functions palette.

- **VIs**—Displays a palette of all VIs in memory not located in `vi.lib`.

- **Type Defs**—Displays a palette of all type definitions in memory not located in `vi.lib`.

- **Globals**—Displays a palette of all global VIs in memory not located in `vi.lib`.

- **Objects in vi.lib**—Displays all VIs, type definitions, and global variables located in `vi.lib`.

- **Others**—Displays a menu from which you can select attribute nodes, breakpoints, and front panel terminals.

- **VIs by Name…**—Invokes the Select VIs by Name dialog box, which displays all VIs in memory by name in alphabetical order. You can type the name of the object to jump quickly to the item in the list you want. With the three checkboxes (**VIs**, **Globals**, and **Type Defs**), you can include and exclude different types of objects to be displayed in the list box.

Finding Text

To search for specific text, click the **Text** button in the Find dialog box.

```
┌───────────────────────────────────────────────────────┐
│                         Find                           │
│  ┌──────────────────────────────────────────────────┐  │
│  │  Search for:  ○ Objects   ● Text                  │  │
│  │  ┌────────────────────────────────┐  ┌─────────┐  │  │
│  │  │ Find What                      │  │  Find   │  │  │
│  │  │ test.vi│                       │  └─────────┘  │  │
│  │  └────────────────────────────────┘  ┌─────────┐  │  │
│  │  ☐ Match Case                        │ Cancel  │  │  │
│  │  ☐ Match Whole Word   ┌────────────┐ └─────────┘  │  │
│  │  ☐ Regular Expression │ More Options... │         │  │
│  │  ┌─Search Scope──────────────────────┐            │  │
│  │  │ <All VIs in Memory>  ▼  │ Select... │           │  │
│  │  │ ☐ Search in Hierarchy Window      │            │  │
│  │  │ ☒ Search VIs in vi.lib            │            │  │
│  │  └───────────────────────────────────┘            │  │
│  └──────────────────────────────────────────────────┘  │
└───────────────────────────────────────────────────────┘
```
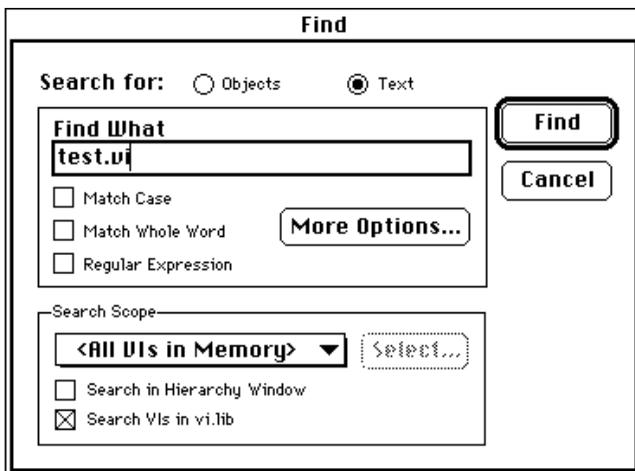
Search String Options

You can type in the text you want to find, and optionally limit the scope of the search to specific areas of VIs (front panel, block diagram, show VI info text, or history information) or components of objects, such as the label or description.

The following menu items are available for indicating the search string.

• **Match Case**—Finds only text strings exactly matching the case (uppercase or lowercase) of the characters you type in.

• **Match Whole Word**—Finds text strings only if they are preceded and followed by a non-alphanumeric character such as a space or a plus sign (+), or the start or end of a line. (If this menu item is not selected, the command matches any string, regardless of if it is a fragment of a larger string or not.)

• **Regular Expression**—Handles the character string as a regular expression. The regular expression has the same specifications the Match Pattern function (see **Online Help»Function and VI Reference** for those specifications).

Text Search Options

The **More Options…** button invokes the Text Search Options dialog box, which has menu items for configuring the scope of the text search for each VI.
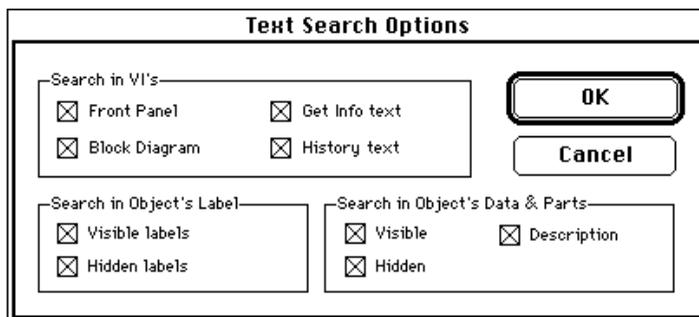
The menu items indicate if and where to search visible or hidden text. They are as follows.

• **Search in VI's**—Indicates whether to search the front panel of a VI, block diagram, Get Info text, or History window text. At least one of these must be checked.

• **Search in Object's Label**—Indicates whether to search visible or hidden object name labels.

• **Search in Object's Data & Parts**—Indicates if you need to search visible or hidden data and parts.The data and parts of an object consist of everything belonging to an object except for the name label of the object. You also have the option of searching the description of various

objects. At least one of the menu items in the **Search in Object's Label** and **Search in Object's Data & Parts** must be checked.

☞ **Note**   *The Text Search options are saved from search to search; therefore, if you do not find a string of text expected to exist, make sure the Text Search Options are set appropriately.*

Narrowing the Search Scope

Three items in the **Search Scope** ring menu indicate the VIs to be searched.

- **All VIs in Memory**—Indicates the search scope as all VIs currently loaded. Two checkboxes are available to include or exclude VIs in `vi.lib` and the Hierarchy window.

- **Selected VIs**—Selects a custom set of VIs to search. The Hierarchy window can be included or excluded from the search scope. The **Select…** button brings up the **Select VIs** menu item in the Search dialog box.

- **Name of a VI**—Searches a specific VI only. The name of the VI in this ring item depends on the active VI when you invoked the Find dialog box.

Click the **Find** button to begin searching for the text you selected.

Search Results Window

After you complete a search, if there is more than one search result, the Search Results window displays all search results, as shown in the following illustration. If only one result is found, the result is immediately highlighted, bypassing the Search Results window.
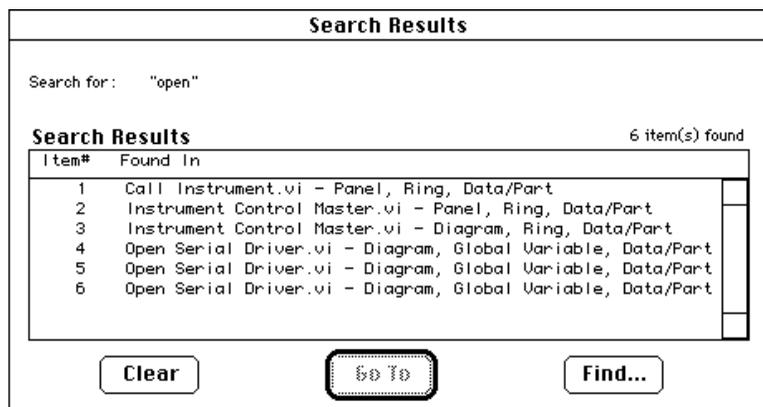
You can display this window also by selecting **Project»Search Results…**.

The following menu items are available in the Search Results window.

- **Clear**—Clears and frees memory used to store search results.
- **Go To**—Highlights the currently selected search result. You also can do this by double-clicking the item. Items already highlighted are checked.
- **Find…**—Invokes the Find dialog box.
- **Stop**—Appears in place of the **Find…** button during a search. If there is a long search through many VIs, the Search Results window updates search results as they are found and shows the status of the search. You use the **Stop** button to interrupt the search.

If you delete an object or VI and it is part of a search result, then the item entry of that result is disabled. Notice, however, modifying text does not update the search results. Therefore, you might highlight a text search result no longer matching the search criteria. In this case, perform another find to update the search results.

Finding Next and Previous Search Items

Select **Project»Find Next** to highlight the next and previous results in the search list. For hidden text, all objects needed to display the text are made visible temporarily as grayed-out objects and the text is selected; when you click the mouse or press a key, the objects are hidden again.

Find Pop-Up Menu for Global and Local Variables and Attribute Nodes

A Find pop-up menu item is available on front panel controls and indicators, global variables, locals variables, and attribute nodes. You can pop up on a local variable or attribute node to find its corresponding control, front panel terminal, or other local variables and attribute nodes. Conversely, front panel controls and indicators have pop-up menus to find all corresponding locals and attribute nodes. From a global node, you can find its corresponding global definition or all other global nodes. Conversely, global controls have a pop-up menu for finding all corresponding global references in memory. If more than one result is found, the Search Results window appears.

4

Executing and Debugging VIs and SubVIs

This chapter describes operating and debugging VIs, and explains the setup of VIs and subVIs for special execution modes.

Executing VIs

Running VIs

You can run a VI by selecting **Operate»Run** or by clicking the Run button. G compiles the VI if necessary.

Run button

While the VI is running at its top level (meaning it has no callers and therefore is not a subVI), the **Run** button changes to look like this.

VI running
at top level

If the VI is executing as a subVI, the **Run** button changes to look like this.

VI caller
is running

Press the **Run Continuously** button to run the VI again and again, until you abort or pause execution.

Run Continuously
button

Pressing the **Abort** button aborts execution of the *top-level* VI. If a VI is used by more than one running top-level VI, the button appears grayed out.

Abort button

Pressing this button pauses execution. You can press it again at any time to continue execution.

Pause button

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

If your VI runs but does not perform correctly, see the sections *Fixing Broken VIs* and *Debugging Executable VIs* later in this chapter.

When you edit a VI, the toolbar contains several tools used for editing VIs, including the Font ring, the Alignment ring, the Distribution ring, and the Reorder ring.

When you run a VI, those tools are replaced with debugging tools, as shown in the following illustration.

Run button

If you pop up on a control while running a VI, you see that all objects have a much simpler set of menu items. These are the same as the menu items in the **Data Operations** submenu that you see in pop-up menus when you edit VIs.
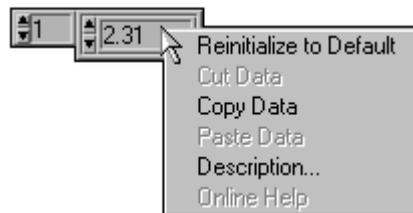
You can cut, copy, or paste the contents of the control, set the control to its default value, and read the description of the control with menu items in this menu. Some of the more complex controls have additional options. For example, an array has menu items to copy a range of values or go to the last element of the array.
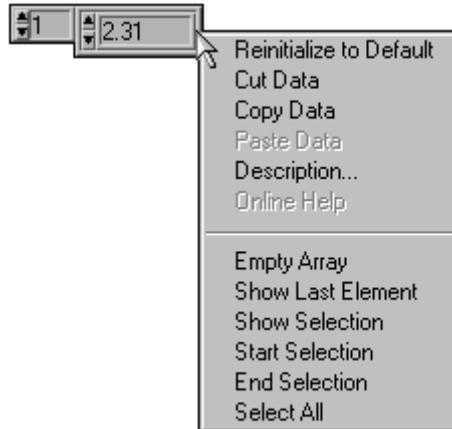
When editing a VI, the pop-up menu of an object contains an editing menu and a **Data Operations** submenu, as shown in the following illustration. This submenu contains the same menu items you see when running a VI plus an additional menu item, **Make Current Value Default**.
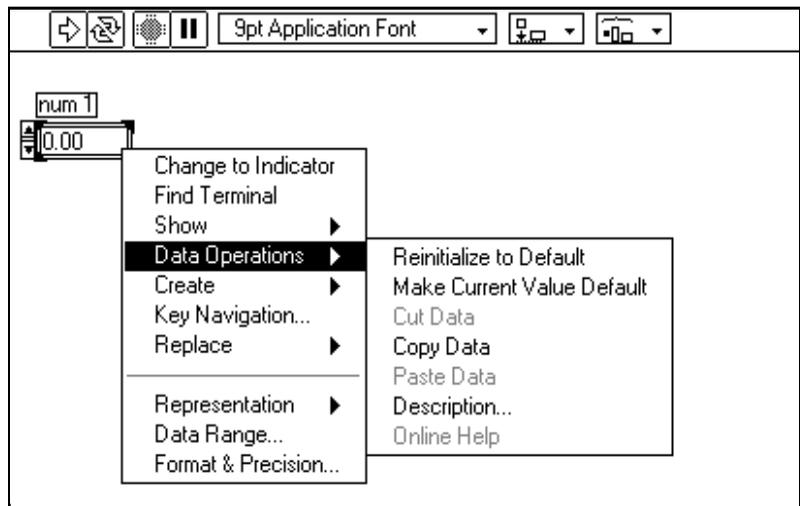
The **Operate** menu on the menu bar contains commands for executing the current VI. The following sections discuss several fundamental execution-related tasks.

Stopping VIs

Abort button

Under ordinary circumstances, you let a VI run until it completes. However, if you need to halt execution immediately, click the **Abort** button or select **Operate»Abort**. The **Abort** command halts the top-level VI at the earliest opportunity. The halted VI most likely did not complete its task, so you cannot rely on any data it produces. Although G closes files open at the time and halts any data acquisition that might be in progress, avoid designing VIs that rely on the **Abort** button or **Abort** menu item. For example, if you create a VI that executes indefinitely within a While Loop until stopped by the operator, wire the conditional terminal of the loop with a Boolean switch on the front panel.

If you want to prevent an operator from inadvertently aborting your VI by clicking the **Abort** button, hide it by deselecting the **VI Setup» Window Options»Show Abort Button** checkbox from the icon pane pop-up menu on the front panel of the VI. See the *Windows Options* section of Chapter 6, *Setting up VIs and SubVIs,* for more information.

Running VIs Repeatedly

Run Continuously button

To execute a VI repeatedly, click the **Run Continuously** button. The VI begins executing immediately, and the **Run Continuously** button changes from outlined arrows to filled arrows while the VI is running. Click the **Run Continuously** button again to stop the VI. The VI stops when it completes normally.

VI running continuously

The behavior of the VI and the state of the toolbar during continuous run is the same as during a single run started with the **Run** button or the **Operate»Run** command.

Data Logging on the Front Panel

Front panel data logging saves a time stamp and the data in all front panel controls of a VI to a datalog file. You can have several different files, each filled with logged data from different tests. You can retrieve this data using interactive retrieval through the VI, programmatic data retrieval, or standard file I/O functions.

Each VI maintains a log-file binding, showing the location of the datalog file in which front panel data is recorded. If the binding is clear when datalogging is enabled, meaning that the location is unspecified, the VI prompts you for the location of the file.

Configure and control front panel data logging by selecting the **Operate** menu and its **Data Logging** submenu, as shown in the following illustration.
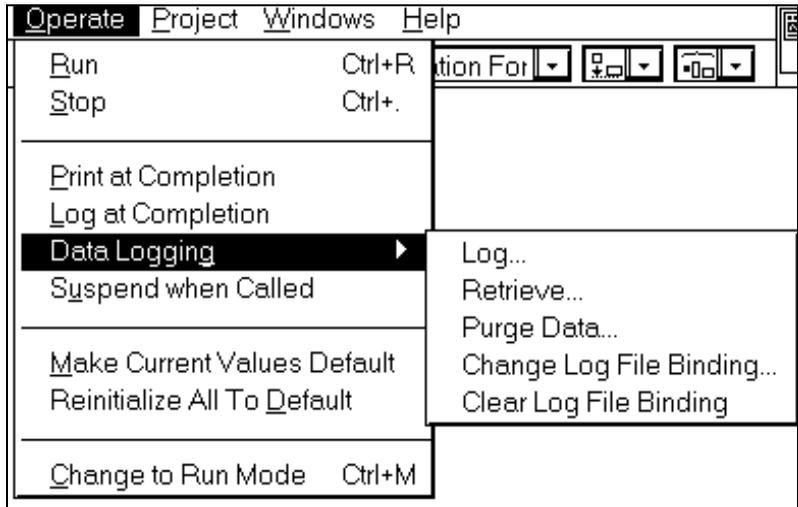
When automatic data logging is enabled for a VI, G logs the front panel of the VI any time the VI completes execution. You can tell if automatic data logging is enabled for a VI by looking at the **Operate»Log at Completion** menu item. Selecting **Operate»Log at Completion** enables automatic data logging for a VI if disabled and disables automatic data logging if enabled.

To interactively log front panel data for a VI, select **Operate»Data Logging»Log…**.

☞ **Note** *A waveform chart returns only one data point at a time with front panel data logging. If you wire an array into the chart indicator, the data log contains the array subset displayed on the chart for that record.*

You can change the log file binding of a VI with the **Operate»Data Logging»Change Log File Binding…** menu item.

You can select a specific record by typing the record number and then clicking the **Enter** button or pressing the `<Enter>` key on the numeric keypad.

Enter button

You can mark the selected record for deletion by clicking the **Delete** button. When the selected record is marked for deletion, the **Delete** button changes to a full trash can. Clicking the full trash can unmarks the selected record for deletion. Selected records are not deleted until you select **Operate»Data Logging»Purge Data…**, or until you switch out of data retrieval mode, either by clicking the **OK** button or selecting **Retrieve…** again. If any records still are marked for deletion when you switch out of data retrieval mode, you are asked if you want to delete the marked records.

Delete button

Record marked for deletion button

Click the **OK** button to return to the VI whose data log you were viewing.

Selecting **Operate»Clear Log File Binding** disassociates the VI from any associated log file. The next time you log from the front panel data of the VI, you are prompted to specify the log file.

To view logged data interactively, select the **Operate»Retrieve…** menu item. The toolbar becomes the data retrieval toolbar, as shown in the following illustration.

| OK | 🗑 | ↕ | 0 | [0..5] | 8/21/95 17:11:52.000 |
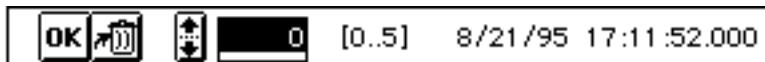
The highlighted number indicates which record currently is selected for viewing. The numbers in square brackets indicate the range of records logged. The date and time indicate when the selected record was logged. You can view the next or previous record by clicking the up arrow or down arrow, respectively. You also can use the up or down arrows on your keyboard.

Retrieving Data Programmatically

You can retrieve data logged from a VI by using the **Enable Database Access** menu item or using standard file I/O functions to read the file as a datalog file.

Accessing Databases

You can retrieve data by using the **Enable Database Access** menu item. Access this from the pop-up menu of a subVI that has logged front panel data in the associated datalog file of the VI, as shown in the following illustration. A *halo* that looks like a file cabinet appears around the datalog

file. This halo has terminals for accessing data from the datalog file. If you run the calling diagram, the subVI does not execute. Instead, you retrieve data from a specified record of the datalog file. It also returns the time the data was logged, and a Boolean value indicating whether the indicated record number is invalid.
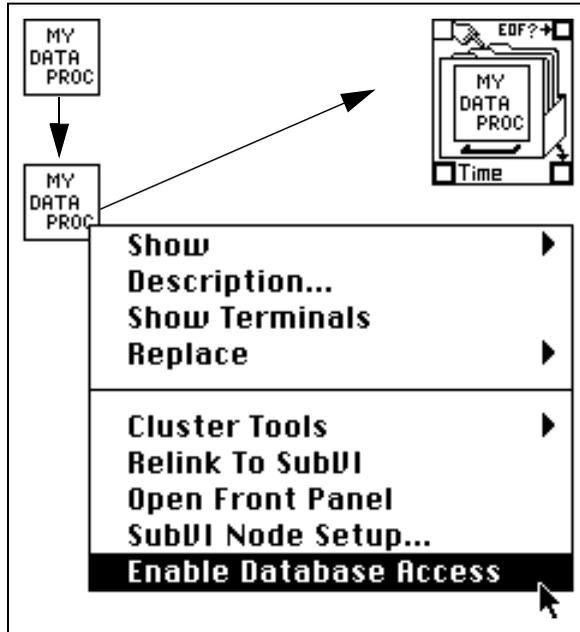
The following illustration shows the halo terminals for accessing the logged data.
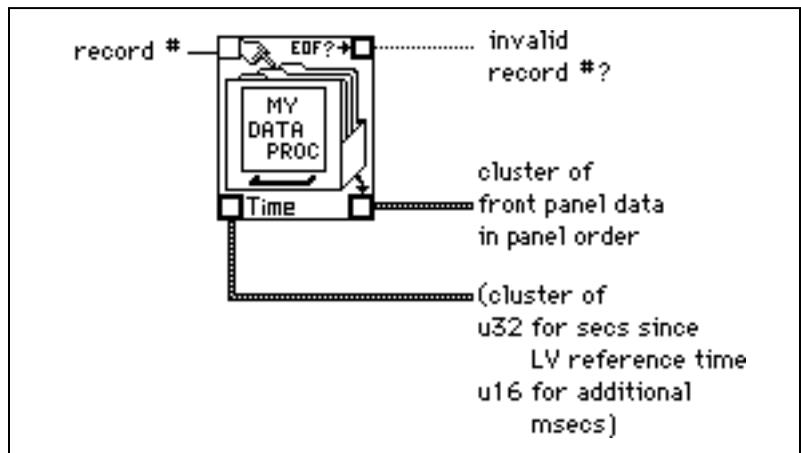
If the subVI has *n* logged records, you can wire any number from –*n* to *n* – 1 to the record # terminal. You can access records relative to the *first* logged record using non-negative record numbers. 0 means first record, 1 means second record, and so on, through *n* – 1, which stands for the last record.

You can access records relative to the *last* logged record using negative record numbers. Keep in mind –1 means last record, –2 means second to the last, and so on, through –n, which means the first record. If you wire a number outside the range –*n* to *n* – *1* to the record # terminal, the invalid record #? output is set to TRUE, and no data is retrieved.

Retrieving Data Using File I/O Functions

You also can retrieve data logged from the front panel of a VI using file I/O functions belonging to G. Each record in the datalog file created by front panel data logging is a cluster containing a time stamp and the front panel data. The time-stamp cluster consists of an unsigned 32-bit integer, representing seconds, and an unsigned 16-bit integer, representing milliseconds elapsed since the G reference time. This cluster is followed by a cluster of the front panel data in panel order.

You can access the records of datalog files created by front panel data logging using the same G file I/O functions you use to access programmatically created datalog files. Enter the appropriate type, described previously, as the type input to the File Open function, as shown in the following example.
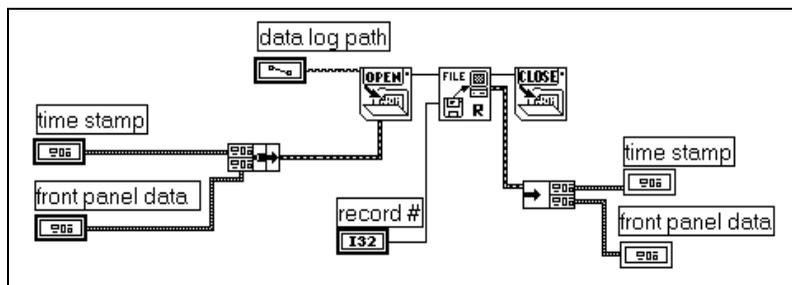
Debugging VIs

This section covers a range of solutions to errors and problems in virtual instruments.

Fixing Broken VIs

A VI cannot compile or run if it is broken. The **VI Run** button typically appears broken while you are creating or editing it, until you wire all the icons in the diagram. If it still is broken after you finish wiring the icons, try selecting **Edit»Remove Bad Wires** or selecting <Ctrl-b> **(Windows)**; <command-b> **(Macintosh)**; <meta-b> **(Sun)**; or <Alt-b> **(HP-UX)**. Often, this fixes a broken VI.

Broken Run
button

To find out why a VI is broken, click the **Broken Run** button. An Error List window appears listing all the errors. This window is shown in the following illustration.
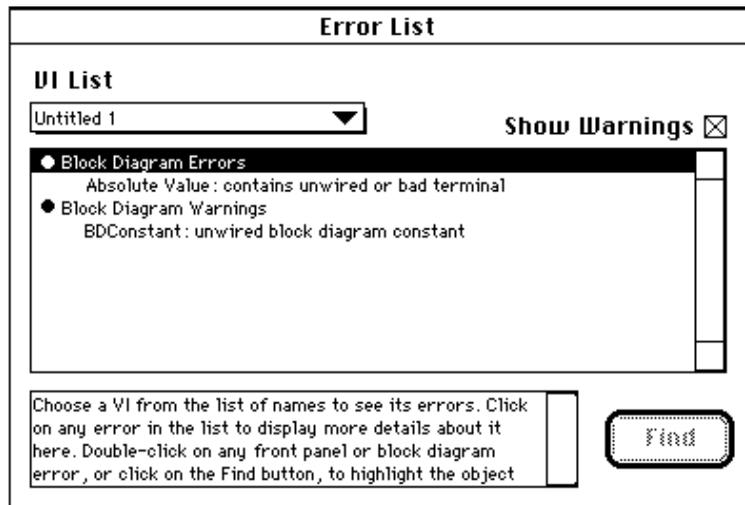
You also can access this box by clicking the **Warning** button of a VI, or by selecting **Windows»Show Error List**. The **Warning** button for a VI is only visible if the VI has a warning (such as overlapping objects) and you check **Show Warnings** in the Error List window. Use the **Preferences** dialog box to configure G to show warnings by default.

Warning
button

To locate an error, double-click the text describing it. The error is displayed by bringing the relevant window to the front and highlighting the object

causing the error. View errors and warnings for other VIs by selecting the name of the VI from the **VI List** pop-up menu.

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 because of a mismatch of data types or a loose, unconnected end. Remove broken wires, including wire stubs you cannot see, with the **Edit»Remove Bad Wires**.

- A subVI is broken, or you edited its connector after you placed its icon in the diagram.

- You might have a problem with an object made invisible, disabled, or otherwise altered through an attribute node. Restore the object using the attribute node to fix the problem, if possible.

Interpreting Error Messages

The following list contains some possible errors in a form similar to the way they look in the Error List window.

Table 4-1. Error Messages

Error Messages	Descriptions
Function Name: contains unwired or bad terminal.	A required input is not wired or the type is inappropriate. Wire the required inputs with the proper data types.
Code Interface Node: object code not loaded.	You did not link the object code of a CIN properly. Pop up on the node and reload the code.
Control: control does not match its type definition.	Edit the control to match, or pop up and update or disconnect the type definition VI from the control.
Enumeration has duplicate entries.	All the items in an enumeration control must be unique and distinct.
More Errors....	The error window shows only 100 errors at one time. Unlisted errors appear after you fix the listed errors.

Table 4-1. Error Messages (Continued)

Error Messages	Descriptions
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 changed since you last loaded the VI and the name in the global or local variable no longer matches. Pop up on the variable and select another name.
Global Variable: subVI is missing.	When the calling VI loaded, the global subVI was not found, perhaps because you changed the name. Replace the bad global subVI with a good one.
Node: A subroutine priority VI cannot contain an asynchronous node.	You cannot use an asynchronous function, such as Wait, in a VI with a priority level equal to subroutine.
Right Shift Register: type is undefined.	You must remove unused shift registers.
Right Shift Register: some but not all left sides are wired.	A shift register must have inputs for all the left sides or for none.
Sequence: One or more sequence locals were never assigned.	You forgot to wire to a Sequence Structure local variable. Remove unused sequence locals.
SubVI name: bad linkage to subVI.	The connector pattern of the subVI changed since you last loaded the subVI, or you forgot to assign controls or indicators of the subVI to connector terminals. In the former case, use the **Relink** command in the subVI pop-up menu.
SubVI name: recursive references (dispose it).	You changed the name of the calling VI to be the same as one of its subVIs, and now G registers this change as the VI calling itself recursively. (The VI and subVI are probably in different directories.) G 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.

Table 4-1. Error Messages (Continued)

Error Messages	Descriptions
SubVI name: subVI is already running.	You cannot run a VI if one of its subVIs already is 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 ar 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.	When the calling VI loaded, the subVI was not found, perhaps because you changed the name. Replace the bad subVI with a good one, or open the missing subVI if you 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.	When the calling VI loaded, the type-definition VI was not found, perhaps because you changed the name. Open the missing type definition VI if you find it, pop up on the bad control, and disconnect it 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 or Unbundle By Name function is either empty or has some components 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.

Debugging Executable VIs

If your program executes but does not produce expected results, you often can solve the problem by taking the following steps.

- Eliminate all VI warnings. If you select **Show Warnings** in the Error List window, the listbox identifies the warnings for your VI. Determine the causes and eliminate them from your VI.

- Check wire paths to ensure that the wires connect to the proper terminals. Triple-clicking the wire with the Operating tool highlights the entire path. It is possible a wire appearing to emanate from one terminal emanates from another. Look closely to see where the end of the wire connects to the node.

- Use the Help window (**Help»Show Help**) to make sure functions are wired correctly.

- Verify the default value for unwired inputs of functions or subVIs are what you expect.

- Use breakpoints, execution highlighting, and single-stepping to determine if the VI is executing as you planned.

- Use the probe feature described in the *Using the Probe Tool* 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, enter the values NaN and ±Inf in addition to normal values.

- Check to make certain execution highlighting is turned off in subVIs, if your VI runs slower than expected. Also, close subVI windows when you are not using them.

- Check the representation of your controls and indicators to see if you are receiving overflow because you converted a floating-point number to an integer or an integer to a smaller integer. Also refer 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.

- Determine if any For Loops inadvertently might execute zero iterations and produce empty arrays. Also refer to the *Execution Highlighting* and *Recognizing Undefined Data* sections of this chapter.

- Verify 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 G detects data type and cluster size mismatches at

edit time, G does not detect mismatches of elements of the same type. Use the **Cluster Order…** menu item on the cluster shell pop-up menu to check cluster order.

- Check the node execution order. Nodes not connected by a wire can execute in any order. The spatial arrangement of these nodes does not control the order. That is, unconnected nodes do *not* execute from left to right, top to bottom on the diagram as statements do in textual languages.

- Check for any extraneous VIs. Unlike functions, unwired subVIs do not always generate errors (unless you configure an input as required or recommended). If you mistakenly place an unwired subVI on the block diagram, it executes when the diagram does, and consequently your program might end up doing extra actions.

- Check that you do not have hidden VIs. You inadvertently might have hidden subVIs three ways—by dropping one directly on top of another node; by decreasing the size of a structure without keeping the subVI in view; or by placing one off the main diagram area. In the last case, scroll the block diagram window to its limits. Also check the inventory of subVIs used in the VI against the **Project** menu items **This VI's SubVIs** and **Unopened SubVIs** to determine if any extraneous subVIs exist. Also look in the Hierarchy window to see the subVIs for a VI, or in the Error List window to see a list of hidden objects (as long as the **Show Warnings** checkbox is selected). To help avoid incorrect results caused by hidden VIs, you can indicate that inputs to VIs are required; for information, see *Required, Recommended, and Optional Connections for SubVIs* in Chapter 3, *Using SubVIs.*

Understanding Warnings

If an object is hidden completely by another object, G generates a warning to indicate all objects are not visible. For example, if a terminal is hidden under the edge of a structure, or tunnels are on top of each other, a warning message is placed in the Error List window. Warnings do not prevent you from running a VI; they are intended to help you debug potential problems in your programs.

Recognizing Undefined Data

There are two symbolic values that can appear in floating-point digital displays to indicate faulty computations or meaningless results. *NaN* (not a number) is the symbol representing a particular floating-point value that operations such as taking the square root of a negative number can produce. *Inf* is another special floating-point value produced, for example, by dividing one by zero.

Undefined data can corrupt all subsequent operations. Floating-point operations propagate NaN and ± Inf, which, when explicitly or implicitly converted to integers or Booleans, become meaningless values. For example, dividing one by zero produces Inf, but converting that value to a word integer produces the value 32,767, which appears to be a normal value. Before converting to integer types, check intermediate floating-point values for validity unless you are sure this type of error does not occur in your VI.

Executing a For Loop zero times can produce unexpected values. For example, the output of an initialized shift register is the initial value. However, if you do not initialize the shift register, the output is either the default value for the data type—0, FALSE, or empty string—or the output is the last value loaded into the shift register when the diagram last executed.

Indexing beyond the bounds of an array produces the default value for the array element data type. You can do this inadvertently 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 can produce undefined output values for certain input values, do 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 using an incoming array to auto-index a For Loop, evaluate the operation of the VI if the input array is empty. 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 still can generate a range error if you send NaN to it.*

Correcting VI Range Errors

Range Error
indicator

A range error indicator appears in place of the **Run** button in the following circumstances.

- You configured a control on a subVI to stop execution when it receives an out-of-range value (through the **Data Range…** menu item of the control), and the control receives such a value.

- You configured an indicator on a subVI to stop execution when it tried to return an out-of-range value to a calling VI, and the indicator attempted to return such a value.

- An operator enters an out-of-range value into a control you set to stop execution on error, as long as the VI is not running at the time.

When a control or indicator is out of range it is surrounded with a thick, red, rectangle, as shown on the right side in the following illustration.
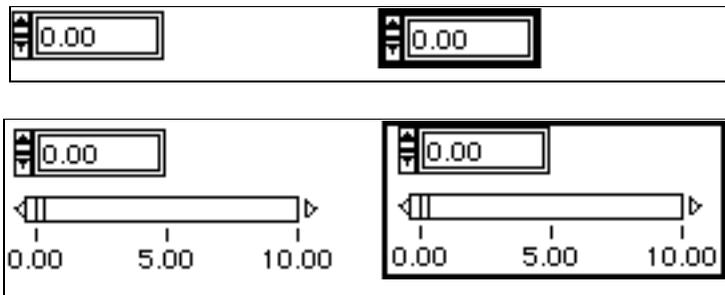
See *Range Options of Numeric Controls and Indicators* in Chapter 9, *Numeric Controls and Indicators*, for more information.

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 starts.*

Debugging Features

This section covers the following debugging features.

- Single-stepping through a VI to observe each execution step

- Highlighting execution to watch the data flow as it occurs

- Using a probe to display data

- Setting breakpoints to pause execution, so that you can single-step, probe wires to see their data, or edit indicators

- Suspending execution to edit indicators, to control the number of times to execute, or to go back to the beginning of the VI

Single-Stepping through VIs

Pause button

For debugging purposes, you might want to execute a block diagram node by node. This form of execution is called single-stepping. Before executing a VI, start single-stepping by pressing the **Pause** button. Execution of a VI can also be paused at any time by clicking the **Pause** button. You can return to normal execution at any time by releasing the **Pause** button. Clicking the **Run** button when a VI is paused starts execution again, but the VI pause is still set. When the VI is called again, execution pauses.

Step Over button

Step Into button

Step Out button

While the VI is running in single-step mode, press any of the active three step buttons to proceed to the next step. The step button you press determines where the next step executes.

If you idle your cursor over any of the step buttons, a Tip strip appears with a description of the next step if you press that button.

You might want to use execution highlighting as you single-step through a VI, to follow data as it flows through the nodes. See the *Execution Highlighting* section in this chapter.

An Example of Single-Stepping through a VI

The following illustration shows an example of a VI that is single-stepping
(Caller 1.vi). This VI currently is running on behalf of the
VI Caller 3.vi. The subVI, Fibonacci.vi, currently is executing
(indicated by the arrow glyph on the **Run** button). The VI, Factorial.vi,
is the next node to be executed. Clicking the **Step Into** button opens its
block diagram and starts single-stepping. Clicking the **Step Over** button
executes the subVI and pauses. Clicking the **Step Out** button finishes
executing the current frame of the loop and pauses.
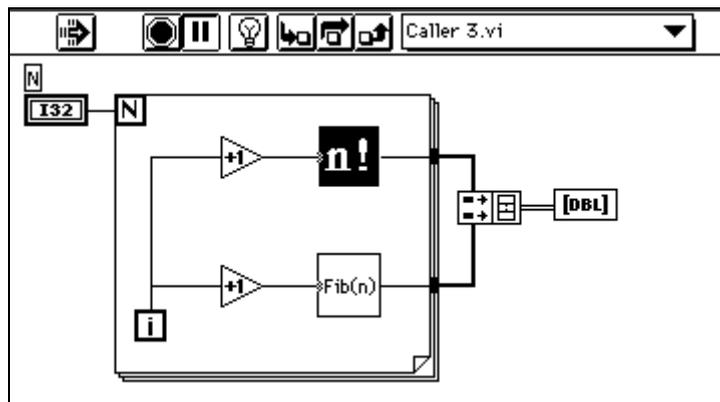
After stepping into the subVI Fibonacci.vi and stepping into the loop,
clicking the **Step Out** button and holding down the mouse for a second, a
menu appears. Select how far the VI executes before pausing. Selecting
Block Diagram runs the VI until all nodes on the block diagram execute,
and execution pauses.

Selecting one of the VIs on the call stack runs that VI until it is finished. At
that point, the caller of the selected VI pauses.

Using Step Buttons

The step buttons are as follows.

Press the **Step Over** button to execute a structure (sequence, loop, etc.) or a subVI and then pause at the next node. The keyboard shortcut is <Ctrl> **(Windows)**; <command> **(Macintosh)**; <meta> **(Sun)**; or <Alt> **(HP-UX)** followed by the right arrow key.

Step Into
button

Press the **Step Into** button to execute the first step of a subVI or structure (sequence, loop, etc.) and then pause at the next step of the subVI or structure. The keyboard shortcut is <Ctrl> **(Windows)**; <command> **(Macintosh)**; <meta> **(Sun)**; or <Alt> **(HP-UX)** followed by the down arrow key. The step buttons affect execution only in a VI or subVI in single-step mode. If a VI in single-step mode has one subVI also in single-step mode and one in normal execution mode, the first subVI single-steps when called but the second executes normally when called. Press the **Step Out** button to finish executing the current block diagram, structure, or VI and pause. The keyboard shortcut is <Ctrl> **(Windows)**; <command> **(Macintosh)**; <meta> **(Sun)**; or <Alt> **(HP-UX)** followed by the up arrow key.

When the VI finishes executing, the step buttons become grayed-out.

Reading Call Chains

When a subVI is paused, a **Call Chain** ring appears. This menu lists the chain of callers from the top-level VI down to this subVI. Keep in mind this is not the same as the **Project»This VI's Caller's** menu item, which lists all calling VIs regardless of whether they currently are executing. When you select a VI from the Call Chain ring, its block diagram opens and the VI calling the current subVI is selected. This helps you distinguish the current instance of the subVI if the block diagram contains more than one instance.

```
┌─────────────────────────────────┐
│ Caller 2.vi                  ▼  │
└─────────────────────────────────┘
          Call Chain ring
```
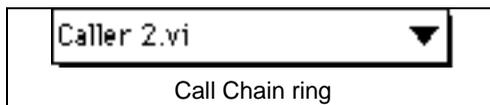
In addition to the keyboard shortcuts given in the *Using Step Buttons* section of this chapter, the following commands are available.

- When single-stepping through a subVI or structure, clicking the **Step Out** button while holding the mouse down brings up a menu. Select how far the VI executes before pausing.

- Double-clicking front panel controls and indicators, local variables, and global variables displays and highlights the corresponding block diagram object.

- Pressing the <Ctrl> **(Windows)**; <option> **(Macintosh)**; <meta> **(Sun)**; or <Alt> **(HP-UX)** key while double-clicking a subVI brings its block diagram to the front.

Execution Highlighting

For debugging purposes, it might be helpful to view an animation of the execution of the VI block diagram.

off   on

Highlight Execution
buttons

To use this feature, click the **Highlight Execution** button. The button changes its appearance. Click 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.

With execution highlighting, the movement of data from one node to another is marked by bubbles moving along the wires. Additionally, in single-stepping, the next node blinks rapidly as shown in the following illustration sequence.
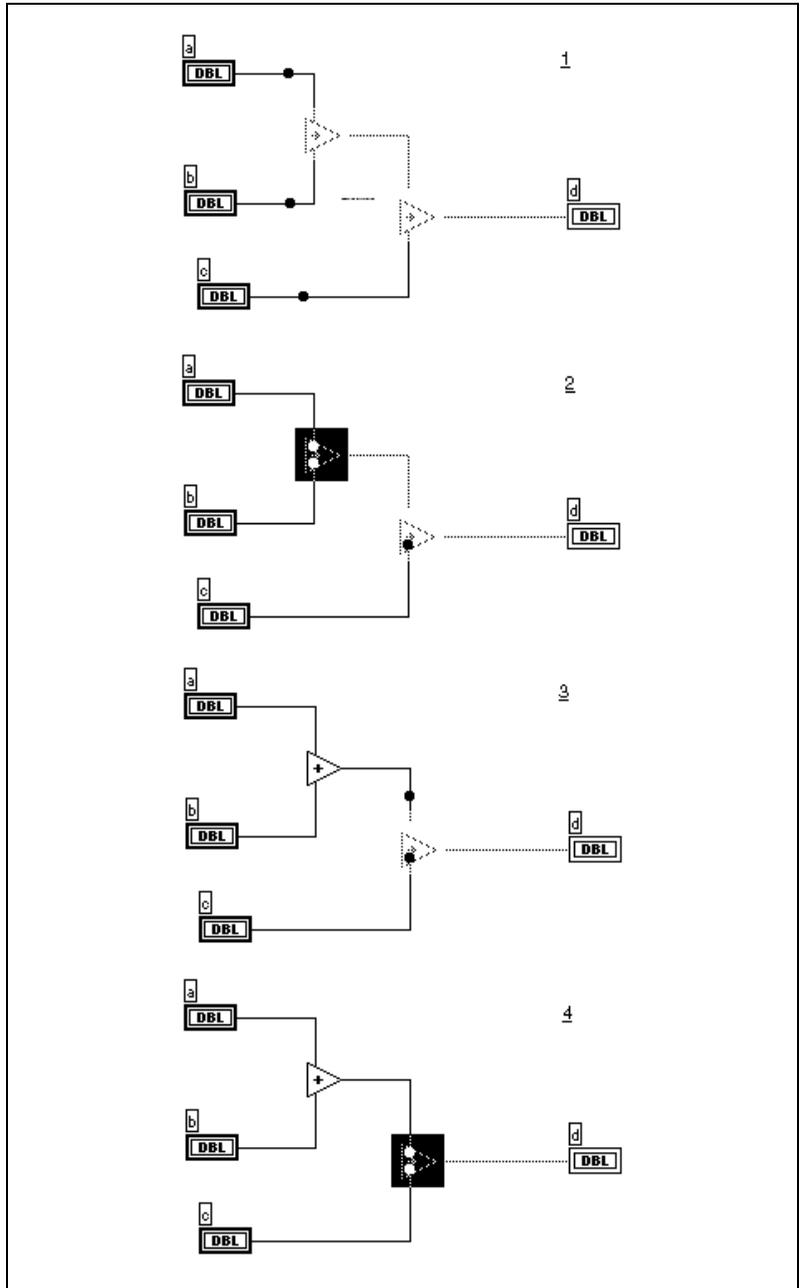
When you single-step through a subVI with the **Highlight Execution** button on, an execution glyph on the subVI icon of the block diagram indicates which VIs are running and which are waiting to run. The arrow in the following illustration indicates a running subVI is waiting to run.

running

Block Diagram
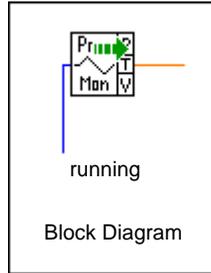
The Hierarchy window displays whether a VI is paused and/or suspended. (For more information about the Hierarchy Window, see the *Using the Hierarchy Window* section of Chapter 3, *Using SubVIs*.) The pause glyph indicates a subVI is paused and/or suspended. A green pause glyph (or a hollow glyph if displayed in black and white) indicates this subVI pauses when called. A red-pause glyph (or a solid glyph if displayed in black-and-white) indicates this subVI currently is paused.

paused

Hierarchy Window
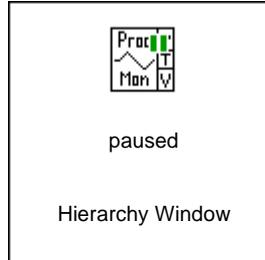
Using the Probe Tool
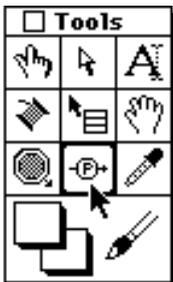
Probe tool selected

You can use the Probe tool to check intermediate values in a VI that executes but produces questionable or unexpected results. For example, you might have a complicated block diagram with a series of operations, any one of which might be returning incorrect data.

One way to look for the source of the questionable results is to wire an indicator to the output wire from one of the operations to display the intermediate results. But placing an indicator on the front panel and wiring its terminal to the block diagram is not a convenient debugging mechanism. It is time consuming and creates unwanted items on your front panel and block diagram you must later delete.

Select the Probe tool and place its cursor on a wire, or pop up on the wire
and select **Probe**. As shown in the following illustration, the floating Probe
window appears, with no data displayed within. As soon as you run the VI,
the Probe window displays data passed along the wire.
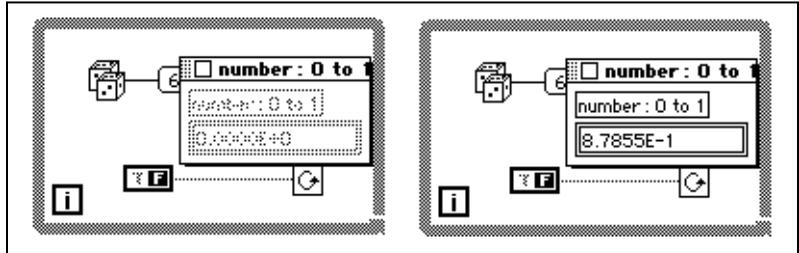
Each probe is numbered automatically and uniquely. The wire to which it
is attached is marked with the same number. This helps keep track of which
probe is associated with which wire. However, if the name of the front
panel control is as long as or longer than the Probe window, the number is
not visible.

You can use the probe in conjunction with execution highlighting,
single-stepping, or breakpoints to view values more easily. During
single-stepping or pausing at a breakpoint, data is updated immediately
if available. Examine the inputs when execution halts at a node.

You can insert the probe before running your VI to see the data. In
single-stepping, create a probe for an executed wire. The probe updates,
showing the wire contents. This is useful when you have set a breakpoint
on a node and you want to examine its inputs.

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

If you pop up on the indicator of the Probe window, find the associated wire, by selecting **Find Wire**, as shown in the following illustration.
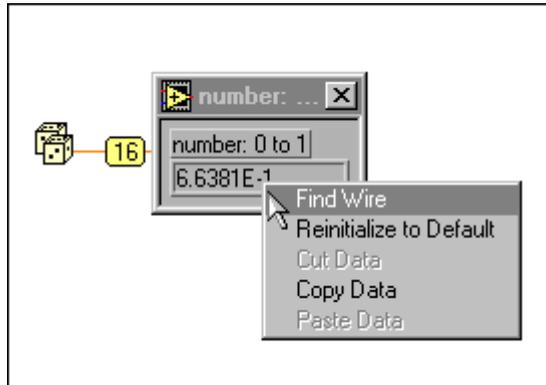
If you select the **Find Wire** menu item, the block diagram containing that wire comes to the front of all VIs and the wire is highlighted.

Use **Reinitialize to Default** to reset the value displayed in the Probe window to its default value.

Use the **Copy Data** menu item to copy the data to other numeric controls in the same VI or in other VIs.

Creating Probes

When you create a probe, you create a default-style probe to match the data type of the wire. With numeric data types, for example, a digital indicator probes the data of a wire.

If you prefer, select a control for the probe from the built-in controls, or from controls saved as custom controls or type definitions. To do so, pop up on the wire, select **Custom Probe**, and select a control from the **Custom Probe** palette appearing to the right. If you choose **Select a Control…**, use the file dialog box to select any custom control or type definition saved in the file system. Type definitions are treated as standard custom controls when used to probe data.
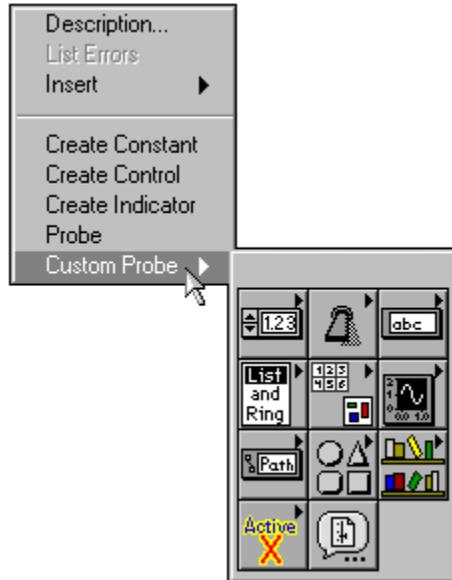
Only those parts of the **Custom Probe** palette that might match the data type of the wire are enabled. If you choose a control that cannot have the same data type as the wire, this mismatch is signaled with a beep and the probe is not created. For example, with arrays and clusters, you cannot use the array and cluster shells from the **Array & Cluster** palette, because they are not completed data types.

Add your own palettes to the end of the **Custom Probe** palette, just as you add them to the **Controls** palette. See Chapter 7, *Customizing Your Environment*, for more information on how to do this.

Placing Breakpoint Tools

You can place a breakpoint on a VI, node, or wire to set a pause to occur during execution. Nodes include subVIs, functions, structures, Code Interface Nodes (CINs), Formula Nodes, and attribute nodes. When a block diagram has a breakpoint, execution pauses after all nodes on the diagram execute. When a wire has a breakpoint, execution pauses after data passes through the wire.

When you reach a breakpoint during execution, single-step through execution using the step buttons, probe wires to see their data, change values of front panel controls, or continue running to the next breakpoint or until execution is completed. Clicking the **Run** button is an easy way to continue execution to the next breakpoint or until the subVI is called again.

To set a breakpoint, select the Breakpoint tool in the **Tools** palette and then click the block diagram, node, or wire. Click this tool on the same object at any time to remove the breakpoint. The appearance of the tool indicates if a breakpoint is set or cleared, as shown at the left.

The following table indicates how breakpoints display and when execution pauses, in relation to where breakpoints are placed.

Table 4-2. Breakpoint Placement

Location of Breakpoint	How Breakpoint Is Highlighted	When Pause Occurs
Block diagram	Red border around block diagram. If the diagram is inside a structure, the red border is inside the structure as well.	When all nodes within the block diagram have finished execution. In the case of loops, this pauses after each loop iteration.
Node	Red border framing the node.	Just before the node executes. At this point all input signals into the node can be probed with the Probe tool.
Wire	Red bullet in the middle of wire. If a probe is attached to the wire, the probe has a red border.	After data passes through the wire.

In the following example, the wire with the attached probe has a
breakpoint. The VI pauses after the probe displays data.
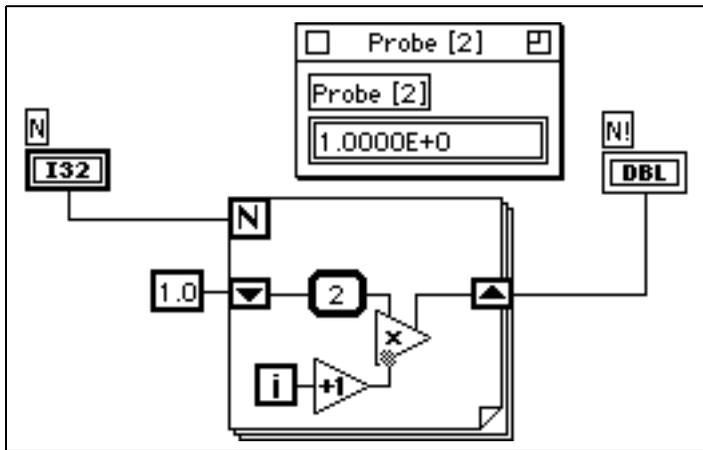
In the next example, the Increment node has a breakpoint. The VI pauses
before Increment executes. One wire also has a breakpoint. The VI pauses
again after Multiply executes.
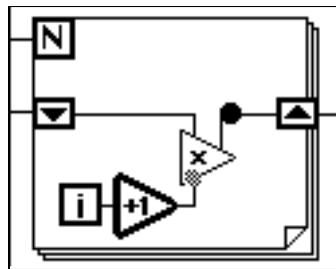
In the following example, the For Loop has a breakpoint. The VI pauses
after Multiply executes.
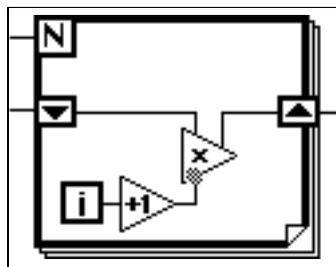
When a VI pauses because of a breakpoint, its block diagram is brought to the front (if the front panel was closed, it is opened as well), and the node or wire causing the break is highlighted with a marquee.

Breakpoints save with a VI, but only become active during execution.

Suspending Execution

You can edit controls and indicators, execute the subVI as many times as you want before returning to the caller, or go back to the beginning of the subVI by suspending execution of a SubVI.

To set a subVI in suspend mode, open the subVI and check the **Operate»Suspend when Called** menu item. This also is accessed by popping up on the connector pane on the front panel while in run mode, and selecting **Setup»Execution Options»Suspend when Called…**. The subVI automatically suspends when it is called. If you check this menu item when single-stepping, the VI does not switch into suspend mode immediately.

If you want to suspend execution at a particular call to a subVI, use the **SubVI Node Setup…** menu item from the pop-up menu of the subVI node, instead of **Suspend when Called**. The **SubVI Node Setup…** suspends execution at that particular instance of the subVI only.

Recognizing Automatic Suspension

A VI suspends automatically if, during its execution, a control or an indicator goes out of range. The terminal or local variable causing the out-of-range error is selected.

A subVI with a control or indicator set to stop on a range error has a conditional breakpoint. If a range error occurs, the subVI pauses as if it encounters a breakpoint. If no range error occurs, the subVI executes normally.

When data passed to a subVI causes a range error, the front panel of the subVI opens or comes to the foreground and the subVI remains in suspend mode. At this time, the values on the subVI controls are the inputs passed by the calling VI, and you can change the values if you want. In fact, you must change them to run the subVI if the range error indicator is on. The indicators of the subVI display either default values or values from the last execution of the subVI during which its front panel was open.

Using Toolbar Buttons When SubVIs Are Suspended

Run button

If you want to execute the current subVI before returning to the caller, press the **Run** button (or select the **Operate»Run** command) while in suspend mode. Repeat the execution as many times as you want.

When the subVI completes, the indicators display results from that execution of the subVI. However, you can change the indicator values if you want to return different values to the calling VI. In fact, the suspend mode is the only time you can set values on indicators. Click the **Return to Caller** button again when you are ready to return the indicator values of the subVI to the calling VI.

Skip to Beginning

If you want to go back to the beginning of the VI, click the **Skip to Beginning** button.

Return to Caller

The **Return to Caller** button appears when a suspended subVI is not executing. Click it to return to the caller VI. Notice you can return to a caller without executing the current VI. If you want to execute it, be sure to press the **Run** button before returning to the caller.

Viewing Hierarchy Windows During Suspension

The Hierarchy window displays an exclamation point glyph to indicate a suspended subVI. The following illustration shows a subVI in the Hierarchy window with **Suspend when Called** on.

Disabling Debugging Features

To reduce memory requirements and to increase performance slightly, you can compile a VI without various debugging features. To do this, go to **VI Setup»Window Options** from the connector pane pop-up menu on the front panel and deselect the checkbox for **Allow Debugging**.

Commenting out Sections of Diagrams

In some cases, you might want to execute a VI with a section of the block diagram disabled. The easiest way to do this is enclose the section of diagram you want to disable in a Case Structure with a constant Boolean wired to the selector. Put the diagram you want to disable in the frame of the Case Structure that does not execute.

If the Case Structure produces any data values, create constants for the output tunnels.

5

Printing and Documenting VIs

This chapter describes a variety of issues involving printing and VI documentation.

Printing

There are four primary ways to print VIs in G.

• Use the **Print Window…** menu item to make a quick printout of the contents of the current window.

• Make a more comprehensive printout of a VI, including information about the front panel, block diagram, subVIs, controls, VI history, and so on, by selecting the **Print Documentation…** menu item.

• Use the programmatic printing feature to make VIs automatically print their front panels under the control of your application.

• Use the VI Server to programmatically print a VI window or VI documentation.

In addition, you can use the Serial Port VIs to send text to the printer if your printer is connected to a serial or parallel port; doing so generally requires some knowledge of the command language of the printer.

In Windows and UNIX, another option for printing text is to use the System Exec VI located in **Functions»Communication**. On the Macintosh, use the AESend Print Document VI located in **Functions»Communication» AppleEvent**. In Windows, use ActiveX automation to tell another application to print data.

Printing Configuration

There are two dialog boxes that affect the appearance of all printouts, regardless of the method you use to print: the Preferences dialog box and the Setup dialog box.

You use the Printing dialog page of the Preferences dialog box to configure printing options.

To configure printer-specific information and formatting, use the **Printer Setup…** (**Page Setup** on the Macintosh) menu item from the **File** menu. For example, with most printers, you change the orientation of printouts (landscape versus portrait) using this dialog box. Many printers also have options for font substitution, paper size, and other printer specific settings. **Printer Setup…** (**Page Setup** for the Macintosh) settings are saved with your VI.

PostScript Printing

If you have access to a PostScript printer, you might wish to select the **Edit»Preferences»Printing»PostScript printing** item rather than using other output options.

A PostScript printer offers the following advantages.

- PostScript printouts reproduce the image of the screen more accurately.
- PostScript facilitates high resolution graphs.
- PostScript reproduces patterns, line styles and fonts more accurately.

You use the Preferences dialog box to indicate if you want to use PostScript printing. See Chapter 7, *Customizing Your Environment*, for more information on the Preferences dialog box.

Printing the Active Window

Use the **Print Window…** menu item to print out the contents of the currently active window (front panel or block diagram). Using this menu item, you can make a quick printout with the minimum number of prompts. For more comprehensive printouts of VIs, use **Print Documentation…**.

Print Window… formats the printout to look the same as it does when you set the VI to print programmatically. The **Execution Options** section of the VI Setup dialog box contains a few options which give more control of the way your VI looks when you print programmatically or using **Print Window….** See the *Setting Page Layout* section later in this chapter for information on these options.

Documenting VIs

This section describes printing your VI documentation, on paper or to RTF and HTML files, and related information on formatting graphics files.

Printing Documentation

If you want a detailed printout of the contents of a VI, use **Print Documentation…**, shown in the following illustration. Use the dialog box to select from several VI formats. You also can create your own custom format.
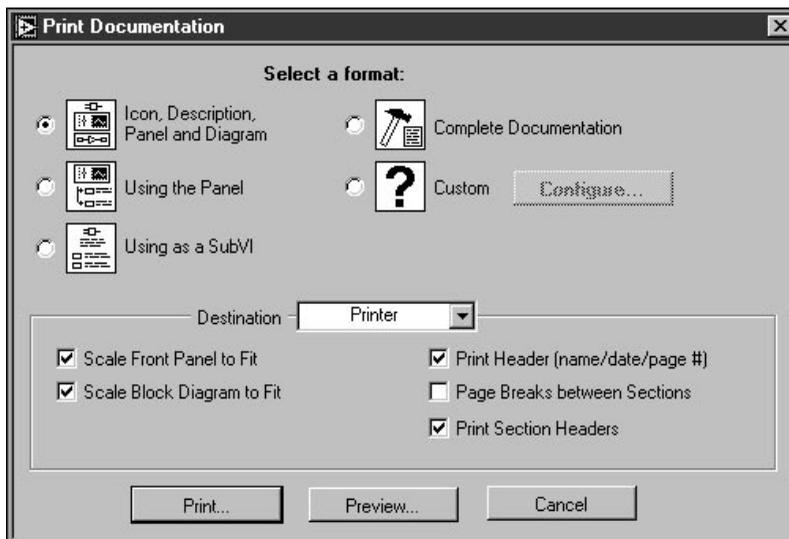
Setting Printout Formats

The Print Documentation dialog box has five different print formats, as well as a custom option to create your own format. The icon for each of these formats is shown to the left of the description.

The **Icon, Description, Panel, and Diagram** format (the default) prints the icon, VI description, front panel and block diagram.

The **Using the Panel** format prints the front panel, VI description, and the control names and their descriptions.

The **Using as a SubVI** format prints the icon, connector, VI description, and the terminals (data types), names, and descriptions of the connected controls. This format is similar to the G function reference format.

The **Complete Documentation** format prints everything, including the icon, connector, description, front panel, information about all front panel controls, the block diagram, and a list of the names of the subVIs.

The **Custom** format prints using the current custom settings. Change the custom settings by selecting **Configure…** after you select the custom menu item. See the *Creating Custom Print Settings* section in this chapter for information on this dialog box.

The **Destination** selection box gives you a choice of sending to Printer, HTML File, RTF File, or Plain Text File. See the *Printing/Exporting Control and VI Descriptions to an RTF or HTML File* section of this chapter for more information on **Destination** features.

Setting Other Print Options

The Print Documentation dialog box contains several page-layout options to control scaling, page breaks, and headers for the printout. These are described as follows.

Scale Front Panel to Fit and **Scale Block Diagram to Fit**—When set through this menu item, the front panel and/or block diagram can be scaled down to one-fourth the original size to better fit on the pages.

Print Header—Prints a header at the top of every page. This header includes the page number, the VI name, and the last modification date of the VI.

Page Breaks Between Sections—Inserts a page break between the following sections.

•   Connector icon and description

•   Front panel

•   List of front panel control details

•   Block diagram

•   Block diagram details

•   VI hierarchy

•   List of subVIs

Print Section Headers—Prints a header for each section (for example, a heading such as List of SubVIs before the subVI information). When printing using **Print Documentation**, this selection is set automatically for each format.

Creating Custom Print Settings

To bring up the Custom Print Settings dialog box, select the **Custom** button in the Print Documentation dialog box. After you select **Custom**, the **Configure** button is activated. Click this button to see the dialog box shown in the following illustration.
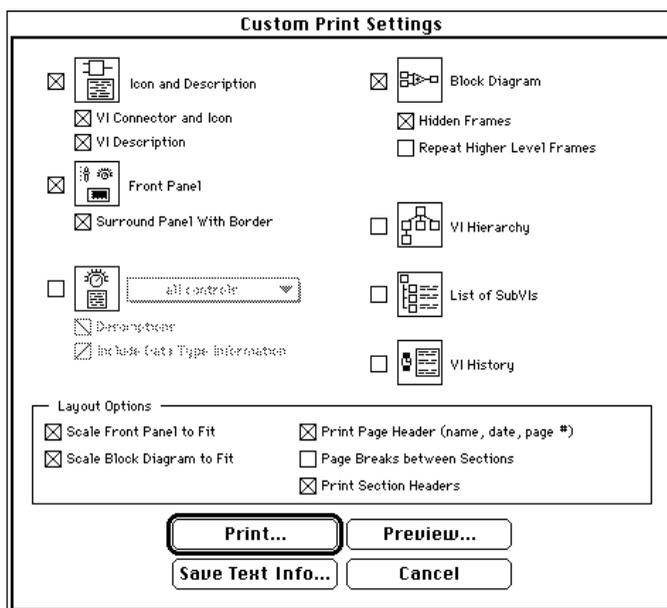
This dialog box displays the categories of items you can print. These categories are listed in the order they appear on the printout. In addition, the page layout options in this dialog box control scaling, page breaks, and headers for the printout.

The icon for each of the custom print setting menu items is shown to the left of the description.

Icon and Description

> **VI Connector and Icon**—Prints a picture of the VI icon along with its inputs and outputs.

> **VI Description**—Prints the VI description.

Front Panel—Prints the front panel.

Controls—Prints a list of the names of the controls and indicators. When an array, cluster, or refnum is encountered, the subcontrols are printed. Select the following menu items from the pop-up menu.

> ✓ all controls
> connected controls
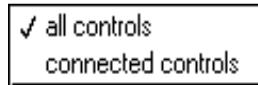
Descriptions—Prints the descriptions next to the control names.

Include Data Type Information—If you print to the printer, then the terminal for each control is printed to the left of the control name. If you save to a text file, the data type is printed after the description.

Block Diagram—Prints the block diagram.

Hidden Frames—Prints the nonvisible frames of Case Structures and Sequence Structures.

Repeat Higher Level Frames—When printing nonvisible frames, prints the visible ones again, in sequence.

VI Hierarchy—Prints a description of the current VI hierarchy in memory, with lines showing connections between VIs and their subVIs. The current VI is highlighted with a box.

List of SubVIs—Prints the icon, name, and path of all subVIs used.

VI History—Prints the history information, if any, for the current VI.

Programmatic Printing

If you want to print something under the control of your VI, rather than interactively as with the Print Window and Print Documentation dialog boxes, there are two ways to accomplish this.

- Using programmatic printing you can set a VI to automatically print its panel every time it completes execution.

- Using the VI Server, you can print the panel or documentation of any VI.

To implement programmatic printing, select **Operate»Print at Completion**.

When this option is on, the VI prints the contents of the front panel any time the VI completes execution. If the VI is a subVI, it prints the panel when that subVI finishes execution, before it returns to the caller.

Controlling When Printouts Occur

In some cases, you do not want a VI to print out every time. You might want it to occur only if the user presses a button, or if some condition occurs, such as a test failure. You also might want more control of the format for your printout. Or, you might want to print only a subset of the controls, or even just one specific control.

You can create a subVI whose panel is formatted the way you want the VI to look. Instead of selecting **Operate»Print at Completion** on your VI, select it from the subVI. When you want a printout, you can call the subVI, passing the data to it to be printed out.

With this organization, you have complete control of the printout appearance, using G controls and tools to create very simple or complex printouts.

An alternative method of controlling when printing occurs is to use the VI Server print methods. With these options, you can print any VI at any time. See Chapter 21, *VI Server*, for more information.

Enhancing Printouts

Using transparency and the Color tool, you can hide portions of controls you do not want to be visible in your printouts. For example, use transparency to simplify the appearance of the edges of controls.

Use graphical objects from the **Decorations** palette to highlight sections of your printout. For example, you might surround a section of controls with a box to set it off from the remainder of your panel.

You also can use bitmapped graphics to customize your printout, adding elements such as company logos to your report.

Setting Page Layout

The settings in **Edit»Preferences** and in **Printer Setup** (**Page Setup** for the Macintosh and UNIX) affect the appearance of your printouts. Use **VI Setup»Execution Options** to implement or disable some layout

options affecting the appearance of your printout. These page layout
options, shown in the following illustration, also affect the behavior of the
Print Window….
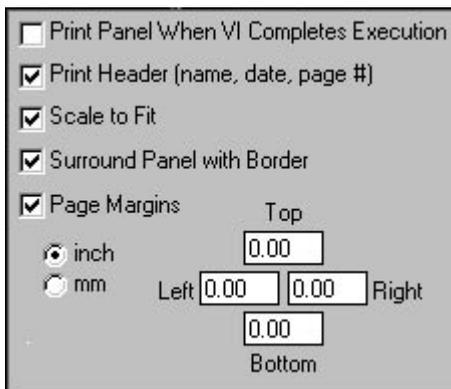
The **Print Panel When VI Completes Execution** menu item is another
way to turn on programmatic printing.

If you select **Print Header**, a header including the VI name, the last
modification date, and the page number appears at the top of each page.

If you select **Scale to Fit**, and the panel is bigger than a single page, the
printout scales down to as little as one-fourth the original size to fit the
panel on as few pages as possible.

If you select **Surround Panel with Border**, a box prints around the panel.

If you select **Page Margins** you can set absolute margins for printouts in
terms of inches or millimeters. All four margins can be specified separately.
Margins are limited by physical parameters of the printer. If you try to set
margins smaller than the printer permits, the actual margins used are the
minimum permitted by the printer.

Using Alternative Printing Methods

You might find the previously mentioned method of printing is not
appropriate for your application. There are some additional techniques that
address various printing concerns.

For example, the previous method of printing is geared towards printing an
entire page of data. In some applications, you might prefer to print data on
a line-by-line basis. If a line-based printer is connected to your serial or
parallel port, use the Serial Port VIs to send text to the printer. Doing this

generally requires some knowledge of the command language of the printer but works well for a number of applications developed by G users.

If printed results are not what you want, consider using another application to print your data by saving the data to a file and then printing from the other application.

For more details on alternative printing methods, see *How can I print a string?* in Appendix B, *Common Questions About G*.

Printing/Exporting Control and VI Descriptions to an RTF or HTML File

You can print or export VI descriptions and controls to Rich Text Format (RTF) and Hyper Text Markup Language (HTML) file formats. This is useful since you can import RTF into most document-publishing software. RTF files also are the source for online help files. Use the HTML format for online documents, especially those you intend to publish on the World Wide Web.

When you print documentation to an RTF file, you can specify whether you want to create a file suitable for online help files (online reference) or for word processing. In the word processing format, graphics are placed in the document. When you print your documentation in RTF format for online help files, the graphics are saved to external bitmap (.BMP) files. For HTML documentation, all graphics are saved externally in the JPEG (.JPG) or Portable Network Graphics (.PNG) file formats.

To print your documentation in RTF or HTML format, select **File»Print Documentation**. The following dialog box appears.
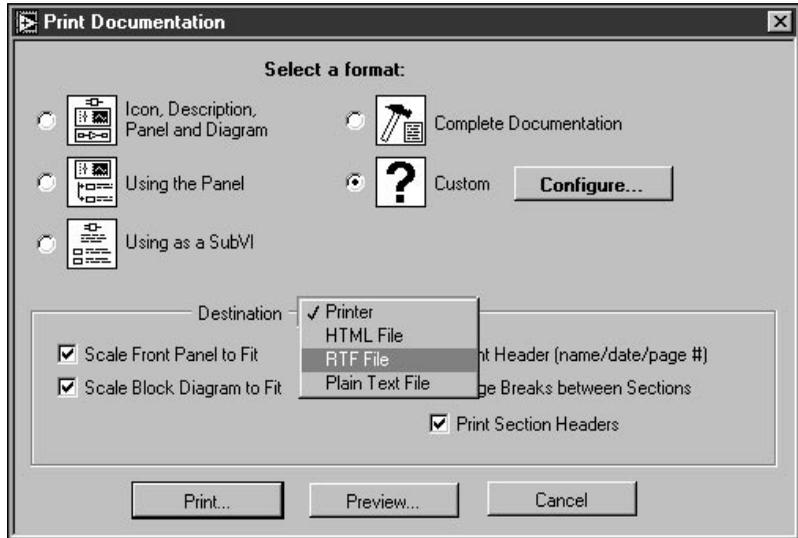
Figure 5-1. Print Documentation Dialog Box

Select **Destination** as RTF File or HTML File.

If you select RTF File, the dialog box shows the additional following menu items. Save your file in an online help format by selecting the **Help compiler source (images written externally)**, which saves the graphics to an external file. Select the **Destination** and color **Depth**.
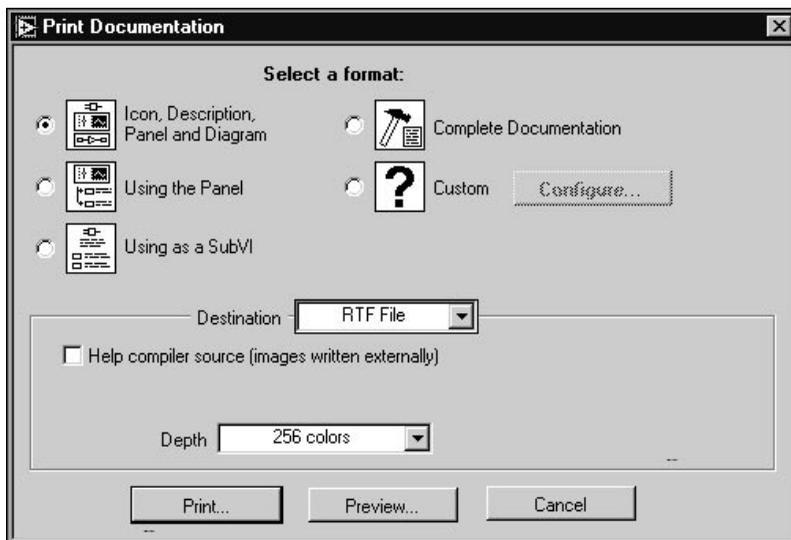
Figure 5-2. Print Documentation Dialog Box, RTF File

If you select HTML File as the format for your control and VI documentation, the dialog box appears as shown below.
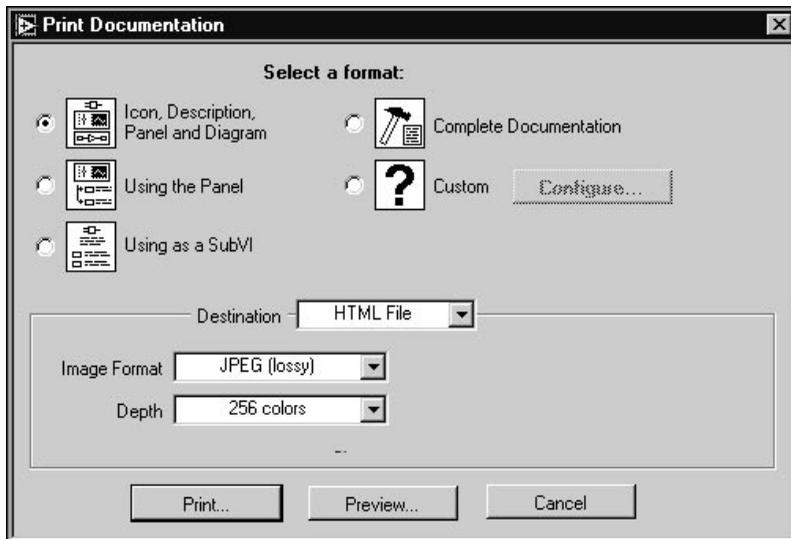
Figure 5-3. Print Documentation Dialog Box, HTML File

You can select the format of the graphics files as well as the color depth. The JPEG format has good compression, but can lose some image detail.

This format works best for photos. For line art, panels, and diagrams, JPEG compression can result in fuzzy images and uneven colors. Images in JPEG are always 24-bit. If you choose a lower color depth with JPEG, black-and-white for example, images capture with the requested depth but the result is still a 24-bit image.

The PNG format also has good compression, although not always as good as the JPEG format. However, PNG compression does not lose any detail. Also, it supports 1-bit, 4-bit, 8-bit, and 24-bit images. For lower-bit depth, the resulting image compresses much better than JPEG. The PNG format was developed to replace the GIF format. While it has a number of advantages over both JPEG and GIF, it is not as well supported.

If you want the graphics in your HTML files saved in the GIF format, you can convert the graphics using one of several shareware graphics-format converters. In this case, modify HTML to refer to the GIF images with the correct extension. If you choose to produce GIF images in this way, we suggest you start out with PNG images because they are lossless reproductions of the original graphics. Then convert those to GIF. Because of licensing issues, GIF is not directly supported, but might be in the future.

When you save documentation for the VI, a dialog box appears prompting you for the name of the document. The default name of any documentation you save in RTF or HTML includes the name of the VI with an `.rtf` or `.html` extension in the file name, respectively (such as `AI Clear.rtf`). (Any documentation you save in HTML on the Windows 3.1 platform includes the VI name with an `.htm` extension.)

The following table is based on an example and lists the VI parameters and their associated JPEG file names. If you select a JPEG graphic format while saving a VI named Test, the following JPEGs result.

Table 5-1. Resulting JPEGs

VI parameter	JPEG name
VI icon	`testi.jpg`
VI connector	`testc.jpg`
VI front panel	`testp.jpg`
VI block diagram	`testd.jpg`
VI hierarchy window	`testh.jpg`
Subvi icons	`name of subVI` + `i.jpg`

Images for control and indicator terminals are saved to image files with consistent names. Thus, if your VI has multiple inputs of the same type, you end up with a single image file for each of the terminals. For example, if your VI has three int32 inputs, a single `ci32.jpg` file is created.

The specific naming scheme is as follows. The first letter is a `c` or an `i`, depending upon whether the terminal is for a control or an indicator. If the terminal represents an array, the letter is followed by the terminal's dimensionality (such as `2d`). Then follows a short name representing the data type. Following are some examples of image file names.

Table 5-2. Image Naming Scheme Examples

Name	Terminal
`ci32.jpg`	control, int32
`i2di32.jpg`	indicator, 2d array of int 32s
`c3dstr.jpg`	control, 3d array of strings

Programmatic RTF and HTML files

For information on creating RTF and HTML files programmatically, refer to Chapter 21, *VI Server*. This chapter also contains information on the export and import of strings for multiple VIs.

Localization Issues

By localizing the strings on the front panel of a VI, you also can localize the VI printouts. For more information on localizing VI strings, see the VI localization section of Chapter 29, *Portability and Localization Issues*.

Creating Your Own Help Files

You can create your own online help or reference documents with the right development tools. Help documents are based on formatted text documents, and you can enter topics in these documents to create connections to your VIs.

The source files for all platforms must be in Windows Help format. It is possible to create help documents for multiple platforms.

When you create source documents, you use a help compiler to create a help document. If you want help files on multiple platforms, you must use the help compiler for the specific platform on which you intend to use the

file. You can use any of the following compilers. The Windows compilers also include tools for creating help documents.

- **(Windows)** RoboHelp from Blue Sky Software, 800 677 4946 for international customers, 619 459 6365

- **(Windows)** Doc-To-Help from WexTech Systems, Inc., 800 939 8324

- **(Macintosh)** QuickView from Altura Software, 408 655 8005

- **(UNIX)** HyperHelp from Bristol Technologies, 203 438 6969

Once you create and compile your help files, you can link them directly to a VI. Pop up on the VI connector pane of the VI which you want to link a help file with and select **VI Setup»Documentation**. Select the **Help Tag** box and type the topic link in the help file. Choose the help file by clicking the **Browse…** button. The file path appears in the **Help Path** box.

6

Setting up VIs and SubVIs

This chapter describes how you customize VI behavior using the VI **Setup** and **SubVI Node Setup** dialog boxes.

Creating Pop-Up Panels

A single front panel is sometimes too restrictive to present numerous options or displays. To solve this problem, organize your VIs so the topmost VI presents high-level options and subVIs present related options.

When G calls a subVI, ordinarily it runs without opening its front panel. You use the **VI Setup** or **SubVI Node Setup** dialog boxes to make a subVI open its front panel when the subVI is called and close the front panel when the subVI completes execution.

You access the **VI Setup** dialog box by popping up on the VI icon in the top right of a front panel and selecting **VI Setup…**, as shown in the following illustration. You must be in edit mode.
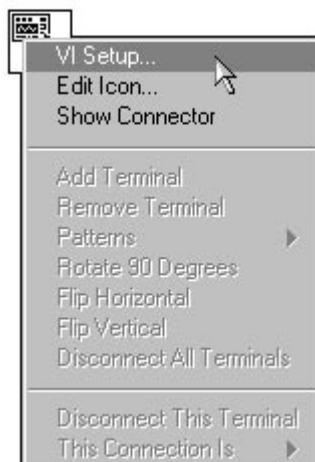
This dialog box contains several options for customizing the look and behavior of the VI. Use the ring at the top of the dialog box to select from four different option categories—**Execution Options**, **Window Options**, and **Documentation**.

VI Setup Options

This section describes options in the execution, visual context, documentation, and menu-item customization of your VIs.

Execution Options

Initially, the **VI Setup** dialog box presents you with execution options for the current VI. The **Execution Options** are shown in the following illustration.
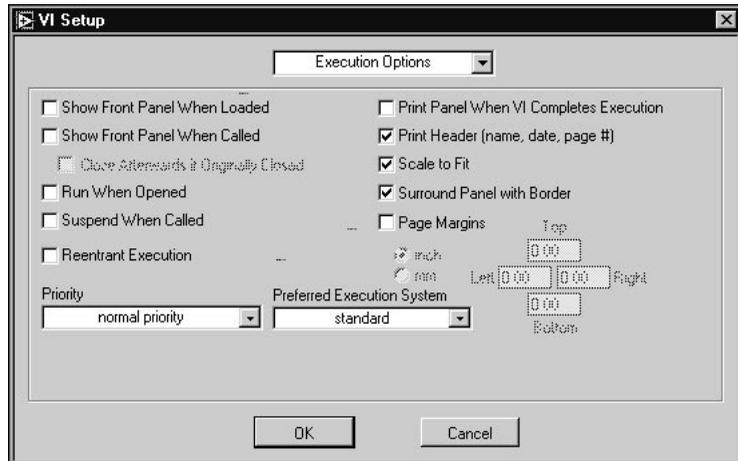
If you want a subVI panel to open as soon as the top-level VI is loaded, turn on the **Show Front Panel When Loaded** option.

Two of the most useful options of this page are **Show Front Panel When Called** and **Close Afterwards if Originally Closed**. If you turn these options on for a subVI, that subVI front panel opens automatically when the subVI is called and then closes automatically if it was originally closed.

The **Run When Opened** option sets up a VI to start running automatically when it opens.

Setting the **Suspend When Called** option is the same as selecting **Operate»Suspend When Called**. This debugging option is described in the *Debugging Features* section of Chapter 4, *Executing and Debugging VIs and SubVIs*.

The **Reentrant Execution, Priority** and **Preferred Execution System** are fairly advanced features affecting the way a VI executes. You need them only in special applications. These options are described in the *Reentrant Execution Overview* and *VI Setup Priority* sections of Chapter 26, *Understanding the G Execution System,* respectively. Another advanced execution feature is *multithreading,* in which simultaneous execution of several VIs takes place while the VIs still respond to mouse and keyboard input, and CPU time shares evenly among the execution threads. This feature also is described in Chapter 26, *Understanding the G Execution System*.

Use the remaining options to print your VI when executed and customize the way it looks when printed. Programmatic printing is discussed in detail in the *Programmatic Printing* section of Chapter 5, *Printing and Documenting VIs*.

Window Options

The window options apply to the VI when it is executing, but not in edit mode. Use these options to control the ability of a user to interact with the program by restricting access to G features, and by changing the way the window looks and behaves. You can make your VI look and act like a dialog box, because the user cannot interact with other windows while this window is open. You also can remove the scrollbars and the toolbar, and set a window to be centered or sized automatically to fit the screen.

You can customize the VI window title to make it more descriptive than the VI file name. This feature is important for localized VIs if the VI window title might be translated to the local language. To change the VI window title while you are editing the VI, select **VI Setup** and **Window Options**

from the top ring, as shown in the following illustration. Deselect **Same as VI Name** and type in the desired VI window title.
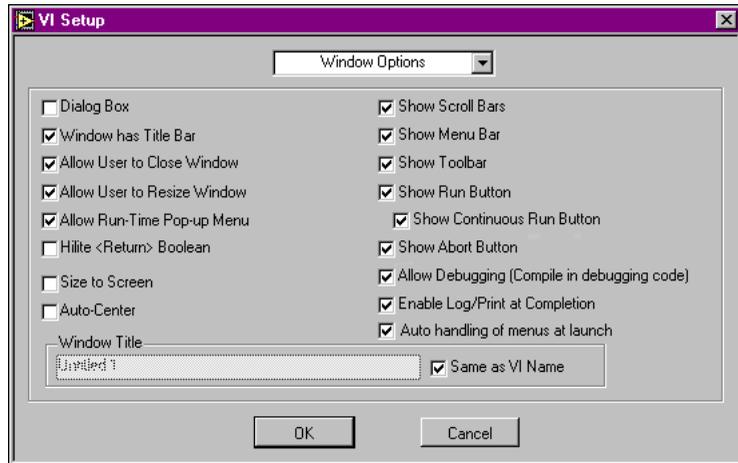
The **Dialog Box** option prevents the user from interacting with other windows while this is open, just as a system dialog box does.

Notice that on UNIX, this menu item does not prevent you from bringing windows of other applications to the front. There is no way to make a window stay in front of all other windows.

Allow Run-Time Pop-Up Menu determines if objects on this front panel display a pop-up menu of data operations in run mode.

When you select the **Hilite <Return> Boolean** menu item, highlighting occurs on any Boolean currently associated with the <Enter> **(Windows** and **HP-UX)**, or <Return> **(Macintosh** and **Sun)** key. You associate keys with controls using the Key Navigation dialog box, described in the *Key Navigation Option for Controls* section of Chapter 8, *Introduction to Front Panel Objects*.

The **Size to Screen** menu item automatically resizes the panel of a VI to fit the screen when you run the VI. The VI does not retain a record of its original size and location, so it stays in the same place when it switches back to edit mode.

Auto-Center automatically centers the front panel on your computer screen.

Other menu items toggle between showing and hiding various window features. Hide individual buttons by deselecting them, or hide the entire toolbar by deselecting the **Show Toolbar** menu item.

Enable Log/Print at Completion enables or disables automatic data logging (see the *Data Logging on the Front Panel* section in Chapter 4, *Executing and Debugging VIs and SubVIs)*, and programmatic printing (see the *Programmatic Printing* section in Chapter 5, *Printing and DocumentingVIs*).

To change the VI window title programmatically, refer to Chapter 21, *VI Server*.

Deselecting **Auto handling of menus at launch** disables the runtime menu bar until you are ready to handle menu selections using the Get Menu selection function. See *Menu Selection Handling* later in this chapter for more information.

⚠ **Caution**   *If you hide the menu bar and the toolbar, there is no visible means of changing from run mode back into edit mode. Use the* <Ctrl-m> **(Windows)**; <command-m> (**Macintosh)**; <meta-m> **(Sun)**; *or* <Alt-m> **(HP-UX)** *hot key corresponding to the* **Operate»Change to Edit Mode** *menu item to change the VI back to edit mode, where the menu bar and palette are visible.*

Documentation Options

The dialog box in **VI Setup»Documentation**, shown in the following
illustration, presents you with menu items concerning entries made to the
History window, which displays the development history of the VI. For
information on the History window, see the *VI History Window* section in
Chapter 27, *Managing Your Applications*.
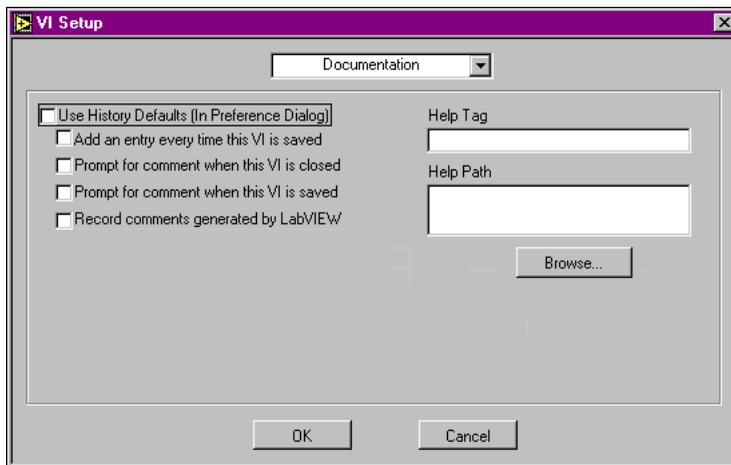
To use any of the history items in the **Documentation** menu, **Use History
Defaults (In Preferences Dialog)** must be deselected. If you select this, the
history preferences are used instead. The history preferences are identical,
except that they set up the defaults used when you create any new VI,
although the VI Setup documentation menu items apply only to the
current VI.

If you check **Add an entry every time this VI is saved**, an entry is added
to the VI history every time you save the VI. If you do not enter a comment
in the **Comment** box of the History window, only a header is added to the
history. The header contains the revision number, the date and time, and the
name of the VI if the menu item is checked in the **Preferences»History**
dialog box.

If you check the **Prompt for comment when this VI is closed** option, the
History window appears. You can enter a comment when you close a VI
that has changed since you loaded it, even if you saved the changes.

If you check the **Prompt for comment when this VI is saved** option, the History window appears when you save. This is useful if you prefer to comment on your changes when you finish making them instead of as you are editing. If you do not set this option, you cannot change the history of the VI after you select **Save** until the save is finished.

☞ **Note**   *You are not prompted to enter a comment when you save or close a VI if the only changes you made were to the history.*

The items to the right of the **VI Setup»Documentation** dialog box create access to online help files you created. Put your cursor in the **Help Tag** box and type in a topic to display for this VI. Then type in the path to the help file in the **Help Path** box, or click the **Browse…** button and find the file. The filename and path of the file appears in the **Help Path** box. After you set these menu items, click the online help icon at the bottom of the Help dialog box to access the help file you selected. See Chapter 5, *Printing and Documenting VIs*, for information on creating your own help files.

SubVI Node Setup Dialog Box

The following dialog box appears when you select **SubVI Node Setup…** from the node pop-up menu of a subVI on the block diagram.
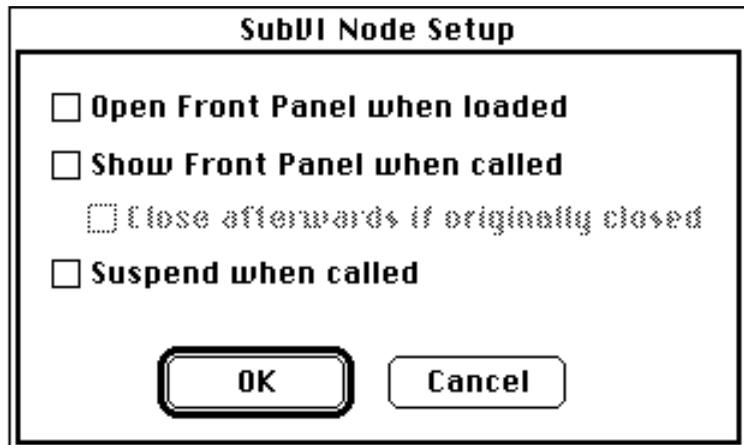
These menu items are a subset of the items in the VI Setup dialog box. The difference between the two is you use the SubVI Setup dialog box to specify items relating to a specific call to a subVI. However, you use the subVI execution items of VI Setup to specify the behavior of all calls to that VI.

Customizing the Menubar

You can customize the menubar for every VI you build, although you can see the custom menubars only when the VI is running.

There are two parts to customizing menus—creating menus and handling menus.

You can build custom menus in two ways—statically at the time of editing, or dynamically at run time. You can statically create a custom menu template and store it in a file called the run-time menu (RTM) file using the Menu Editor (see the *Menu Editor* section in this chapter for more information). The Menu Editor also lets you associate a RTM file with the VI in question. When the VI runs, it loads the menu from the associated RTM file. You can insert, delete and modify menu items programmatically, at run time, from the diagram using G functions (see the *Dynamic Menu Functions* section in this chapter for more information).

The diagram handles custom menu items. Each menu item has a unique case-insensitive string identifier called a tag. Whenever you select a menu item, you can retrieve the tag of the selected menu item programmatically using the G function **Get Menu Selection** (see the *Menu Selection Handling* section in this chapter for more information). Based on the tag value, you can provide a handler in the diagram for each of the menu items.

☞ **Note**   *Custom menu bars also are referred to as run-time menus.*

Menu Editor

The Menu Editor provides an interface to create and edit RTM files and associate them with the VI in question. You invoke the Menu Editor on a VI by selecting **Edit»Edit Menu…**. Currently, you can associate 3 types of menus with a VI, **Default**, **Minimal** and **Custom** types as shown in Figure 6-1. The Default option provides the standard menu. The Minimal option removes the infrequently used items from the standard menu. The Custom option lets you associate a RTM file with the VI. The Default and Minimal types cannot be edited.
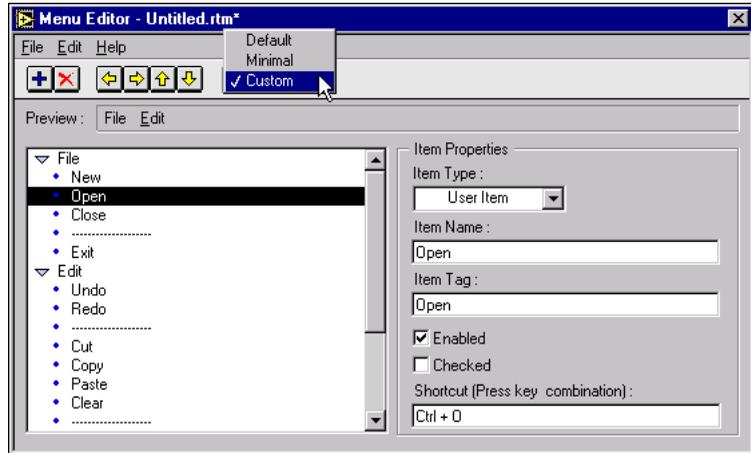
Figure 6-1. Menu Editor

The Menu Editor displays the menu hierarchy on the left hand side and the properties of the menu item selected from the menu hierarchy on the right hand side. You can view the menu as it appears at run-time in the **Preview** area. You can insert a new menu item by clicking the button from the tool bar shown at left, or by selecting the insert options from the **Edit** menu.

A menu item can be of three types: *User Item*, *Application Item* or *Separator*. You can change the type of a menu item by choosing from **Item Type** ring.

A **User Item** can be handled programmatically in the diagram. It has a name, which is the string appearing in the actual menu, and a tag which is a unique case-insensitive string identifier. A tag identifies the User Item in the diagram. For ease in editing, whenever you enter a name it is copied to the tag. However, you can always edit the tag to make it different from the name. In order for a menu item to be valid, its tag must have a value. Invalid menu items are shown as **???**. The Menu Editor always ensures the tag is unique to a given menu hierarchy by appending numbers when necessary.

A User item can be enabled or disabled, checked or unchecked, by setting the respective attributes. You can set a shortcut (accelerator) for a User item by typing an appropriate key combination. The shortcut has the <Menu Key[-Shift]-key> format.

Note   *PC users can use underscores in item names to facilitate menu selection using the <ALT> key. For example, to select an item named File/Exit using < ALT+F+X>, use _File and E_xit as the item names.*

☞ **Note**   *Shift key and function keys in shortcuts do not show up under the Macintosh OS 7 or lower. To make menus portable on the Macintosh OS 7 avoid using shift or function keys in shortcuts*

☞ **Note**   *On the Macintosh platform, all items in the menubar must be pull down menus. However, Windows 95/NT and UNIX users can create selectable items in the menubar. Avoid using selectable items in menubars on the Macintosh.*
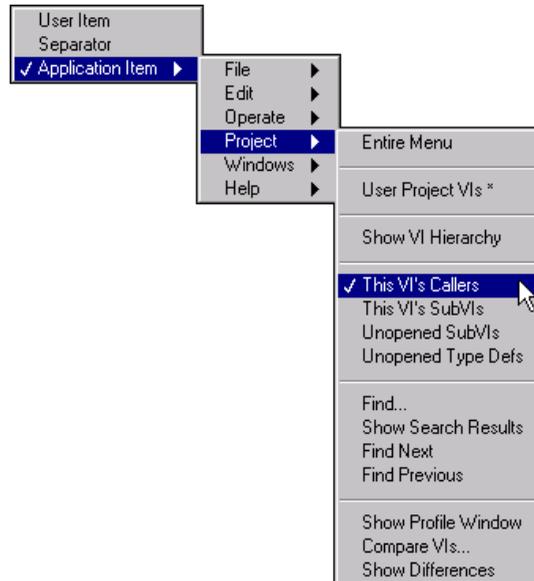
Figure 6-2. Choosing an Application Item

An **Application Item** is one provided by the G-language software. These are the items that are part of default menu. To select a particular item, choose **Item Type»Application Item** and follow the hierarchy as shown in the illustration. You can add individual items or entire submenus by using this process. Application items are handled implicitly by the software. These item tags do not show up in diagrams. You cannot alter the name, tag or other attributes of an Application item. The software reserves tags starting with APP_ for Application items. The tags for Application items are listed in Table 6.1 at the end of this chapter.

A **Separator** item inserts a separation line in the menu. You cannot set any attributes for a Separator item.

You can move items in the menu hierarchy by clicking the arrow buttons in the tool bar or using the hierarchy manipulation options in **Edit** menu or

by using simple drag and drop. Hierarchies can be collapsed or expanded for viewing comfort by clicking the submenu glyphs or choosing from expand/collapse options in **Edit** menu.

You can load or save RTM files using options from the **File** menu. You can use the **Edit** menu options to cut, copy, paste, insert and delete menu items (see the *Menu Editor Menu Options* section in this chapter for more information). You can get help on any control in the Menu Editor by choosing **Help»Show Help** and pointing the mouse at the control in question.

Menu Editor Menu Options

The following options are available in the **File** menu.

Open—Opens an existing run-time menu file. Opening a RTM file causes the Menu Editor to switch to Custom type.

New—Creates a new run-time menu file. Prompts to save any previously edited RTM file.

Save—Saves the current run-time menu.

Save As—Saves the current run-time menu to a different file.

Close—Exit the Menu Editor.

The following options are available under the **Edit** Menu.

Cut—Delete the currently selected menu item or text and copy it to clipboard.

Copy—Copy the currently selected menu item or text to clipboard.

Copy Entire Menu—Copy the entire menu hierarchy to clipboard.

Paste—Paste the contents of clipboard.

Collapse/Expand—Collapse or Expand the currently selected submenu item.

Collapse All—Collapse all submenu items.

Expand All—Expand all submenu items.

Insert User Item—Insert a user menu item past the selected menu item.

Insert Separator—Insert a separator item past the selected menu item.

Insert Application Item—Brings up a submenu to select from one of the available application items.

Delete Item—Delete the selected menu item.

Make Super Item—Makes the items following the selected item as sub-items of the selected item.

Make Sub Item—Make the selected item a submenu item of the preceding menu item.

Move Item Up—Move the selected item up. If the selected item has sub-items, they are moved along with the selected item.

Move Item Down—Move the selected item down. If the selected item has sub-items, they are moved along with the selected item.

Menu Editor Tool Bar Options

The following options are available under the toolbar.

Insert a user menu item past the selected menu item.

Delete the selected menu item.

Make the items following the selected item as sub-items of the selected item.

Makes the selected item a submenu item of the preceding menu item.

Move the selected item up. If the selected item has sub-items, they are moved along with the selected item.

Move the selected item down. If the selected item has sub-items, they are moved along with the selected item.

Menu Selection Handling

This section provides a detailed description of the **Get Menu Selection** and **Enable Menu Tracking** functions, descriptions of which appear in the *Menu Selection Handling Functions* section later in this chapter. The functions operate on menus identified by a refnum. The menu refnum of a VI is obtained through the constant current VI menu, shown at left, and in the following illustrations, Figure 6-3 and Figure 6-4.
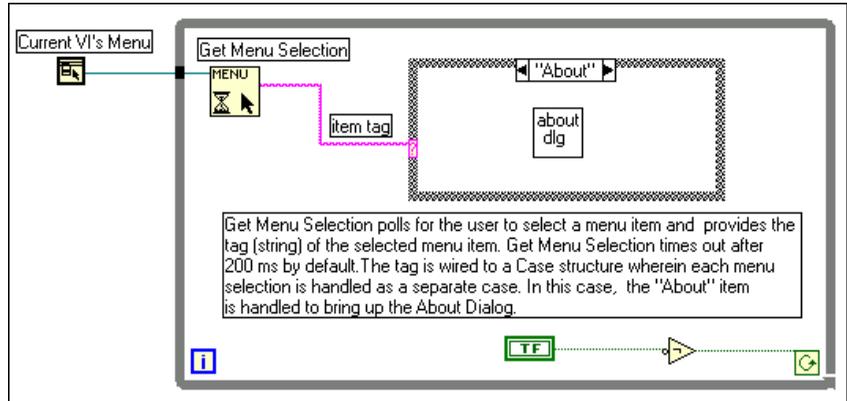
Figure 6-3. Get Menu Selection Function

Once you select an item, you cannot select another item until the **Get Menu Selection** function reads the first item. However, once an item is read, you can select a new menu item, even if the previous item has not processed. This is not usually done when it takes a while to process a selection. Under such conditions, **Get Menu Selection** is invoked under block menu mode, wherein menu tracking is blocked out after a selection is read. The menu is enabled after you process the selection using the **Enable Menu Tracking** function, as shown in the following illustration.
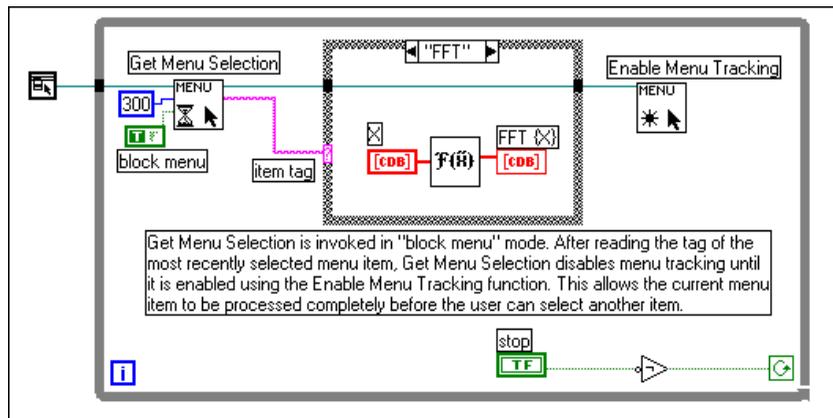
Figure 6-4. Enable Menu Tracking Function

Menu Selection Handling Functions

The following Menu Selection Handling functions are available.

Get Menu Selection

Returns the **item tag** of the last selected menu item, optionally waiting `timeout milliseconds`. **Item path** is a string describing the position of the item in the menu hierarchy, which is the format of a list of menu tags separated by a colon (:). If **block menu** is set to True, Menu selection is blocked out after an item tag is read.
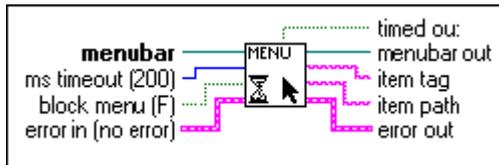
Enable Menu Tracking

Enables or disables tracking of menu selections.
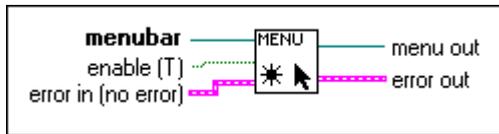
The functions operate on menus identified by a refnum. The menu refnum of a VI is obtained through the menu of the constant current VI, shown at left. Items are identified by an item tag (string) and sometimes by an item path (string), which is a list of item tags from the menu tree root up to the item and separated by colons.

Dynamic Menu Functions

While a VI executes, you can change items in the VI menubar by using the menu management functions **Insert Menu Items**, **Delete Menu Items**, **Get Menu Item Info**, **Set Menu Item Info**, and **Get Menu Shortcut Info**. You find these functions on the **Application Control** palette, which you access by selecting **Functions»Application Control**. The functions operate on menus identified by a refnum. The menu refnum of a VI is obtained through the menu of the constant current VI, shown at left.

Insert Menu Items

Inserts menu items into a menubar or a submenu within the menubar.
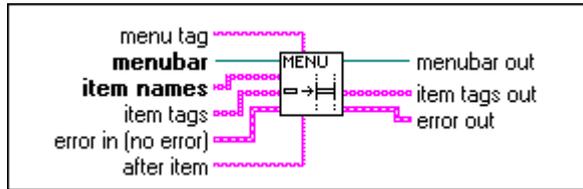
Menu tag specifies the submenu where items are inserted. If you do not specify **menu tag**, the items are inserted into the **menubar**.

Item names and **item tags** identify the items to be inserted into the menu. The type of **item names** and **item tags** can be an array of strings (for inserting multiple items) or just a string (for inserting a single item). You can wire in either the **item names** or **item tags**, in which case both names and tags get the same values. If you require a different name and tag for each item, you must wire in separate values for **item names** and **item tags**.

After item specifies the position where the items are inserted. **After item** can be a tag (string) of an existing item or a position index (zero based integer) in the menu. To insert at the beginning of the menu, wire a number less than 0 to **after item**. To insert at the end of the menu, wire a number larger than the number of items in the menu. You can insert Application items by using application tags from Table 6.1 at the end of this chapter. You can insert a separator using the application tag APP_SEPARATOR. The function always ensures that the tags of all the inserted menu items are unique to the menu hierarchy by appending numbers to the supplied tags, if necessary.

Item tags out returns the actual tags of the inserted items. If **menu tag** or **after item** (tag) is not found, the function returns an error.

The example in the following illustrations dynamically builds a menu tree consisting of two top-level menus, **File** and **Edit**, and a submenu **Test**.
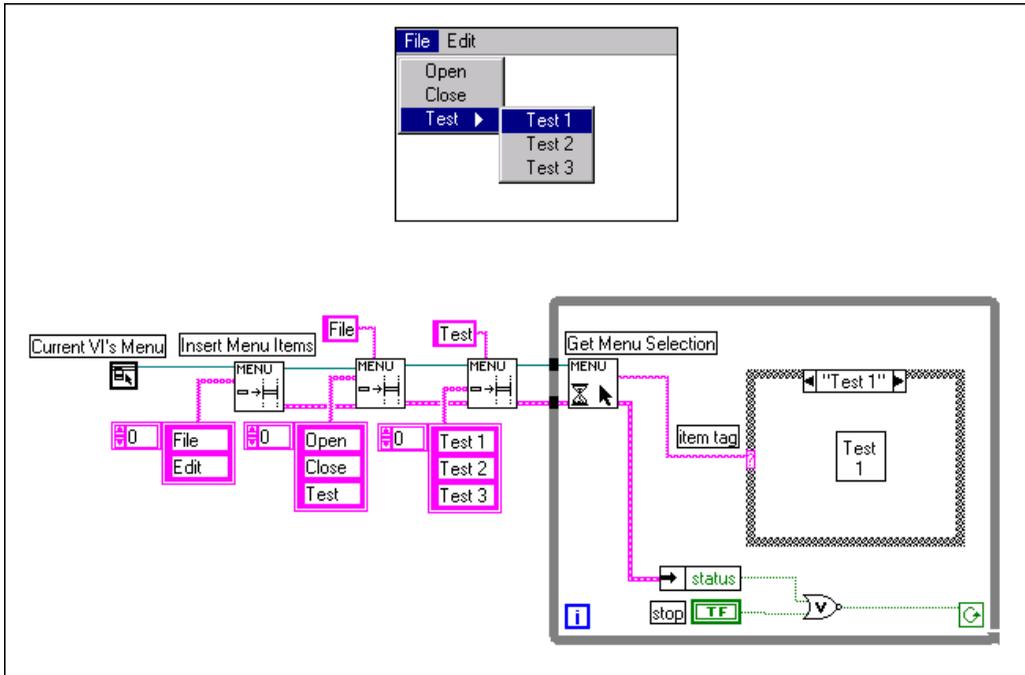
Figure 6-5. Dynamic Menu Insertion

Delete Menu Items

Deletes menu items from the menubar or a submenu within the menubar.
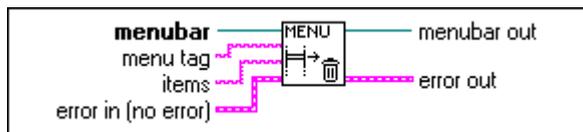
If **menu tag** is specified, the items are deleted from the submenu specified by **menu tag**, or else the items are deleted from the menubar. The function returns an error if **menu tag** or one of the items specified is not found.

Items can be a tag (string) of an existing item, an array of tags of existing items, a position index (zero-based integer) of an item in the menu or an array of position indices of items in the menu. If you do not wire **items**, all the items in the menubar are deleted. If there is a submenu in any of the

specified items, the submenu and all its contents are deleted automatically. You can delete Application items by using application tags listed in Table 6.1 at the end of this chapter. Because separators do not have unique tags, they are best deleted by using their positional indices.

Get Menu Item Info

Returns the attributes of the menu item specified through item tag.
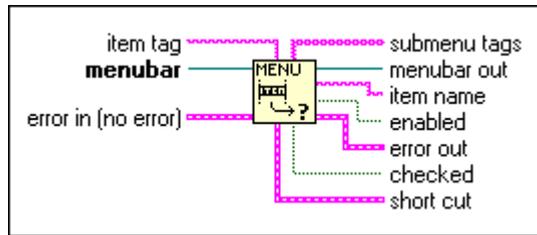
Item attributes are **item name** (the string that appears in the menu), **enabled** (false designates that the item is grayed out), **checked** (specifies whether there is a check mark next to the item), and **shortcut** (key accelerator). If the item has a submenu, its item tags are returned as an array of strings in **submenu tags**. If **item tag** is unwired, the menubar items are returned. If **item tag** is not valid, an error is returned.

Set Menu Item Info

Sets the attributes of a menu item specified through menu and item tag. Item attributes are item name (the string that appears in the menu), enabled (false designates that the item is grayed out), checked (specifies whether there is a check mark next to the item), and shortcut (key accelerator). Attributes that are not wired remain unchanged. If item tag is not valid, an error is returned.
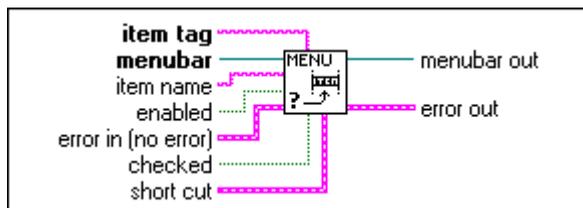
Get Menu Shortcut Info

Returns the menu item that is accessible through a given shortcut.

```
menubar  ──────────┐ MENU ┌────── menubar out
short cut ─────────┤ ▣→? ├─────── item tag
error in (no error) ┤     ├─────── item path
                    └─────┘└────── error out
```
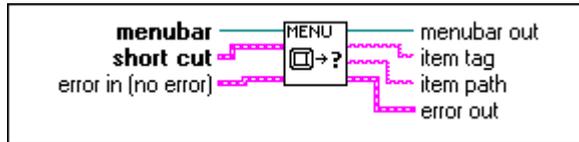
Item path is a string of menu item tags separated by a colon (:).

Short cut consists of a string (key) and a Boolean (specifies whether the shift key is included or not).

Application Item Tags

You can insert application items by using the Application item tags listed in the following table.

Table 6-1. Application Item Tags

Application Item	Item Tag
File	APP_FILE
New	APP_NEW
Open...	APP_OPEN
Close	APP_CLOSE
Save	APP_SAVE
Save As...	APP_SAVEAS
Save A Copy As...	APP_SAVE_A_COPY_AS
Save with Options...	APP_SAVE_WITH_OPTIONS
Printer Setup...	APP_PRINTER_SETUP
Print Documentation...	APP_PRINT_DOCUMENTATION
Print Window...	APP_PRINT_WINDOW
Edit VI Library...	APP_EDIT_VI_LIBRARY
Edit Template...	APP_EDIT_TEMPLATE
Mass Compile...	APP_MASS_COMPILE

Table 6-1. Application Item Tags (Continued)

Application Item	Item Tag
Convert CVI FP File...	APP_CONVERT_CVI
Exit	APP_EXIT
Edit	APP_EDIT
Undo	APP_UNDO
Redo	APP_REDO
Cut	APP_CUT
Copy	APP_COPY
Paste	APP_PASTE
Clear	APP_CLEAR
Import Picture from File...	APP_IMPORT_PICT_FROM_FILE
Preferences...	APP_PREFERENCES
User Name...	APP_USER_NAME
Clear Password Cache	APP_CLEAR_PASSWORD_CACHE
Select Palette Set	APP_SELECT_PALETTE_SET
Edit Control & Function Palettes...	APP_EDIT_PALETTES
Operate	APP_OPERATE
Stop	APP_STOP
Print at Completion	APP_PRINT_AT_COMPLETION
Log at Completion	APP_LOG_AT_COMPLETION
Data Logging	APP_DATA_LOGGING
Log...	APP_LOG
Retrieve...	APP_RETRIEVE
Purge Data...	APP_PURGE_DATA
Change Log File Binding...	APP_CHANGE_LOG_BINDING
Clear Log File Binding	APP_CLEAR_LOG_BINDING
Suspend when Called	APP_SUSPEND_WHEN_CALLED

Table 6-1. Application Item Tags (Continued)

Application Item	Item Tag
Reinitialize All To Default	APP_REINIT_ALL_TO_DEFAULT
Project	APP_PROJECT
User Project VIs *	APP_USER_PROJECT_VIS
Show VI Hierarchy	APP_SHOW_VI_HIER
This VI's Callers	APP_THIS_VIS_CALLERS
This VI's SubVIs	APP_THIS_VIS_SUBVIS
Unopened SubVIs	APP_UNOPENED_SUBVIS
Unopened Type Defs	APP_UNOPENED_TYPEDEFS
Find...	APP_FIND
Show Search Results	APP_SHOW_SRCH_RSLTS
Find Next	APP_FIND_NEXT
Find Previous	APP_FIND_PREV
Show Profile Window	APP_SHOW_PROFILE_WINDOW
Compare VIs...	APP_COMPARE_VIS
Show Differences	APP_SHOW_DIFFERENCES
Export Strings...	APP_EXPORT_STRINGS
Import Strings...	APP_IMPORT_STRINGS
Windows	APP_WINDOWS
Show Panel	APP_SHOW_PANEL_DIAGRAM
Show VI Info...	APP_SHOW_VI_INFO
Show History	APP_SHOW_HISTORY
Show Functions Palette	APP_SHOW_CTRLS_FCTNS
Show Tools Palette	APP_SHOW_TOOLS
Show Clipboard	APP_SHOW_CLIPBOARD
Show Error List	APP_SHOW_ERRORS
Tile Left and Right	APP_TILE_LEFT_RIGHT
Tile Up and Down	APP_TILE_UP_DOWN

Table 6-1. Application Item Tags (Continued)

Application Item	Item Tag
Full Size	APP_FULL_SIZE
Open Windows *	APP_OPEN_WINDOWS
Help	APP_HELP
Show Help	APP_SHOW_HELP
Lock Help	APP_LOCK_HELP
Simple Help	APP_SIMPLE_HELP
Online Reference...	APP_ONLINE_REF
Online Help for VI	APP_ONLINE_HELP_FOR_VI
Help Files *	APP_HELP_FILES
About LabVIEW...	APP_ABOUT

7

Customizing Your Environment

This chapter explains how to customize your environment by setting editor preferences for features such as printing, display, and **Undo**, and by changing the contents of the **Controls** and **Functions** palettes.

Setting Preferences

To customize your application and configure the editor, use the Preferences dialog box.

You access the Preferences dialog box by selecting **Edit»Preferences…**, as shown in the following illustration.
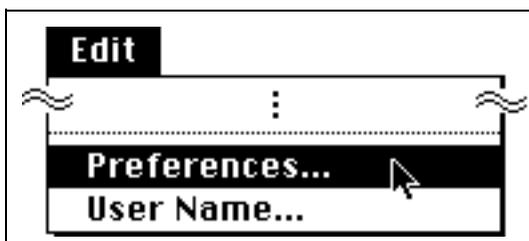
Use the pull-down menu at the top of the dialog box to select among the different categories of preferences, as shown in the following illustration.
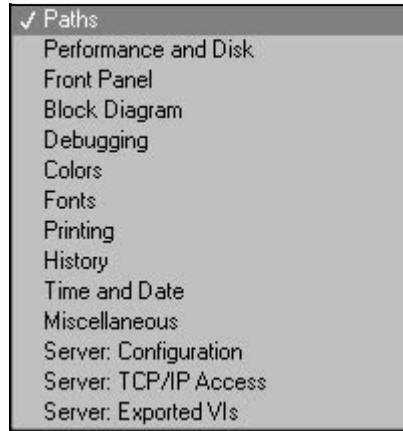
Path Preferences

You can specify the directories searched when looking for VIs as well as the paths used for temporary files and the library directory. If not already selected, select **Paths** from the pull-down menu in the Preferences dialog box to bring up the dialog box shown in the following illustration.
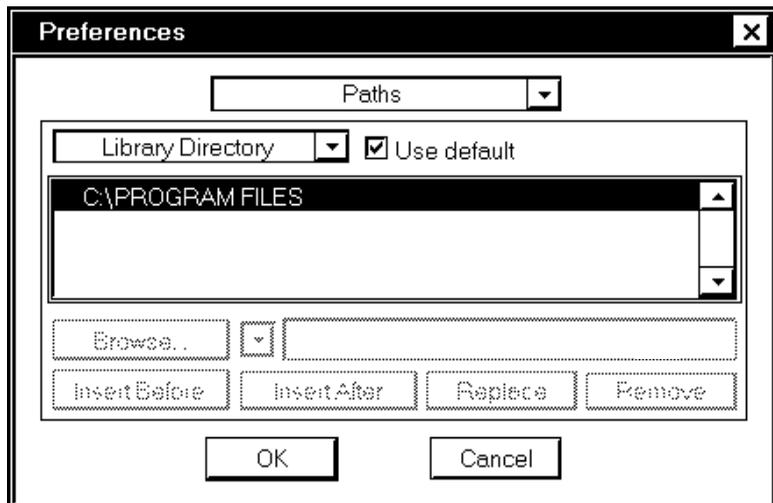
Note   *The format of pathnames — the use of colons, slashes, or backslashes — differs slightly on different platforms.*

In the Path Preferences dialog box is another pull-down menu, shown in the following illustration, from which you select the path category you want to view or edit.
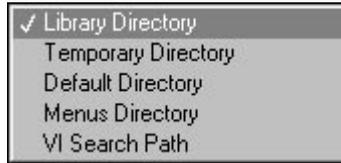
In the dialog box that appears, many items are grayed out and inactive because G is set to use the default path. If you want to change one of these preferences, deselect the **Use Default** checkbox, as shown in the following illustration.
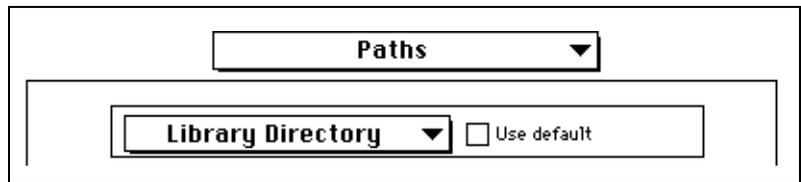
Library, Temporary, Default, and Menus Directories

The **Library Directory**, **Temporary Directory**, **Default Directory** and **Menus Directory** are single directories (folders). When you edit one of these paths, you have options to type in a new path and replace the existing path, or browse the file dialog box to select a path. This is shown in the following illustration.
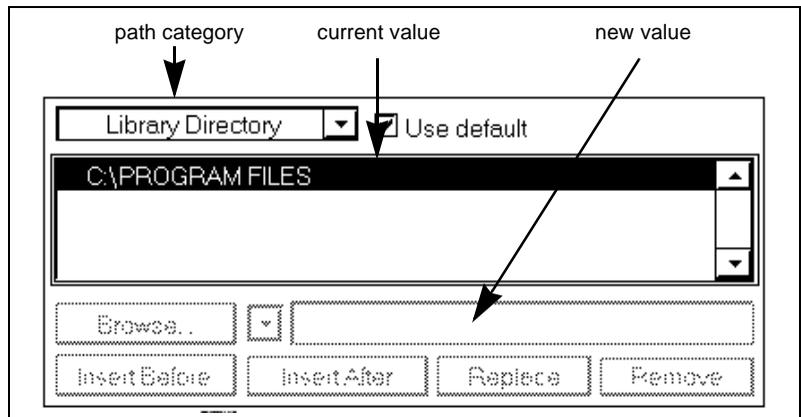
The **Library Directory** shows the absolute pathname to the directory containing `vi.lib` and any library directories you supply. The default is the directory containing your G development environment.

The **Temporary Directory** shows the absolute pathname to the directory for temporary files, which varies according to platform, as follows.

- **(Windows)** The default is the directory containing your G development environment.

- **(Macintosh)** Under System 7, the default is an invisible folder called `Temporary Items` at the top of your hard drive, which is where Apple recommends storing temporary files. If your development environment crashes, the temporary files move into the trash when you restart.

- **(UNIX)** The default is the `/tmp` directory.

The **Default Directory** shows the absolute pathname to the default directory, which is the initial directory the file dialog box displays. The **Default Directory** function in the **Functions»File I/O»File Constants** palette also returns this value. The default is the current working directory. Changes to the default directory take place immediately

The **Menus Directory** shows the path to the directory containing the palette menu files describing the organization of the **Controls** and **Functions** palettes. By default, this is the **Menus Directory** in your **Library Directory**. In general, it is not necessary to change this, although you might want to change it if you have a network installation and individual users require their own custom menu sets.

☞ **Note**   *All changes to the Library Directory, Temporary Directory, Menu Directory and Default Directory options take effect when you relaunch your G development environment.*

VI Search Path

The **VI Search Path** is used only when your G development environment searches for a subVI, control, or external subroutine that was not in the expected location. It lets you list the path of directories for G to search. When you edit the search path, you are given more options than for the other paths — you can add new items in specific locations, remove paths, and select from a list of *special* paths.
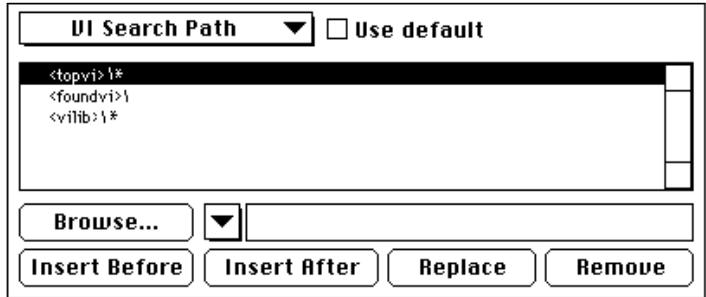
The list contains paths your G development environment searches, in the order it searches them. To add a new directory, first decide when your G development environment searches that directory relative to other directories. Select an adjacent directory from the list. Then use either the **Browse…** button, which displays a file dialog box, or the special pull-down menu, which contains a list of special paths, to select the directory your G development environment searches. Edit or enter this path in the string control next to these options. Finally, use **Insert Before**, **Insert After**, or **Replace** to add that to the list.

When you select a path, G normally searches that directory, but not its subdirectories. Make the search hierarchical by appending a * as a new path item.

Some examples on different platforms.

- **(Windows)** To search the directory `C:\VIs\` recursively, you enter `C:\VIs\*`

- **(Macintosh)** To search the `VIs` folder recursively on the HD disk, you enter `HD:VIs:*`

- **(UNIX)** To search recursively `/usr/home/gregg/vis`, you enter `/usr/home/gregg/vis/*`.

Use the special pull-down menu shown in the following illustration, located to the right of the **Browse…** button, to select from several special directories.

```
✓ <vilib>
  <vilib>\*
  <topvi>
  <topvi>\*
  <foundvi>
  <foundvi>\*
```

These directories are as follows.

- <vilib> as a path entry is a symbolic path referring to the vi.lib directory inside of the **Library Directory**.

- <topvi> refers to the directory containing the top-level VI being opened.

- <foundvi> refers to a list of directories LabVIEW builds each time a VI is loaded. During a search, any directory in which LabVIEW finds a subVI, or a directory you manually select, is added to this list. Use this symbolic path so that if you move or rename a directory of VIs, and then open a calling VI, you must find that directory only once for that load.

Remove a path using the **Remove** button, which places the removed path into the string box, in case you decide to re-insert it elsewhere in the list.

☞ **Note**   *Changes to* **VI Search Path** *take effect upon the next launch of the application.*

Performance and Disk Preferences

Some editing operations require the use of temporary files on disk. For example, when you save a VI, it is first saved into a temporary file. If the save is successful, the new file replaces the original. This might require a fair amount of disk space. To avoid problems, the available disk space in the temporary directory is checked at launch time. If you have less than 500 KB of disk space free, you are notified and the launch is aborted. If you have less than 2 MB available, you are notified, but the launch continues. You can turn these warnings off or change the levels at which they alert you using the Performance and Disk dialog box, accessed from the Preferences pull-down menu.

The Performance and Disk Preferences dialog box is shown in the following illustration. Notice only one menu item is available on Windows 3.1, and some other options are available only on the Macintosh.
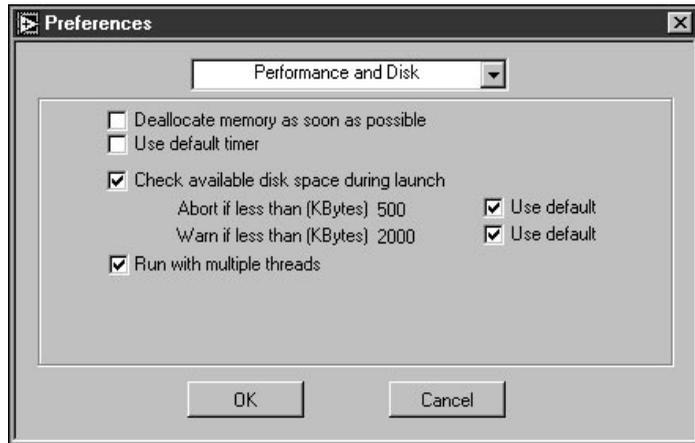
The options in this dialog box are as follows.

Deallocate memory as soon as possible — Deallocates the memory of every VI after it completes execution. Doing this can improve memory usage in some applications because subVIs deallocate their memory immediately after executing. However, it slows performance because G must allocate and deallocate memory more frequently and in some instances it might lead to excessive memory fragmentation.

(Windows 3.1) Use default timer — The timing functions in G use the built-in Windows timer. This timer has, by default, a resolution of 55 ms (it increments 18 times a second). This means the Tick Count, Wait, and Wait Until Next ms Multiple VIs in G only have 55 ms resolution. Notice, however, you can specify more accurate timing rates to the DAQ VIs, because they use the built-in timers of the DAQ boards to control the acquisition rate.

Increase the resolution of the Tick Count, Wait, and Wait Until Next ms Multiple VIs to 1 ms by increasing the resolution of the built-in timer of Windows to 1 ms. If you increase the timer resolution, your software timing loops are more accurate.

Under certain circumstances, you might not want to increase the timer resolution from 55 ms. Changing the resolution increases the number of timer tick interrupts from 18 interrupts per second to 1,000 interrupts per second. If you are performing other interrupt intensive operations, you

might exceed the capacity of your PC to handle the interrupt load, which might lock up or crash your PC.

DAQ operations using programmed I/O instead of DMA to transfer data between the DAQ board and PC memory are interrupt intensive. Do not change the timer tick resolution to 1 ms if you plan to perform any of the following operations at high transfer rates.

- Buffered analog input using the PC-LPM-16

- Buffered analog output using the AT-MIO-16, MC-MIO-16, or Lab series boards

- Buffered digital I/O using the DIO-24, DIO-96, or Lab series boards

- Buffered analog input or output with DMA disabled in WDAQCONF using any DAQ board

Transfer rates of approximately 30 ksamples/sec or higher on a 386 25 MHz machine causes interrupt load problems with the 1 ms timer resolution. Transfer rates of approximately 75 to 80 ksamples/sec or higher on a 486 33 MHz machine, or 100 ksamples/sec or higher on a 486 50 MHz machine, might cause interrupt load problems.

The timing functions in G for Windows 95/NT have a timer resolution of 1 ms, the maximum resolution possible under Windows 95/NT. Slower machines might have lower resolution.

(Macintosh) Compact memory during execution—Directs G to compact memory approximately once every 30 seconds. Compaction reduces fragmentation by moving allocated memory to a single location. A disadvantage to this is compaction takes time and might cause a short lull in performance while G is compacting memory.

(Macintosh) Cooperation level—With this option, you can control how cooperative G is with other background applications. If you set it to **Low**, G does not share time very frequently with other applications. Setting it low improves the performance of G, but it causes the performance of any other applications that run in parallel to decrease. If you set it to **High**, G performance suffers, but other applications have more time to execute.

Check available disk space during launch—Checks amount of disk space available in the temporary directory as G launches. You can check whether you want G to use the default in two cases: abort if less than 500 KB, or warn if less than 2,000 KB.

Run with multiple threads—This option is only available on operating systems that support multithreading. If you turn this off, the execution

system behaves as though you only have a single thread, so both User
Interface interaction and execution of VIs happen in the same thread.
You might use this for compatibility reasons if you have VIs that are not
well-behaved in the multithreaded execution system.

☞ **Note**   *Changes to* **Check available disk space during launch, Use default timer and
Run with multiple threads** *take effect when you relaunch G. Changes to the*
Compact memory during execution*,* **Deallocate memory as soon as possible** *and*
Cooperation level *options take effect immediately.*

Front Panel Preferences

The Front Panel Preferences dialog box is shown in the following
illustration.
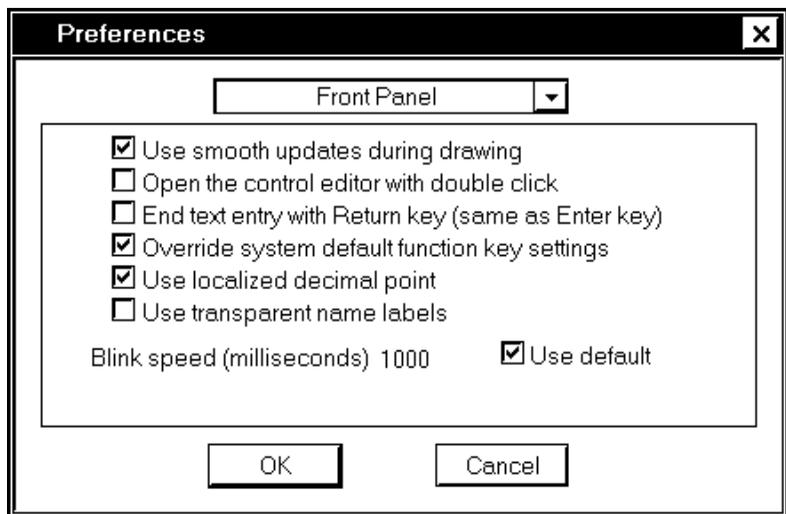
The options in this dialog box are as follows.

Open the control editor with double click—Makes easy access available
to the Control Editor window, where you can customize the appearance of
a front panel control or indicator.

End text entry with Return key (same as Enter key)—Makes the
<Return> key on the alphanumeric keyboard function like the <Enter>
key on the numeric keypad, ending text entry. If you select this option, you
can embed new lines by pressing <Ctrl-Enter> **(Windows)**;
<option-return> **(Macintosh)**; <meta-Return> **(Sun)**; or
<Alt-Return> **(HP-UX)**.

(Windows, Macintosh) Override system default function key settings—
By default, your operating system might use some function keys for system
purposes. For example, on the PC, pressing F10 is the same as pressing the
`<Break>` key. The Macintosh usually treats F1 to F4 as **Cut**, **Copy**, **Paste**,
and **Clear**. If you turn this override on, the function keys are not used for
system purposes but are passed instead to G as standard function keys.

Use localized decimal point—Uses the decimal separator for the system
instead of the period. For example, in many countries, the comma is used
as a decimal point. Turn this on if you want G to operate within your system
configuration. Turn it off if you want G to use periods in all cases for the
decimal point.

Use smooth updates during drawing—When G updates a control with
smooth updates off, it erases the contents of the control and draws the new
value. This can result in a noticeable flicker as the old value is erased and
replaced. Using smooth updates, G 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 requires more application memory because an offscreen
drawing buffer must be maintained.

Use transparent name labels—Directs the software to use see-through
labels.

(Sun) Support numeric keypad on Sun keyboards—Turns on support for
the Sun keyboard, including keypads, arrow keys, and the Help key. Do not
turn this on if you are using a non-Sun keyboard (for example, if you are
using an X terminal or a PC running X software).

Blink speed—Sets the blinking speed for front panel objects, to the default
1,000 milliseconds or another speed you type in. Blinking is a basic
attribute turned on with attribute nodes. See Chapter 22, *Attribute Nodes*,
for more information.

☞ **Note**   *Changes to options in the Front Panel Preferences dialog box take effect
immediately.*

Block Diagram Preferences

The Block Diagram Preferences dialog box is shown in the following
illustration.
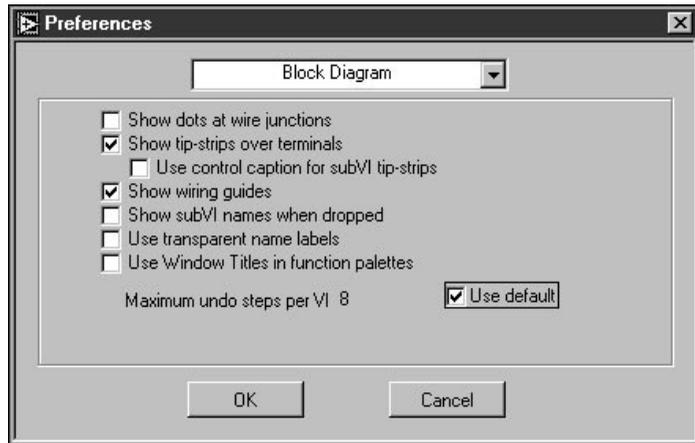
The options in this dialog box are as follows.

Show dots at wire junctions—Places dots where a wire branches to more
than one destination. This makes it easier to differentiate between such
junctions and wires that simply cross one another.

Show tip-strips over terminals—Places parameter names under
terminals when you idle your cursor over them in functions and subVIs.

> **Use control caption for sub-VI tip-strips**—Uses control
> captions instead of control names when showing help for VIs or
> when displaying tip-strips for VI terminals.

Show wiring guides—Shows wire stubs indicating data type and data
direction for each parameter when the cursor idles over a block diagram
node.

Show subVI names when dropped—Directs the software to label a
subVI with the name of the original VI when you drop it on the block
diagram, and to show the label.

Use transparent name labels—Directs the software to use see-through
labels.

Use Window Titles in function palettes—In the function palettes, uses the window title of a VI instead of the name of a VI when displaying the palette name of a VI.

Maximum undo steps per VI—Set the number of actions that you can undo and redo. If the number of actions is zero, undo is disabled. Each VI records its own undo instances, therefore the number you enter pertains to each VI. Setting this number lower conserves memory resources. See the *Undo* section in this chapter for more information.

☞ **Note**   *Changes to options in the Block Diagram Preferences dialog box take effect immediately.*

Debugging Preferences

The Debugging Preferences dialog box is shown in the following illustration.
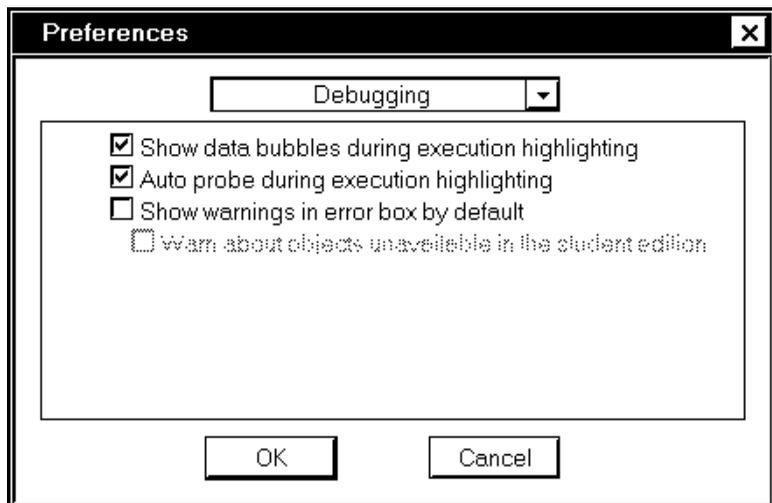
The options in this dialog box are as follows.

Show data bubbles during execution highlighting—Animates execution flow by drawing bubbles along the wires. Turn this feature off for faster debugging with less animation.

Auto probe during execution highlighting—Probes scalar values automatically, drawing their values on the diagram. Turn off this feature if you find it clutters the display.

Show warnings in error box by default—Displays warnings in addition
to errors in the Error List dialog box. A warning does not mean the VI is
incorrect; it just points out a potential problem in your block diagram.

Warn about objects unavailable in the student edition—If you select
the previous menu item about warnings, this item includes warnings about
objects unavailable in the LabVIEW Student Edition.

☞ **Note**   *Changes to options in the Debugging Preferences dialog box take effect
immediately.*

Color Preferences

The Color Preferences dialog box is shown in the following illustration.
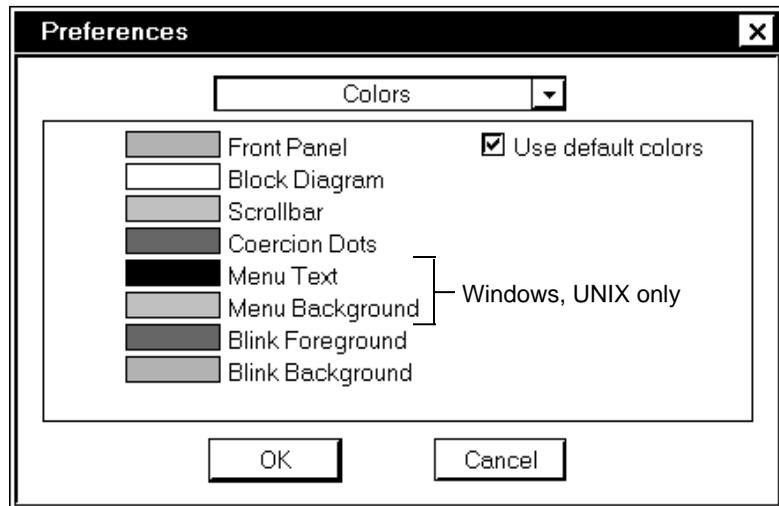
This dialog box lets you change the colors used by G for various items.
If you do not select the **Use default colors** checkbox, you can click any
rectangle to change the color. The options are the following.

Front Panel—Select a color for the front panel of new VIs. Changing the
color does not affect old VIs.

Block Diagram—Select a color for the block diagram of new VIs.
Changing the color does not affect old VIs.

Scrollbar—Select a color for scrollbars. This affects only the VIs
currently open.

Coercion Dots—Select a color for the dots indicating coercion of numerical data. This affects only the VIs currently open.

(Windows, UNIX) Menu Text—Select a color for text used in menus.

(Windows, UNIX) Menu Background—Lets you select a color for the background used in menus.

Blink Foreground—Lets you select the foreground color for a blinking object. Affects only a blinking object in its blink state. Blinking is a basic attribute turned on with attribute nodes. See Chapter 22, *Attribute Nodes*, for more information.

Blink Background—Lets you select the background color for a blinking object. Affects only a blinking object in its blink state.

Use default colors—Uses the default choice of colors for all items listed in this dialog box. You must turn this off if you wish to change any of these colors.

(UNIX) Provide extra colors—Toggles between a 216-color palette and a 125-color palette. A larger palette might cause a noticeable flash when you switch between windows if your system only supports 256 colors.

☞ **Note**   *Changes to options in the Color Preferences dialog box take effect immediately.*

Font Preferences

The Font Preferences dialog box lets you change three categories of predefined fonts: the Application font, System font, and Dialog font. You select the category of font you wish to change from the pop-up menu above the text box. The dialog box with **Application Font** selected is shown in the following illustration.
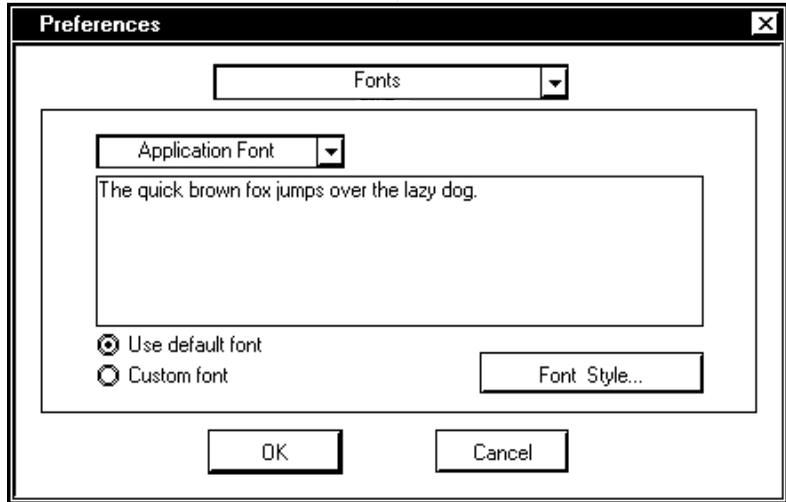
G uses the three categories of fonts for specific portions of its interface. The
fonts are predefined according to the platform, as follows.

Use default font—Uses the default font, as defined by G in keeping with
the different platforms, as described previously.

Custom font—This checkbox is checked automatically if you change any
font characteristics through the **Font Style…** option.

Font Style…—Brings up a dialog box to change font characteristics.

☞ **Note**   *Changes to options in the Font Preferences dialog box do not take effect
                  immediately. The software must be restarted.*

For specific information on how the fonts are predefined or for information
about the portability of fonts, see the section *Resolution and Font
Differences* in Chapter 29, *Portability and Localization Issues*.

Printing Preferences

The Printing Preferences dialog box is shown in the following illustration. Keep in mind some options are available on certain platforms only.
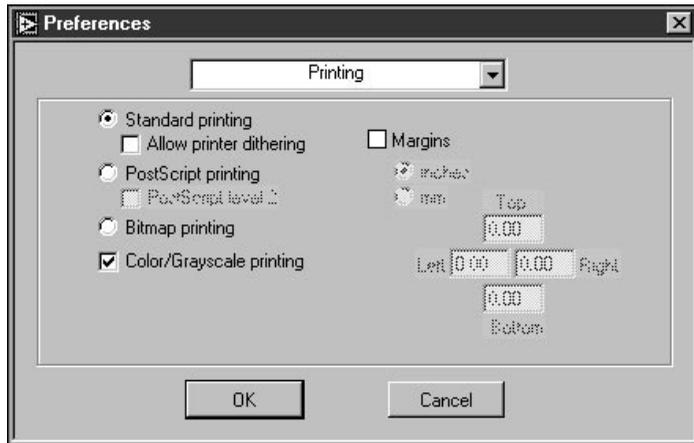
(Windows, Macintosh) Standard printing—Formats the VI print data (front panel, diagram, icon, etc.) and print it using standard drawing commands. Use this if you want to print to a file (assuming your printer driver supports printing to a file), if your printer does not have PostScript support, or if you want your printer driver to do the PostScript translation instead of the G-language software.

(Windows only) Allow printer dithering—Uses printer dithering when creating the drawing commands. Applicable for black-and-white printing, printer dithering is a method of adding gray scales to an image instead of only black-and-white areas. This menu item is not dependent on printer type.

PostScript printing—Translates the VI print data into PostScript (.ps) format and sends it to the printer as PostScript data. If you have a PostScript printer and a PostScript printer driver, the printout is a graphics image. Do not select this menu item if your printer or printer driver does not support PostScript printing.

PostScript level 2—Tells the software the connected printer supports **PostScript level 2**. If this box is checked, G sends **PostScript level 2** code to the printer. Select this only if your printer supports **PostScript level 2**.

(Windows, UNIX) Bitmap printing—Creates a bitmap, draw all data for that page into the bitmap, and then send the bitmap to the printer. This method

takes longer to print than the other two, but can yield a more accurate representation of the text and fonts than standard printing, even though the printout is not as high resolution as with PostScript printing. The bitmap image is in color for color printers and black-and-white otherwise.

(Windows only) Color/Grayscale printing—This menu item generates 8 bits-per-pixel (BPP) color/grayscale output to a printer, overriding the depth settings obtained from the printer driver. Most printers correctly report the color capabilities, but certain laser printer drivers report a bit depth of 1, when the printer can actually handle 8 BPP.

Margins—You can set absolute margins for printouts in terms of millimeters or inches. All four margins (top, bottom, left, and right) can be specified separately. Margins are limited by the physical parameters of the printer. If you set margins smaller than the printer accepts, the actual margins are the minimum accepted by the printer. The margins setting in the reference dialog box affects newly created or converted VIs. If you want to set margins for a specific VI, use **VI Setup**.

☞ **Note**   *Changes to options in the Printing Preferences dialog box take effect immediately.*

History Preferences

As explained in Chapter 27, *Managing Your Applications*, each VI has a History window displaying the development history of theVI. With the History Preferences dialog box, you can choose the default settings for the History window of new VIs.

The History Preferences dialog box is shown in the following illustration.
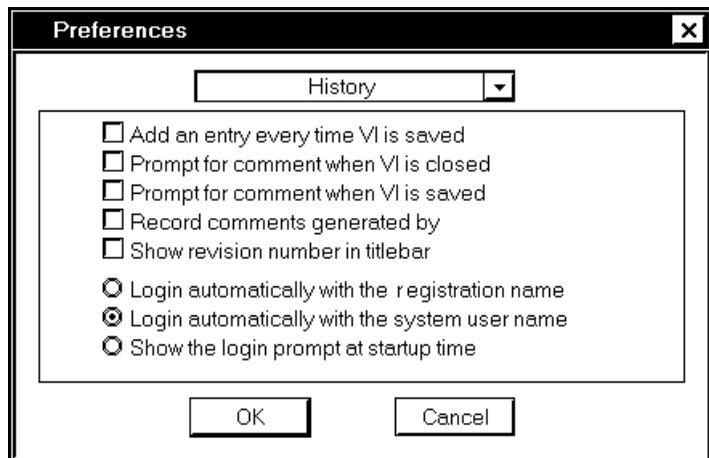
The options in the History dialog box are in two groups. The first group contains five options for indicating when and how entries to the History window are added for new VIs.

The second group of options in the History Preferences dialog box concerns how you log into LabVIEW or BridgeVIEW. This in turn determines the name inserted into the headers to comments entered in the VI History window.

Options concerning History window entries are specified for *individual* VIs through the **VI Setup»Documentation** dialog box. See *Documentation Options* in Chapter 6, *Setting up VIs and SubVIs*, for information about accessing the options.

The options in this dialog box are as follows.

Add an entry every time VI is saved—Adds an entry to the VI history every time you save the VI. If you do not enter a comment in the **Comment** box of the History window, only a header is added to the history. (The header contains the revision number if the menu item further down in this dialog box is checked as well as the date and time, and the name of the VI.)

Prompt for comment when VI is saved—Brings up the History window for comments on changes sine the previous save. This is useful if you prefer to comment on your changes when you finish making them instead of as you are editing. If you do not set this option, you do not have a chance to change the history of the VI after you select **Save** until the save is finished.

Prompt for comment when VI is closed—Similar to the previous option, except that G only prompts you when you close a VI, rather than every time you save. You are prompted if the VI changed any time since you loaded it, even if you saved the changes.

☞ **Note**   *You are not prompted to enter a comment when you save or close a VI if the only changes you made were changes to the history.*

Record comments generated by the editor—Causes the editor to insert comments into the History window when certain events occur. The events causing automatic comments are conversion to a new version of LabVIEW or BridgeVIEW, subVI changes, and changes to the name or path of the VI.

Show revision number in titlebar—Adds the revision number to the header of the History window. The revision number starts at zero and increases incrementally every time you save the VI. It does not increase however, if the only changes you made were changes to the VI history.

The login options are as follows.

(Windows and Macintosh only) Login automatically with the registration name—Uses the name of the registered user of the application.

Show the login prompt at startup time—Prompts you for a user name when your application is started. The user name can be changed any time by selecting **Edit**»**User Name**. The prompt displays a menu of all user names previously entered into LabVIEW with this option.

☞ **Note**   *This is a LabVIEW-only feature. BridgeVIEW has its own login security system in which* **Edit**»**User Name** *does not appear.*
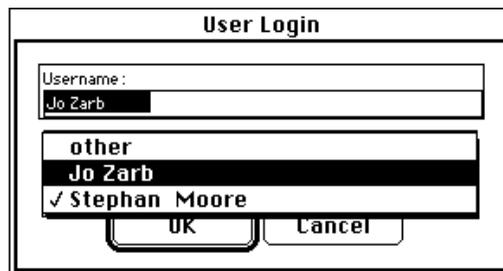
You can switch to another user name (the system user name or your application registration name) in Windows or on the Macintosh, or the system user name in UNIX, without having to type it in by using the pull-down menu shown in the preceding illustration. You also can type in a name, which appears in the list thereafter.

(Windows NT, Macintosh with file sharing only, and UNIX) Login automatically with the system user name—Assuming a user name is defined, this causes the software to use the system login name of the current user.

☞ **Note**   *Changes to options in the History Preferences dialog box take effect immediately.*

Time and Date Preferences

The Time and Date Preferences dialog box is shown in the following illustration.
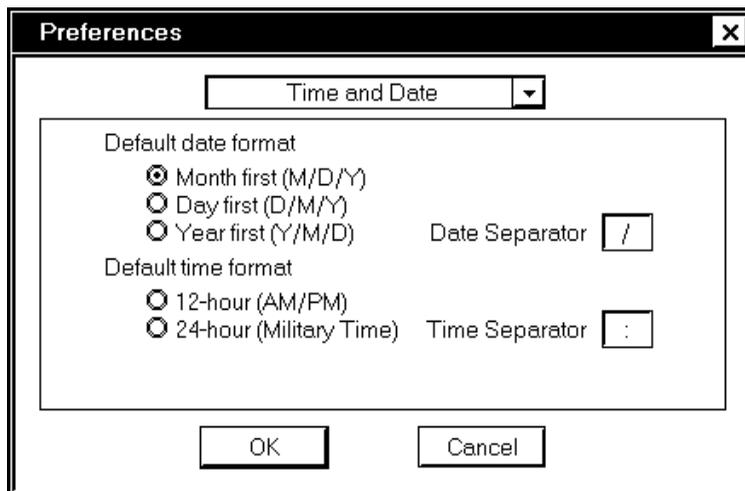
The options in this dialog box control the default displays of time and date in the digital displays of new controls and indicators. Override the defaults in this dialog box for individual digital displays, as explained in the *Format and Precision of Digital Displays* section of Chapter 9, *Numeric Controls and Indicators*.

The options in this dialog box are as follows.

Default date format—Determines whether the month, day, or year comes first in digital displays.

Default time format—Determines whether time in digital displays is based on a 12-hour or 24-hour clock.

Date Separator—Determines the character used to separate the month, day, and year (in the order selected in the Default date format option) of digital displays.

Time Separator—Determines the character used to separate the hours and minutes past the hour in digital displays.

☞ **Note**   *Changes to options in the Time and Date Preferences dialog box take effect immediately.*

Miscellaneous Preferences

The Miscellaneous Preferences dialog box is shown in the following
illustration.
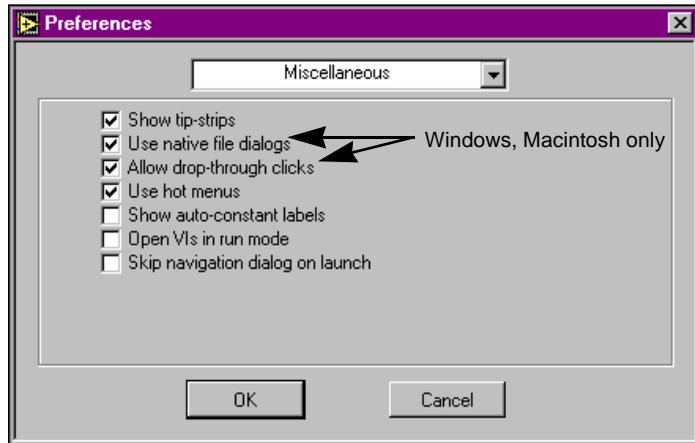
The options in this dialog box are as follows.

Show tip-strips—Toggles the display of tip strips.

(Windows, Macintosh) Use native file dialogs—Uses the operating system's
native file dialog boxes so they look similar to those of other applications
on your machine (with a few exceptions discussed at in the next paragraph).
When this preference is not selected, the G-language software uses its own
platform-independent file dialog boxes, which add some convenient
features, such as providing a list of recent paths and reducing the steps
necessary to save VIs in VI libraries.

(Windows, Macintosh) Allow drop-through clicks—You can set G so that
clicking the mouse only once activates and selects an object in an inactive
window. This reduces the number of required mouse clicks from two to
one. (Normally, the first click activates the window and the second click
passes through as a click in the window.)

Use hot menus—With this option, you can navigate the menus using the
mouse without keeping the left mouse button pressed. This ergonomic
feature relieves strain on the hand.

Show auto-constant labels—This toggles the creation of labels for
auto-created constants. When this is set, labels automatically show the
name of the terminal from which the constant was created.

Open VIs in run mode—Opens VIs in run mode rather than edit mode.

Skip navigation dialog on launch—Skip the initial window that permits navigation to common dialogs and simply open the Untitled VI.

☞ **Note**   *Changes to options in the Miscellaneous Preferences dialog box take effect immediately.*

Server: Configuration

The Server: Configuration dialog box is shown in the following illustration.
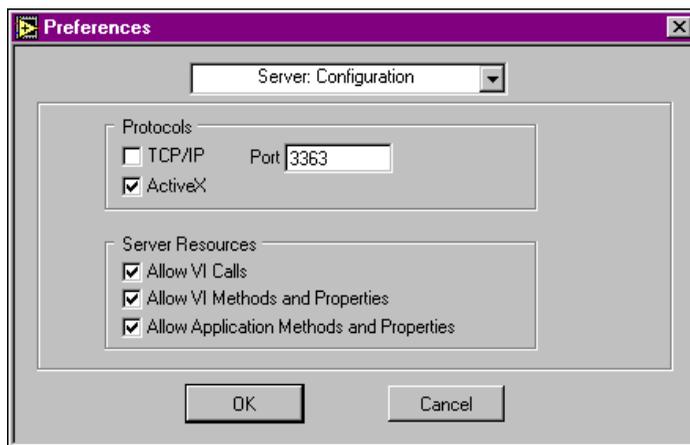
These options specify through which communication protocols other applications access the VI Server, either TCP/IP or ActiveX protocols. If you enable **TCP/IP**, you must enter the **Port** number that client applications use to connect to the server. When you permit other applications to connect using TCP/IP, it is best to configure which Internet hosts have access to the server. For more details see the section *Server: TCP/IP Access* in this chapter. For more information about the VI server ActiveX interface, refer to Chapter 21, *VI Server*.

With **Server: Configuration** selected, you also specify which of the following server resources are available to applications accessing the VI Server.

- **Allow VI Calls** permits applications to make calls to VIs exported on the server. When you permit other applications access to VIs, it is best to specify which VIs are exported using the Exported VI Preference page. For more information see the section *Server: Exported VIs*, in this chapter.

- **Allow VI Methods and Properties** permits applications to read and set the properties of VIs on the server. As above, you can specify which VI in the Exported VI Preference page. For more information see the section *Server: Exported VIs*, in this chapter.

- **Allow Application Methods and Properties** permits applications to read and set the properties of the server.

The default server settings are **ActiveX** enabled and **TCP/IP** disabled. By default, **VI Calls** is enabled, but **VI Methods and Properties** and **Application Methods and Properties** are disabled.

Server: TCP/IP Access

When you permit remote applications to access the VI Server using the TCP/IP protocol, it is best to specify which Internet hosts have access to the server.

The **Server: TCP/IP Access** dialog box is shown in the following illustration.
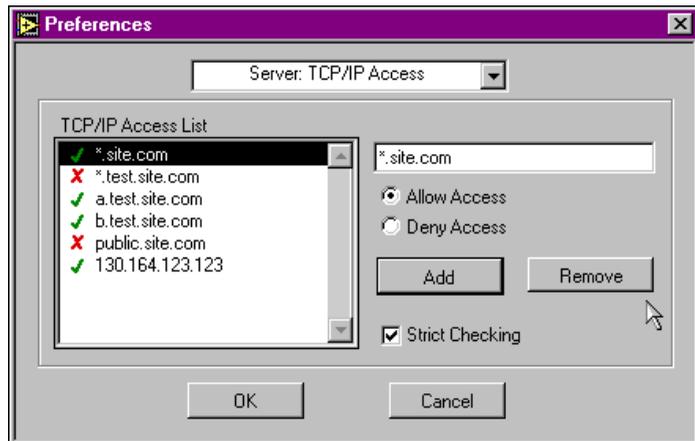
Using **Server: TCP/IP Access** you specify which clients access the VI Server. The TCP/IP Access List describes clients that have access or are denied access to the server. To change an entry, select it from the list and type in the text box at the right of the TCP/IP Access List. Click the **Allow**

Access radio button to permit client access to the server. Click the **Deny Access** radio button to deny the client access to the server. Click the **Add** button to insert a new entry after the current selection. Click the **Remove** button to remove the current selection from the list. Click and drag an entry to change its position within the TCP/IP Access List. If an entry permits access from an address, a check mark appears next to the entry. If an entry denies access from an address, an *X* appears next to the entry. If no symbol appears next to the entry, the syntax of the entry is incorrect.

When a client tries to open a connection to the server, the server examines the entries in the TCP/IP Access List to determine if the client is permitted access. If an entry in the list matches the client address, the server either permits or denies access, based on how you set up the entry. If a subsequent entry also matches the client address, that permission is used in place of the previous permission. For example, in the illustration above, `a.test.site.com` and `b.test.site.com` are permitted access even though the list indicates (by the `*` wildcard) that all addresses ending in `.test.site.com` are not permitted access. If no entry matches the client address, access is denied. See Table 7.1 for more information on the `*` wildcard and permitting matching access entries.

An Internet (IP) address, such as `130.164.123.123`, might have one or more associated domain names, such as `www.natinst.com`. The conversion from a domain name to the corresponding IP address is called *name resolution*. The conversion from an IP address to its domain name is called *name lookup*.

Name lookups and name resolutions are done through system calls that access domain name system (DNS) servers on the Internet. A name lookup or resolution can fail when the system does not have access to a DNS server, or when the address or name is not valid. A resolution problem occurs when an entry contains a domain name that cannot be resolved into an IP address. A lookup problem occurs when an entry contains a partial domain name, such as `*.natinst.com`, and the lookup for the client IP address fails.

Strict Checking determines how the server treats access list entries that cannot be compared to a client IP address because of resolution or lookup problems. When Strict Checking is enabled, a denying access list entry in the TCP/IP Access List encountering a resolution problem is treated as if it matched the client's IP address. When **Strict Checking** is disabled, an access list entry encountering a resolution problem is ignored.

To specify an Internet host address, enter its domain name or IP address. Use the `*` wildcard when specifying Internet host addresses. For example,

you can specify all hosts within the domain domain.com with the entry
*.domain.com. You can specify all hosts in the subnet whose first two
octets are 130.164 with the entry 130.164.*. The entry * matches all
addresses.

The following table shows examples of TCP/IP Access List entries.

Table 7-1. Server: TCP/IP Access

Access String	**Matches**
*	All hosts
test.site.com	The host whose domain name is test.site.com
*.site.com	All hosts whose domain names end with *.site.com
130.164.123.123	The host with the IP address 130.164.123.123
130.164.123.*	All hosts whose IP addresses start with 130.164.123.

In the previous illustration, all hosts in the site.com domain have access
to the server, with the exception of all hosts in the test.site.com
domain. Additionally, the hosts a.test.site.com, b.test.site.com
and 130.164.123.123 also have access to the server. The host
public.site.com does not have access, even though it is in the
site.com domain.

The default TCP/IP Access settings permit access only to clients on the
server machine.

☞ **Note**   *If the VI Server runs on a system that does not have access to a DNS server, do not
use domain name entries in the TCP/IP Access list. Requests to resolve the domain
name or an IP address fail, slowing down the system. For performance reasons,
place frequently matched entries toward the bottom of the TCP/IP Access List.*

Server: Exported VIs

If you permit remote applications to access VIs on the VI Server, it is best to specify which VIs these applications can access.

The **Server: Exported VIs** dialog box is shown in the following illustration.
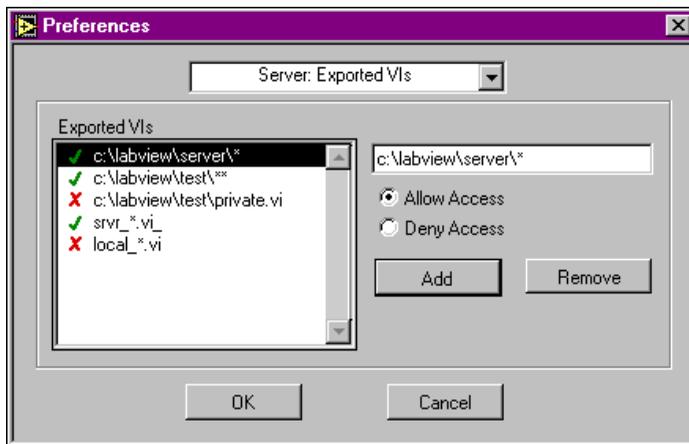
The **Server: Exported VIs** menu items let you specify which VIs other applications can access through the VI Server. The Exported VIs list specifies which VIs are exported. To change an entry, select it from the list, then type into the text box at the right of the Exported VIs list. To specify whether remote computers can or cannot access that VI, click the **Allow Access** or **Deny Access** radio buttons. Click the **Add** button to insert a new entry after the current selection. Click the **Remove** button to delete the current selection. Click and drag an entry to change its position within the Exported VIs list. If an entry permits access to VIs, a check mark appears next to the entry. If an entry denies access to VIs, an **X** appears next to the entry. If no symbol appears next to the entry, the syntax of the entry is incorrect.

Each entry in the list describes a VI name or a VI path and might contain wildcard characters. Entries that contain path separators are compared against VI paths, while entries that do not contain path separators are compared against VI names only. When a remote client tries to access a VI, the server examines the Exported VIs list to determine whether to grant access to the requested VI. If an entry in the list matches the requested VI, the server either permits or denies access to that VI, based on how that entry is set up. If a subsequent entry also matches the VI, its access permission is

used in place of the previous permission. If there is not a VI in the list that matches the requested VI, access to the VI is denied.

You can use wildcard characters in the Exported VIs list so an entry in the list matches more than one VI. Use the following wildcard characters.

'?'	matches exactly one arbitrary character, except for the path separator
'*'	matches zero or more arbitrary characters, except for the path separator
'**'	matches zero or more arbitrary characters, including the path separator

If you want to match a VI with a name that contains a wildcard character, you must escape that character using '\' (**Macintosh** and **UNIX**), or '`' (**Windows**).

The following table shows examples of Exported VI list entries. The examples use UNIX path separators.

Table 7-2.   Server: Exported VI List Entries

VI Access String	Matches
`*`	All VIs
`/usr/labview/*`	All VIs in the directory `/usr/labview/`.
`/usr/labview/**`	All VIs in the directory `/usr/labview/` and any of its sub-directories.
`test.vi`	Any VI named `Test.vi`.
`OK\?`	Any VI with the name `OK?`

In the previous illustration, all VIs in the `c:\labview\server` directory are exported. All VIs in the `c:\labview\test` directory and all its sub-directories are exported as well, with the exception of the VI `c:\labview\test\private.vi`. Additionally, any VI that begins with the string `srvr_` and ends with the string `.vi` is exported. No VI that

begins with the string `local_` and ends with the string `.vi` is exported, even if it is located within the `c:\labview\server` directory.

The default Exported VIs settings permit access to all VIs.

How Preferences Are Stored

Usually, it is not necessary to edit preference information manually or know its exact format, because the Preferences dialog box takes care of it for you. Preferences are stored differently on each platform, as described in the following paragraphs.

(Windows) If you are a LabVIEW user, Preference information stores in a `LabVIEW.INI` file in your `LabVIEW` directory, if you are a LabVIEW user. The file format 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 `offscreenUpdates=True`. If a configuration value is not defined in `LABVIEW.INI`, LabVIEW checks to see if there is a `[LabVIEW]` section of your `WIN.INI` file, and if so, checks for the configuration value in that file.

If you are a BridgeVIEW user, preference information stores in a `BridgeVIEW.INI` file in your `BridgeVIEW` directory. The format for this file is similar to other `.INI` files. It begins with a section marker `[BridgeVIEW]`.

☞ **Note**   *The references to `WIN.INI` files applies only to Windows 3.1. If you are a BridgeVIEW user, this information does not apply.*

You also can specify a preference file on the command line when you start LabVIEW or BridgeVIEW. For example, to use a file named `lvrc` instead of `labview.ini`, type `labview -pref lvrc`.

(Macintosh) By default LabVIEW creates a text file in the Preferences folder of the System Folder called `LabVIEW Preferences`.

If you want, copy your LabVIEW Preferences file to the LabVIEW folder. When launched, LabVIEW always looks for the Preferences file in the LabVIEW folder. If not found there, it looks in the Preferences folder, and if not found there, it creates a new one in the Preferences folder. By moving Preferences files between desktop and the LabVIEW folder, you can create multiple Preferences for multiple users or uses.

(UNIX) Preference information normally is stored in a text file in your home directory named `.labviewrc`. If you change a parameter from the Preferences dialog box, G writes the information to this file. The following is for those who want more information on the storage format and rules for finding preference information.

Preference entries consist of a *preference name* followed by a colon and a value. The preference name is the executable name followed by a period (.) and a token. When LabVIEW searches for preference names, the search is case-sensitive. You have the option to enclose the preference value in double or single quotation marks. For example, to use a default precision of double and use a search path that recursively searches a particular directory, specify the following preference entries.

```
labview.defPrecision : double

labview.viSearchPath : "/usr/lib/labview/*"
```

LabVIEW also searches for preferences in a file named `labviewrc` in the applications directory. For example, if you installed your LabVIEW files in `/opt/labview`, preferences are read out of both `/opt/labview/labviewrc` and the `.labviewrc` file in your home directory.

Entries in the `.labviewrc` file override conflicting entries in the `labviewrc` file. You might want to use this global preference file to store things that are the same to all users such as the VI search path.

You also can specify a preference file on the command line when you start LabVIEW. For example, to use a file named `lvrc`, type `labview -pref lvrc`.

⚠ **Caution**   *The* `.labviewrc` *file is still read in this case. Also, keep in mind changes made in the Preferences dialog box are written to the* `lvrc` *file in this example.*

Undo

Using **Edit»Undo** and **Edit»Redo,** you can undo an action immediately after it is performed, and once you undo an action you can redo it. The key command for **Undo** is `<Ctrl-Z>` (or `<Cmd-Z>` on the Macintosh) and the key command for **Redo** is `<Ctrl-Shift-Z>` (or `<Cmd-Shift-Z>` on the Macintosh). The items in the **Edit** menu, **Undo** and **Redo**, each contain a brief (one- or two-word) description of the action that can be undone or

redone. In the following illustration if you select **Redo Resize**, the item returns to its original size.

```
Edit  Operate  Project  Windows
  Undo Move              Ctrl+Z
  Redo Resize        Ctrl+Shift+Z
```
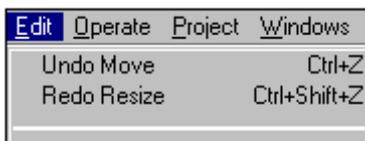
Set the number of actions that you can undo or redo in **Preferences»Block Diagram»Maximum undo steps per VI**. If the number of actions is zero, **Undo** and **Redo** are disabled. See the section *Preferences Dialog Box Options*, *Block Diagram Preferences*, in this chapter for more information on setting the maximum undo steps for a VI.

Each VI records its own undo instances, therefore the number you enter for **Maximum undo steps per VI** pertains to each VI. Setting this number low can conserve memory resources.

VI callers automatically update when the VI changes its interface. In this situation, all undo information in the callers is thrown away. This occurs when you do any one of the following.

•   Change a VI connector pane pattern or change the data type of a control on the connector pane.

•   Change a type definition appearance (only if it is strict), change the data type or strictness, or change a type definition to a custom control.

•   Change the data type or name of a control in a global VI, or add or remove a control to a global VI.

Customizing the Controls and Functions Palettes

G has a powerful set of tools for customizing the **Controls** and **Functions** palettes in any of the following ways.

•   Add your own VIs and controls to the palettes by saving them in the user.lib or instr.lib directories.

•   Add your own VIs and controls to the palettes by using the **Edit» Edit Control & Function Palettes** option.

•   Set up the palettes to change automatically as you add or remove files from specific directories.

•   Rearrange the built-in palettes to make frequently used functions more easily accessible.

- Set up different views for different users, hiding some functions to make G easier to use for one user while providing the full capabilities for another user.

- On Windows and the Macintosh, the **Function** palette is hidden when a front panel is the active window and the **Control** palette is hidden when a block diagram is the active window. If you prefer to have all palettes visible at all times, edit your Preferences file to add a line that sets the showAllPalettes preference to True.

Adding VIs and Controls to user.lib and instr.lib

The simplest method for adding new entries to the **Controls** and **Functions** palettes is to save them inside of the user.lib directory. When you restart G, the **User Libraries** subpalette of the **Functions** palette contains subpalettes for each directory, .llb, or .mnu file in user.lib and entries for each file in user.lib.

The **Instrument Drivers** palette of the **Functions** palette corresponds to the instr.lib directory. You might want to put instrument drivers in this directory to make them easily accessible from the palettes.

☞ **Note**   *The* vi.lib *directory contains files from National Instruments overwritten when you upgrade your software. Therefore, do not save your own files into* vi.lib*.*

Installing and Changing Views

Controls and **Functions** palette information is stored in the menus directory of your application directory. The menus directory contains directories corresponding to each view you create or install. If you are running off a network, you might want to define individual menus directories for each user. You can add a line to your preference file that sets the menusDir preference to an alternative path, one that is unique for each user preference file.

This organization makes it easy to transfer views to other people. To give someone else a copy of a view, give the person a copy of the directory of the view from the menus directory. After placing it in his or her menus directory, the view becomes available for the next launch.

To switch to another view, select **Edit»Edit Control & Function Palettes**. Then select the view you want from the **menu setup** ring.

To create your own view, see the description in the following *Palettes Editor* section. For more details on how views are organized, see the *How Views Work* section in this chapter.

Palettes Editor

For more control of the layout and contents of the **Controls** and **Functions** palettes, use the **Edit»Select Palette Set** option. When you select this option, you enter the Palettes Editor. The Edit Palettes dialog box appears.

In this editor, you can rearrange the contents of palettes by dragging objects to new locations. If you want to delete, customize, or insert objects, pop up on a palette as shown in the following illustration, or pop up on an object within a subpalette.
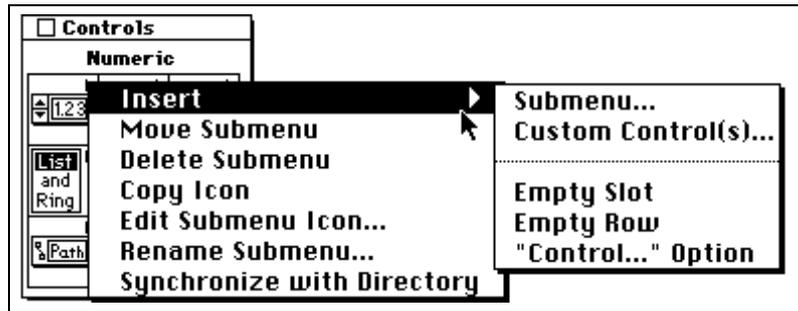
With the pop-up options, you can modify anything within the **User Libraries** menu (corresponds to `user.lib`) or the **Instrument I/O** menu (corresponds to `instr.lib`). If you want to edit the top-level **Controls** or **Functions** palettes or any other predefined menus, you must first create a new view by selecting **new setup…** from the **menu setup** ring in the Edit Palettes dialog box. After selecting a new setup, any menu you modify that is part of the default-menu set copies to the directory of your view in the `menus` directory before making changes. This protection of the built-in palettes ensures that you can experiment with the palettes without corrupting the default view.

If you want to add a new object in a new row or column of a subpalette, pop up in the space at the right edge or bottom of the subpalette. You also can create new rows or columns by dragging objects to the area to the right or bottom of the palette.

Creating Subpalettes

When you add a palette, you can move it to a new location, edit the subpalette icon, or rename the palette using the Palettes Editor.

If you want to create a palette from scratch or hook in a palette not in `user.lib` or `vi.lib`, you can use **Insert»Submenu…** from the pop-up

menu in the Palettes Editor. When you select this option, you see the
following dialog box.

Insert Submenu

- ◉ **Create a new menu file (.mnu)**
- ○ **Link to an existing menu file (.mnu)**
- ○ **Link to a library (.llb)**
- ○ **Link to a directory**

[**OK**]   [**Cancel**]

You can store submenu information in VI libraries or in .mnu files. A menu
file or `.LLB` file can contain one **Functions** palette and one **Controls**
palette.

Select **Create a new menu file** to insert a new, empty palette. You are
prompted for a palette name and a file to contain it. It is recommended you
add an `.mnu` extension to the file to indicate it is a menu (palette). In
addition, it is recommended you either store the .mnu file in the directory of
your view in the `menus` directory or in the same directory containing the
controls or VIs the menu largely represents.

Select **Link to an existing menu file** if you have an existing palette you
want to add to the Controls or Functions palette file.

Select **Link to a directory** if you want to create a palette with entries for
all the files in the directory. Selecting this also recursively creates
subpalettes for each of the subdirectories, VI libraries, or .mnu files within
the directory. These palettes automatically update if you add new files to or
remove files from the directories you selected. Enable or disable the
automatic update setting for a subpalette by selecting the **Automatic
Update From Directory** pop-up option. This menu item actually creates
menu (palette) files inside each subdirectory.

Select **Link to a library** if you want to link to the **Controls** palette or **Functions** palette that is a part of a VI library. As with the previous setting, the palette for a library automatically updates as you add files to the library.

Notice you can mix VIs, functions, and subpalettes within a palette freely. Also, a palette can contain VIs from different locations.

Moving Subpalettes

Because clicking a subpalette opens it, you cannot move a subpalette by dragging it. To move a subpalette, select the **Move Submenu** menu item from its pop-up menu. As a shortcut, hold down the `<Shift>` key while you click a subpalette to drag it instead of opening it.

How Views Work

Both `.mnu` files and VI library files are binary files that can contain one **Controls** palette and one **Functions** palette. In addition, both types of files contain an icon for the **Controls** and **Functions** palettes. Each submenu you create must store in a separate file.

When you select a view, G looks for a directory in the `menus` directory. It builds the top-level **Controls** and **Functions** palettes from the `root.mnu` file within that directory. The top-level palettes then link to other `.mnu` or VI library files for each subpalette within the palette.

If you link to a directory, you initiate a search to check if the directory contains a `dir.mnu` file. If so, the directory uses that `.mnu` file as the subpalette for the directory. Otherwise, your application creates a `dir.mnu` file based on the contents of the directory. For each VI (or control), an entry is created. For each subdirectory, `.mnu` file, or VI library, your application creates a subpalette.

G automatically updates the palettes within a VI library as you add files to or remove files from the VI library. You have the option to set an `.mnu` file to update based on the contents of a directory. To change this setting, select **Synchronize with Directory** from the pop-up menu of the subpalette icon. Although it is a good idea to keep the `.mnu` file in the same directory with which it is synchronized, you actually can select any directory. In addition to reflecting a specific directory, you can put additional VIs or controls from other directories in the same palette. You also can remove VIs from the palette by selecting **Delete Item** from the pop-up menu.

Part II

Front Panel Objects

This section contains information about front panel objects, controls and indicators and ActiveX controls.

Part II, *Front Panel Objects*, contains the following chapters.

- Chapter 8, *Introduction to Front Panel Objects*, introduces the front panel and its two components—controls and indicators. It also explains how to import graphics from other programs to use in your controls.

- Chapter 9, *Numeric Controls and Indicators,* describes how to create, operate, and configure the various styles of numeric controls and indicators.

- Chapter 10, *Boolean Controls and Indicators*, describes how to create, operate, and configure Boolean controls and indicators.

- Chapter 11, *String Controls and Indicators*, describes how to use string controls and indicators, and the table. You can access these objects through the **Controls»String & Table** palette.

- Chapter 12, *Path and Refnum Controls and Indicators*, describes how to use file path controls and refnums, which are available from the **Controls»Path & Refnum** palette.

- Chapter 13, *List and Ring Controls and Indicators*, describes the listbox and ring controls and indicators, which are available from the **Controls»List & Ring** palette.

- Chapter 14, *Array and Cluster Controls and Indicators*, describes how to use arrays and clusters. You access arrays and clusters from the **Controls»Array & Cluster** palette.

- Chapter 15, *Graph and Chart Controls and Indicators*, describes how to create and use the graph and chart indicators in the **Controls»Graph** palette.

- Chapter 16, *ActiveX Controls,* describes the ActiveX Container capability, which enhances the interactions between G-based software and other applications.

8

Introduction to Front Panel Objects

This chapter introduces the front panel and its two components — controls and indicators. It also explains how to import graphics from other programs to use in your controls.

Building the Front Panel

Controls and indicators on the front panel are the interactive input and output terminals of the VI. You use controls to supply data to a VI, and indicators display the data generated by the VI. This section explains a few editing options common to all controls and indicators.

The **Controls** palette on the front panel is shown in the following illustration.

If you idle your cursor over any square of the palette, the name of the collection of controls is displayed at the top of the palette window.

The types of controls in the palette, from left to right, top to bottom, are as follows.

- **Numeric controls**—For entering and displaying numeric quantities.

- **Boolean controls**—For entering and displaying TRUE/FALSE values.

- **String & Table controls**—For entering and displaying text.

- **List & Ring controls**—For displaying and/or selecting from a list of options.

- **Array & Cluster controls**—For grouping sets of data.

- **Graph controls**—For plotting numeric data in chart or graph form.

- **Path & Refnum controls**—For entering and displaying file paths and for passing refnums from one VI to another.

- **Decorations**—For adding graphics to customize front panels. These objects are for decoration only, and do not display data.

- **User Controls**—For automatically saving customized controls into `user.lib` file.

- **ActiveX**—For enhancing ActiveX support. These objects include the ActiveX Container and the ActiveX Variant.

- **Select a Control…**—For choosing a custom control of your own design.

Front Panel Control and Indicator Options

When you pop up on a control or indicator on the front panel while editing a VI, you see 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. Below the line, for example, **Representation**, **Data Range…**, and **Format & Precision…** are customized depending on the object.
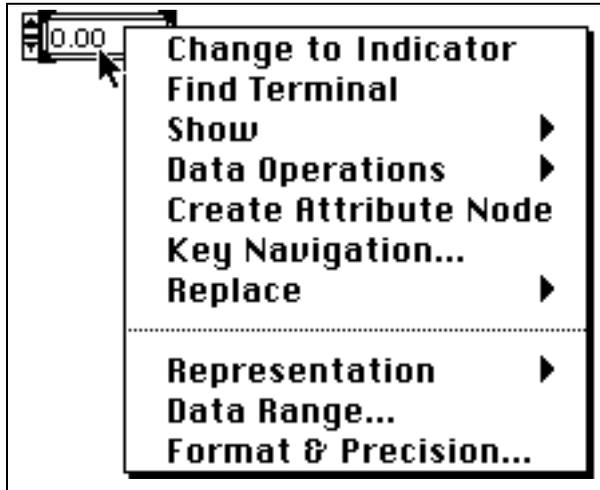
Objects in the **Controls** palette initially configure as controls or indicators based on how they are typically used. For example, if you choose a toggle switch from the **Boolean** palette, it appears on the front panel as a control, because a toggle switch is usually an input device. Conversely, 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 indicators to controls, by choosing the **Change to Control** or **Change to Indicator** commands from the pop-up menu of the object. 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** menu item on the control and indicator pop-up menus highlights the block diagram terminal for the control or indicator. This item is useful for identifying a particular object on a crowded block diagram.

The **Create Attribute Node** menu item creates an Attribute Node for the object. Use attribute nodes to control various properties of the object programmatically.

The **Show** submenu shows a list of the parts of a control you can choose to hide or show, such as the name label.

When editing a VI, the pop-up menu for a control contains a **Data Operations** submenu. Using items from this menu, you can cut, copy, or paste the contents of the control, set the control to its default value, make the current value of the control its new default, and read or change the control description. (When editing a VI, you can paste data into an

indicator.) Some of the more complex controls have additional options; for example, the array has options 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 the pop-up menu of a control available while the VI is running.

```
Reinitialize to Default
Make Current Value Default
Cut Data
Copy Data
Paste Data
Description...
Online Help
```
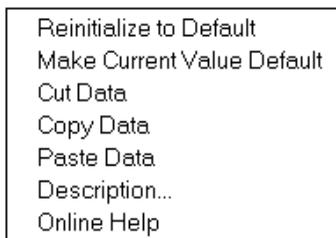
If you pop up on a control while a VI is running, you can change only the value of a control. While running a VI, you cannot change most characteristics of a control, such as its default value or description.

Replacing Controls

The **Replace** menu item of the pop-up menu of an object displays the **Controls** palette from which you can choose a control or indicator to take the place of the current item on the front panel.

Use **Replace** when you want to choose a different style of control, but do not want to lose all of the editing options you selected for the control up to that point. Selecting **Replace** from the pop-up menu preserves as much information as possible about the original control, such as its name, description, default data, dataflow direction (control or indicator), colors, size, and so on. However, the new control keeps its own data type. Wires from the terminal of the control or local variables remain connected on the block diagram.

The more similar the control is to the one being replaced, the more the original characteristics can be preserved. As an example, the following illustration shows a slide being replaced by a different style of slide.
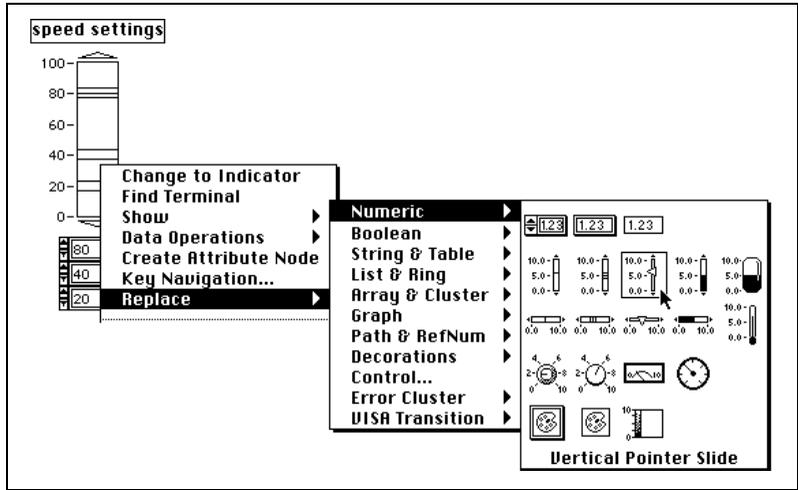
The next illustration shows the result.
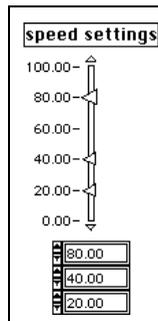
The new slide has the same height, scale, value, name, description, and so on.

If you were to pop up and replace the slide with a string instead, only the name, description, and data flow direction are preserved, because a slide does not have much in common with a string.

Another way to replace a control that does not use the **Replace** pop-up menu and does not preserve any characteristics of the old control involves copying the control to the Clipboard. This method does keep the connections on the block diagram and VI connector. Copy the new control to the Clipboard. Then select the old control you want to replace with the Positioning tool, and select **Edit»Paste**. The old control is discarded and the new control is pasted in its place.

Key Navigation Option for Controls

All front panel controls have a **Key Navigation…** menu item. You use this to associate a keyboard key combination with a control. When a user enters that key combination while running the VI, G acts as though the user had clicked on that control. The associated control becomes the key focus. If the control is a text control, any existing text within that control is highlighted, ready for editing. If the control is a Boolean control, the state of the button is toggled.

☞ **Note**   *The* **Key Navigation…** *menu item is disabled for indicators, because you cannot enter data into an indicator.*

You also can use **Key Navigation…** to indicate whether a control is included when the user tabs from control to control while running.

You can use **Key Navigation…** to associate function keys with various buttons that control the behavior of a panel. You can use it to define a default button for VIs that behave like a dialog box, so that pressing the <Enter> **(Windows and HP-UX)**, or <Return> **(Macintosh and Sun)** key becomes the same as clicking the default button.

If you associate the <Enter> or the <Return> key with a dialog box button, G automatically draws that button with a special thick border around it, as shown in the following illustration.
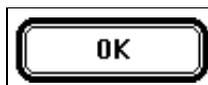
There are two important things to know about the <Enter> or the <Return> key.

1. If you associate the <Enter> or the <Return> key with a control, then no control on that front panel ever receives a carriage return. Consequently, all strings on the front panel are limited to a single line.

2. When you tab from control to control, buttons are treated specially. If you tab to a button and press the <Enter> or <Return> key, the button you tabbed to is toggled, even if another control has the <Enter> or <Return> key as its key navigation setting.

You cannot assign the same key combination to more than one control in a given front panel.

When you select the **Key Navigation…** option, G displays the following dialog box.
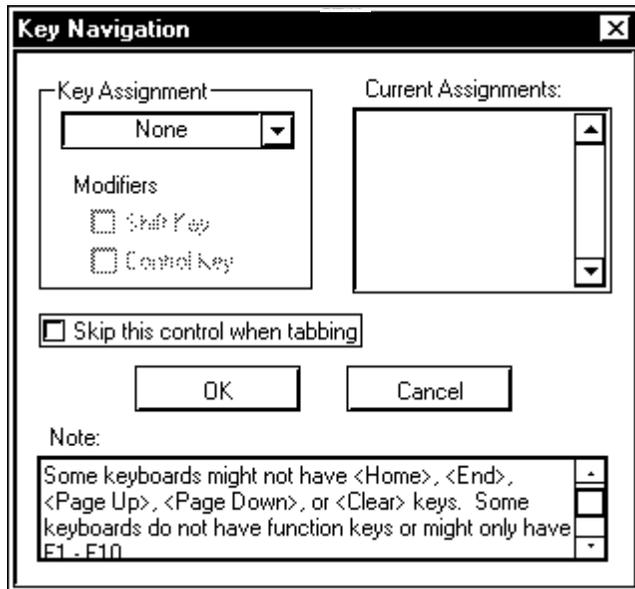
You use the ring at the top left of this dialog box to select the key for this control. You can define the key to be a single key, or a key held in combination with a modifier key, such as the <Shift> key. In addition to the <Shift> key modifier, you can choose the menu key, which is the <Alt> **(Windows and HP-UX)**; <command> **(Macintosh)**; or <meta> **(Sun)** key. The options in this ring are shown in the following illustration.

A list of the current keyboard associations, labeled Current Assignments (shown in the preceding illustration), is located at the top right of the **Key Navigation** dialog box.

```
√ None
   <Return>
   <Escape>
   <Home>
   <End>
   <Page Up>
   <Page Down>
   <Clear>
   F1
   F2
   F3
   F4
   F5
   F6
   F7
   F8
   F9
   F10
   F11
   F12
```
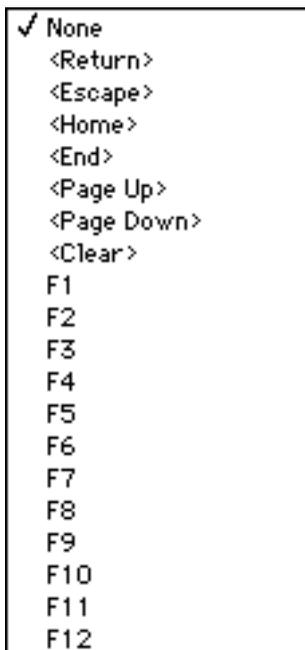
Panel Order Option

Controls and indicators on a front panel have a logical order, called panel order, unrelated to their position on the front panel. The first control or indicator you create on the front panel is element 0, the second is 1, and so on. If you delete a control or indicator, the panel order adjusts automatically. Change the panel order by selecting **Edit»Panel Order…**. The appearance of the front panel changes, because it is now in panel order edit mode, as shown in the following illustration.
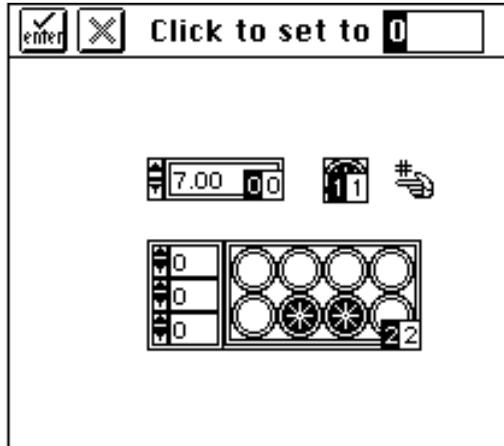
The white boxes on the controls and indicators show their current places in
the panel order. Black boxes show the new place in the panel order of the
control or indicator. Clicking an element with the panel order cursor sets
the position of the element in the panel order to the number displayed inside
the toolbar. Change this number by typing a new number into it. When the
panel order is the way you want it, click the **Enter** button to set it and exit
the panel order edit mode. Click the **X** button to revert to the old panel order
and exit the panel order edit mode.

X Button

The panel order determines the order in which the controls and indicators
appear in the records of datalog files produced by logging the front panel.

Customizing Dialog Box Controls

dialog ring

There are several types of dialog box controls: dialog rings (which are
numeric), dialog buttons (which are Boolean), dialog checkmarks, and
dialog buttons.

dialog button

The dialog controls change appearance depending on which platform you
are using. Each appears with the color and appearance typical of that
platform.

(Windows) The dialog button appears as a three-dimensional rectangular gray button. The dialog ring appears as a flat black-and-white rectangle, as shown in the following illustration.
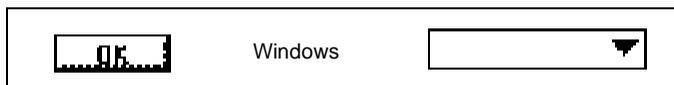
(Macintosh) The dialog button appears as a two dimensional black-and-white oval. The dialog ring appears as a black-and-white rectangle with a drop shadow, as shown in the following illustration.
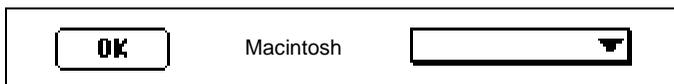
(UNIX) The dialog button appears as a three-dimensional rectangular gray button. The dialog rings appears as a flat, black-and-white rectangle with drop shadow, as shown in the following illustration. You cannot color the dialog button or the dialog ring.
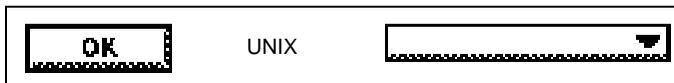
Because these controls change appearance, you can create VIs with controls whose appearance is compatible with any of the computers that can run G. Using these controls, along with the checkmark and radio button Booleans, simple numeric controls, simple strings, and the dialog fonts, you can create a VI that adapts its appearance to match any computer you use the VI on. By using the **VI Setup** options to hide the menu bar and scrollbars and control the window behavior, you can create VIs that look like standard dialog boxes for that computer.

Customizing Controls Using Imported Graphics

You can import graphics from other programs for use as background pictures, as items in ring controls, or as parts of other controls. For more information on using graphics in these controls, see Chapter 13, *List and Ring Controls and Indicators*, and Chapter 24, *Custom Controls and Type Definitions*.

Before you can use a picture you must load it into the clipboard used by BridgeVIEW or LabVIEW. There are one or two ways to do this, depending on your platform, as described below.

(Windows) If you can copy an image directly from a paint program to the Windows Clipboard and then switch to G-language software, the picture is automatically imported to the G-language clipboard. Drag a graphics file directly from the file system or, use the **Import Picture from File…** menu item from the Windows **Edit** menu to import a graphics file into the G Clipboard. In Windows 3.1, use the latter method on BMP, CLP, EMF, GIF, JPG, LZW, PCX, PNG, TARGA, TIFF, WMF, and WPG files. On Windows 95/NT, use BMP, CLP, EMF, JPG, PNG, and WMF files.

Under Windows 95/NT you can drag and drop bitmaps and metafiles directly on LabVIEW and BridgeVIEW panels.

(Macintosh) If you can copy an image directly from a paint program to the Windows Clipboard and then switch to G-language software, the picture is automatically imported to the G-language clipboard.

(UNIX) Use the **Import Picture from File…** menu item from the UNIX **Edit** menu to import a picture of type X Window Dump (XWD), which you can create using the xwd command, or a JPG or PNG picture.

(All Platforms) When a picture is on the G-language software Clipboard, you can paste it as a static picture on your front panel, or you can use the **Import Picture** menu item of a pop-up menu, or the **Import Picture** options in the Control Editor.

For an example of how to import graphics from other programs, see examples\general\controls\custom.llb.

9

Numeric Controls and Indicators

This chapter describes how to create, operate, and configure the various styles of numeric controls and indicators.

When you select **Controls»Numeric** from the palette, a new palette of controls and indicators appears, as shown in the following illustration.
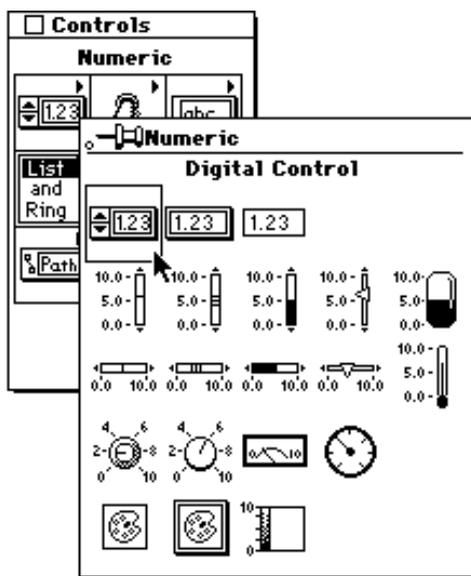
Numeric controls and indicators are either digital, slide, rotary, ring, enumerated, color box, or color ramp controls.

Digital Controls and Indicators

A digital numeric control and indicator are shown in the following illustration.
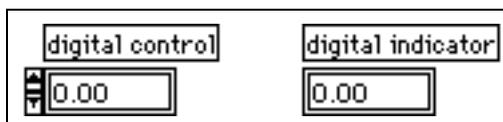
Digital numerics are the simplest way to enter and display numeric data.

Operating tool

Use the Operating tool and any of the following methods to increase or decrease the displayed value.

- Click inside the digital display window and enter numbers from the keyboard.

- Click the increment or decrement arrow of the digital control or indicator.

- Place the cursor to the right of the digit you want to change and press the up or down keyboard arrow.

Enter button

The **Enter** button appears on the toolbar to remind you the new value replaces the old only when you press the <Enter> key on the numeric keypad, click outside the display window, or click the **Enter** button. While the VI is running, any of these actions prevent the software from interpreting intermediate values as input. For example, while changing a value in the digital display to 135, you do not want the VI to receive the values 1 and 13 before getting 135. This rule does not apply to values you change using the increment/decrement arrows.

Numeric controls accept only the following—decimal digits, a decimal point, +, –, uppercase or lowercase e, and the terms Inf (infinity) and NaN (not a number), and in absolute time format the following characters: /, :, and uppercase or lowercase am and pm. If you exceed the limit for the selected representation, the software coerces the number to the natural limit. For example, if you enter 1234 into a control set for byte integers, the software coerces the number to 127. If you incorrectly enter non-numeric values such as aNN for NaN, or Ifn for Inf, the application ignores them and uses the previous value.

The increment buttons usually change the ones digit. Increasing a digit beyond 9 or decreasing below 0 affects the appropriate adjacent digit. Increasing and decreasing repeats automatically. If you click an increment button and hold the mouse button down, the display increases or decreases repeatedly. If you hold the <Shift> key down while increasing the value repeatedly, the size of the increment increases by successively higher orders of magnitude. For example, by ones, then by tens, then by hundreds, and so on. As the range limit approaches, the increment decreases by orders of magnitude, slowing down to normal as the value reaches the limit.

In a digital control for time and date, individual components (second, minute, hour, day, month, year) as well as individual relative time components can be increased and decreased.

To increase 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 might become too large to fit in the digital display on the control. You can view the complete value by resizing the control, increasing its length 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 numeric pop-up menu. A d appears on the housing of the numeric display as shown in the following illustration.
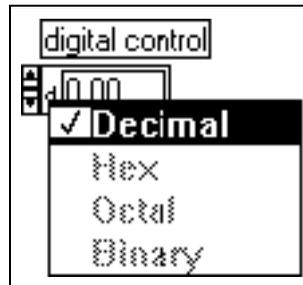
If you click the d, the menu shown in the previous illustration appears. The number 32,753 is displayed in each radix in the following illustration.
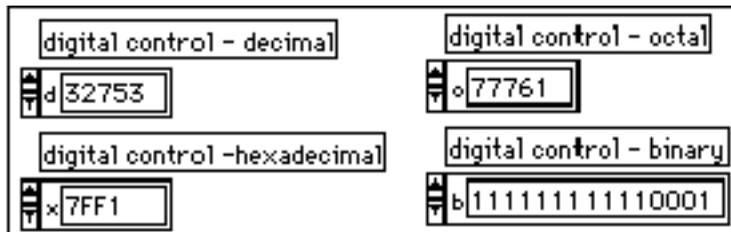
Changing the Representation of Numeric Values

You can choose from 12 representations for a digital numeric control or indicator. Use the **Representation** item from the control or indicator pop-up menu to change to 32-bit single-precision (SGL), 64-bit double precision (DBL), extended-precision (EXT) floating-point numbers, or one of the six integer representations: signed (I8) or unsigned (U8) byte (8-bit), signed (I16) or unsigned (U16) word (16-bit), or signed (I32) or unsigned (U32) long (32-bit) integers. You also can choose complex extended-precision (CXT), complex double-precision (CDB), or complex single-precision (CSG) floating-point numbers. These choices are shown in the following illustration.
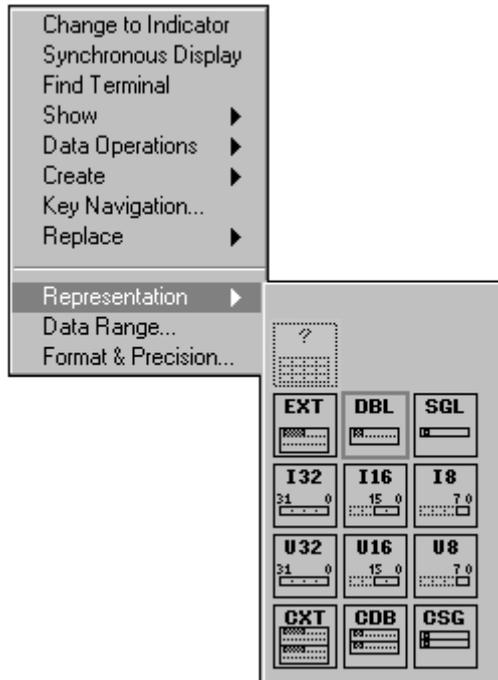
Range Options of Numeric Controls and Indicators

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

Table 9-1. Range Options of Floating-Point Numbers

Precision	Single	Double	Extended (Platform Dependent)
Minimum Positive Number	1.5E–45	5.0E–24	1.9E–4951
Maximum Positive Number	3.4E38	1.7E308	1.1E4932
Minimum Negative Number	–1.5E–45	–5.0E–324	–1.9E–4951
Maximum Negative Number	–3.4E38	–1.7E308	–1.1E4932

☞ **Note**   *Although G can process in the range shown in Table 9-1,* Range Options of
Floating-Point Numbers, **the range of extended floating-point numbers it can
represent and display in text format is ±9.9999999999999999E999. For HP-UX,
Concurrent and Power Macintosh, the range for extended floating-point numbers
is the same as for double-precision numbers.**

You can choose other limits within these natural bounds with the **Data
Range…** item from the pop-up menu. The following dialog box appears.

Numeric Range Checking

You also can 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 need to 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   G does not change or flag invalid values.
Clicking the increment and decrement
arrows changes the value by the
increment you set, but the value does not
go beyond the minimum or maximum
values.

Coerce   G 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. G coerces the value 0 to 3,
                                    the value 6 to 7, and the value 100 to 10.

Suspend   G suspends execution of a VI or subVI
                                    when a value is invalid. When you choose
                                    to suspend on invalid values, G keeps a
                                    copy of all the front panel data in memory
                                    in case you need to open the front panel to
                                    show an error.

A top-level VI with controls whose values are invalid cannot run. If the
value of a control is invalid before a subVI runs (when a subVI is about to
execute), the VI is suspended. The front panel of the VI opens (or becomes
the active window) and the invalid control(s) are outlined in red (or a thick
black line on a black-and-white monitor). The toolbar on the block diagram
of a suspended subVI looks like the following illustration.

You must set the control to a valid value before you can proceed. When all
control values are valid, the toolbar looks like the following illustration.
Click the **Run** button to continue execution.

Run button

Return to Caller button
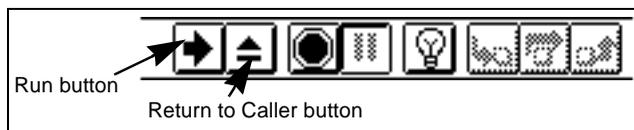
If the value of an indicator or control becomes invalid while a subVI is running, execution pauses as if there were a breakpoint. The front panel of the VI opens (or becomes the active window). The indicator(s) and control(s) currently invalid are outlined in red (or a thick black line on a black-and-white monitor). When a suspended VI finishes execution, if there are any invalid values, the **Return to Caller** button is disabled, as shown in the following illustration.
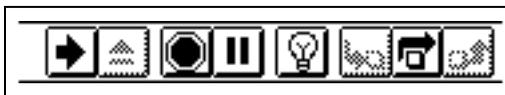
You must set indicators to valid values before you can return to the calling VI. You also can change the control values to produce valid outputs and run the subVI again by clicking the **Run** button. When all indicator values are valid, the toolbar looks like the following illustration. Click the **Return to Caller** button to continue execution.

Format and Precision of Digital Displays

You can select the format of your digital displays for numerics or for time and date. If numeric, you can choose if the notation is floating-point, scientific, engineering, or relative time in seconds. You can select their precision, that is how many digits to the right of the decimal point they display, from 0 through 20. The precision you select affects only the display of the value; the internal accuracy still depends on the representation.

To change any of these parameters of the digital display, select **Format & Precision…** item from the display pop-up menu. The dialog box opens in numeric format, as shown in the following illustration.

```
┌─────────────────────────────────────────────────┐
│              Format & Precision                  │
│  ┌───────────────────────────────────────────┐   │
│                                                  │
│   Format:              Example:                  │
│   ┌──────────────────┐  3.14                     │
│   │ Numeric      ▼ │                            │
│   └──────────────────┘                           │
│                                                  │
│        ┌──────────────────────────┐              │
│        │  Decimal            ▼ │                │
│        └──────────────────────────┘              │
│     ┌────────────────────────────────────┐       │
│     │  ┌──────┐                           │       │
│     │  │    2 │ Digits of Precision       │       │
│     │  └──────┘                           │       │
│     │  ● Floating Point Notation          │       │
│     │  ○ Scientific Notation              │       │
│     │  ○ Engineering Notation             │       │
│     │  ○ Relative Time (seconds)          │       │
│     └────────────────────────────────────┘       │
│                                                  │
│         ┌────────┐   ┌──────────┐                │
│         │   OK   │   │  Cancel  │                │
│         └────────┘   └──────────┘                │
│  └───────────────────────────────────────────┘   │
└─────────────────────────────────────────────────┘
```
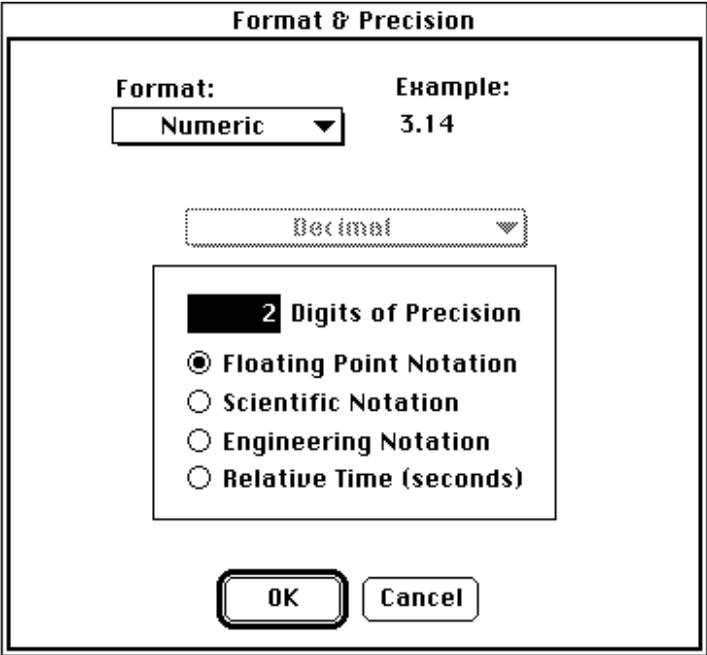
You can see an example at the top of the dialog box as you make selections.
Examples of format and precision settings on a digital control are shown in
the following illustration.

```
┌──────────────────────────────────────────────────┐
│  ┌────────────┐          ┌────────────────┐        │
│  │ 65.7123    │          │ 6.571231E+0    │        │
│  └────────────┘          └────────────────┘        │
│  floating-point notation  scientific notation      │
│  4 digits after the decimal  6 digits after the decimal │
│                                                    │
│  ┌────────────┐          ┌────────────┐            │
│  │ 65.710E+0  │          │ 65:57:05   │            │
│  └────────────┘          └────────────┘            │
│  engineering notation     relative time            │
│  3 digits after the decimal  seconds               │
└──────────────────────────────────────────────────┘
```
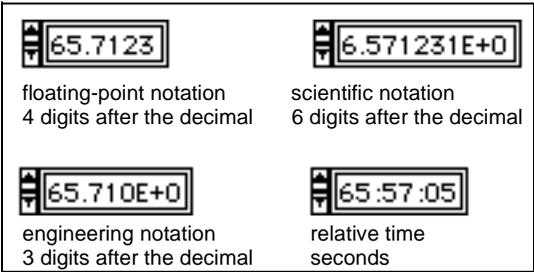
To format absolute time and/or date, select **Time & Date** from the Format ring at the top of the dialog box. The dialog box changes, as shown in the following illustration.

```
┌─────────────────────────────────────────────────┐
│              Format & Precision                  │
│ ┌─────────────────────────────────────────────┐ │
│ │   Format:            Example:               │ │
│ │  ┌──────────────┐    06:28:39 PM            │ │
│ │  │ Time & Date ▼│    02/17/1995             │ │
│ │  └──────────────┘                           │ │
│ │ ⊠ Time───────────────────────────────────   │ │
│ │   ◉ AM/PM            ○ HH:MM                 │ │
│ │   ○ 24-hour          ◉ HH:MM:SS             │ │
│ │       Seconds Precision: [   0 ]            │ │
│ │ ⊠ Date───────────────────────────────────   │ │
│ │   ◉ M/D/Y            ○ Don't Show Year       │ │
│ │   ○ D/M/Y            ○ 2 Digit Year          │ │
│ │   ○ Y/M/D            ◉ 4 Digit Year          │ │
│ │        (  OK  )   ( Cancel )                │ │
│ └─────────────────────────────────────────────┘ │
└─────────────────────────────────────────────────┘
```

You can format for either time or date, or both. If you enter only time or only date, unspecified components are inferred. If you do not enter time, 12:00 a.m is the assumed time. If you do not enter date, the previous date value is assumed. If you enter date, but the control is not in a date format, the month, day, and year ordering are assumed, based on the settings in the Preferences dialog box. If you enter only two digits for the year, the following is assumed: any number less than 38 is in the twenty-first century, otherwise the number is in the twentieth century.

Though absolute time is displayed as a time and date string, it is represented internally as the number of seconds since 12:00 a.m. Jan. 1, 1904, Universal Coordinated Time (UTC), formerly known as Greenwich Mean Time (GMT). The software keeps track of these components internally.

☞ **Note**   *When a control is in absolute time format, you always have the option to enter time, date, or time and date. If you do not want a date assumed, use relative time.*

Notice the examples at the top right of the dialog box, which change as you make selections.

The valid range for time and date differs across computer platforms as follows.

(Windows) 12:00 a.m. Jan. 2, 1970 – 12:00 a.m. Feb. 4, 2106

(Windows 95/NT) 12:00 a.m. Jan. 2, 1970 – 12:00 a.m. Jan. 17, 2038

(Macintosh) 12:00 a.m. Jan. 2, 1904 – 12:00 a.m. Jan. 2, 2040

(UNIX) 12:00 a.m. Dec. 15, 1901 – 12:00 a.m. Jan. 17, 2038

These ranges might be up to a few days wider depending on your time zone and if daylight saving time is in effect.

Slide Numeric Controls and Indicators

The slide controls and indicators are shown in the following illustration.
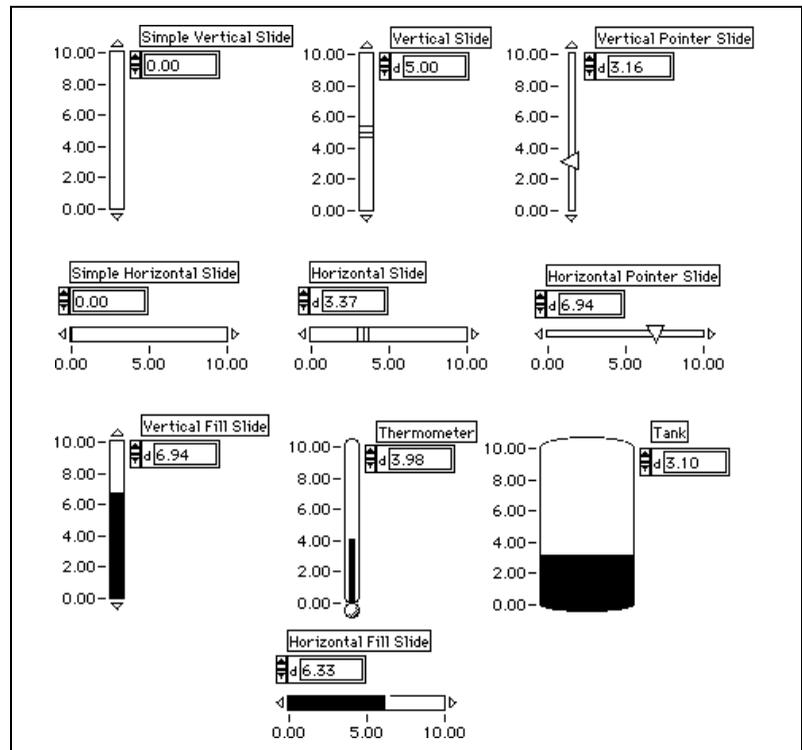
Each slide has a digital display. Use the digital displays to enter data into slide controls, as explained in the *Digital Controls and Indicators* section of this chapter. Use the Operating tool on such parts as the slider, the slide housing, the scale, and increment buttons to enter data or change values. The slider is the part that moves to show the value of the control. The housing is the stationary part that the slider moves on or over. The scale indicates the value of the slider, and the increment buttons are small triangles at either end of the housing. An example of a slider is shown in the following illustration.
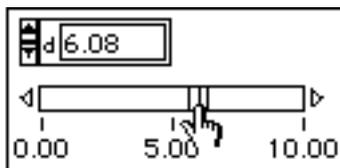
You can drag the slider with the Operating tool to a new position. If the VI is running during the change, intermediate values might pass to the program, depending on how often the VI reads the control.

You also can click a point on the housing and the slider snaps to that location as shown in the following illustration. Intermediate values do not pass to the program.
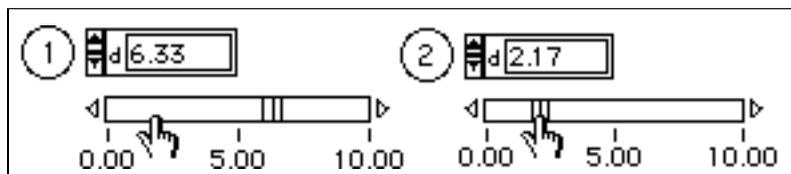
If you use a slide that has increment buttons, you can click an increment button, as shown in the following illustration, and the slider moves slowly in the direction of the arrow. Hold down the <Shift> key to move the slider faster. Intermediate values might pass to the program.
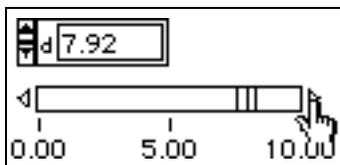
Just like the digital numerics, slides have **Representation**, **Data Range…**, and **Format & Precision** options in their pop-up menus. These options work the same as they do for digital displays, except that slides cannot represent complex numbers. Slides also have other options. The following illustration shows a slide pop-up menu.

The **Show»Digital Display** menu item of the pop-up menu controls the visibility of the digital display of the slide.

Slide Scale

The scale submenu menu items apply only to the scale. The scale pop-up menu is shown in the following illustration.
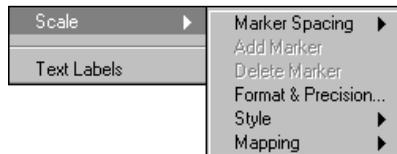
The **Marker Spacing** item is described in the *Scale Markers* section of this chapter.

The **Format & Precision** item functions as described in the *Format and Precision of Digital Displays* section of this chapter.

The **Style** item shows 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.
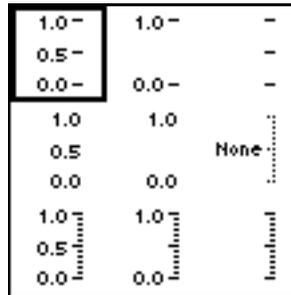
The **Mapping** item shows you the option of linear or logarithmic scale spacing, as shown in the following illustration.
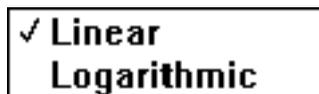
If you change to Logarithmic spacing, and the low scale limit is less than or equal to 0, the limit automatically becomes a positive number, and other markers are revalued accordingly. Keep in mind that scale options, including the mapping functions, are independent of the slide data range values, which you change with the **Data Range** pop-up option. If you want to limit your data to logarithmic values, change the data range to eliminate values less than or equal to zero.

Scale Markers

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.

Changing Scale Limits

The outer two markers are the scale limits. They are not required to coincide with the range limits, but they 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.
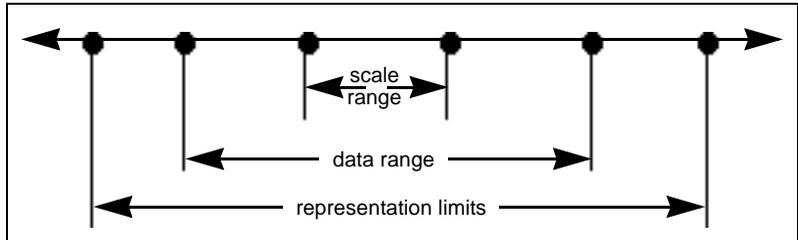
You can change the minimum, maximum, and increment of a scale interactively in five ways using either the Operating tool or the Labeling tool. If you want to change a scale programmatically, use the attribute node. See Chapter 22, *Attribute Nodes*, for more information on changing a scale programmatically using the attribute node.

- If you type a new maximum value into its display, the minimum stays the same, and G recalculates the increment automatically.

- If you type a new minimum value into its display, the maximum stays the same, and G recalculates the increment automatically.

- If you type the current minimum value into the maximum display, G flips the scale so that what was formerly the minimum is now the maximum, and can flip the scale so that what was formerly the maximum is now the minimum. G 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 the markers do not overlap.

Selecting Non-Uniform Scale Marker Distribution

By default, scale markers are evenly spaced. If you want, you can indicate exactly where inner markers for a scale are located. This is useful for marking a few specific points on a slide (such as a set point or threshold). If you want non-uniform marker distribution, select **Scale»Marker Spacing»Arbitrary** from the pop-up menu of the scale. Then, you can pop up on the slide and select **Scale»Add Marker** or pop up on the marker and

select **Add Marker**. A marker appears at an arbitrary location, as shown in the following illustration.
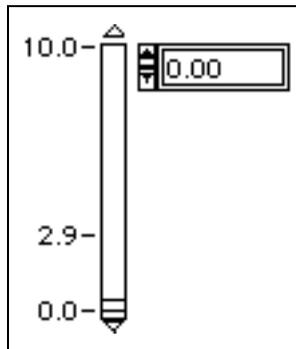
To delete an arbitrary marker, pop up on the slide and select **Scale»Delete Marker** or pop up on the marker and select **Delete Marker**.

When a marker is created, you can type a number in the marker; the location changes automatically to match the number. You also can move markers by dragging ticks, the small lines beside the data points of the scale. To do so, idle the Operating tool over the ticks. The cursor changes to show the ticks can move, as shown in the following illustration.
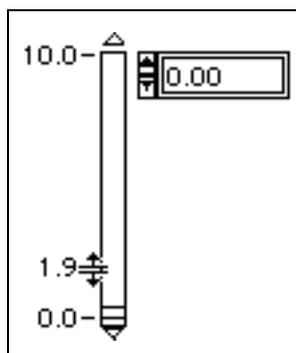
In Uniform mode, dragging a tick changes the marker distribution. In Arbitrary Markers mode, dragging a tick moves only the current marker, without changing the other markers. Pressing the <Ctrl> **(Windows)**; <option> **(Macintosh)**; <meta> **(Sun)**; or <Alt> (**HP-UX**) key while dragging creates a new marker. (If the ticks are hidden, as chosen in the **Scale»Style** dialog box, you cannot drag them.)

The arbitrary markers affect only the inner markers, not the two end markers. If no arbitrary markers are visible in the current range, the scale reverts temporarily to uniform markers.

Text Scale

You also can use text labels on numeric scales. Labels are useful because you can associate strings with integer values. This configuration is useful for selecting mutually exclusive options. To implement this option, select **Text Labels** from the slide pop-up menu. The slide control appears with default text labels, min and max, and you can begin typing immediately to enter the labels you want. Press <Shift-Enter> **(Windows** and **HP-UX)**, or <Shift-Return> **(Macintosh** and **Sun)** after each label to create a new label, and <Enter> on the numeric keypad after you finish your last label.

You can use the Labeling tool to edit min and max labels in the text display or on the slide itself.
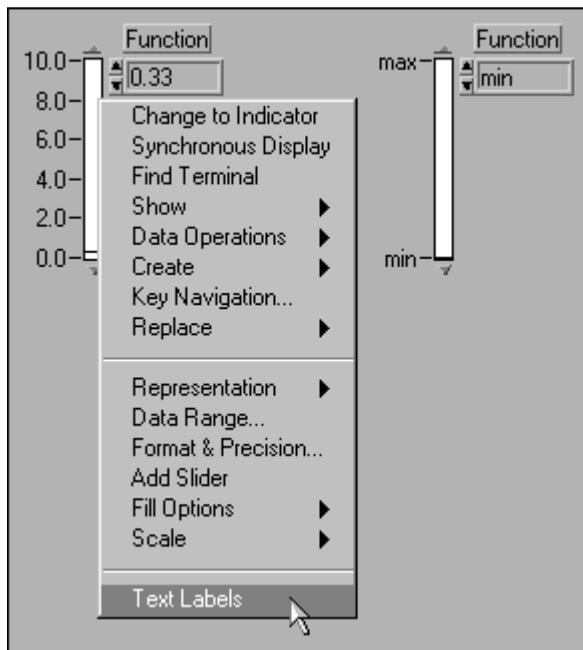
You can pop up on the text display and select **Show»Digital Display** to find out what numeric values are associated with the text labels you create. These values always start at zero (on the bottom of vertical slides and on the left or right of horizontal slides) and increase by one for each text label.

Use the **Add Item After** or **Add Item Before** item from the text display pop-up menu to create new labels, as shown in the following illustration.
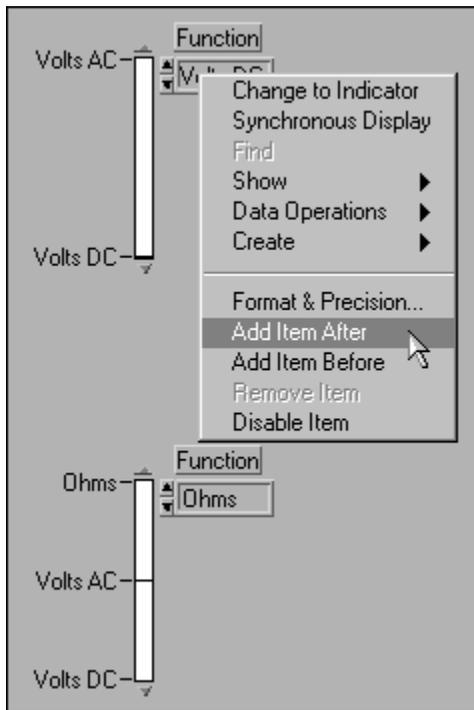
You also can press `<Shift-Enter>` to advance to a new item when you are editing the existing items.

Filled and Multivalued Slides

The **Numeric** palette contains four controls 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 slides, and you can turn fill slides into regular slides. Normally, you have three choices, as shown in the following illustration. These are fill from the minimum value to the slider location, fill from the maximum value to the slider location, or use no fill.
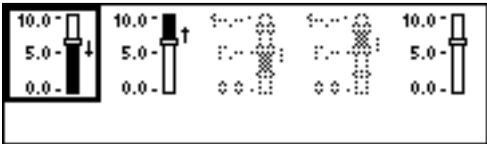
You also can show more than one value on the same slide. To do so, choose
Add Slider from the pop-up menu. A new slider appears along with a new
digital display as shown in the following illustration.
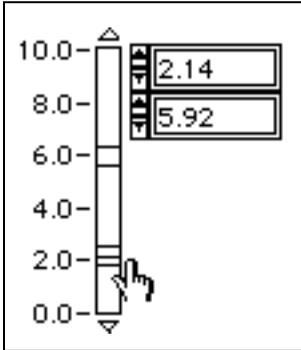
When you do this, two more options appear in the **Fill Options** palette: **Fill
to Value Above** and **Fill to Value Below**. These options apply to the active
slider. All five options appear in the following illustration.
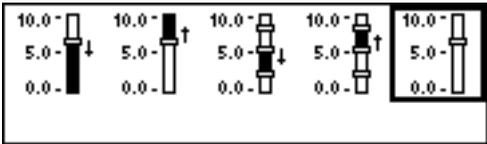
For an example of slide controls and indicators, see
`examples\general\controls\alarmsld.llb`.

Rotary Numeric Controls and Indicators

The rotary numeric controls and indicators are shown in the following illustration.
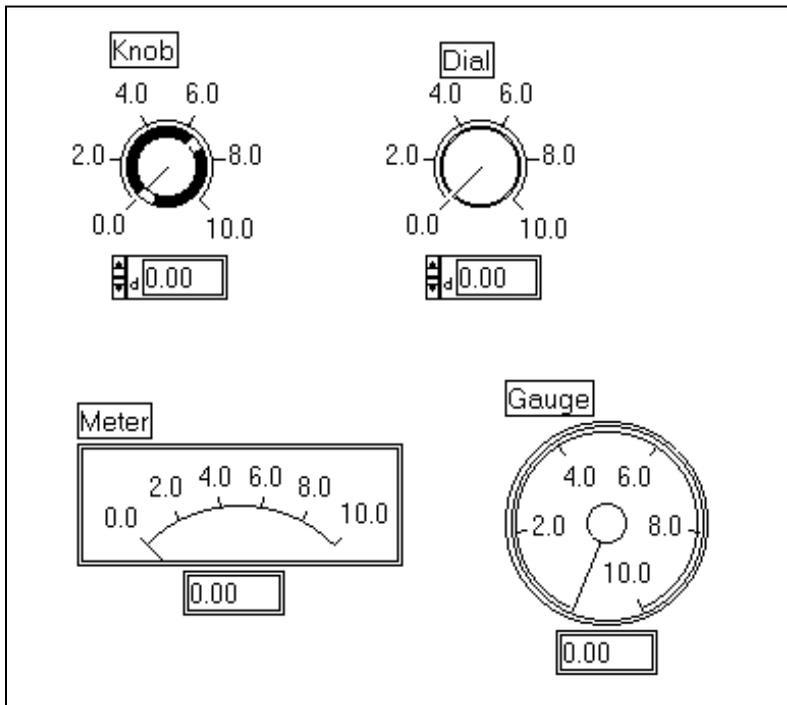
The rotary objects have most of the same options as the slide. The sliders (or needles) in rotary objects turn rather than slide, but you operate them the same way you operate slides.
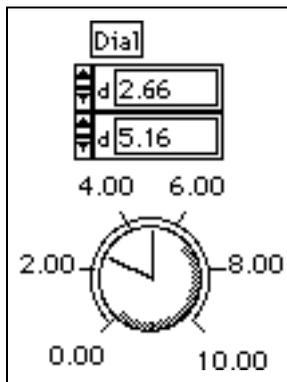
Rotary controls, like linear controls, can display more than one value, as shown in the preceding illustration. You add new values by selecting **Add Needle** from the pop-up menu, in the same way you add new values to slides.

If you move the Positioning tool to the scale, the tool changes to a rotary cursor. If you click and drag the outer limits of the scale, you change the arc the scale covers. If you click and drag the inner markers of the scale, you rotate the arc. It still has the same range but different starting and ending angles. This procedure is shown in the following illustration. Hold down the `<Shift>` key to snap the arc to 45-degree angles.

Place the cursor on the knob scale. It changes appearance to look like a horseshoe with arrowheads on either end.
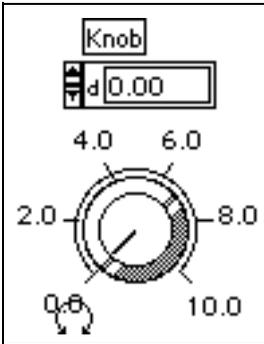
Dragging outer markers changes the size of the scale arc.
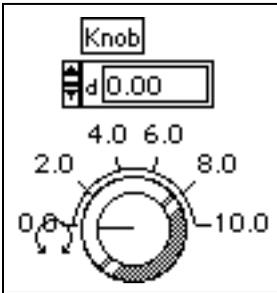
Dragging inner markers rotates the arc.
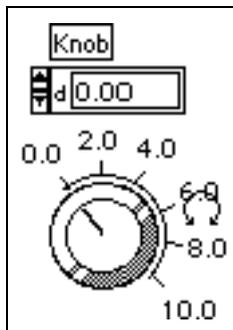
Tick marks of rotary scales can be dragged with the Operating tool, just like linear scales. Ticks also can be unevenly distributed in a similar manner to linear scales. See the section *Selecting Non-Uniform Scale Marker Distribution* earlier in this chapter for more information.

For more information on ring controls and indicators, see Chapter 13, *List and Ring Controls and Indicators*.

Color Box

The color box displays a color corresponding to a specified value. The color value is expressed as a hexadecimal number with the form RRGGBB, in which the first two numbers control the red color value, the second two numbers control the green color value, and the last two numbers control the blue color value. An example is shown in the following illustration.

Set the color of the color box by clicking it with either the Operating or Color tool to display a **Color** palette, as shown in the following illustration. Releasing your mouse button on the color you want selects that color.
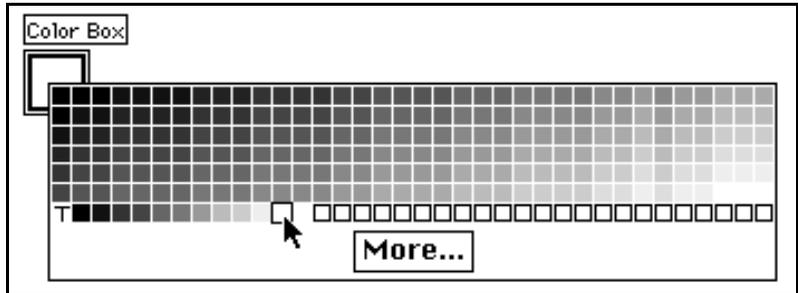
Color boxes typically are used on the front panel as indicators. The easiest way to set the color of a color box is to use the color box constant, from the **Functions»Numeric»Additional Numeric Constants** palette in the block diagram window. Wire the color box constant to the color box terminal on the block diagram. Clicking the color box constant with either the Operating tool or the Color tool displays the same **Color** palette you use to set the color box. If you want to change the color box indicator on the front panel to various colors which indicate different conditions, you can use a series of color box constants inside a Case Structure.

The small T in the lower left corner of the **Color** palette represents the transparent color.

Color Ramp

The color ramp uses color to display its numeric value. You can configure a color scale, which consists of at least two arbitrary markers, each consisting of a numeric value and the display color corresponding to that value. As the input value changes, the color displayed changes to the color

corresponding to that value. An example is shown in the following illustration.
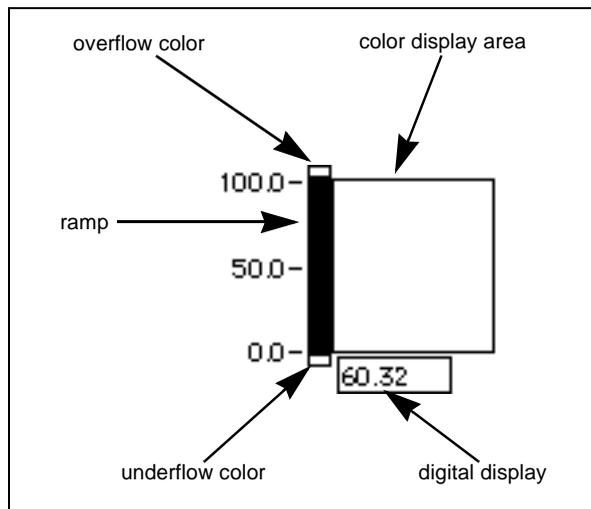
You create these pairs using the arbitrary markers of the color scale. Each arbitrary marker specifies a value and has an associated color. You change the associated color with a marker by popping up on the marker, not on the ramp. You can add markers, change the values, and set the colors as you choose.

Use the **Interpolate Colors** item of the color ramp pop-up menu to control whether intermediate shades of color for values between the indicated markers are displayed. 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 always is sorted by marker values. 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 redistributed automatically between the new values.

The color scale has a number of options, which you can display or hide. These options include the unit label, the digital display, and the ramp.

The ramp component of this control has an extra color at the top and the bottom of the scale. Use these to select a color to display if overflow or an underflow occurs. The control value is above the maximum of the color scale or below the minimum of the color scale. Click 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; a given value always maps to the same color.

🖝 **Note**   *The color ramp can have as many colors as your monitor can handle, but no more colors than what are available on your monitor.*

You can use the color scale to indicate color tables for the Intensity Graph and Intensity Chart controls. See Chapter 15, *Graph and Chart Controls and Indicators*, for more information on these controls.

Unit Types

Any numeric control 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 item from the pop-up menu, as shown in the following illustration.
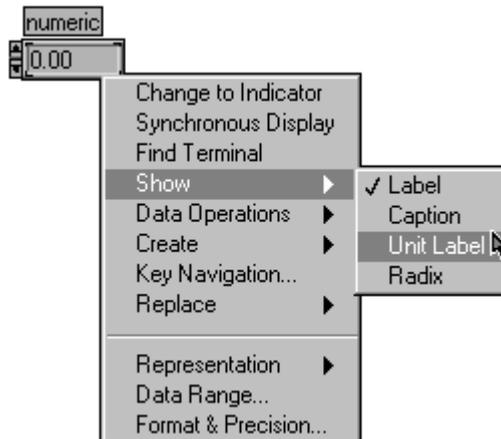
When the unit label is displayed, you can enter a unit using standard abbreviations such as m for meters, ft for feet, s for seconds, and so on.

If you are not familiar with which units are acceptable, 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 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.
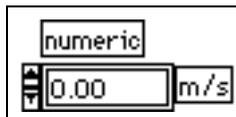
Table 9-1 through Table 9-5 list all the units you can use with the units feature.

Table 9-2. 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 9-3. Derived Units with Special Names

Quantity Name	Unit	Abbreviation
frequency	hertz	Hz
force	newton	N
pressure	pascal	Pa
energy	joule	J
power	watt	W

Table 9-3. Derived Units with Special Names (Continued)

Quantity Name	Unit	Abbreviation
electric charge	coulomb	C
electric potential	volt	V
capacitance	farad	F
electric resistance	ohm	Ohm
conductance	siemens	S
magnetic flux	weber	Wb
magnetic flux density	tesla	T
inductance	henry	H
luminous flux	lumen	lm
illuminance	lux	lx
Celsius temperature	degree Celsius	degC
activity	becquerel	Bq
absorbed dose	gray	Gy
dose equivalent	sievert	Sv

Table 9-4. 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

Table 9-4. Additional Units in Use with SI Units (Continued)

Quantity Name	Unit	Abbreviation
energy	electron volt	eV
mass	unified atomic mass unit	u

Table 9-5. CGS Units

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

Table 9-6. 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, G flags the invalid unit by placing a ? in the label, as shown in the following illustration.
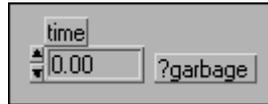
Notice you must enter units using the correct abbreviations or G flags it, as shown in the following illustration.
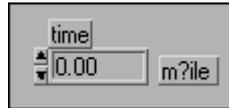
Also, keep in mind you cannot enter ambiguous units. Thus, m/ss is flagged in the following illustration, because it is not clear if it means meters per second squared, or (meters/seconds) * seconds. To resolve ambiguity, you must enter the units differently, as shown in the following three examples.
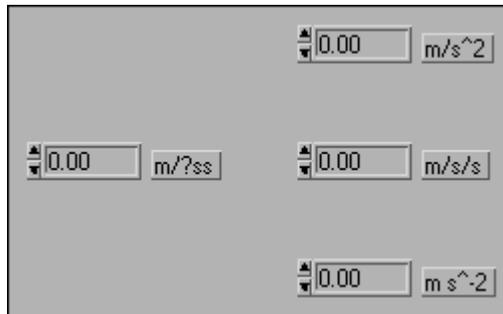
You cannot select units for a chart or graph unless you wire them to an object that has an associated unit. Then, you can show the unit that the chart or graph acquired through the connection. You can change this but only to a compatible unit that measures the same physical phenomenon. For example, if an input to a chart carries the unit mi, you can edit the chart unit to ft, in, or m, but not to N, Hz, min, acre, or A.

Units and Strict Type Checking

A wire connected to a source that has a unit associated with it connects only to a destination with a compatible unit, as shown in the following illustration.
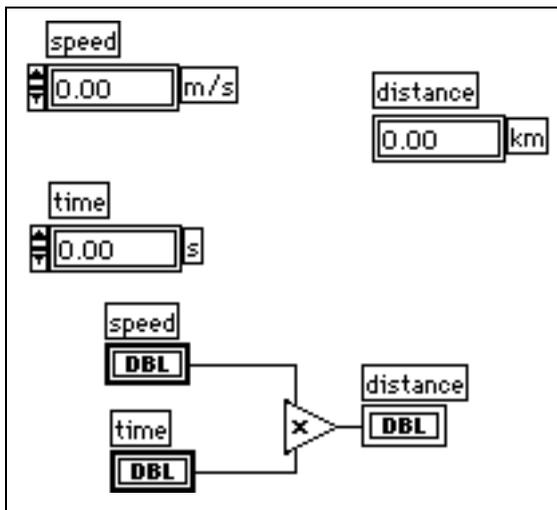
In the case of the previous illustration, the distance display is scaled automatically to display kilometers instead of meters, because that was the specified unit for the indicator.

You cannot connect signals with incompatible units, as shown in the following illustration. If you select **List Errors** from the pop-up menu for the broken wire shown below, the List Errors window indicates the error is `Wire: unit conflict.`
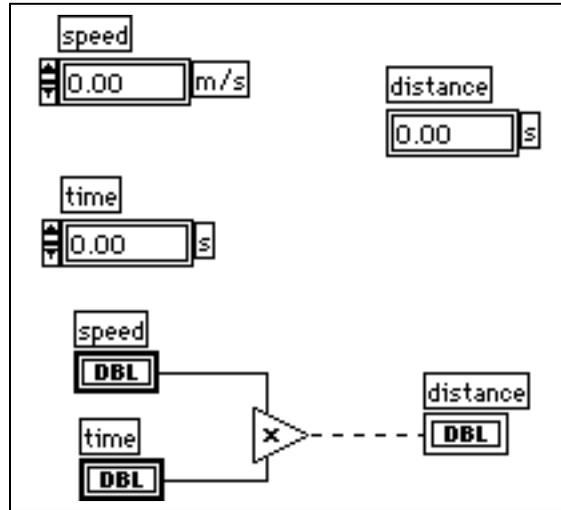
Some functions are ambiguous with respect to units and cannot be used with signals that have units. For example, the Add One function is ambiguous with respect to units. If you are using distance units, Add One cannot tell whether to add one meter, one kilometer, or one foot. Because of this ambiguity, the Add One function and similar functions cannot be used with data that has associated units.

One way around this difficulty is to use a numeric constant with the proper unit and the addition function on the block diagram to create your own Add One unit function.
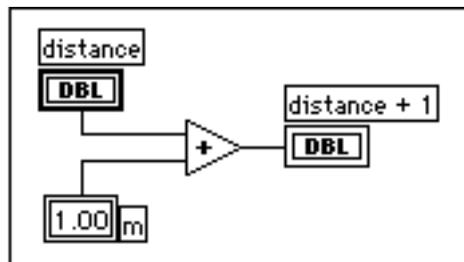
☞ **Note**   *You cannot use units in Formula Nodes.*

Polymorphic Units

If you want to create a VI that computes the root mean square value of a waveform, you must define the unit associated with the waveform. A separate VI is necessary for voltage waveforms, current waveforms, temperature waveforms, and so on. For one VI to do the same calculation, regardless of the units received by the inputs, G-language software has polymorphic unit capability.

Create a polymorphic unit by entering $x, where x is a number (for example, $1). Think of this as a placeholder for the actual unit. When the VI is called, G substitutes the units you pass in for all occurrences of $x in that VI. The $x place holder always gets passed a unit expressed in base units, for example, meters, meters/sec, not feet, inches, and so on.

A polymorphic unit is treated as a unique unit. It is not convertible to any other unit and propagates throughout the diagram just as other units do. When it is connected to an indicator that also has the abbreviation $1, the units match and the VI can compile.

$1 can be used in combinations like any other unit. For example, if the input is multiplied by 3 seconds and then wired to an indicator, the indicator must be $1 s units. If the indicator has different units, the block diagram shows a bad wire.

If you need to use more than one polymorphic unit, you can use the abbreviations $2, $3, and so on.

A call to a subVI containing polymorphic units computes output units based on the units received by its inputs. For example, suppose you create a VI that has two inputs with the polymorphic units $1 and $2 that creates an output in the form $1 $2/s. If the subVI is wired with inputs of m/s to the $1 input and kg to the $2 input, the output unit is 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 this VI is wired with inputs of m/s to the $1 input and m/s^2 to the $1/s input, the output unit is computed as m^2/s^2. If this VI is wired with inputs of m to the $1 input and kg to the $1/s input, however, the subVI call is broken. One of the inputs is declared to be a unit conflict and the output is computed (if possible) from the other. A polymorphic VI can have a polymorphic subVI because the respective units are kept distinct.

10

Boolean Controls and Indicators

This chapter describes how to create, operate, and configure Boolean controls and indicators.

Creating and Operating Boolean Controls and Indicators

A Boolean control or indicator has two values—TRUE or FALSE. Boolean controls and indicators are available from the **Controls»Boolean** palette, as shown in the following illustration.
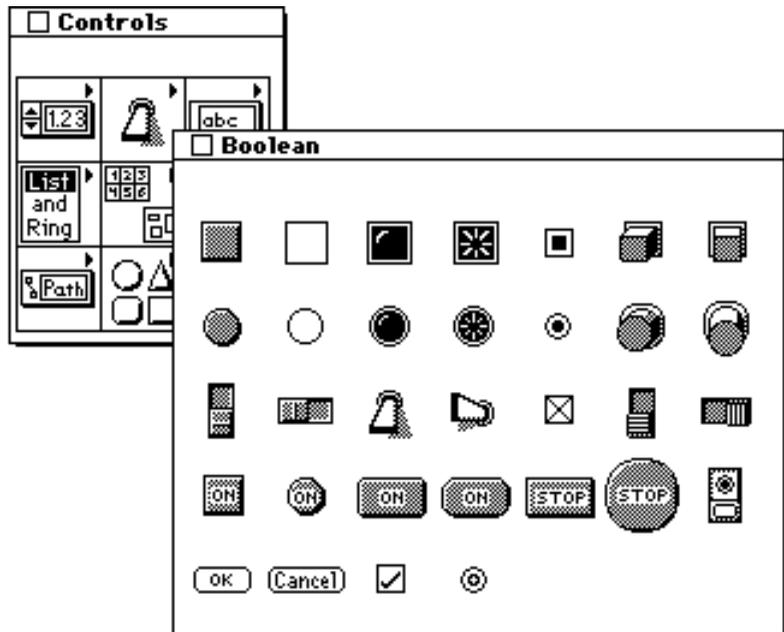
Some Boolean controls that simulate mechanical pushbuttons, toggle switches, and slide switches are shown in the following illustration.
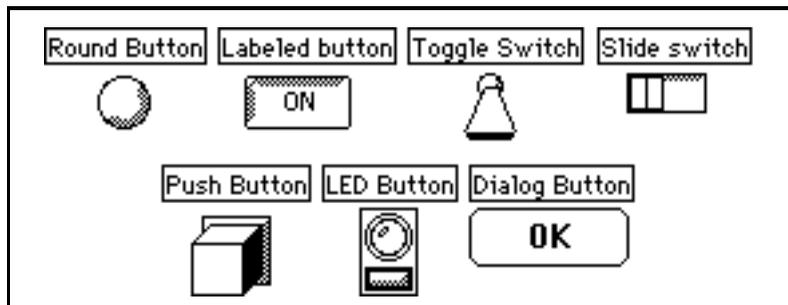
Note   *The dialog button, dialog checkmark, and dialog radio button look different on each platform. For information about dialog controls on various platforms, see the Customizing Dialog Box Controls section in* Chapter 8*, Introduction to Front Panel Objects.*

Some Boolean indicators that simulate LEDs and lights are shown in the following illustration.
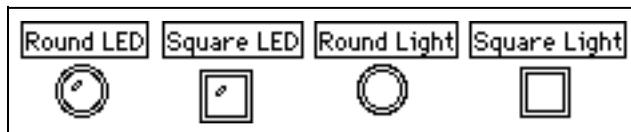
Clicking a Boolean control with the Operating tool toggles it between its TRUE (on) and FALSE (off) states. In run mode, clicking an indicator has no effect because indicators are for output only. In edit mode, you can operate both controls and indicators.

Configuring Boolean Controls and Indicators

Each Boolean control or indicator has several items available in its pop-up menu. The pop-up menu for a Boolean object is shown in the following illustration.
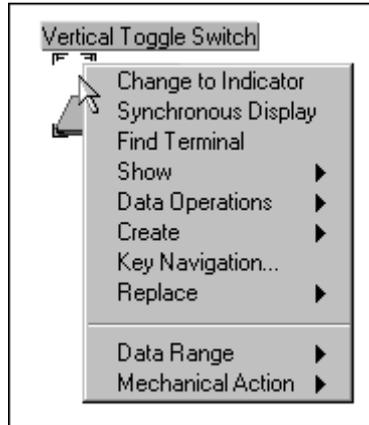
The menu items above the line in the pop-up menu are common to all controls and indicators and are described in the *Front Panel Control and Indicator Options* section of Chapter 8, *Introduction to Front Panel Objects*. The **Data Range** and **Mechanical Action** items are described later in this chapter.

Changing Boolean Labels

Several controls in the **Boolean** palette display text and are called *labeled Booleans*. Initially, the buttons display the word ON in their TRUE state and the word OFF in their FALSE state, as shown in the following illustration. Then, when you click the button with the Operating tool, the control toggles to the opposite state. In edit mode, use the Labeling tool to change the text in either state, for example, YES instead of ON.

By default, the text centers on the button. If you want to move the text, select **Release Text** from the pop-up menu.

At this point, use the Positioning tool to reposition the text, or choose the item **Lock Text in Center**. Select the Boolean text, then use the Font ring on the tool bar to change the font, size, and color of the text. You also can hide the Boolean text from both states by toggling the **Boolean Text** item on the **Show** submenu.

The Boolean controls that do not appear in the palette as labeled Booleans are unlabeled by default. However, you can select **Boolean Text** from the **Show** submenu to make them labeled. You can move the Boolean text in these Booleans. Their text is not locked in the center of the button unless you select **Lock Text in Center** from the pop-up menu.

If you want to change the font of either the name label or the Boolean text without changing both, select what you want to change with the Labeling tool, and then use the menu items in the Font ring to make the changes you want.

Boolean Data Range Checking

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

Configuring the Mechanical Action of Boolean Controls

Boolean controls have six types of mechanical action. Select the appropriate action for your application from the pop-up menu **Mechanical Action** palette, shown in the following illustration. In these palette symbols, M stands for the motion of the mouse button when you operate the control, V stands for the output value of the control, and RD stands for the point in time the VI reads the control.
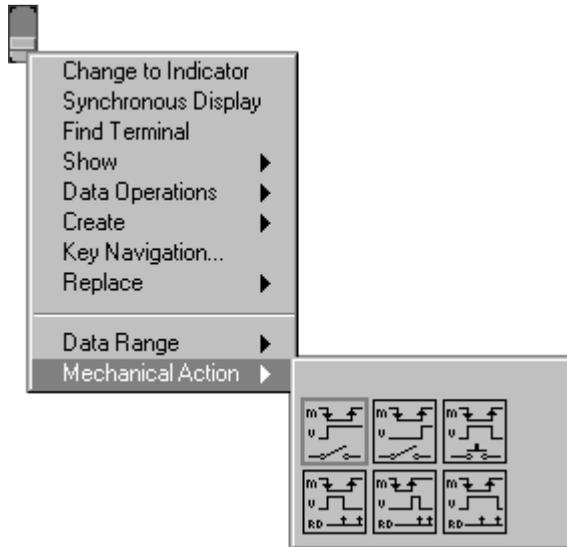
 The **Switch When Pressed** action changes the control value each time you click it with the Operating tool, in a manner similar to a light switch. The action is not affected by how often the VI reads the control.

 The **Switch When Released** action changes the control value only after you release the mouse button during a mouse click within the graphical boundary of the control. 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 it, and retains the new value until you release the mouse button. At this time, the control reverts to its original value, similar to the operation of a door buzzer. The action is not affected by how often the VI reads the control.

 With **Latch When Pressed** action, the control changes its value when you click it, and retains the new value until the VI reads it once. At this point, the control reverts to its default value, whether or not you keep pressing the mouse button. This action is similar to a circuit breaker and is useful for stopping While Loops or for getting the VI to do something only once each time you set the control.

The **Latch When Released** action changes the control value only after you release the mouse button within the controls graphical boundary. When your VI reads it once, the control reverts to the old value. This action works in the same manner as do dialog-box buttons and system buttons.

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

For an example of Boolean controls and indicators, see
`examples\general\controls\booleans.llb`.

Customizing Booleans 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 *Using the Control Editor* section of Chapter 24, *Custom Controls and Type Definitions*.

11

String Controls and Indicators

This chapter describes how to use string controls and indicators, and the table. You can access these objects through the **Controls»String & Table** palette, shown in the following illustration.
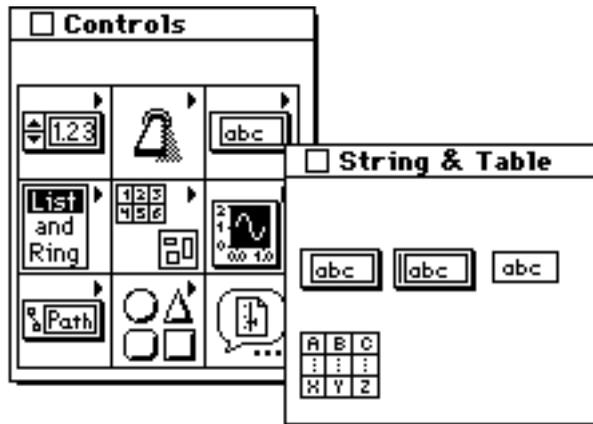
Using String Controls and Indicators

A string control and string indicator are shown in the following illustration.
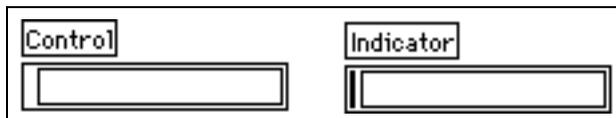
You enter or change text in the string control using the Operating tool or the Labeling tool. By default, new or changed text does not pass to the diagram until you press the `<Enter>` key on the numeric keypad, click the **Enter** button in the **Tools** palette, or click outside the control to terminate the edit session. However, if you select the **Update Value while Typing** item from the pop-up menu, the control value changes with each character. Pressing the `<Return>` key on the alphanumeric keyboard enters a carriage return.

When the text reaches the right border of the string display, the string wraps to the next line, breaking on a natural separator such as a space or tab character.

☞ **Note**   *When running a VI, you use the* <Tab> *key to move to the next control. When editing a VI, pressing the* <Tab> *key changes tools. To enter a tab character into a string, select the* **'\'** **Codes Display** *menu item from the string pop-up menu and type* \t. *To enter a linefeed into a string, press the* <Enter> **(Windows** *and* **HP-UX)**, *or* <Return> **(Macintosh** *and* **Sun)** *key on the alphanumeric keyboard, or select the* **'\'** **Codes Display** *menu item from the string pop-up menu and type* \n. *To enter a carriage return into a string, select the* **'\'** **Codes Display** *menu item from the string pop-up menu and type* \r. *Refer to* Table 11-1, *G '\' Codes, in the* **Display Types** *section of this chapter for a complete list of* **'\'** *codes.*

For information on manipulating strings, see the **Online Reference» Function and VI Reference»String Functions** topic. Also, see the examples in examples\general\strings.llb.

String Control and Indicator Menu Items

Strings have special features you access through the string pop-up menu, shown in the following illustration.
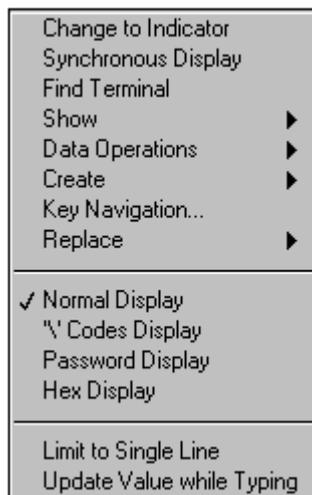
Scrollbar Menu Items

The **Scrollbar** menu item from the string pop-up **Show** submenu is disabled unless you increase the size of your string enough to fit a scrollbar. If you select this item, a vertical scrollbar appears on the string control or indicator as shown in the following illustration so that you can display text not visible in the string control. You also can use this item to minimize the space taken up on the front panel by string controls containing a large amount of text. If the menu item is dimmed, you must make the string taller to accommodate the scrollbar before choosing this item.
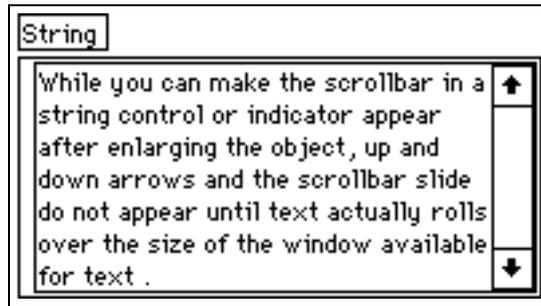
Display Types

Menu items in the middle section of the string pop-up menu let you select whether a string displays data as it normally does, displays backslash codes in place of nonprintable characters, displays in password mode (using the * symbol in place of each character), or displays hexadecimal characters.

Normal Display

The **Normal Display** menu item displays all characters as typed (with the exception of characters created with special keys such as the <Tab> or <Esc> key, which are nondisplayable).

Backslash ('\') Codes Display

By choosing **'\' Codes Display** from the string pop-up menu you instruct the software to interpret characters immediately following a backslash (\) as a code for nondisplayable characters. The following table shows how G interprets these codes.

Table 11-1. G '\' Codes

Code	G Interpretation
\00 - \FF	Hex value of an 8-bit character; must be uppercase.
\b	Backspace (ASCII BS, equivalent to \08).
\f	Form feed (ASCII FF, equivalent to \0C).
\n	Linefeed (ASCII LF, equivalent to \0A).
\r	Carriage return (ASCII CR, equivalent to \0D).
\t	Tab (ASCII HT, equivalent to \09).
\s	Space (equivalent, \20).
\\	Backslash (ASCII \, equivalent to \5C).

Use uppercase letters for hexadecimal characters and lowercase letters for the special characters, such as form feed and backspace. Thus, G interprets the sequence \BFare as *hex BF* followed by the word *are*, whereas G 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, G assumes a *0* follows the backslash, so G interprets \B as *hex 0B*. Any time a backslash is not followed by a valid hex character, G does not recognize the backslash character.

You can enter some nondisplayable characters from the keyboard, such as a carriage return, into a string control, whether or not you select **'\' Codes Display**. However, if you enable the backslash mode when the display window contains text, G redraws the display to show the backslash representation of any nondisplayable characters as well as the \ character itself.

Suppose the mode is set to Normal Display and you enter the following string.

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

When you enable the mode, the following string appears because the carriage return after `left` and the backslash characters following it are shown in backslash form as \n\\.

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

Suppose now you select **'\' Codes Display** and you enter the following string.

```
left
```

```
\right\3F
```

When you disable the mode, the following string appears because G 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 **'\' Codes Display** again, the following string appears.

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

Indicators behave in the same way.

In these examples, 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 useful for debugging programs, and for sending nonprintable characters to instruments, serial ports, and other devices.

Password Display

With the **Password Display** item, the string control displays the * symbol for each character entered into it. When you read the data of the string from the block diagram, however, you read the actual data the user entered. If you try to copy data from the control, only the * characters are copied.

Hex Display

Use the **Hex Display** item to display the string as hex characters rather than alphanumeric characters. As with **'\' Codes Display**, the **Hex Display** item is useful for debugging and when communicating with instruments.

Limit to Single Line

The **Limit to Single Line** menu item prevents you from entering a carriage return into a string while typing into it.

Update Value while Typing

Select the Update Value while Typing menu item to let the value of a control change as characters are entered instead of waiting until the user clicks the Enter button or otherwise terminates the edit session. This behavior can be useful to check the correctness of the input, limit input, or give user feedback. For example, it is possible to make a string control that limits input to alphanumeric characters.

Tables

A table is a 2D array of strings. The following illustration is an example of a table with all its features shown.
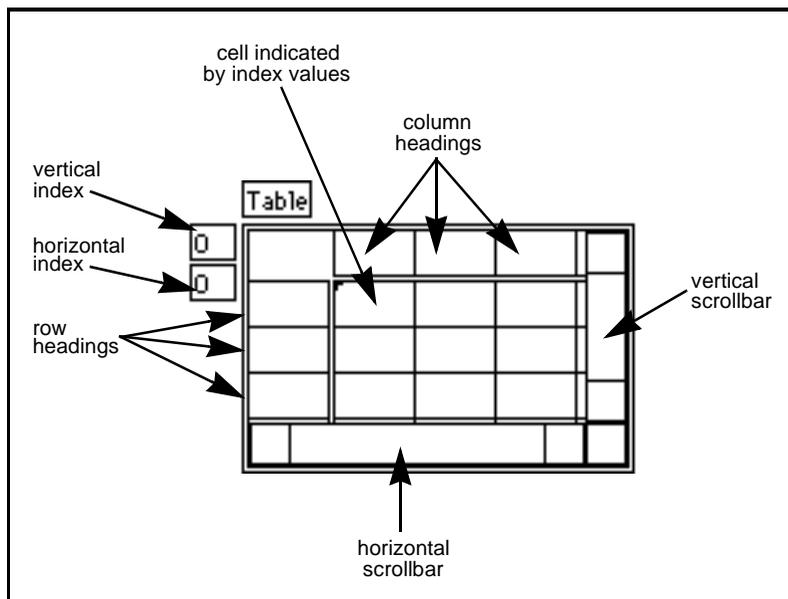
A table has row and column headings which are separated from the data by a thin, open border space. You enter headings when you place the table on the front panel, and you can change them using the Operating tool or Labeling tool. You can update or read headings using the attribute node.

The index display indicates which cell is visible at the upper left corner of the table. You can operate these indices just as you do on an array.

Resizing Tables, Rows, and Columns

You can resize the table from any corner with the Resizing tool. You can resize individual rows and columns in a table by dragging one of the border lines with the Positioning tool. When the tool is placed properly to drag a line, one of the drag cursors shown in the following illustration appears. Click a line and drag it to resize the row or column.
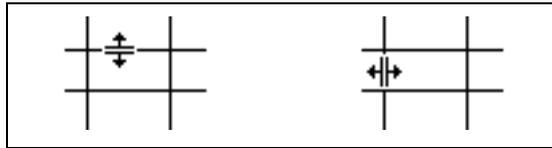
You can use the `<Shift>` key while dragging a border line to size multiple rows or columns to be all the same size. If the row or column you are resizing is inside an area of the table you selected (outlined in blue or bold), all the rows or columns in the table also are equally sized. Thus, the uneven horizontal rows in the following illustration to the left become evenly spaced in the illustration to the right because the `<Shift>` key was used.
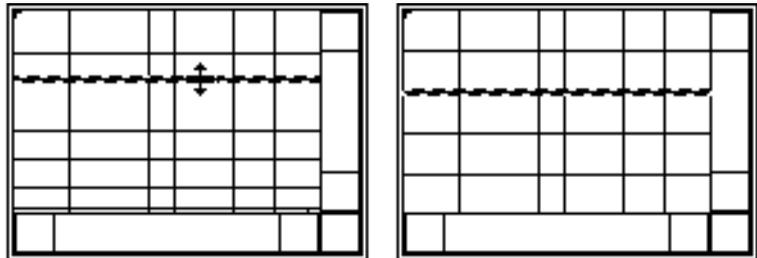
Entering and Selecting Data Tables

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, type in your data.

The `<Return>` key on the alphanumeric keyboard enters your text and moves the cursor into the cell below. The `<Enter>` key on the numeric keypad enters your data and terminates the entry. Pressing the `<Shift>` key while pressing the arrow keys moves the entry cursor to adjacent cells.

You can select individual cells with the Operating tool or by double-clicking, and you can extend the selection by pressing `<Shift-click>` and then dragging.

Moving outside the current contents of the table scrolls the table while extending the selection. A border appears around selected cells to indicate they are selected. You can turn scrolling on or off using the **Selection Scrolling** item in the table pop-up menu.

You also can select all the data in a table, or all the data in a row or column. Position the Operating tool at the upper left corner of the table to select all data. A special, double-arrow cursor appears when the tool is positioned properly as shown in the following illustration. Click the area to select the data.
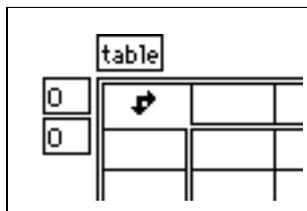
To select an entire row or column of data, position the Operating tool at the left border of a row, or at the top of a column. Again, a special arrow cursor appears when the tool is properly positioned, shown in the following illustration. Click and drag across width of column or height of row to select the data.
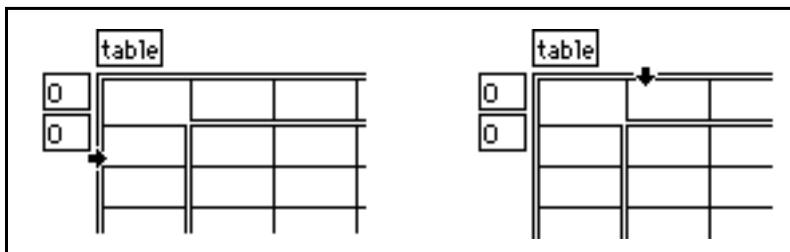
You can copy, cut and paste data, using the **Copy Data**, **Cut Data**, and **Paste Data** menu items in the **Data Operations** submenu of the pop-up menu.

If you cut any row or rows of data, all rows below it move up, as shown in the top portion of the following illustration. If you cut any column of data, all columns to the right scroll left, as shown in the bottom portion of the following illustration.

If you cut a portion of a row or column, the entire row or column is removed from the table but only the selected data is copied to the clipboard.

Pasting without first selecting a target causes rows and columns to be added to the table as needed.

Selecting a row   After cutting row

Selecting a column   After cutting column
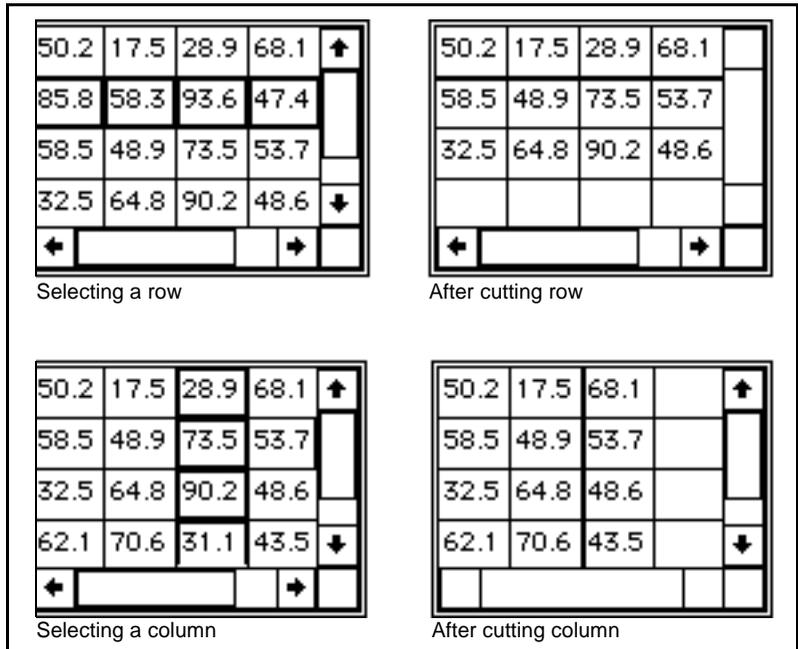
To show a selected area of data, you can turn the item on or off by selecting or deselecting **Data Operations»Show Selection**.

The row and column headings of a table are not part of the table data. Table headings are a separate piece of data, and you can read and set them using the attribute node.

The table is the same as a two-dimensional array of strings as far as the block diagram is concerned. Because of this, string functions can manipulate tables. See the **Online Reference»Function and VI Reference»String Functions** topic for information on using these functions.

12

Path and Refnum Controls and Indicators

This chapter describes how to use file path controls and refnums, which are available from the **Controls»Path & Refnum** palette, shown in the following illustration.
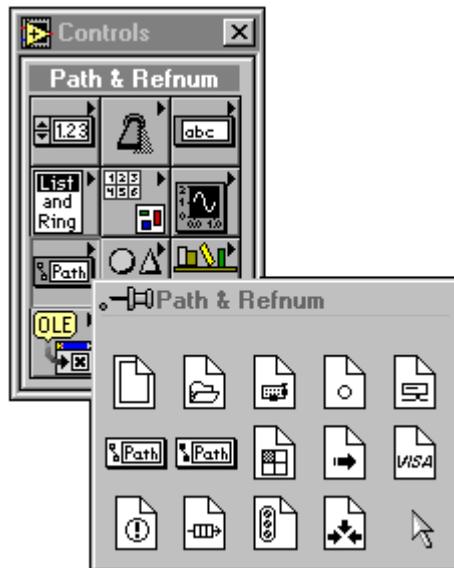
Path Controls and Indicators

A path control and path indicator are shown in the following illustration.
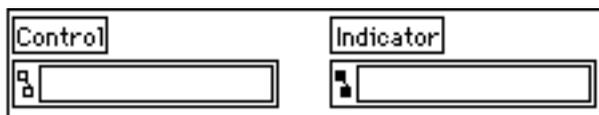
You use path controls and indicators to enter and display, respectively, the location of a file or directory in a file system using the standard syntax for

Path Symbol

Not a Path Symbol

a given platform. If a function that is supposed to return a path fails, it returns an invalid path. When an invalid path is displayed by a path control or indicator, the Path symbol changes to the Not a Path symbol, and `<Not A Path>` appears in the text field to indicate the path is invalid.

You change the value of a path control from a valid path to an invalid path by clicking the Path symbol and selecting the **Not a Path** item from the menu. This is shown in the following illustration. In the same way, you change the value of a path control or indicator from an invalid path to a valid empty path by clicking the Not a Path symbol and selecting the **Valid Path** item from the menu.
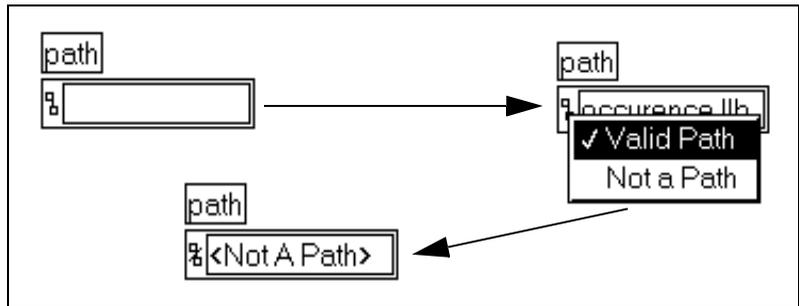
You might use this **Not a Path** value as the default value for a path control on your VI. In this way, you detect when the user has not entered a path. In this case use a file dialog box for choosing a path.

An empty path in a path control appears as an empty string in the control. When this empty path is wired to the File I/O functions, particularly File/Directory Info VI and the List Directory VI, on the Windows platform the empty path refers to the list of drives mapped to your computer. On the Mac platform, the empty path refers to the files on your desktop. On UNIX platforms, the empty path refers to the root directory.

Refnum Controls and Indicators

The **Path & Refnums** palette contains several refnum controls, shown in the following illustration. A refnum is a unique identifier used to identify an object, such as a file, a device, or even a network connection to another machine. Many refnums are used to specify upon which object an I/O operation is intended to act, such as the file read by a File Read function. A refnum control passes a refnum into a VI and a refnum indicator passes a refnum out of a VI. For each refnum control, there is a corresponding

indicator. You create a refnum indicator by first creating a refnum control. Then pop up on the control and select **Change to Indicator** from the menu.

The following illustration shows the different kinds of refnums.
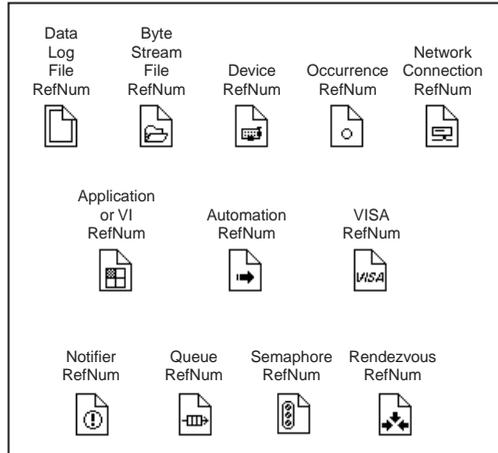
When you open a file, you must specify the path of the file you want to open. A *refnum* is returned identifying that file. You use this refnum in all subsequent operations relating to that file. You can think of this refnum as a unique number produced each time you open a file to identify it. When you close the file, the refnum is disassociated from the file. Following are descriptions of the refnums in the above illustration, found on the **Controls»Path & Refnum** palette.

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

• **Data Log File RefNum**—Because datalog files have an inherent structure, the Data Log File RefNum passes the refnum as well as a description of the file type to (or from) calling VIs. The Data Log File RefNum is resizeable, like a cluster. You place a control inside the refnum defining the structure of the file. For a file containing numbers, you create a datalog refnum containing a number. If each record in the file contains a pair of numbers, you place a cluster inside the refnum, and then place two numeric controls inside the cluster.

• **Byte Stream File RefNum**—The Byte Stream File RefNum is used with byte stream files, either text or binary files. Typically, you use it when you open or create a file in one VI but want to perform I/O on the file in another VI. You must have a refnum control on the front panel

of the VI that performs I/O, and a refnum indicator on the front panel of the VI that opens or creates the file.

The remaining refnums are easy to use, because they do not have any options.

- **Device RefNum**—The Device RefNum is used with the device I/O functions available for the Macintosh. It is used only when you open a device in one VI but want to perform I/O on the device in another VI.

☞ **Note**   *A Device RefNum control on your front panel is rarely necessary. These controls are used in the Instrument I/O VIs in* vi.lib *to access the G serial device drivers. A Device RefNum control is only necessary to write a special-purpose G device driver for the Macintosh.*

- **Occurrence RefNum**—The Occurrence RefNum is used with the occurrence functions. You use it only when you generate an occurrence in one VI but want to set or wait for the occurrence in another VI.

- **Network Connection RefNum**—The Network Connection RefNum is used with the TCP/IP VIs. You generally use it when you open a network connection in one VI but want to perform I/O on the network connection in another VI.

- **Application or VI Refnum**—The Application or VI refnum control is used with the VI Server functions. This refnum control is used when you open a reference to either a LabVIEW application, or a VI within a LabVIEW, and you wish to pass the reference as a parameter to another VI. By passing one of these refnums to one of the VI Server functions, you can control the behavior of the application and of specific VIs. By selecting **Select VI Server Class** from the refnum control submenu, you can specify the data type of the control to be an application refnum, a VI refnum, or a strictly-typed VI refnum. Strictly-typed VI refnums have data type information that includes the connector pane of the VI and can therefore be used to call a dynamically loaded VI using the Call By Reference function. You can select a VI from disk that has the desired connector pane using **Select VI Server Class»Browse....** You can also drag and drop a VI icon/connector pane, or a subVI icon from a diagram or the hierarchy window to a VI refnum control to specify the type of a strictly-typed VI refnum control.

  While VI refnum controls are usually used to pass a VI refnum from one VI to another, a strictly-typed VI refnum control is also required as a type-specifier input to the Open VI Reference function when you want to get a strictly-typed reference to a VI. In this case, the value of

the refnum control is unimportant. Only the type is used by
the function.

- **Automation RefNum**—The Automation refnum is a reference to a
  Automation object.

- **VISA RefNum**—A VISA RefNum is a reference to a VISA resource.

- **Notifier RefNum**—The Notifier RefNum is used with the notification
  VIs. You generally use it when you create a notifier in one VI but want
  to wait on a notification or send a notification in another VI.

- **Queue RefNum**—The Queue RefNum is used with the queue VIs.
  You generally use it when you create a queue in one VI but want to
  insert or remove queue elements in another VI.

- **Semaphore RefNum**—The Semaphore RefNum is used with the
  semaphore VIs. You generally use it when you create a semaphore in
  one VI but want to acquire or release the semaphore in another VI.

- **Rendezvous RefNum**—The Rendezvous RefNum is used with the
  rendezvous VIs. You generally use it when you create a rendezvous
  in one VI but want to wait at the rendezvous in another VI.

The following illustration shows the front panel and block diagram of a
VI that opens or creates a file and passes out a byte stream file refnum,
through a byte stream file refnum indicator, that can be used to access the
file in other VIs.
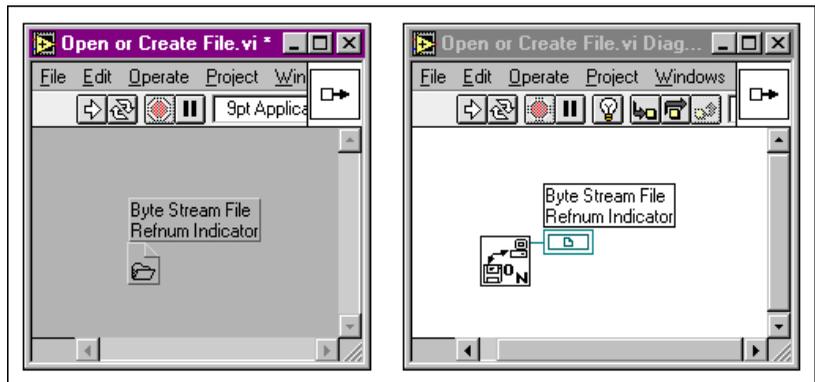
The following illustration shows the front panel and block diagram of a VI that writes data to and then closes the file specified by the byte stream file refnum passed into the VI through a byte stream file refnum control.
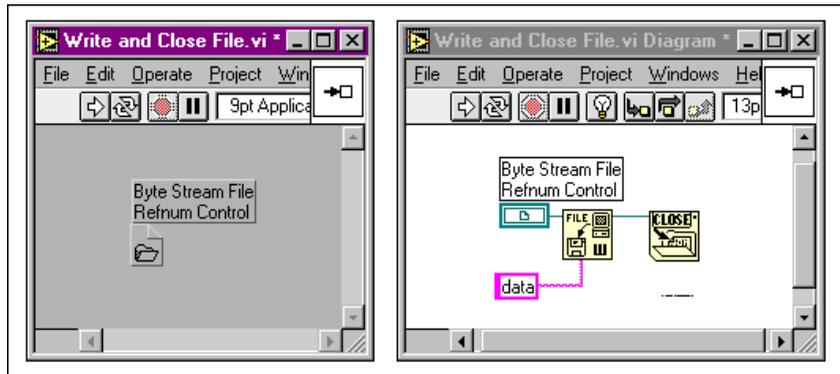
The following illustration shows the front panel and block diagram of a VI that passes a byte stream file refnum used to access a file opened or create by one subVI to another subVI that writes data to and then closes that file.
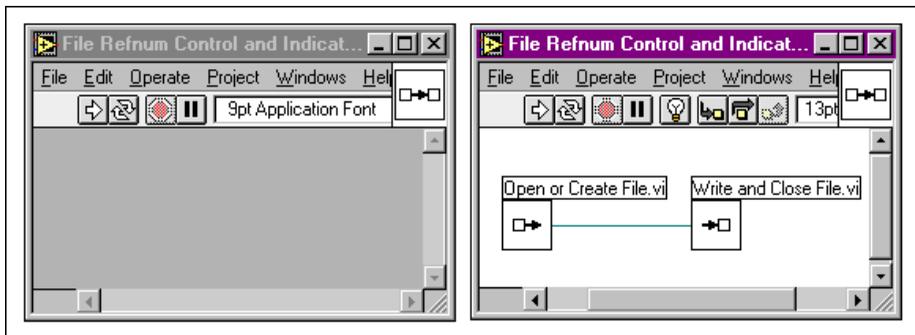
Note   *Use a refnum control or indicator if you want to pass a refnum into or out of a VI.*

13

List and Ring Controls and Indicators

This chapter describes the listbox and ring controls and indicators, which are available from the **Controls»List & Ring** palette, shown in the following illustration.
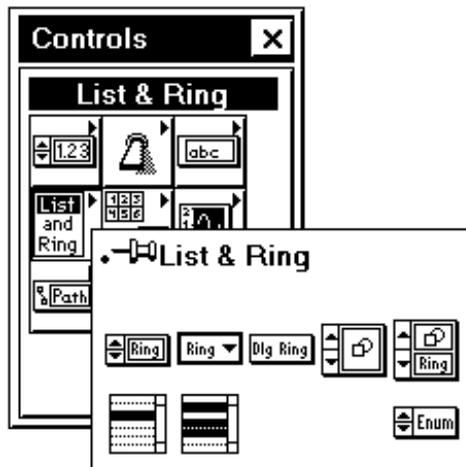
The **List & Ring** palette contains eight controls—Text Ring, Menu Ring, Dialog Ring, Pict Ring, Text & Pict Ring, Single Selection Listbox, Multiple Selection Listbox, and Enumerated Type. This chapter discusses the ring controls, listbox controls, and the enumerated type controls, in that order. For information about the attributes of listbox and ring controls, see Chapter 22, *Attribute Nodes*.

Listbox Controls

Listbox controls present users with a list of items. An example is shown in the following illustration. There are two types of listbox controls—Single Selection in which only one item can be selected, and Multiple Selection, in which one or more items can be selected.
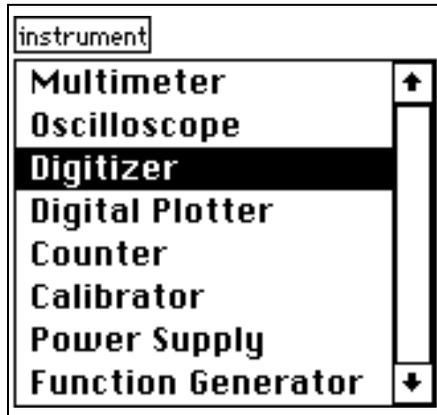
In addition to displaying a list of text items, you have one of several different symbols drawn next to each item (as in the **Save** dialog box, where directories and files have different symbols). You can also disable individual items, and set separator lines between items. You detect the currently selected item(s) by reading the value of the control. By using an attribute node, you also detect which item, if any, the user double-clicked. Finally, you can set the item strings, symbols and whether items are disabled using the attribute node.

Creating a List of Items

When you put a listbox on a front panel, it contains no items. You create the list of items in edit mode by typing them with the Labeling tool, or in run mode by using the attribute node. Separate individual items by entering a carriage return. If you can determine the items when you create the VI, you might want to type in the items while in edit mode. If you can determine the items at run time (for example, a list of files), use the attribute node.

Selecting Listbox Items

There are three ways to select items in a listbox—by using the mouse, by using the arrow keys, or by typing part of the name of the item you want.

•   Mouse—Use the Operating tool to select items. With the Multiple Selection Listbox, you select more than one item by `<Shift>`-clicking additional items.

•   Arrow keys—Use the arrow keys to select items. To add the currently highlighted item to the list of selected items, press the spacebar. You can deselect a currently selected item by using the arrow keys to move to the item and then pressing the spacebar.

•   Typing part of the name of the item you want—Type the name of the item to select the item. You also can use the left or right arrow keys to go to the previous or next items that match the typed letters.

Listbox Data Types

The data type for a Single Selection Listbox is Int32. The data value for the Single Selection Listbox is a number that represents the currently selected item, with the first item having a value of zero. If no item is selected, the value is –1.

The Multiple Selection listbox is an array of Int32s, where the value(s) in the array represent the currently selected item(s). If no item is selected, the value is an empty array.

Listbox Pop-Up Menu Items

The listbox pop-up menu includes the items shown in the following illustration. The items unique to listbox controls are described in this section.
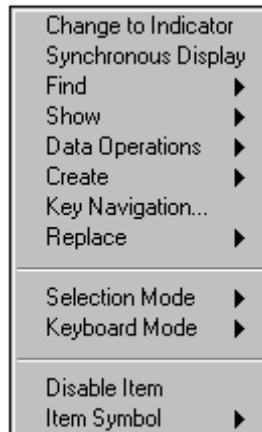
Show

You use **Show** from the pop-up menu, shown in the following illustration, to select which components of the listbox are visible. You also can choose to show or hide the label, the scrollbar, and the symbols.
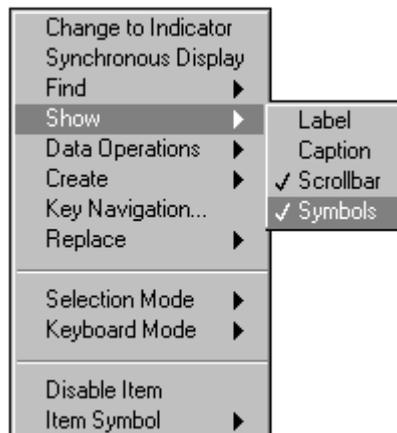
If you select **Show»Symbols**, the listbox adds an extra column to the left side of the list for symbols. An example of this is shown in the following

illustration. Initially, no items have symbols. You can add symbols using
Item Symbol or **Attribute Node**.

Selection Mode

You use **Selection Mode** from the pop-up menu to indicate how many
items you can select at once.

For a Single Selection Listbox, selection mode items configure the listbox
to support either zero or one selection, or one selection, as shown in the
following illustration.
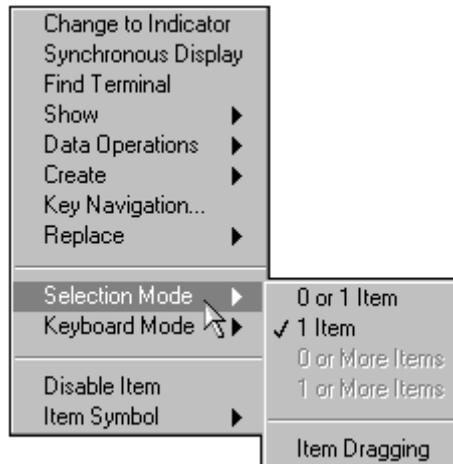
If you select **1 Item** (the default setting), then one item in the listbox always
is selected (highlighted). When you click a different item, the new item is
selected (highlighted), and the previous item is deselected. If you select
0 or 1 Item, the behavior is the same, except that a user can deselect the
current item by <Shift>-clicking it, leaving no items selected. You can use
Item Dragging to move items in the listbox by dragging them with the
mouse.

In a Multiple Selection Listbox, you have the same selection mode items as the Single Selection Listbox. In addition, you can indicate multiple items be selected, as shown in the following illustration.
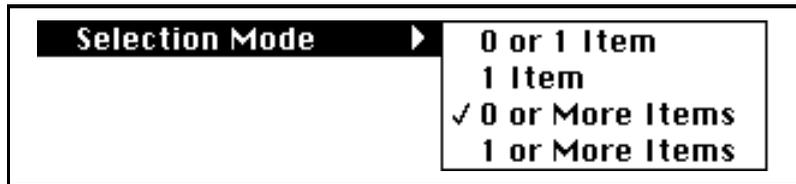
By default, the Multiple Selection listbox has a selection mode in which the user can select **0 or More Items**. You also can select **1 or More Items** to put it into a multiple selection mode where at least one item must be selected.

Keyboard Mode

You use **Keyboard Mode** from the pop-up menu, shown in the following illustration, to indicate how the listbox treats upper and lower case characters when you select items by typing their initial letters.
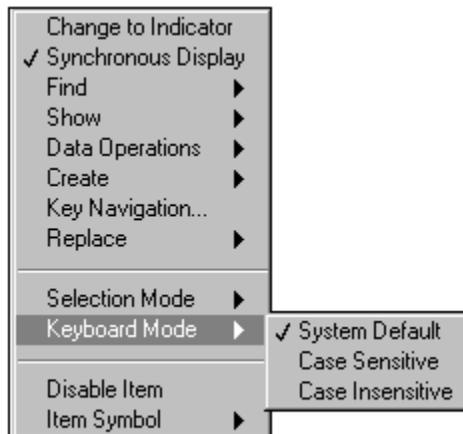
If you select **Case Sensitive**, an item is selected only if the uppercase and lowercase letters of what the user types exactly match the item. If you select **Case Insensitive**, case is ignored. If you select **System Default** (the default setting for a listbox), the listbox behaves the same way other listboxes do for a given platform. Specifically, in Windows and on the Macintosh, the listbox is case insensitive. In UNIX, the listbox is case sensitive.

Disable Item

You can disable a listbox item by popping up on an item and selecting **Disable Item** from the pop-up menu. To enable a disabled item, pop-up on the item and select **Enable Item**.

To enable/disable item programatically, use the **Disabled Items** attribute node (which is an array of items) as shown in the following illustration.

Item Symbol and Dividing Line

You can set the symbol of a listbox by popping-up on the item and selecting one of the items from the **Item Symbol** submenu. Selecting the item in the lower right corner replaces the list item with a dividing line.
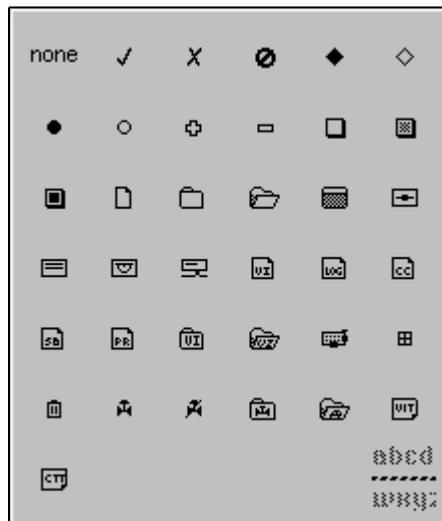
To set list item symbols programatically, use the **Item Symbols** attribute node (which is an array of item symbols) shown in the following illustration.

You can specify item symbols using the **Listbox Symbol Ring** constant in the **Functions»Numeric»Additional Numeric Constants** palette.

If you use a value of –1 for the symbol of an item, the listbox draws a grayed-out dividing line instead of drawing the name of the item.

Ring Controls

Rings are special numeric objects that associate numeric values with strings, pictures, or both. The different styles of rings are shown in the following illustration.
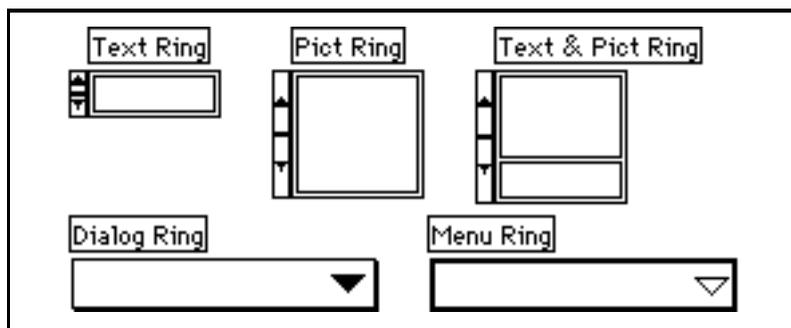
Rings are useful particularly for selecting mutually exclusive items, such as trigger modes. For example, you might want users to be able to choose from continuous, single, and external triggering, as shown in the following illustration.
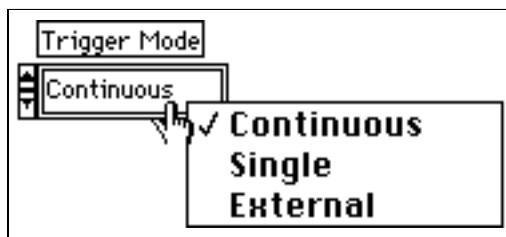
In the preceding example, the ring labeled Trigger Mode has three mode descriptors, one after another, in the text display of the ring. G arranges items in a circular list (like a Rolodex) with only one item visible at a time. Each item has a numeric value, which ranges from zero to $n – 1$, where n is the number of items (three in this example). The value of the ring, however, can be any value in its numeric data range. It displays the last item

(`External`, above) for any value greater than or equal to two and the first
item (`Continuous`, above) for any value less than or equal to zero.

You can pop up and select **Show»Digital Display** to display the numeric
value associated with the current item of a ring.

Users can use this ring to choose an item from an easy-to-understand
list without having to know what value that item represents. The value
associated with the selected item passes to the block diagram, where, for
example, you can select a case from a Case Structure (conditional code)
that carries out the selected item.

You can select an item in a ring control two ways. 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 also can select any item directly by clicking the ring
with the Operating tool and then choosing the item you want from the menu
that appears. However, the menu ring looks and operates like a pull-down
menu. It has no increment buttons. You select an item by clicking the ring
and selecting an item from the menu that appears.

Adding Text Items to Rings

A new text ring has one item with a value of zero and a display containing
an empty string. You enter or change text in the ring text area as you do
with labels, using the Labeling tool. Press the `<Enter>` **(Windows** and
HP-UX) or `<Return>` **(Macintosh** and **Sun)** key or click outside the text area
to finish typing. **Add Item After** from the text ring pop-up menu creates
a new empty item following the current one. **Add Item Before** inserts a
new item in front of the current item. If you are editing item 0 when you
select **Add Item After**, for example, item 1 is created, ready for you to
enter text for the new item. You also can press `<Shift-Enter>` **(Windows**
and **HP-UX)**, or `<Shift-Return>` **(Macintosh** and **Sun)** after typing in an
item to advance to a new 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 numeric values of the items
adjust automatically.

To disable ring items while editing, pop up on a ring control and select
Disable Item as shown in the following illustration. Then select the item
you want to disable.
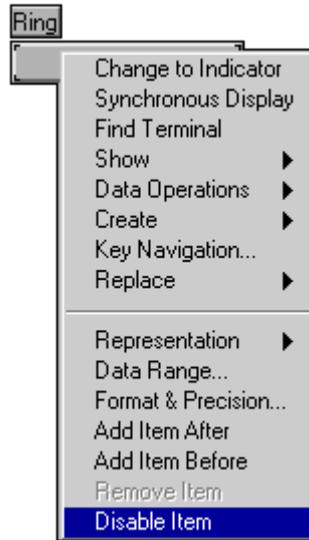
If you want to enable a disabled item while editing, pop up and select
Enable Item. To disable/enable a ring item programmatically, use the
Disable Items attribute node (which is an array of indices) shown in the
following illustration.

Adding Picture Items to Rings

A new Pict Ring or Text & Pict Ring has one item with an empty picture
display. You must copy a picture to the Clipboard before you can import
the picture into a Pict Ring. When you have a picture on the Clipboard,
pop up on the Pict Ring and select **Import Picture**. To add another
picture, copy it to the Clipboard, then pop up on the Pict Ring and
select **Import Picture Before** or **Import Picture After**, as shown in the
following illustration.

Pict Ring

| Change to Indicator |
| Synchronous Display |
| Find Terminal |
| Show ▶ |
| Data Operations ▶ |
| Create ▶ |
| Key Navigation... |
| Replace ▶ |
| Representation ▶ |
| Data Range... |
| Format & Precision... |
| Import Picture |
| Import Picture After |
| Import Picture Before |
| Remove Item |
| Disable Item |
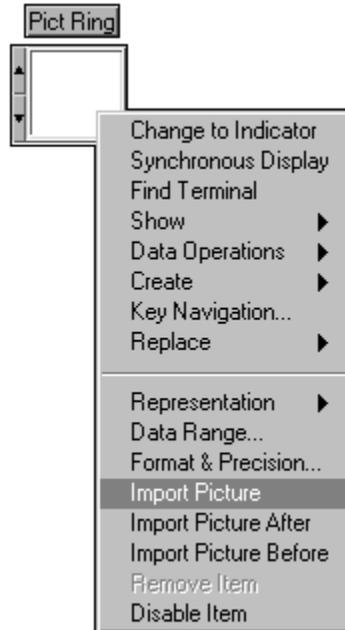
Changing the Size and Text of Text & Pict Rings

You can enlarge the Text & Pict Ring to make more room for the contents, just as you can enlarge all controls. You also can decrease the size. When you resize the Text & Pict Ring from one of its bottom corners, the height of the text area changes. When you resize the Text & Pict Ring from one of

its top corners, the height of the picture area changes. These resizing capabilities are shown in the following illustration.
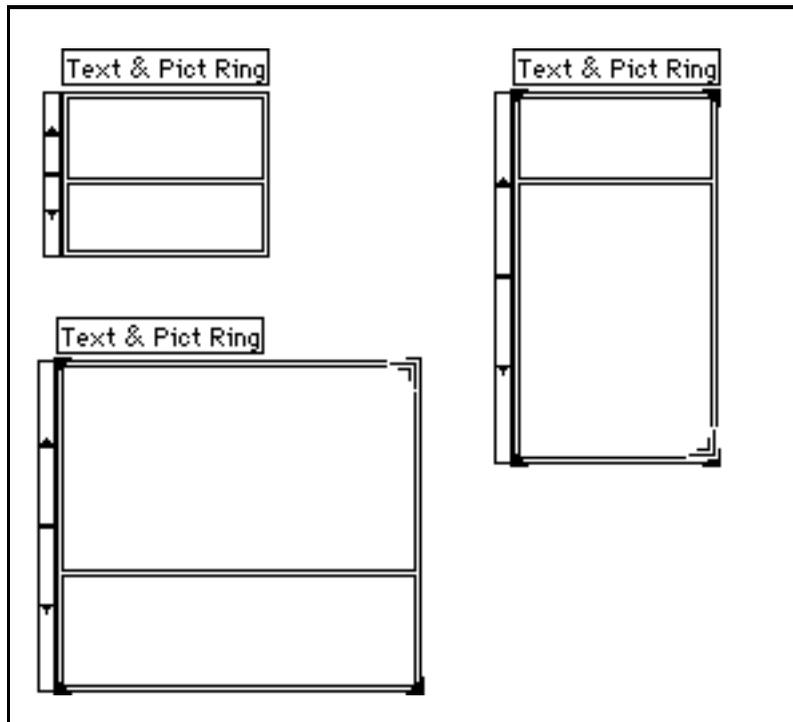
Enumerated Type Controls

An enumerated type control is similar to a text ring control. If data from an enumerated type is connected to a Case Structure, however, the case displays the string, or text of the item, instead of a number. The enumerated type data type is unsigned byte, unsigned word, or unsigned long, selectable from the **Representation** palette. An example is shown in the following illustration.
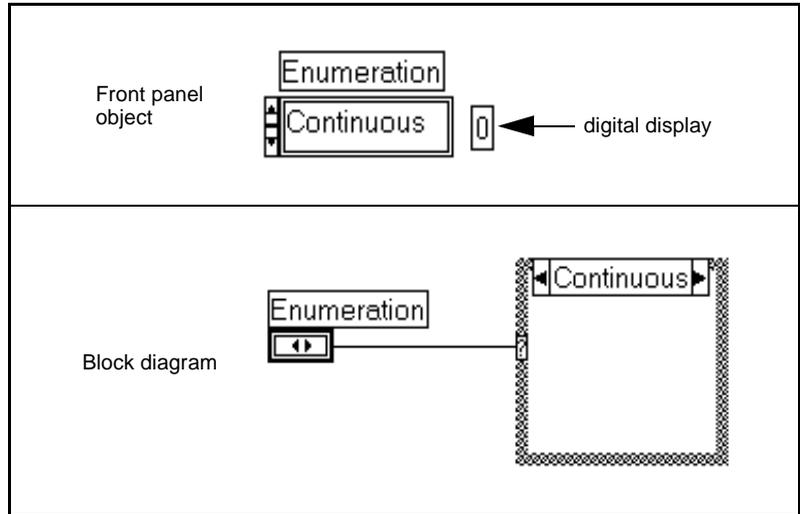
One advantage of using enumerated types instead of ring controls is if you use an enumerated type on the connector pane of a subVI, then pop up to create a constant, control, or indicator, you create an enumeration with the proper strings.

You enter items into an enumerated type the same way you enter items into a ring, using **Add Item After** or **Add Item Before**. Or, you can use the Labeling tool. Press `<Shift-Enter>` **(Windows** and **HP-UX)**, or `<Shift-Return>` **(Macintosh** and **Sun)** to enter a new item. Click any area outside the enumerated type when you finish entering items.

If an enumerated type is wired to a Case Structure, the Case Structure must have cases which handle every item in the enumerated type. If the Case Structure has a Default case, it handles items not explicitly handled by other cases.

All arithmetic operations except Add One and Subtract One treat the enumerated type the same as an unsigned numeric. Add One increments the last enumeration to the first, and Subtract One decrements the first enumeration to the last. In all other respects, the enumerated type is treated as an unsigned value. When coercing a signed integer to an enumerated type, negative numbers are pinned to the first enumeration, and out-of-range positive numbers to the last enumeration. Unsigned integers always pin to the last enumeration value.

If you connect a numeric value to an enumerated type indicator, the number is converted to the closest enumeration item, with out-of-range numbers handled as above. If you connect an enumeration control to a numeric value, the value is the enumerated type index. To wire an enumeration type to an enumeration indicator, the enumeration items must match, although the indicator might have additional items beyond the items in the type.

14

Array and Cluster Controls and Indicators

This chapter describes how to use arrays and clusters. You access arrays and clusters from the **Controls»Array & Cluster** palette, shown in the following illustration.
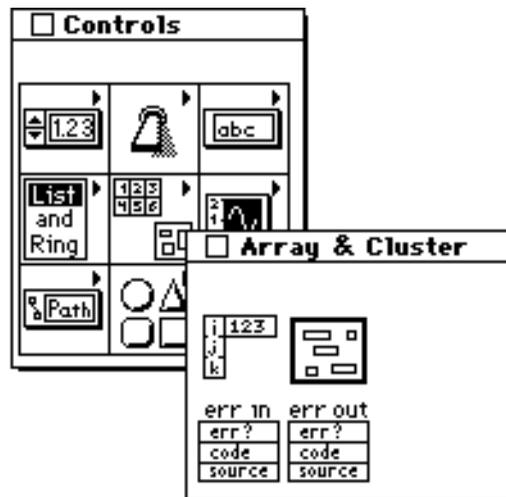
See the examples in `examples\general\arrays.llb`.

You can use **Help»Online Reference»Function and VI Reference** to find descriptions of G functions, many of which operate on arrays in addition to scalars. The *Array Functions* and the *Cluster Functions* topics describe the functions exclusively designed for array and cluster operations.

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	11-Element Array
0	Melissa
1	Greg
2	Gregg
3	Don
4	Duncan
5	Thad
6	Dean
7	Stepan
8	Kate
9	Mary
10	Mark

A simple example of an array is a list of names, represented in G 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   0   1   2   3   4   5   6   7   8

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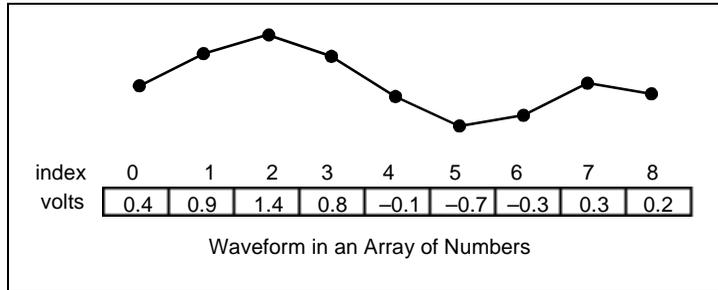
A more complex example is a graph represented as an array of points where each point is a pair of numbers representing the X and Y coordinates, as shown in the following illustration.

index	0	1	2	3	4	5	6
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
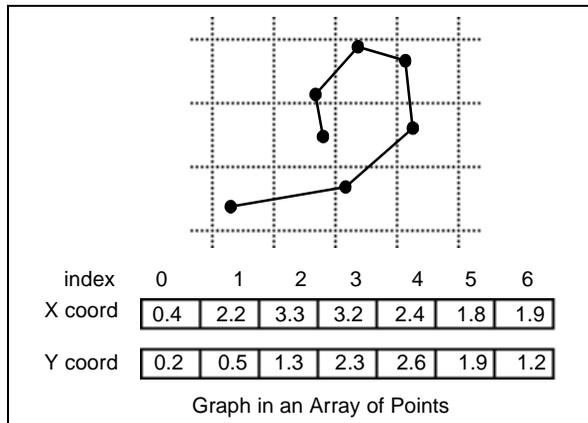
The preceding examples are *one-dimensional* (1D) arrays.
A *two-dimensional* (2D) array requires two indices to locate an element—a column index and a row index, both of which are zero-based. In this case, we speak of an N column by M row array, which contains N times M elements, as shown in the following illustration.

6 Column by 4 Row
Array of 24 Elements
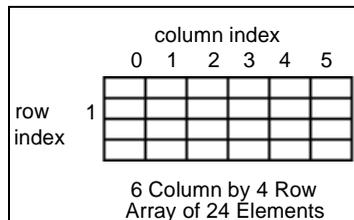
A simple example is a chess board. There are eight columns and eight rows for a total of 64 positions, each of which can be empty or have one chess piece. You can represent a chess board as a two-dimensional array of strings. Each string has the name of the piece occupying the corresponding location on the board, or it is 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 the one-dimensional array examples can be generalized to two dimensions. A collection of waveforms represented as a two-dimensional array of numbers is shown in the following illustration. The row index selects the waveform and the column index selects the point on the waveform.

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

Multiple Waveforms in a 2D Array of Numbers
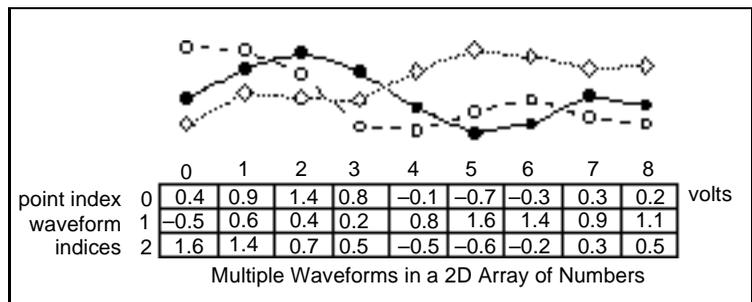
Arrays can have an arbitrary number of dimensions, but one index per dimension is required to locate an element. The data type of the array elements can be a cluster containing an assortment of types, including arrays whose element type is a cluster, and so on. You cannot have an array of arrays. Instead, use a multidimensional array or an array of cluster of arrays.

Consider using arrays when you work with a collection of similar data. Arrays are frequently helpful when you perform repetitive computations or I/O functions. Using arrays can make your application smaller, faster, and easier to develop because of the large number of array functions and VIs in G.

Creating Array Controls

You create an array control by first selecting an array from
Controls»Array & Cluster, as shown in the following illustration.
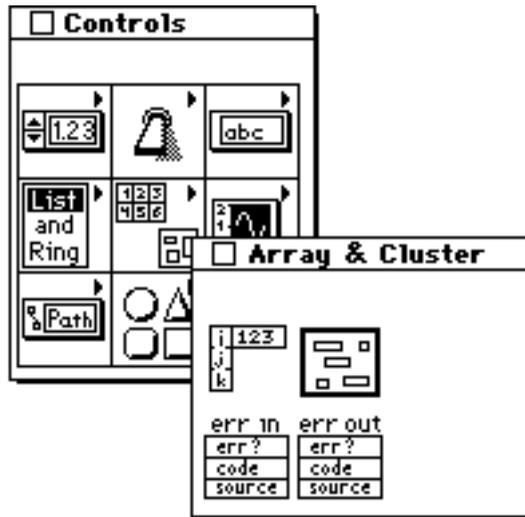
You create an array control or indicator by combining an *array shell* from
the **Array & Cluster** palette, as shown in the preceding illustration, with
a valid *element*, which can be a numeric, Boolean, string, path, refnum or
cluster. The element cannot be another array or a chart. Also, if the element
is a graph then only the graph data types containing clusters instead of
arrays at the top level are valid.

Selecting an array from the **Array & Cluster** palette places an array shell
on the front panel, as shown in the following illustration.
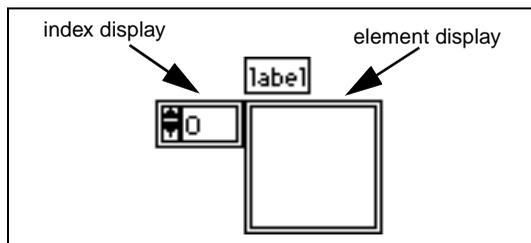
A new array shell has one *index display*, an empty *element display*, and an
optional label.

There are two ways to define an array type. Drag a control or indicator of the desired type into the element display window, or deposit the control or indicator directly by popping up on the front panel, selecting a control, and dragging it into the element display. Either way, the control or indicator you insert fills the empty display.

For example, if you want to define an array of Booleans, pop up, select a Boolean control, and drag it into the element display as shown in Figures 14-1 through 14-4.
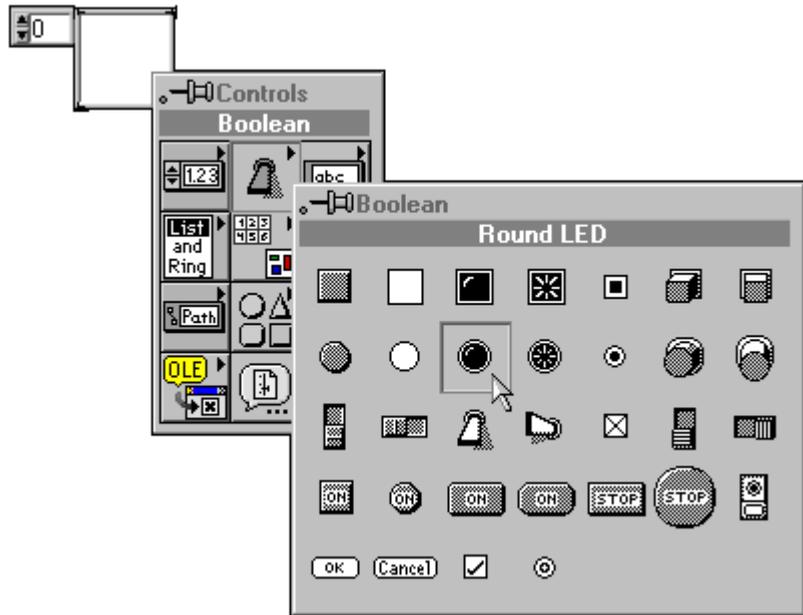
Figure 14-1. Pop up in the element display and select the Boolean control.
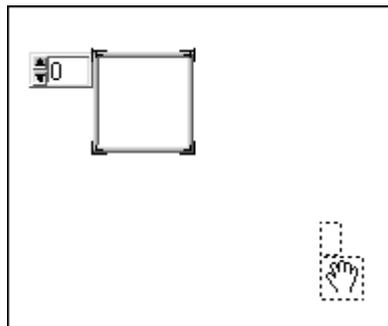
Figure 14-2. Release the mouse button. The menu disappears and the cursor changes to the Scroll Window icon.
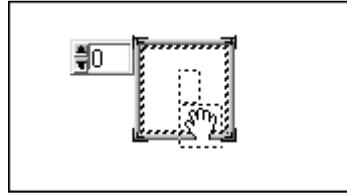
Figure 14-3. Drag the Scroll Window icon into the element display.
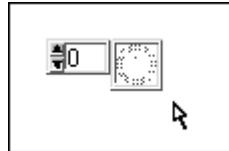
Figure 14-4. Click the mouse button. The control is deposited inside the display.

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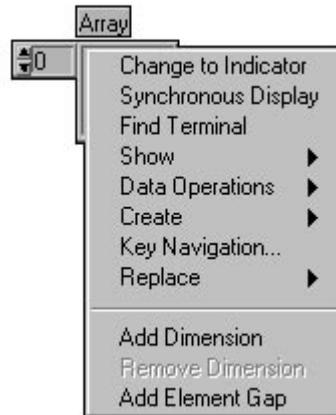
Array Dimensions

A new array has one dimension and one index display. You resize the index display vertically or select the **Add Dimension** item from the index display pop-up menu to add a dimension. You shrink the index vertically or select **Remove Dimension** to delete it. An additional index display appears for each dimension you add. Two methods for resizing are shown in the following illustrations.
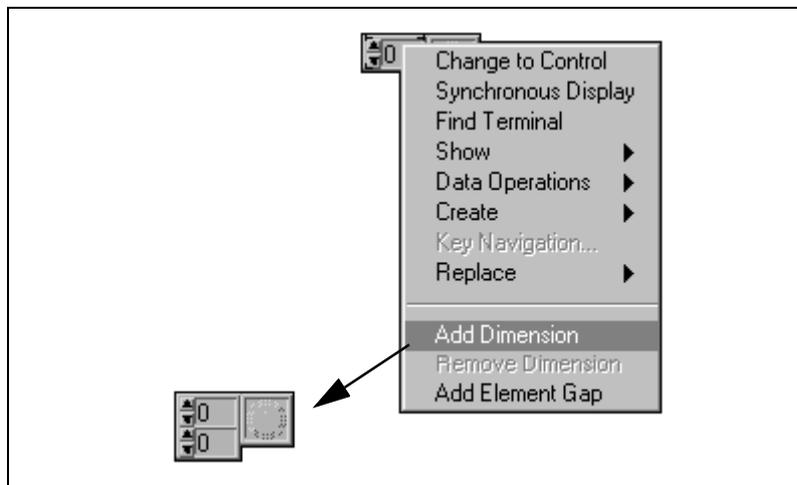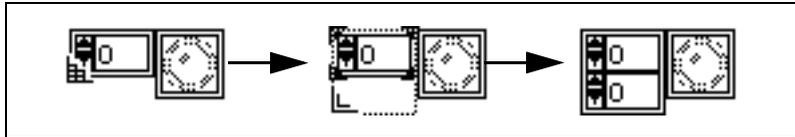
The dimension of an array is a choice you make when you decide how many identifiers it takes to locate an item of data. For example, to locate a word in a book you might go to the sixth word on the twenty-eighth line on page 192. Three indices, 6, 28, and 192 specify the word, so one possible representation for this book is a three-dimensional (3D) array of words. To locate a book in a library you indicate the position on the shelf, which shelf, which bookcase, which aisle, and which floor. If you represented this as an array it uses five dimensions. The location of a word in a library then requires eight indices.

Notice that when you add a dimension to an array, the wire changes in style, thickness and color. For more information see Chapter 18, *Wiring the Block Diagram*.

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 following illustration.
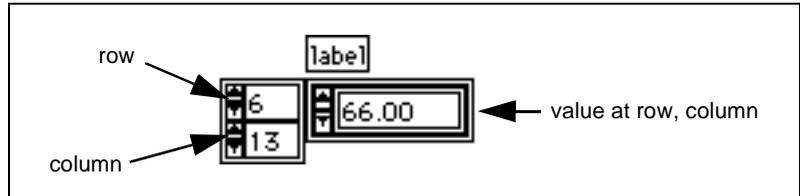
For example, suppose you have an array of 3 rows and 4 columns, with values as shown in the following figure.

0	1	2	3
4	5	6	7
8	9	10	11

Rows and columns are zero-based, meaning the first column is column 0, the second column is column 1, and so forth. Changing the index display to row 1, column 2 displays a value of 6, as shown in the following illustration.
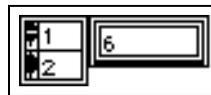
If you try to display something out of range, the value display is grayed-out to indicate there is no value currently defined. For example, if you try to display an element in row 10 of the array shown previously, the display is grayed-out as shown in the following illustration.
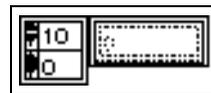
Array Resizing
tool

Usual Resizing
symbol

Displaying Arrays in Single-Element or Tabular Form

A new array appears in single-element form. The array displays the value of a single element—the one referenced by the index display. You also can display the array as a table of elements by resizing the array shell from any of the four corners surrounding the element. The Array Resizing tool is slightly different from the usual Resizing tool when it is over the array shell resizing handles, and the cursor changes to the usual symbol when you begin to resize. An example is shown in the following illustration.
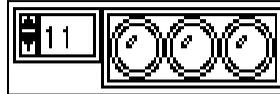
The following illustration shows how to resize a one-dimensional array either vertically or horizontally to show more elements simultaneously.
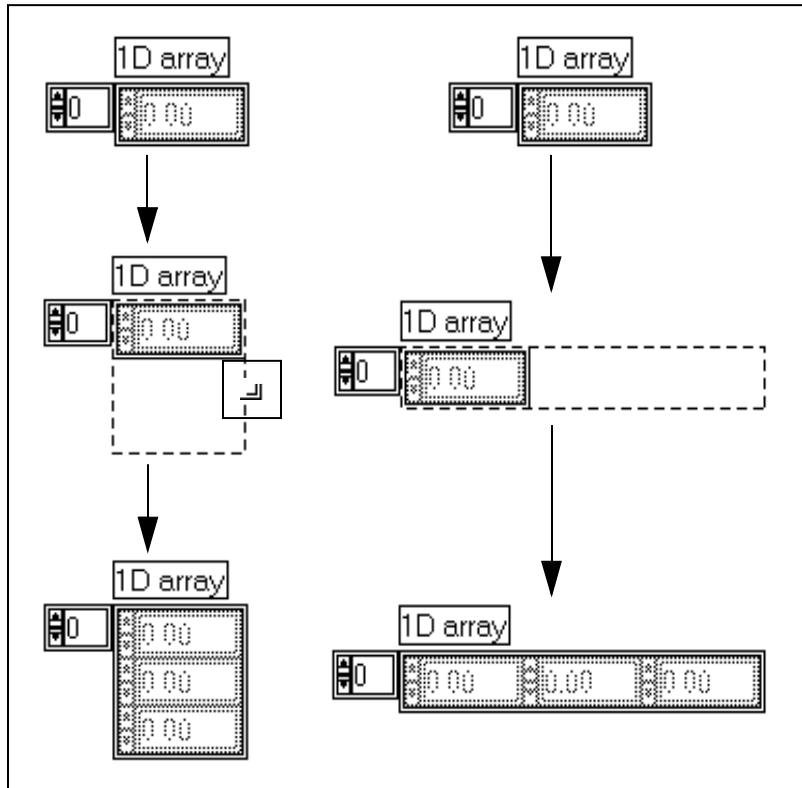
The index display contains the index of the element in the left or upper left corner of the table. By changing the index, you can display different

sections of a large or multidimensional array. For one-dimensional arrays, the index identifies the column of the left-most visible element. For two-dimensional 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 array as a book of pages 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 three of the array.
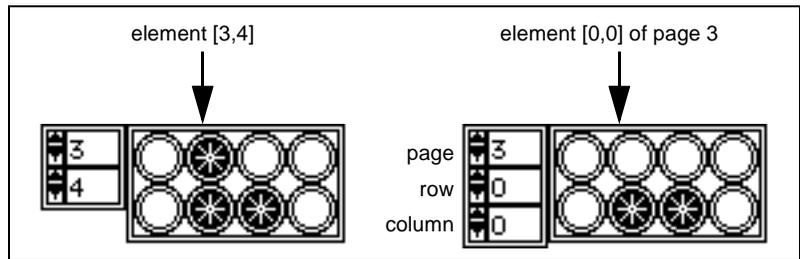
If you want to hide the array indices, pop up on the outer frame and turn off the **Show»Index Display** item from the array pop-up menu. The following illustration shows an example of what a tabular array looks like without the index display.
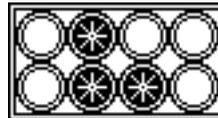
If you want to display an array in an orientation different from the tabular array format, display the elements in a cluster on the front panel and process them in an array. To do this, use the Array To Cluster and Cluster To Array functions explained in the **Online Reference»Function and VI Reference»Cluster Functions** topic.

Operating Arrays

You operate an element of the array the same as 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 value of the element.

Default Sizes and Values of Arrays

You cannot limit the size of an array by a fixed number of elements. However, when you set the default values of an array control, you also can set the default size. *Do not make the default size larger than necessary.* If you set an array to have a large default size, all of the default data saves 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 cannot be used in a program. After you deposit a control in the array, it is an *empty array* (its length is 0) with the defined type of the deposited control. Although it also has no defined elements, you still can use it in the VI. The array control dims all elements, indicating they are undefined.

The following illustrations show an array control in single-element form wired to an array indicator in tabular form with five elements displayed. The first figure shows both arrays are empty (element values are undefined).
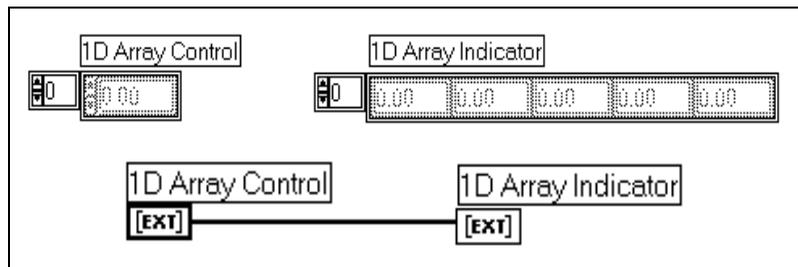
If you set the index display of the array control to 9 and set the element value at index 9 to 5.4, the data in the array expands to 10 elements (0 to 9). The value of element 9 is the value you set, 5.4 in this example. G assigns to the other elements the default value of the digital control, 0.00. If you now select **Make Current Value Default** from either the shell or the pop-up menu of the array control, the default data of the array now consists of the specified 10 elements. After running the VI, the indicator displays the same values.
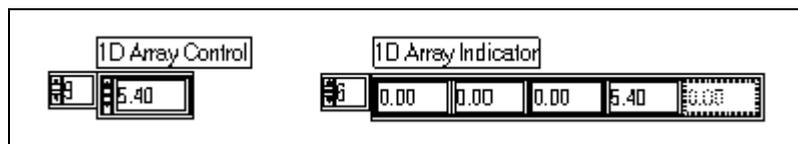
Selecting **Reinitialize to Default** for an element resets that element only to its default value.

Now assume you change the default value of a numeric element from 0.0 to 1.0. 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**. 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, but elements 10 and 11 have the new default value of 1.0, and element 12 has the assigned value of 7.0. This example is shown in the following illustration.
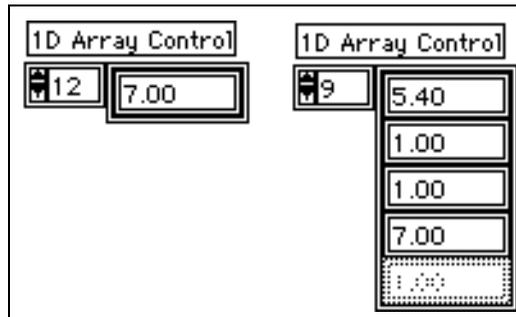
If you now select **Reinitialize to Default** from either the shell or the index pop-up menus, the array reverts to the 10-element array shown in the previous example.

To increase the size of an array from [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 **Data Operations»Empty Array** command from the index array pop-up menu, then set the array and values to the size you want, or select the unwanted subset of the array and select the **Data Operations»Cut Data** item from the pop-up menu, as explained in the following sections.

Array Elements

When in run mode, use the `<Tab>` key either to move the key focus between front panel controls or between elements within a single array. Initially, the `<Tab>` key moves the key focus between front panel controls. To move it between the elements within a specific array, first `<Tab>` to that array. Then use the `<Ctrl>` **(Windows)**; `<command>` **(Macintosh)**; `<meta>` **(Sun)**; or `<Alt>` **(HP-UX)** key and the down arrow, to move the key focus inside the array. You can now use the `<Tab>` key to sequence through the array elements and the array indices. To return tabbing between controls, use the `<Ctrl>` **(Windows)**; `<command>` **(Macintosh)**; `<meta>` **(Sun)**; or `<Alt>` **(HP-UX)** key and the up arrow.

Finding the Size of Arrays

To find the size of an array control or indicator, select **Data Operations»Show Last Element** from the index array pop-up menu.

Moving or Resizing Arrays

Always move the array by clicking the shell border or the index display and then dragging it. If you drag the element in the array, it separates from the array, because the shell and elements are not locked together. If you inadvertently drag the element when you meant to drag the shell, cancel the operation by dragging the element back into the shell, or past the front panel window border, or by using the `<Esc>` key before releasing the mouse button. You cancel a resizing operation by dragging the border past the front panel window before releasing the mouse button. If the array is part of the current selection, then grabbing an element drags the array.

Resize the array index vertically from any corner, or horizontally from the left.

Selecting Array Cells

To copy data from or paste data to arrays you must first select an area of the array. You can copy or paste by taking the following steps.

1. To select the data, set the array index to the first element in the data set you want to copy.

2. Then, select the **Data Operations»Start Selection** item from the array pop-up menu.

3. Next, set the array index to the last element in the data set you want to copy.

4. Then, select the **Data Operations»End Selection** item from the array pop-up menu.

5. Finally, select **Data Operations»Copy** or **Data Operations»Paste** to copy or paste the selected items.

When you are in run mode, these menu items are directly available when you pop up on an array. The selection process is detailed in the following example.

An Example of Selecting Array Cells

Begin by selecting **Show Selection** from the pop-up menu. Activate **Show Selection** for display of visibly marked selected cells.

A border surrounds selected cells. Select **Data Operations»Add Element Gap** from the pop-up menu. This separates the cells by a thin open space. Although it is not necessary to add the element gap, this open border slightly changes and might improve the appearance of the array. When you select cells, this open space is filled by a thick border, denoting the selected cells. If you do not add the element gap, the selection border appears over the selected cell edge.

You define the cells for selection with the array index. The following example uses a two dimensional array.

Set the array index for 1,1. Choose **Start Selection** from the pop-up menu. Set the array index to 3,3. Choose **End Selection** from the pop-up menu.
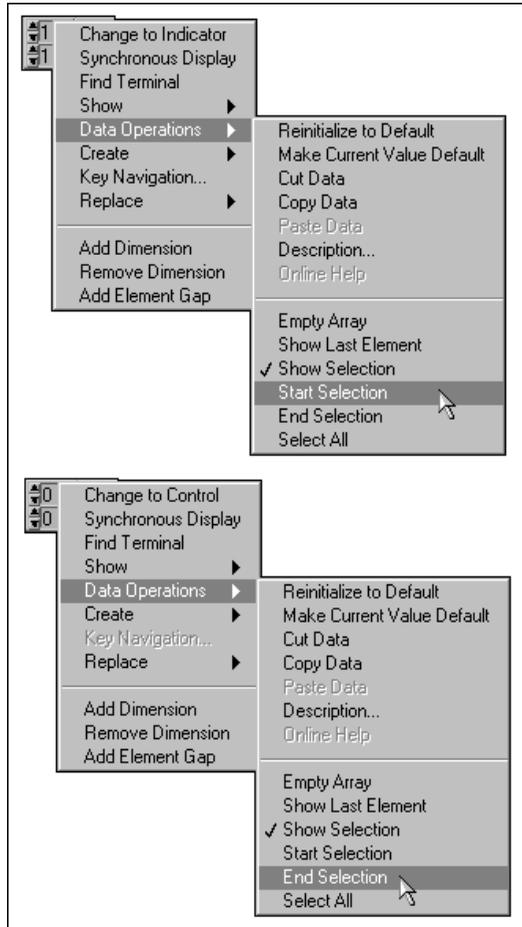
The selection includes the cells denoted by lower index numbers but not the cells denoted by the higher index numbers. If you want to include the cells from 1,1 to 3,3, set the index to 4,4 before choosing **End Selection**. Notice that cutting the selected items actually removes the entire set of rows and columns spanned by the selection, but places only the selected area into the clipboard.

After you select cells, cut or copy the data to paste into other cells. When you finish, notice one border line remains highlighted. This is an insertion point, and remains in the array. To hide this line, deselect the **Show Selection** item 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.

G Arrays and Arrays in Other Systems

Most programming languages have arrays, although few have the number of built-in array functions G contains. Other languages typically stop at the fundamental array operations of extracting or replacing an element, leaving it to users to build more complex operations. G has these fundamental operators, so you can map a program in another language directly to a VI in G. However, you usually can create a simpler and smaller diagram if you start again in G and use the higher-level array functions. The following example shows a fragment of a C program that sums the squares of the elements of an array, the direct translation to a G block diagram, and two redesigned versions more efficient than the translation.

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

G Direct Translation

G Auto-Indexing Version

G Optimal Version
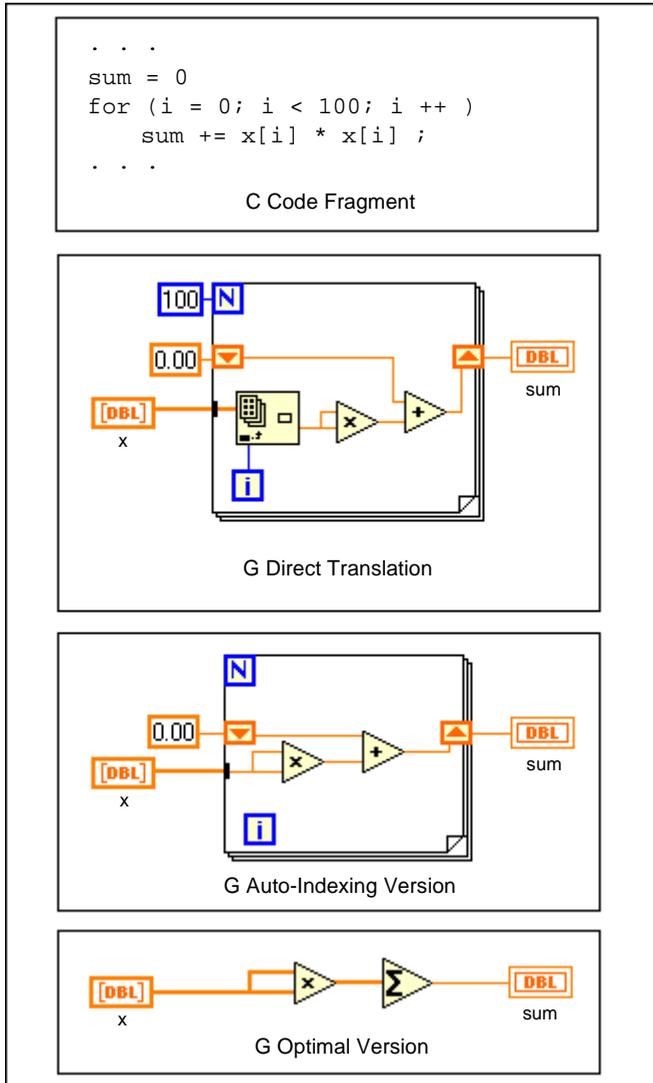
Index Array
function icon

The indexing operation for extracting an element is represented in C by
brackets [ ] following the array name and enclosing the index. In G, 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 G version. Disregard the +1 in the FORTRAN computation of J; the addition is necessary because FORTRAN arrays begin indexing with one rather than zero.

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

G Direct Translation
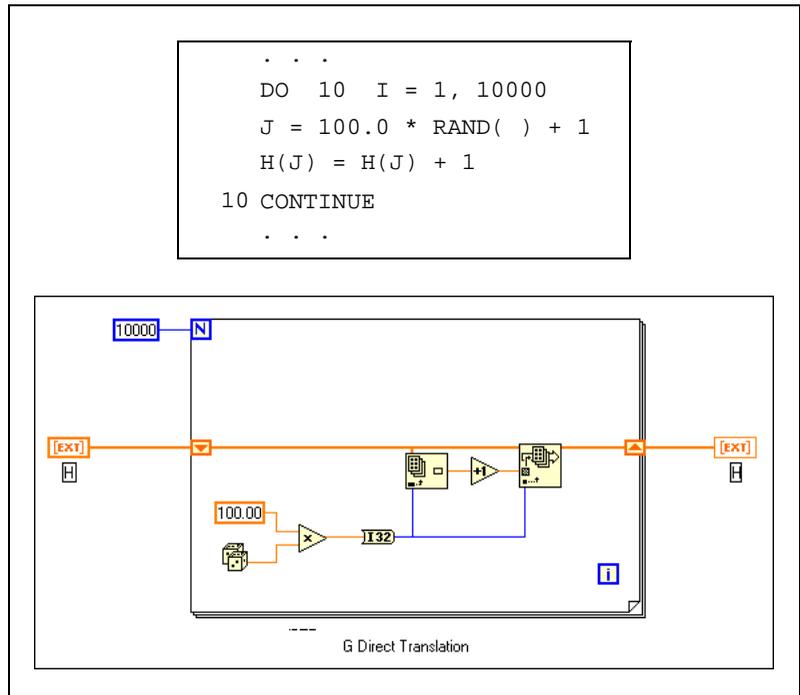
The indexing operation for inserting or replacing an element is represented in FORTRAN by parentheses( ) on the left side of the equal sign following the array name, and enclosing the index. In G, this operation is represented by the Replace Array Element function icon (shown at left). The array is wired 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.

Replace Array
Element function

Before you can use an array in most programming languages you first must declare it and, in some cases, initialize it. In G, 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 indicate 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 must keep track of the size yourself. In G, it is not necessary to declare a maximum size for arrays, because G automatically preserves the size information for the array.

Furthermore, with G, you cannot store a value outside the bounds of an array. In many conventional programming languages, there is no such checking, and you can corrupt memory inadvertently.

Clusters

A G 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 G 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 appearing 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.

Common Cluster Wire Pattern

Cluster of Numbers Wire Pattern

You can connect terminals only if they have the same type. For clusters, this means both clusters must have the same number of elements, and corresponding elements—determined by the cluster order—must match in type. G coerces numbers of different representations to the same type.

The Cluster Order is a numeric order given to elements in a cluster as described in the *Cluster Element Order* section later in this chapter.

For information on functions that manipulate clusters, see the **Online Reference»Function and VI Reference»Cluster Functions** topic.

Creating Clusters

You create a cluster control or indicator by installing any combination of Booleans, strings, charts, graph scalars, arrays, or even other clusters into a *cluster shell*. You access the cluster shell from the **Controls** palette, as shown in the following illustration.
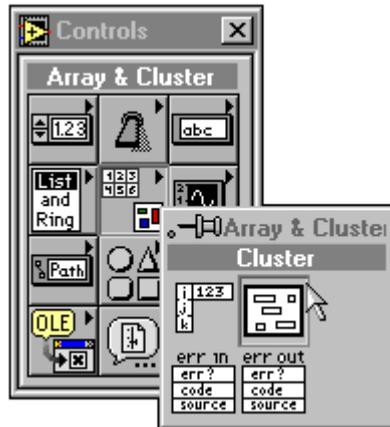
A new cluster shell has a resizable border and an optional label.

When you pop up in the empty element area, the **Controls** palette appears. You can place any element from the **Controls** palette in a cluster. You can drag existing objects into the cluster shell or create and deposit them directly inside by selecting them from the control pop-up menu and dragging them into the cluster shell. 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 from controls to indicators.

Operating and Configuring Cluster Elements

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

Cluster Elements

When in run mode, you can use the `<Tab>` key either to move the key focus between front panel controls, or between elements within a single cluster. Initially, the `<Tab>` key moves the key focus between front panel controls. To move it between the elements within a specific cluster, first `<Tab>` to that cluster. Then use the `<Ctrl>` **(Windows)**; `<command>` **(Macintosh)**; `<meta>` **(Sun)**; or `<Alt>` **(HP-UX)** key and the down arrow, to move between the cluster elements. To return tabbing between controls, use the `<Ctrl>` **(Windows)**; `<command>` **(Macintosh)**; `<meta>` **(Sun)**; or `<Alt>` **(HP-UX)** key and the up arrow.

Cluster Default Values

To set the default value of the entire cluster to the current value of each individual element, pop up on the cluster frame and select the **Data Operations»Make Current Value Default** command from the cluster pop-up menu. To reset all elements to their individual default configurations, select the **Data Operations»Reinitialize To Default Values** command from the cluster pop-up menu.

To change the default value of a single element in a cluster, pop up on that particular element and select **Make Current Value Default**.

Cluster Element Order

Cluster elements have a logical order 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…** item from the cluster pop-up menu. The appearance of the element changes, as shown in the following illustration.
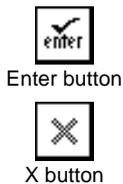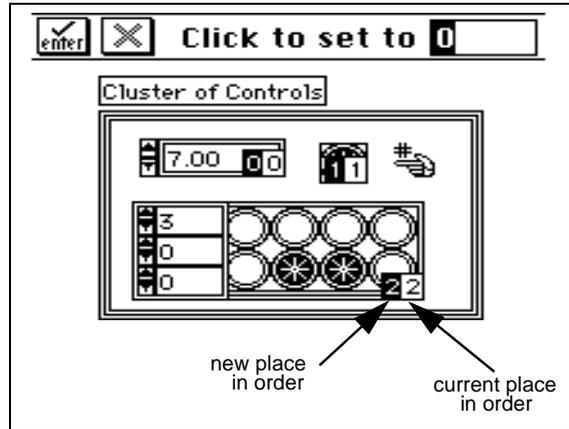
The white boxes on the elements show their current places in the cluster
order. The black boxes show the new position of the element in the order.
Clicking an element with the cluster order cursor sets the position of the
element in the cluster order to the number displayed inside the toolbar.
Change this order by typing a new number into that field and clicking the
corresponding control. When the order is as you want it, click the enter
button to set it and exit the cluster order edit mode. Click the X button to
revert to the old order.

Enter button

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.
For more information on these functions, see the **Online Reference»
Function and VI Reference»Cluster Functions** topic.

Moving or Resizing Clusters

Cluster elements are not fixed 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 of the shell, click the shell when
you want to move a cluster, and resize clusters from the shell border. If you
inadvertently drag an element when you meant to drag the whole cluster,
you can cancel the operation by dragging the element back into the shell
or past the front panel window border or by using the <Esc> key before
releasing the mouse button. You also cancel a resizing operation by
dragging the border past the front panel window or by using the
<Esc> key before releasing the mouse button.

You can shrink clusters to fit their contents by selecting **Autosizing** from
the pop-up menu, shown in the following illustration.
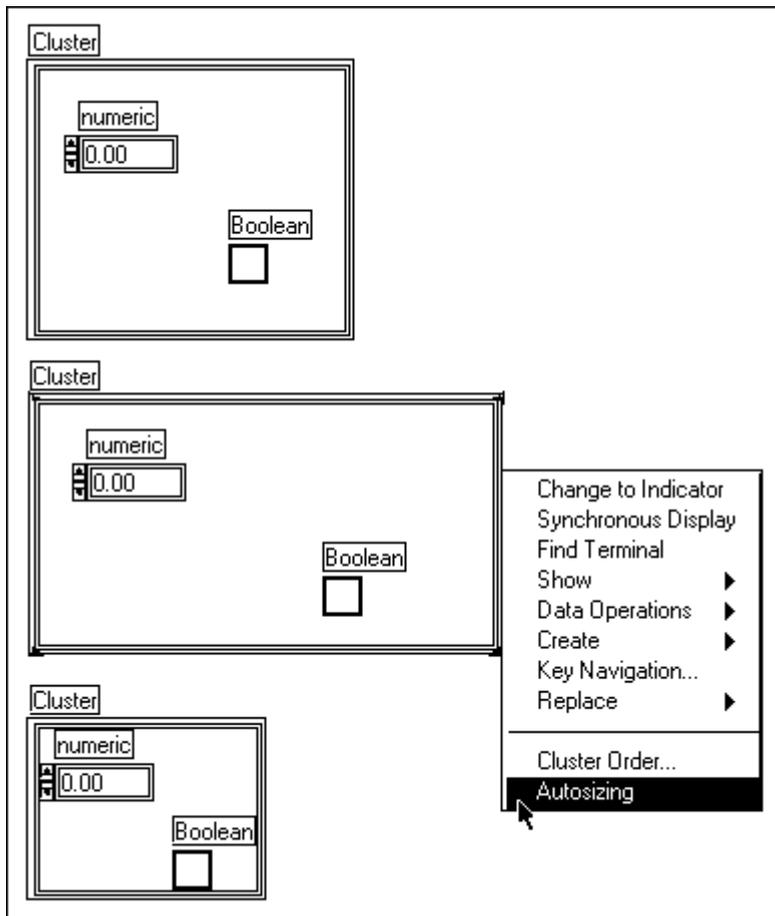
Cluster Assembly

G 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** palette, appears on the block diagram with two element input terminals on the left side. Resize the icon vertically to create as many terminals as you need. The element you wire to the top terminal becomes element 0 in the output cluster, the element you wire to the second terminal becomes element 1 in the output cluster, and so on.

As you wire to each input terminal, a symbol representing the data type of the wired element appears in the formerly empty terminal shown in the following illustration. You must wire all the inputs you create.
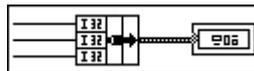
In addition to input terminals for elements on the left side, the Bundle function also has an input terminal for clusters in the middle as shown in the following illustration. You use this terminal to replace one or more elements of an existing cluster wire without affecting the other elements.
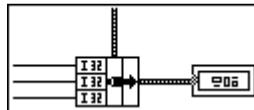
See the *Replacing Cluster Elements* section in this chapter for an example of using this function to replace elements in a cluster.

Bundle By Name Function

Bundle By Name function

The Bundle By Name function, also in the **Functions»Cluster** palette, is similar to the Bundle function. Instead of referencing fields by position, however, you reference them by name. Unlike the Bundle function, you can show only the elements you want to access. For each element you want to access, you must 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 use the Bundle By Name function only to replace elements by name, not to create a new cluster. You might, however, wire a data type cluster to the middle terminal, and then wire all of the new values as names to produce 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

must first connect an input cluster to the middle terminal of the Bundle By Name function.

After you wire the middle terminal, select elements you want to modify by popping up on any of the left terminals of the Bundle By Name function. You see 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.
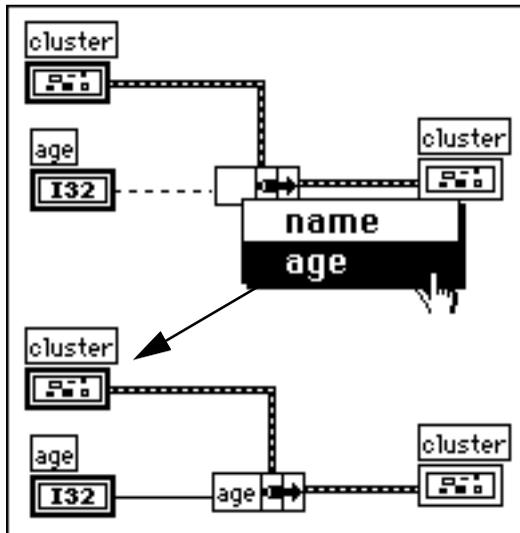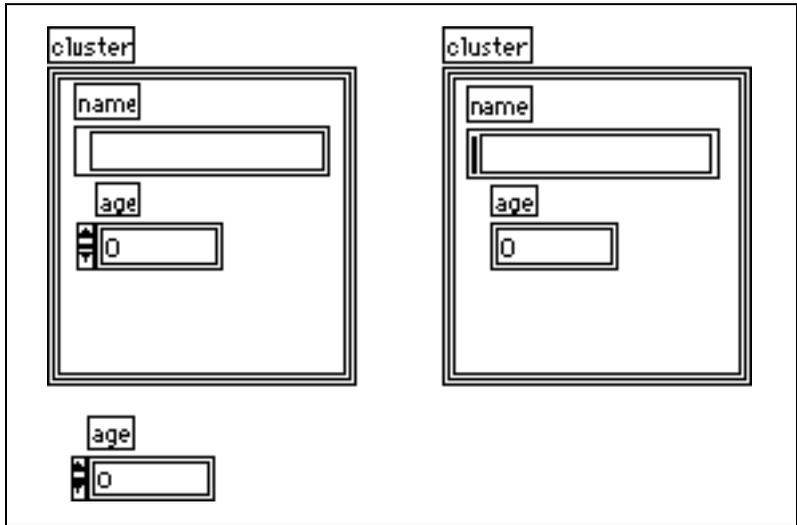
Resize the Bundle By Name function to show as many or as few elements as you need. This is different from the Bundle function, which requires you to size it to have the same number of elements as are in the resulting cluster.

If you pop up on an individual element of the Bundle By Name function when the cluster contains a cluster, you find a pull to the side submenu, **Select Items**, you can use 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 might change during the development process. If your modification involves the addition of a new element, or a reordering of the cluster, the change does not require changes to the Bundle By Name function, because the names are still valid. For example, if you modified the previous example by adding the new element and weight to the source and destination clusters, the VI is still correct. This example is shown in the following illustration.
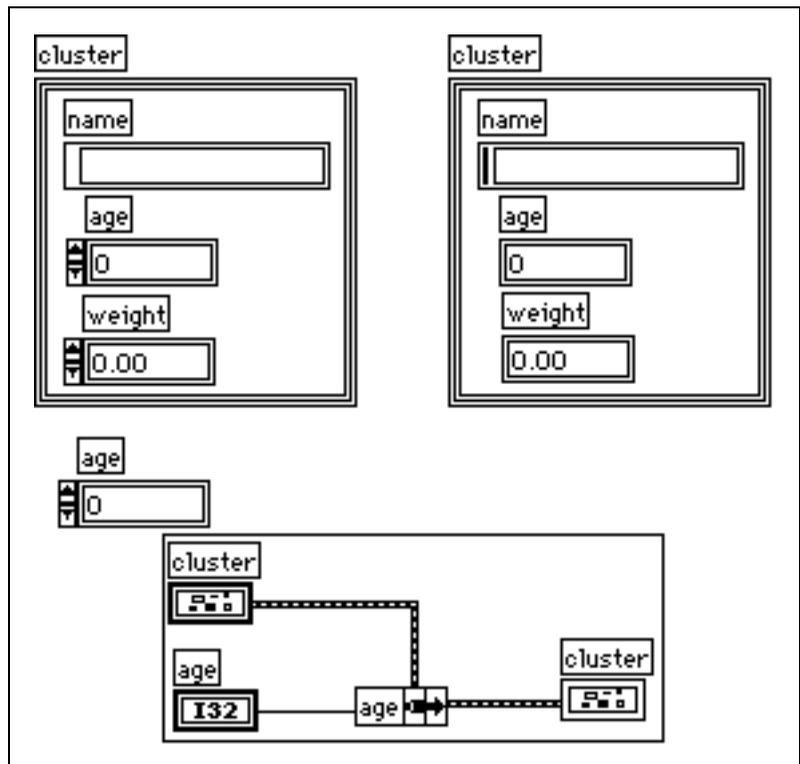
Bundle By Name is particularly useful in larger applications in conjunction with type definitions. By defining a cluster that might change as a type definition, any VI using that control can be changed to reflect a new data type by updating the file of the type definition. If the VIs use only Bundle 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 24, *Custom Controls and Type Definitions*, 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 **Cluster** or **Array** palettes of the **Functions** palette, converts the elements of a 1D array to elements of a cluster. This function is useful when you want to display elements of the same type within a front panel cluster indicator, but you also want to manipulate the elements by index value on the block diagram.

Use the **Cluster Size…** item from the function pop-up menu to configure the number of elements in the cluster. The dialog box that appears is shown in the following illustration.
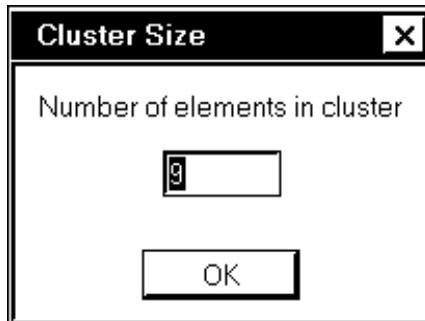
The function assigns the zero array element to the zero cluster element, and so on. If the array contains more elements than the cluster, the function ignores the remaining elements. If the array contains fewer elements than the cluster, the function assigns the extra cluster elements the default value for the element data type.

The following illustration 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 might 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. The loop shown simply counts the number of elements in the array set to true.
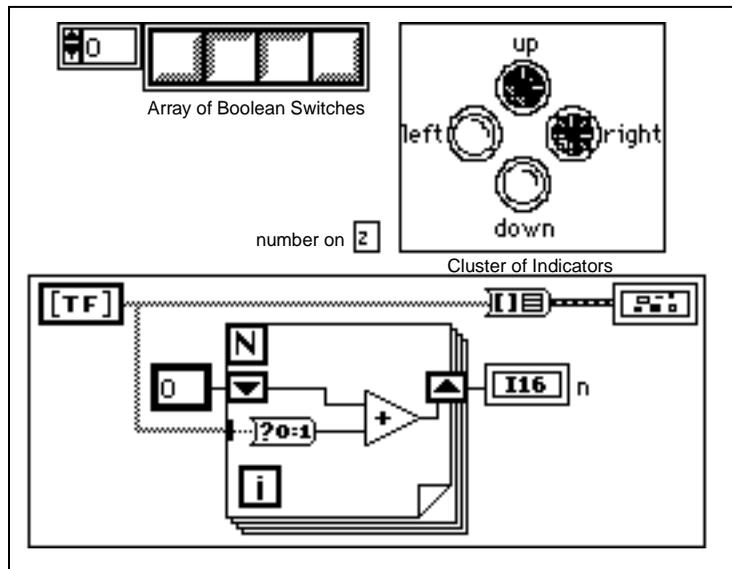
Cluster Disassembly

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

Unbundle Function
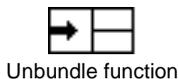
Unbundle function

The Unbundle function, which you obtain from the **Array & Cluster** palette of the **Functions** palette, has two element output terminals on the right side. You adjust the number of terminals with the Resizing tool the same way you adjust the Bundle function. Element 0 in the cluster order passes to the top output terminal, element 1 passes to the second terminal, and so on.

The cluster wire remains broken until you create the correct number of output terminals. When the number of terminals is correct, the wire becomes solid. The terminal symbols display the element data types, as shown in the following illustration.

Unbundle By Name Function
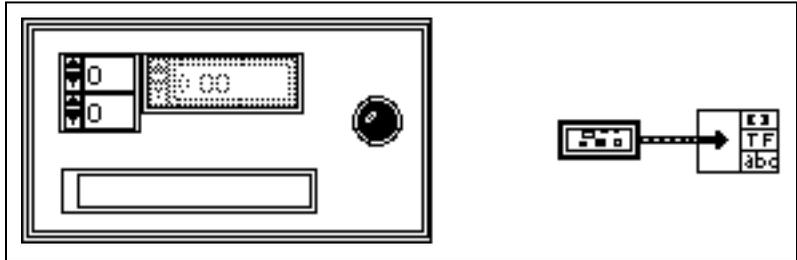
Unbundle By Name function
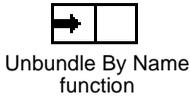
The Unbundle By Name function, also in the **Array & Cluster** palette of the **Functions** palette, is similar to the Unbundle function. Instead of referencing fields by position, however, you reference them by name. Unlike the Unbundle function, you show only the elements you want to read. It is not necessary to have one output for every cluster element.

For example, the following is a panel with a cluster of three elements, weight, height, and age.
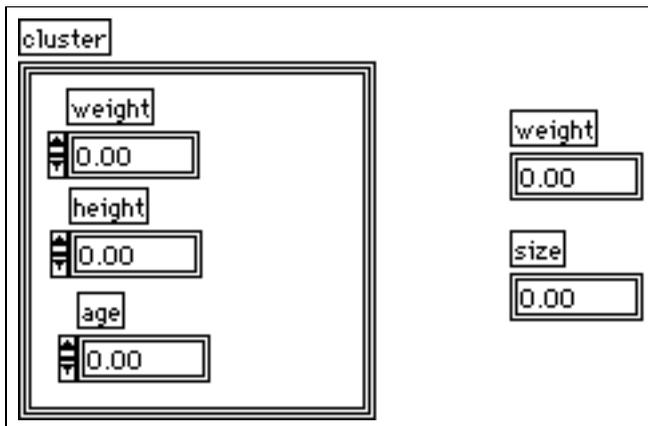
When connected to an Unbundle By Name function, you can pop up on an output terminal of the function and choose **Select Item** to select an element you want to read from the names of the components of the cluster, as shown in the following illustration.
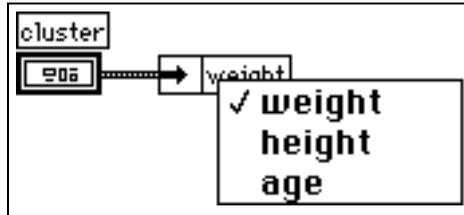
Resize the function to read other elements, as shown in the following illustration. Remember, you do not have to read all of the elements and you can read them in arbitrary order.

One of the advantages of the Unbundle By Name function compared to the Unbundle function is that it is not as closely tied to the data structure of the cluster. The Bundle function always must have exactly the same number of terminals as it has elements. If you add or remove an element from a cluster connected to an Unbundle function, you end up with broken wires. With the Unbundle By Name function, you can add elements to and remove elements from the cluster without breaking the VI, as long as the elements were not referenced on the diagram.

For example, with the previous cluster, you add a string for birthdate, and the diagram still is correct. This example is shown in the following illustration.
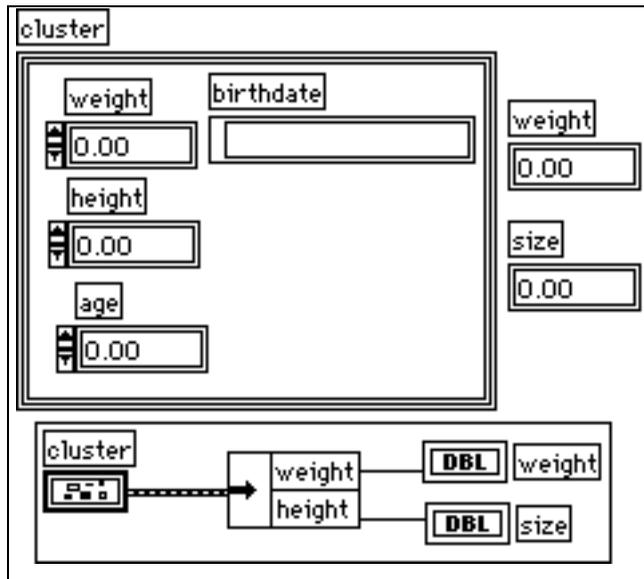
As with Bundle By Name, the Unbundle By Name function is particularly useful in larger applications in conjunction with type definitions. See the *Bundle By Name Function* section of this chapter and the *Type Definitions* section of Chapter 24, *Custom Controls and Type Definitions*, for more information.

Cluster To Array Function

Cluster To Array function

The Cluster To Array function, which you obtain from the **Conversion** palette of the **Functions** palette, converts the elements of a cluster into a 1D array of those elements. The cluster elements must all be the same type. The array contains the same number of elements as the cluster.

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 also want to process them as an array on the block diagram, as shown in the following illustration.
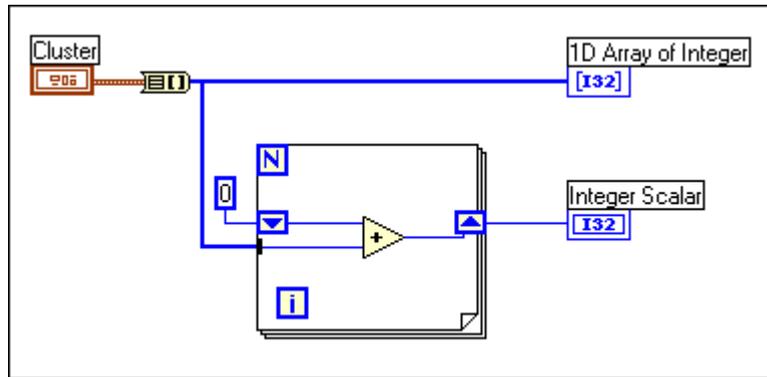
Replacing Cluster Elements

Sometimes you want to replace, or change the value of, one or two elements of a cluster without affecting the others. Do this by unbundling a cluster and wiring the unchanging elements directly to a Bundle function along with the replacement values for the other element. The following example shows a more convenient method. This example computes the number of hours until New Year using the date-time cluster. Because the cluster input terminal (middle terminal) of the Bundle function is wired, the only element input terminals that must be wired are those with replacement values.
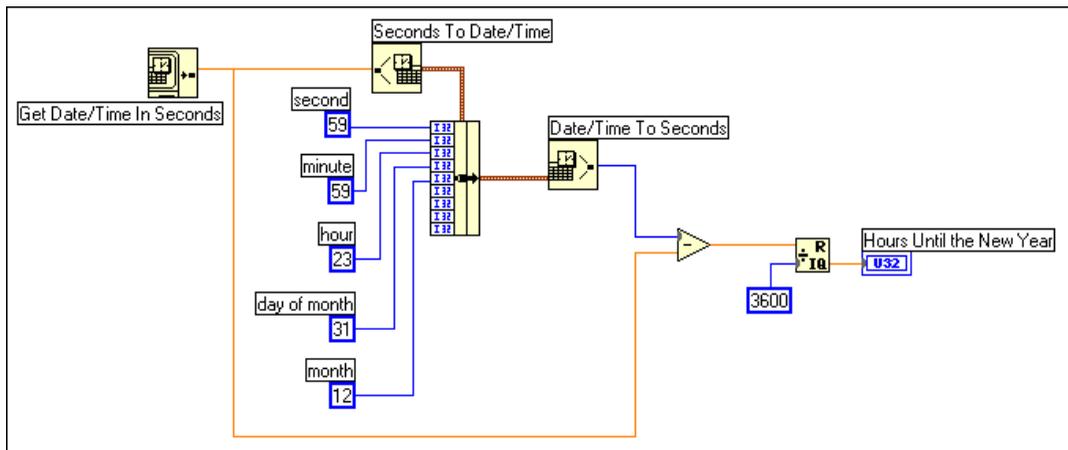
15

Graph and Chart Controls and Indicators

This chapter describes how to create and use the graph and chart indicators in the **Controls»Graph** palette.

A graph indicator is a two-dimensional (2D) 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 it helpful to study the graph and chart examples. These examples are located in `examples\general\graphs`.

G has three kinds of graphs and two kinds of charts. These are shown in the following illustration starting at the top from left to right—waveform chart, waveform graph, *x*y graph, intensity chart, and intensity graph.
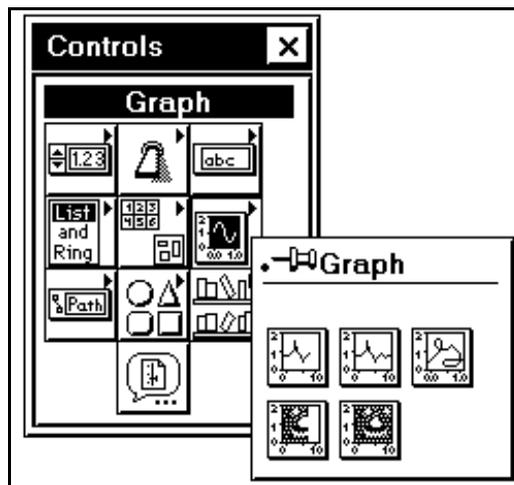
Creating Waveform and XY Graphs

Waveform graphs are for displaying evenly sampled measurements. *XY graphs* can display any set of points, whether evenly sampled or not. You can obtain a waveform graph and *xy* graph from the **Controls» Graph** palette. Examples of these graphs are shown in the following illustration.
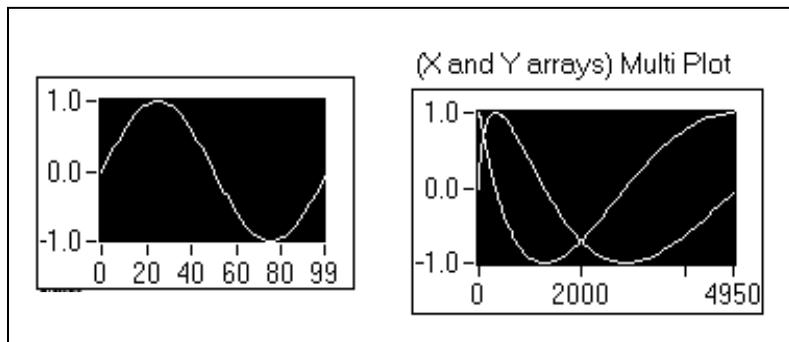
The waveform graph plots only single-valued functions with points evenly distributed along the *x*-axis, such as acquired time-varying waveforms. The *xy* graph is a general-purpose, Cartesian graphing object you can use to plot multivalued functions such as circular shapes or waveforms with a varying timebase. Both graphs can display any number of plots.

Each of these graphs accepts several data types. This minimizes the extent to which you must manipulate your data before displaying it. These data types are described in the next section.

Following the section on graph data types are sections describing some of the more advanced options of the graph. These include customizing the appearance of the graph, displaying and manipulating cursors, and displaying a legend for the graph.

Plotting Single-Plot Graphs

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 the waveform graph accepts is a single array of values. The graph interprets the data as points and increments them by one, starting at $x = 0$. The following diagram illustrates how you create this kind of data.
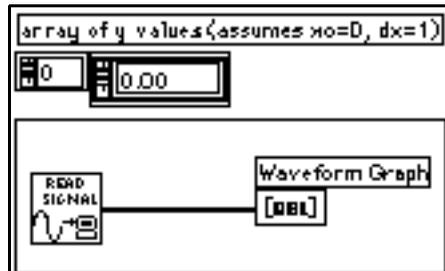
The second data type is a cluster of an initial x value, a Δx value, and an array of y data. The following diagram illustrates how you create this kind of data.
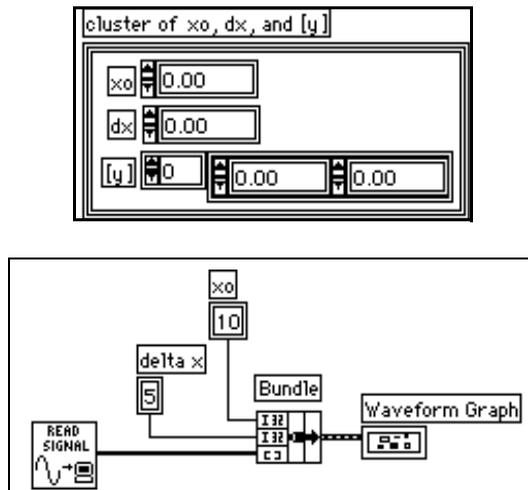
XY Graph Data Types

The *xy* graph accepts two data types for a single-plot graph. These data types are described in the following paragraphs.

The first data type the *xy* graph accepts is a cluster containing an *x* array and a *y* data array. The following diagram illustrates how you create this kind of data.
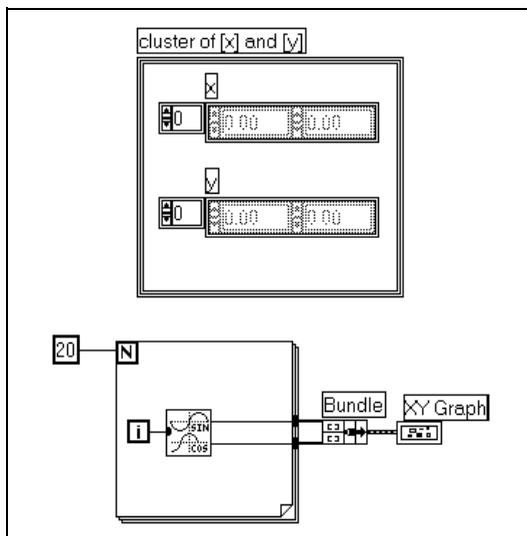
The second data type is an array of *points*, where a point is a cluster of an *x* value and a *y* value. The following diagram illustrates how you create this kind of data.
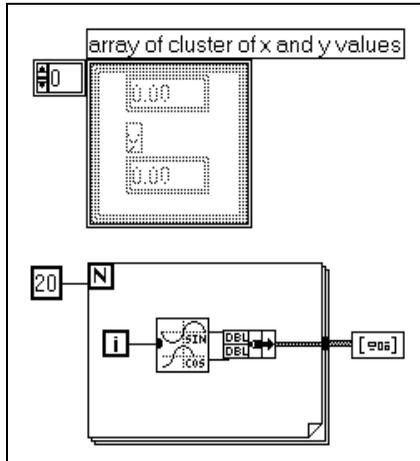
Plotting Multiplot Graphs

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 G does not
accept arrays of arrays, in a case where creating an array of a single plot
data type produces 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, described in the
following paragraphs.

The first data type a multiplot graph waveform accepts is a
two-dimensional array of values, where each row of the data is a single plot.
The graph interprets this data as points, with the points starting at $x = 0$ and
increasing incrementally by one.
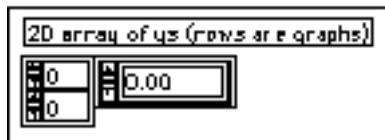
If you select the **Transpose Array** item from the graph pop-up menu,
it handles each column of data as a plot. This is useful particularly when
sampling multiple channels from a data acquisition board, because that
data returns as 2D arrays, with each channel stored as a separate column.

The following example shows how you use a graph to display two signals, where each signal is a separate row of a two-dimensional array.
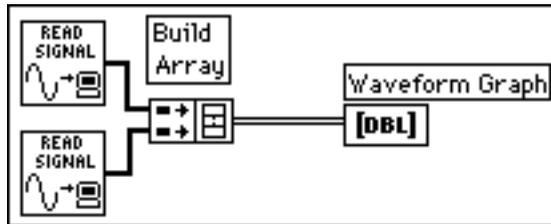
The second data type is a cluster of an *x* value, a Δ*x* value, and a two-dimensional array of *y* data. The *y* data is interpreted as described for the previous data type. This data type is useful for displaying multiple signals all sampled at the same regular rate. The following diagram illustrates how you create this kind of data.
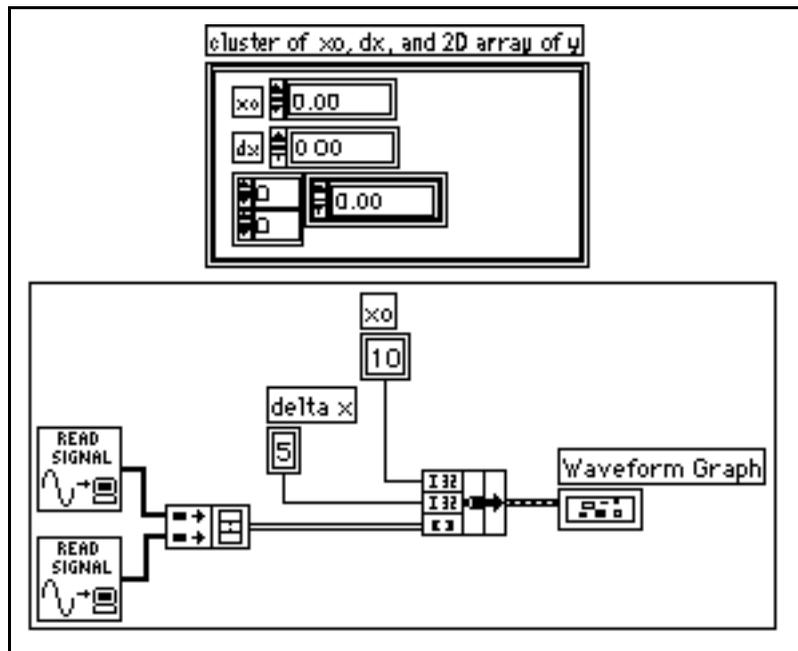
The third data type is an array of clusters of an array of *y* data. The inner array describes the points in a plot, and the outer array has one element for each plot.
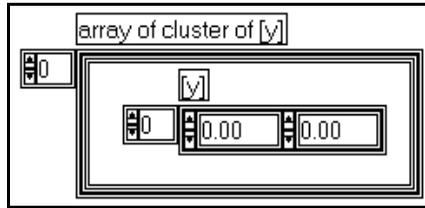
Use this data structure instead of a two-dimensional array if the number of elements in each plot is different. For example, use this to sample data from several channels but not for the same amount of time from each channel. Use this data structure because each row of a two-dimensional array must have the same number of elements, but the number of elements in the interior arrays of an array of clusters can vary.

The following illustration shows how to create the appropriate data structure from two arrays.
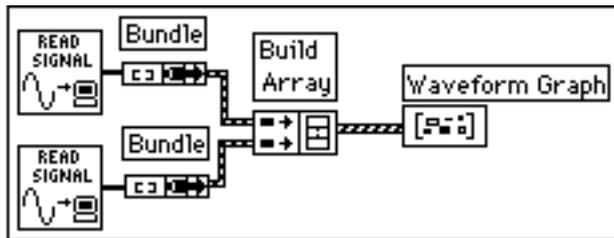
To see an example of this concept, look at the Waveform Graph VI in the `examples\general\graphs\gengraph.llb` directory.

Another way to convert arrays into 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, use this to sample data from several channels, but not for the same amount of time from each channel. 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 create this kind of data.
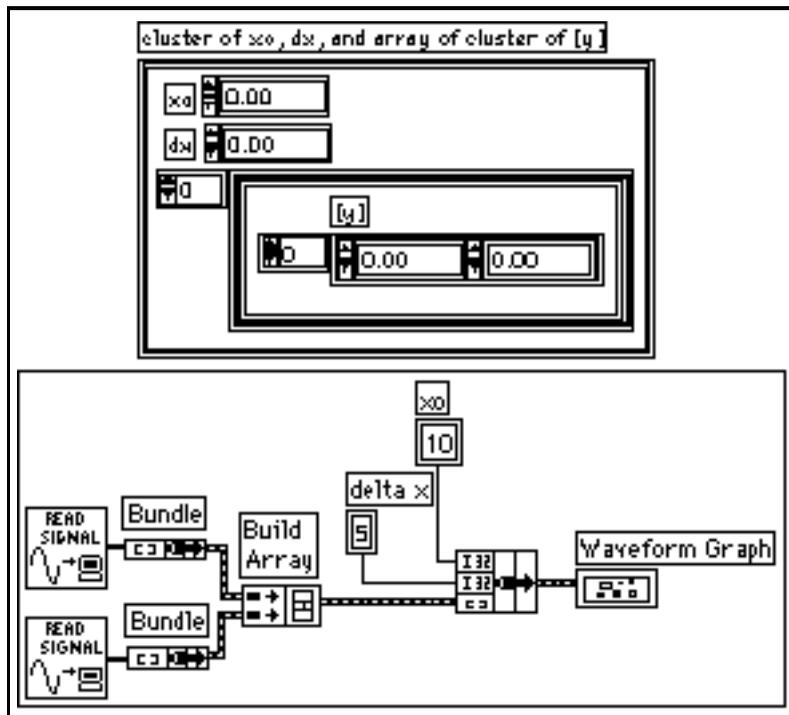
Notice the arrays are bundled into clusters using the Bundle function, and the resulting clusters built into an array with the Build Array function. You can use the Build Cluster Array, which creates arrays of clusters of specified inputs, instead.

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 you can specify a unique starting point and increment for the x-axis of each plot. The following diagram illustrates how to create this kind of data.

array of cluster of xo, dx, and [y]

xo [0.00]

dx [0.00]

[y] [0] [0.00]

xo
[15]

delta x
[2.5]

READ
SIGNAL

Bundle
I32
DBL
[]

xo
[10]

delta x
[5.0]

READ
SIGNAL

Bundle
I32
DBL
[]

Build
Array

Waveform Graph
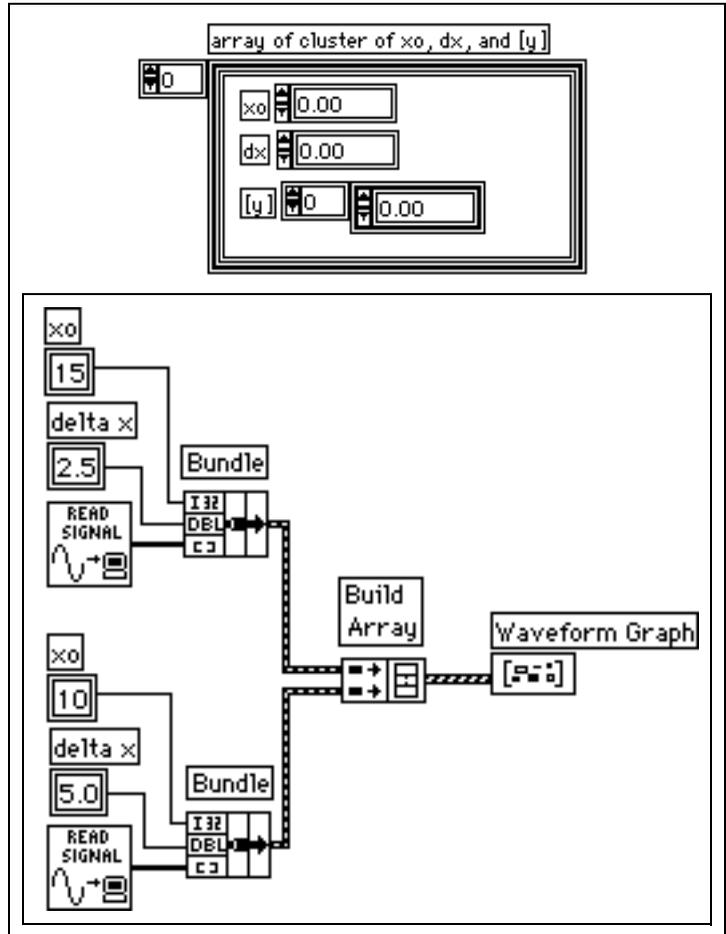
XY Graph Data Types

The *xy* graph accepts two multiplot data types, described in the following paragraphs. Both of these data types are arrays of clusters of the single plot data types described previously.

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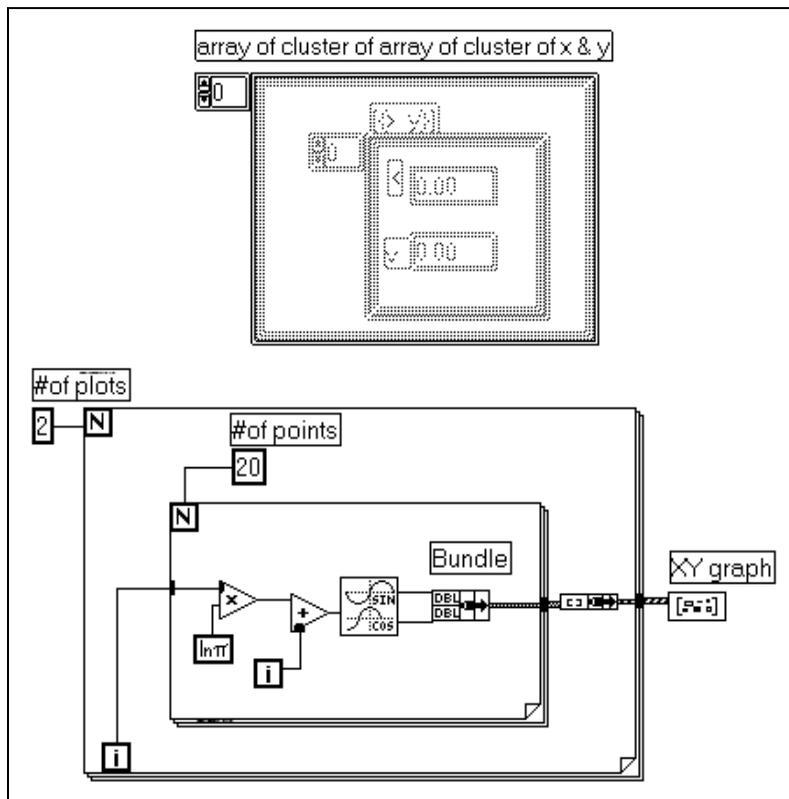
The second data type is an array of plots, where a plot is a cluster of an
x array and a *y* array. The following diagram illustrates how you create this
kind of data.
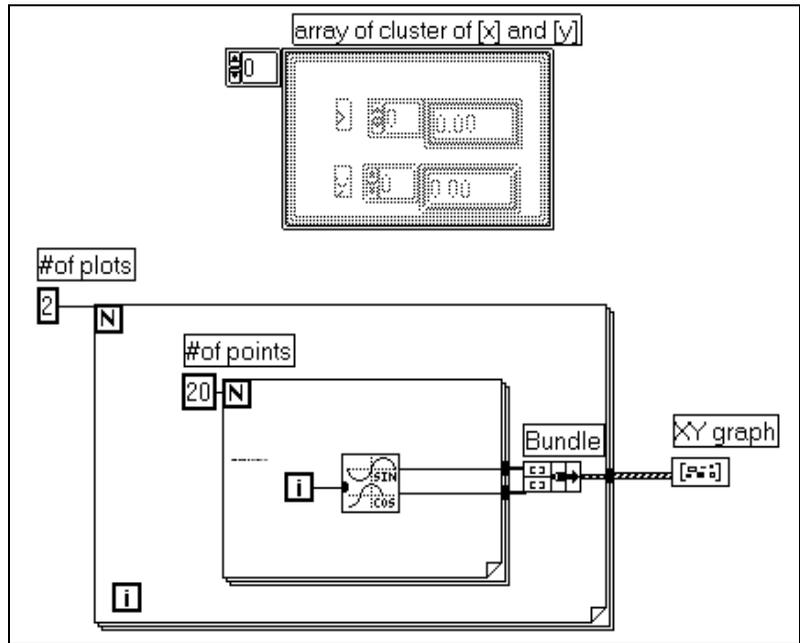
Setting Custom Options on a Graph

Both graphs have optional parts that can be shown or hidden using the
Show submenu of the graph pop-up menu. These items include a legend,
from which you can define the color and style for a particular plot; a palette
used to change scaling and format options while the VI is running; and a

Cursor palette used 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.
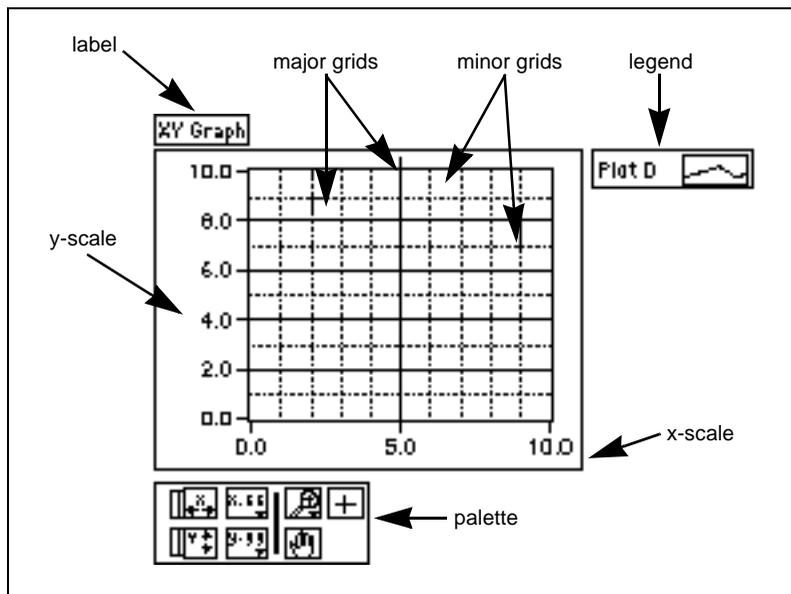
Graphs have many options to customize your data display. The graph pop-up menu is shown in the following illustration. **Transpose Array** is available with the waveform graph only.
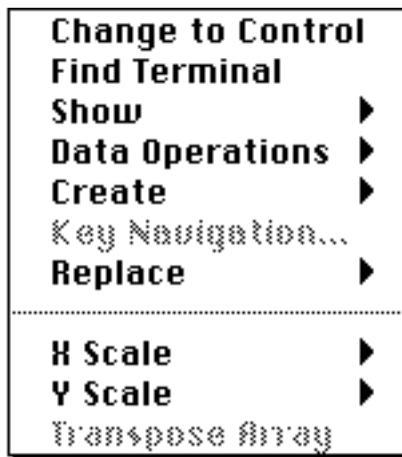
Scale Options

Graphs can automatically adjust their horizontal and vertical scales to reflect the data wired to them. Turn this autoscaling feature on or off using the **Autoscale X** and **Autoscale Y** menu items from the **Data Operations** or the **X Scale/Y Scale** submenus of the pop-up menu for the graph. You also can control these autoscaling features from the palette for the graph, as described later in this chapter. Autoscaling on is the default setting for graphs. However, autoscaling can cause the graph to run more slowly.

You can change the horizontal or vertical scale directly using the Operating or Labeling tool, just as you can with any other control or indicator.

The **Data Operations** submenu of the graph pop-up menu includes a **Smooth Updates** item that uses an offscreen buffer to minimize flashing. This feature can cause the graph to run more slowly, depending on the computer and video system you use.

The *x* and *y* scales each have a submenu of items, as shown in the following illustration.
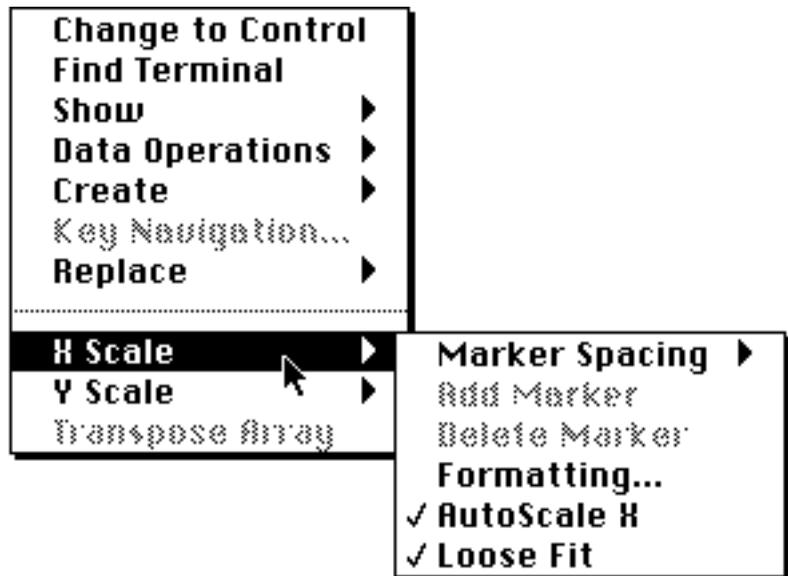
Marker Spacing

By default, marker values for *x* and *y* scales are uniformly distributed. If you prefer, you can specify marker locations anywhere on the *x* or *y* scale. This is useful for marking a few specific points on a graph (such as a set point or threshold).

If you want non-uniform marker distribution, choose **X** or **Y Scale»Marker Spacing»Arbitrary Markers** from the pop-up menu for the scale. After making this selection, if you move the Operating tool over a tick mark, the cursor changes to the double arrow cursor shown in the following illustration. You can now create a new marker by dragging the existing tick mark with the `<Ctrl>` **(Windows)**; `<option>` **(Macintosh)**; `<meta>` **(Sun)**; or `<Alt>` **(HP-UX)** key selected, or you can move the existing tick mark anywhere on the scale you want by dragging it.

You can add or delete a marker by popping up on the graph or the scale and selecting **Add Marker** or **Delete Marker**. When a marker is created, type a number in the marker to change its location.

Formatting

Click **Formatting…** to bring up the dialog box shown in the following illustration.
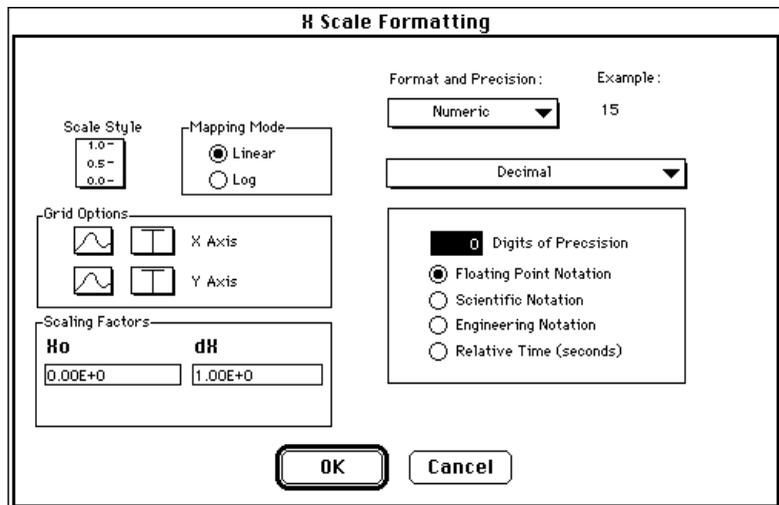
Select the items in this dialog box to set the following graph properties.

Scale Style—Use this item to select major and minor tick marks for the scale. Major tick marks are points corresponding to scale labels and minor tick marks are interior points between labels. You also use this palette to decide if you want the markers for a particular axis to be visible.

Mapping Mode—Use this item of the scale menus to decide if the scale maps data using a linear or a logarithmic mode.

Grid Options—Clicking these items brings up a palette of grid types to select if you want no gridlines, gridlines only at major tick mark locations (points corresponding to scale labels) or major and minor tick marks (minor tick marks are interior points between labels). The control next to the grid pop-up is a color menu you can use to select the color for the gridlines.

Scaling Factors—You use the scaling factors to specify the initial value and spacing between points on a waveform chart or graph, or along the scales of an intensity chart or graph. You also can use these scaling factors to scale your data for display. For example, if your data is binary sampled in a range of –2,048 to 2,047, scale this data to its appropriate voltage values of 0 to 5 by specifying an additive offset of 2.5 and a scaling factor of 5/4095= 0.001221.

Format & Precision—The x and y scale format, in numerics or time and date, is selected through a menu ring at the top of the dialog box.

If you select **Numeric** formatting, you can choose if the notation is floating-point, scientific, engineering, or relative time in seconds; you can choose if the notation is decimal, unsigned decimal, hexadecimal, octal, or binary; and you can select the precision (how many digits are displayed to the right of the decimal point) from 0 through 20. The precision you select affects only the display of the value; the internal accuracy still depends on the representation. Examples are shown in the dialog box as you select each combination of items.

If you select **Time & Date** formatting, the dialog box changes, as shown in the following illustration.
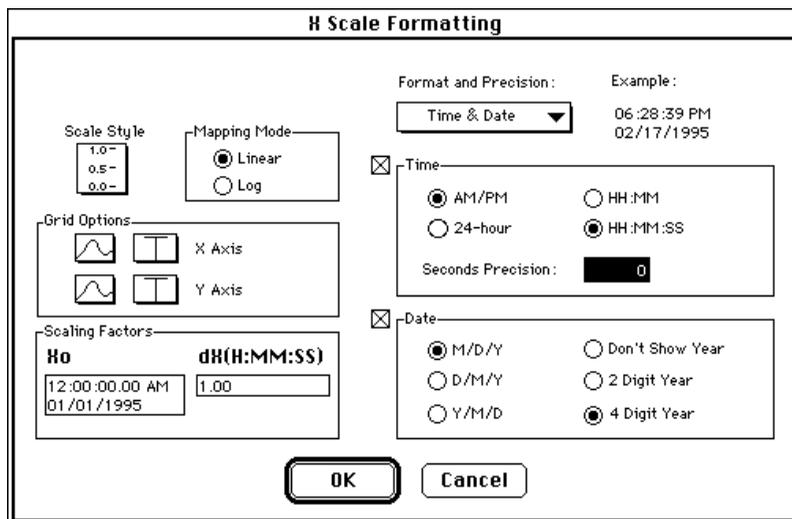
You can format for either absolute time or date, or both. If you edit a scale and enter only time or only date, the unspecified components are inferred. If you do not enter time when editing, it assumes 12:00 a.m. If you do not enter a date, the previous date value is assumed. If you enter date, but the scale is not in a date format, the month, day, and year ordering are assumed based on the settings in the **Preferences** dialog box. If you enter only two digits for the year, the following is assumed: any number less than 38 is in the twenty-first century, otherwise the number is in the twentieth century.

Although absolute time displays as a time and date string, it is represented internally as the number of seconds since 12:00 a.m. January 1, 1904, Universal Time Coordinated (UTC), formerly known as Greenwich Mean Time (GMT).

Notice the examples at the top right of the dialog box, which change as you make selections.

The valid range for time and date differs across platforms as follows.

- **(Windows)** 12:00 a.m. Jan. 2, 1970 – 12:00 a.m. Jan. 3, 2040
- **(Windows NT)** 12:00 a.m. Jan. 1, 1904 – 12:00 a.m. Jan. 3, 2040
- **(Macintosh)** 12:00 a.m. Jan. 2, 1904 – 12:00 a.m. Jan. 2, 2040
- **(UNIX)** 12:00 a.m. Jan. 1, 1904 – 12:00 a.m. Jan. 17, 2038

These ranges might be up to a few days wider depending on your time zone and if daylight saving time is in effect.

Autoscale

Use **AutoScale (X** or **Y)** to turn autoscaling on or off.

Loose Fit

With **Loose Fit** on, the end markers are rounded to a multiple of the increment used for the scale. If you want the scales to be set to exactly the range of the data, turn off the **Loose Fit** item in the graph pop-up menu.

Panning and Zooming Options

The **Graph** palette is included with any graph you drop onto the front panel. This palette has controls for panning (scrolling the display area of a graph) and for zooming in and out of sections of the graph. A graph with its accompanying **Graph** palette is shown in the following illustration.
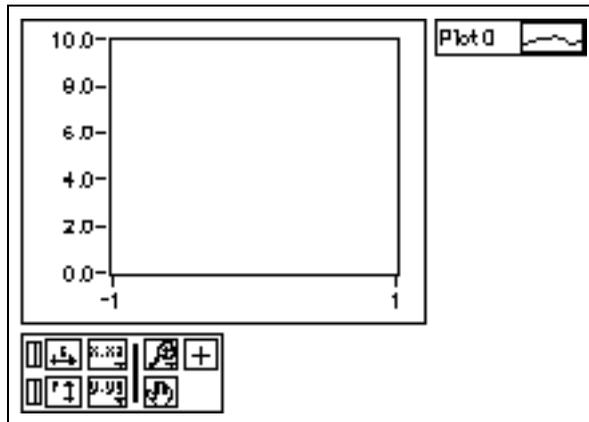
If you press the *X* **Autoscale** button, shown at the left, the graph autoscales the *x* axis. If you press the *y* autoscale button, shown at the left, the graph autoscales the *y* axis. If you want the graph to autoscale either of the Scales continuously, click the lock switch, shown at the left, to lock autoscaling On.

By using the **Scale Format** buttons, shown left, you can maintain run-time control of the format of the *x* and *y* scale markers respectively.

You use the remaining three buttons to control the operation mode for the graph.

Normally, you are in standard operate mode, indicated by the plus or crosshatch. In operate mode, you can click in the graph to move cursors around.

If you press the Panning tool, shown to the left, you switch to a mode in which you can scroll the visible data by clicking and dragging the plot area of the graph.

If you press the Zoom tool, shown at the left, you can zoom in or out on the graph. If you click the Zoom tool, you see a pop-up menu to choose methods of zooming. This menu is shown in the following illustration.
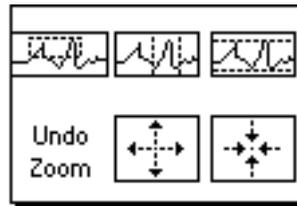
A description of each of these items follows.
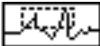
Zoom by rectangle.

Zoom by rectangle, with zooming restricted to *x* data (the *y* scale remains unchanged).

Zoom by rectangle, with zooming restricted to *y* data (the *x* scale remains unchanged).

Undo last zoom. Resets the graph to its previous setting.

Zoom in about a point. If you hold down the mouse on a specific point, the graph continuously zooms in until you release the mouse button.

Zoom out about a point. If you hold down the mouse on a specific point, the graph continuously zooms out until you release the mouse button.

Note   *For the last two modes, zoom in and zoom out about a point,* <Shift>-*clicking zooms in the other direction.*

Legend Options

The graph uses a default style for each new plot unless you create 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 also can indicate 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, the graph assigns those characteristics to the plot, regardless of if the legend is visible. If the graph receives more plots than are defined in the legend, the graph 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 make more room on the labels, or on the right to make more room on the plot samples.

By default, each plot is labeled with a number, beginning with zero. You can modify this label the same way you modify other 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 displays with the characteristics you assign it in the graph legend.

The plot sample pop-up menu is shown in the following illustration.
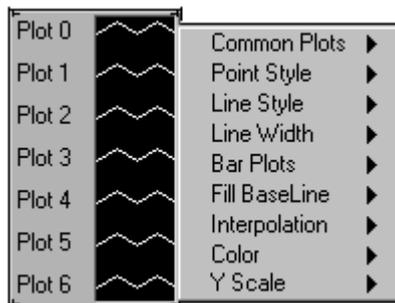
The **Common Plots** item helps you configure a plot for any of six popular plot styles, including a scatter plot, a bar plot, and a fill to zero plot. Items in this subpalette are preconfigured for the point, line, and file styles in one step.

The **Point Style**, **Line Style**, and **Line Width** items display styles you can use to distinguish a plot. The line width subpalette contains widths thicker than the default 1 pixel, as well as the hairline item. The latter item has no effect on the screen display, but prints a very thin line if the printer and print mode support hairline printing.
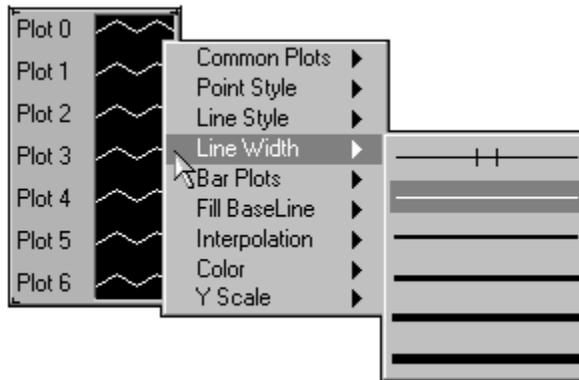
Note   *In Windows, wide pens can be only in the solid style.*

The **Bar Plots** item has a selection of vertical bars, horizontal bars, or no bars at all.

The **Fill Baseline** item sets what the baseline fills to. **Zero** fills from your plot to a baseline generated at 0. **Infinity** fills from your plot to the positive edge of the graph. **–Infinity** fills from your plot to the negative edge of the graph. By using the bottom portion of this menu, you can select a specific plot of this graph to fill to.

The **Interpolation** item brings up the palette shown in the following illustration, in which you choose how the graph draws lines between plotted points. The first item does not draw a line, making it suitable for a scatter plot. The item at the bottom left draws a straight line between plotted points. The two stepped items, which link points with a right-angled elbow, are useful for creating histogram-like plots. The item at the top right plots in the *y* direction first, and the item at the bottom right plots in the *x* direction first.
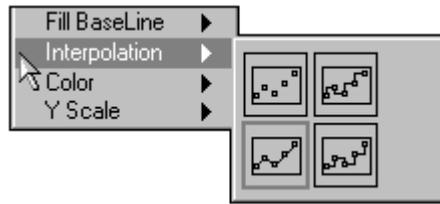
The **Color** item displays the palette for selecting the plot color. You also can color the plots on the legend with the Color tool, and you can change the plot colors while running the VI.

The **Y scale** item displays a list of the *y* scales on the graph. This is used on stacked charts to define on which scale each plot is plotted.

Waveform Charts

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

For an example of a waveform chart, see
`examples\general\graphs\charts.llb`.

Waveform Chart Data Types

You can pass charts either a single value or multiple values at a time. As with the waveform graph, each value is handled 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 handles these inputs as new data for a single plot.

Following are diagrams illustrating how you use the chart for each of these kinds of data.
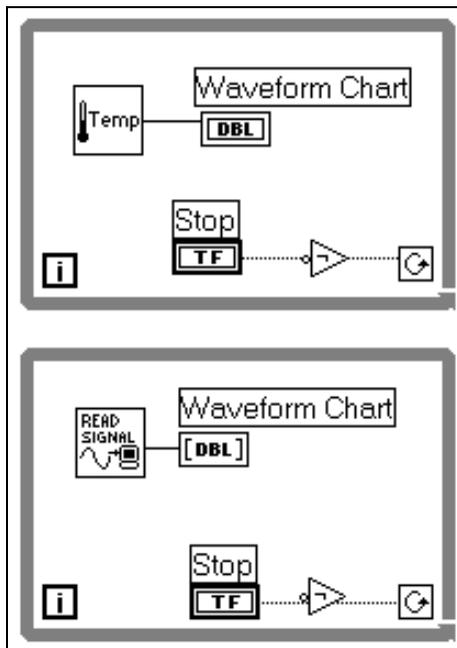
The chart redraws less frequently when you pass it multiple points at a time than if drawn once for each point.

You can pass data for multiple plots to a waveform chart in several ways. The first method is to bundle the data together into a cluster of scalar numerics, where each numeric represents a single point for each of the plots. An example of this is shown in the following illustration.
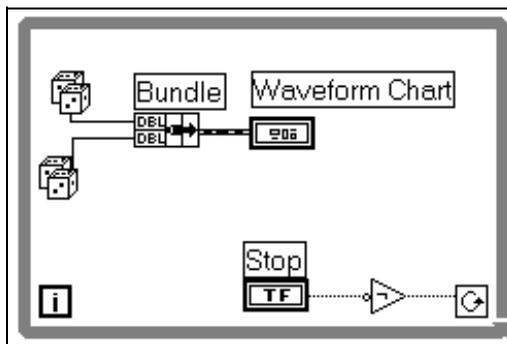
If you want to pass multiple points for plots in a single update, you can wire an array of clusters of numerics to the chart. Each numeric represents a single point for each of the plots. An example of this is shown in the following illustration.
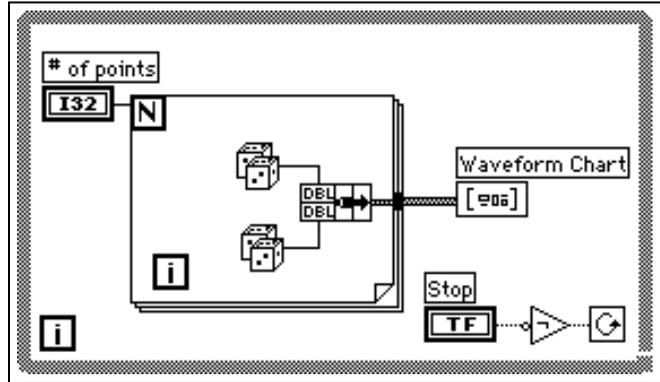
If the number of plots you want to display cannot be determined until runtime, or you want to pass multiple points for plots in a single update, you can wire a two-dimensional array of data to the chart. As with the waveform graph, rows are handled normally as new data for each plot. You can use the **Transpose Array** item of the waveform chart pop-up menu to handle columns as new data for each plot.
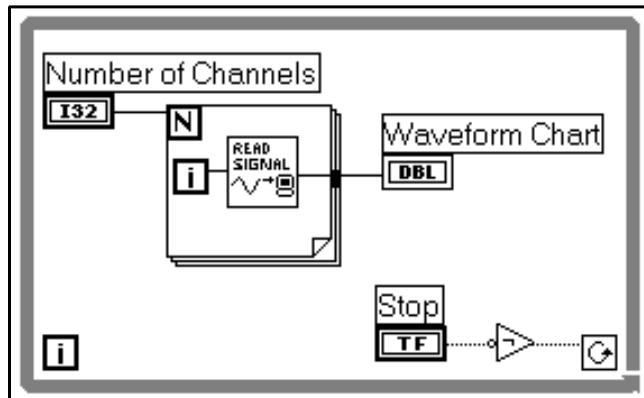
Waveform Chart Options

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

With the **Show** submenu of the chart pop-up menu, you can show or hide optional digital display(s) and a scrollbar. The **Digital Display** item displays the latest value being plotted. The last value passed to the chart from the diagram is the latest new value for the chart. There is one digital display per plot.

You can view past values contained in the buffer by scrolling the *x*-axis to a range of previously plotted values using the scrollbar, or by changing the *x* scale range to view old data.

There is a limit to the amount of data the chart can hold in the buffer to avoid running out of memory. The default size of this buffer is 1,024 points. When the chart reaches that limit, the oldest point(s) are thrown away to make room for new data. You can change the length of this buffer using the **Chart History Length…** item from the chart pop-up menu.

Chart Update Modes

To change the way the chart behaves when new data is added to the display, use the **Update Mode** item from the **Data Operations** submenu of the pop-up menu of the chart, shown in the following illustration.
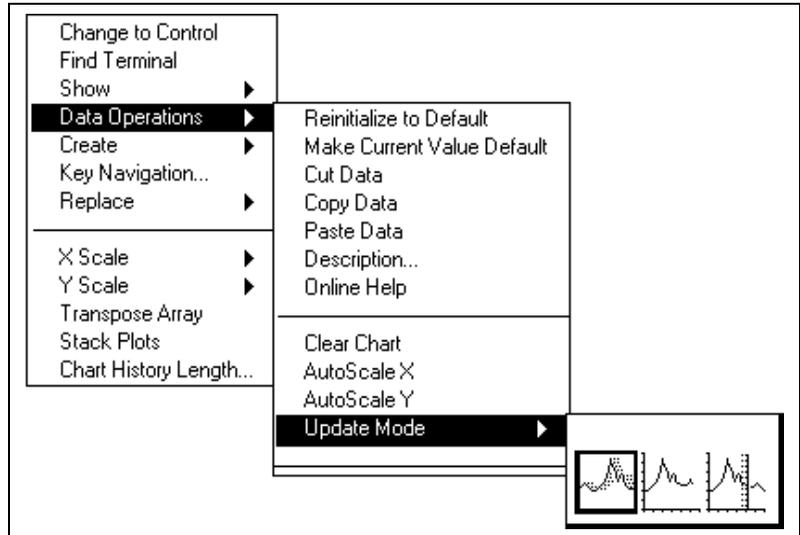
The three items—strip chart, scope chart, and sweep chart—are illustrated in the following pictures, and described in the subsequent paragraphs. The default mode is strip chart.

The *strip chart* mode has a scrolling display similar to a paper tape strip chart recorder. As each new value is received, it is plotted at the right margin, and old values shift to the left.
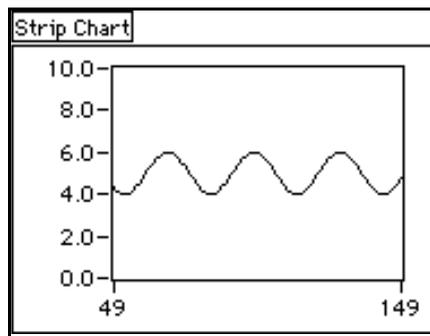
The *scope chart* mode has a retracing display similar to an oscilloscope. As each new value is received, it is plotted to the right of the last value. When the plot reaches the right border of the plotting area, the plot is erased and plotting begins again from the left border. The scope chart is

significantly faster than the strip chart because it is free of the processing overhead involved in scrolling.
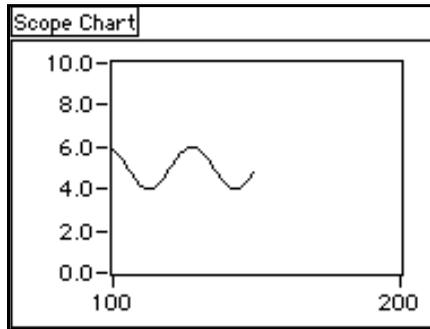
The *sweep chart* mode acts much like the scope chart, but it does not blank when the data hits the right border. Instead, a moving vertical line marks the beginning of new data and moves across the display as new data is added.
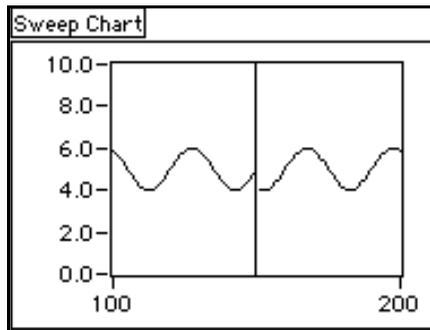
Stacked Versus Overlaid Plots

By default, the chart displays multiple plots by overlaying one on top of the other, like graphs drawn on the same grid. An example of overlaid plots is shown in the following illustration.
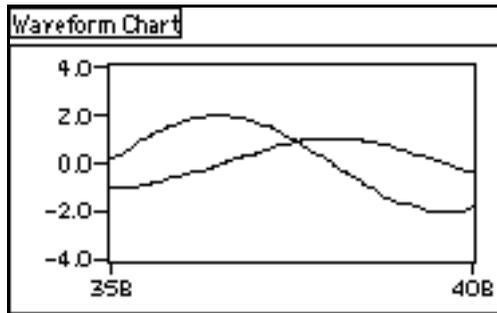
Alternatively, you can display multiple plots stacked one above the other with a different *y* scale for each plot, by selecting the **Stack Plots** item from the chart pop-up menu. If you do this, the *y* scale for each chart can have a different range. An example of stacked plots is shown in the following illustration.
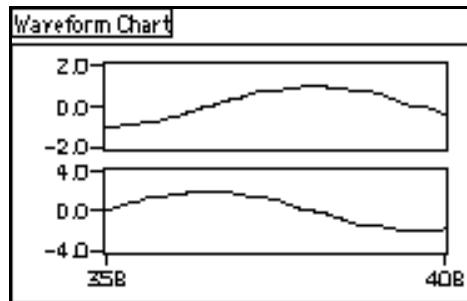
When you input data to the chart as a cluster, the chart automatically stacks the correct number of plot displays. When you input data as a 2-D array, you must create the correct number of plot displays using the Legend, available in the **Show** submenu of the pop-up menu. As you enlarge the Legend display to increase the number of plots, the number of stacked plot displays increases to match.

Graph Cursors

For each graph, you can show a **Cursor** palette used to put cursors on the graph. You can label the cursor on the plot, and use a cursor as a marker. When you use a cursor as a marker, you lock the cursor to a data plot so the cursor follows the data.

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

Waveform Graph

Intensity Graph
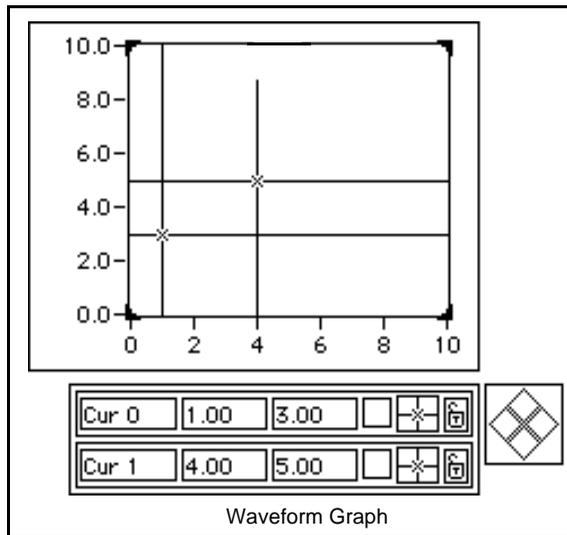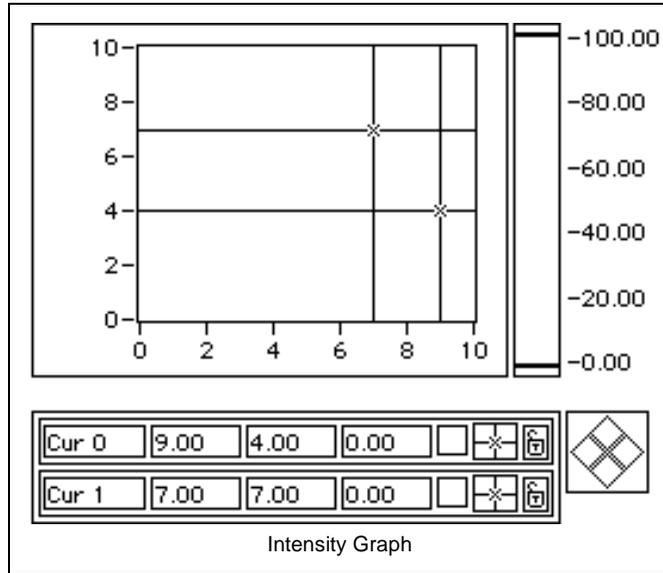
Each cursor for a graph has the following parts.

- A label
- *x* and *y* coordinates, and *z* coordinate, if applicable
- A button that marks the plot for movement with the plot cursor pad
- A button that controls the look of the cursor
- A button that determines if the cursor is locked to a plot or moved freely

These parts are shown in the following illustration.
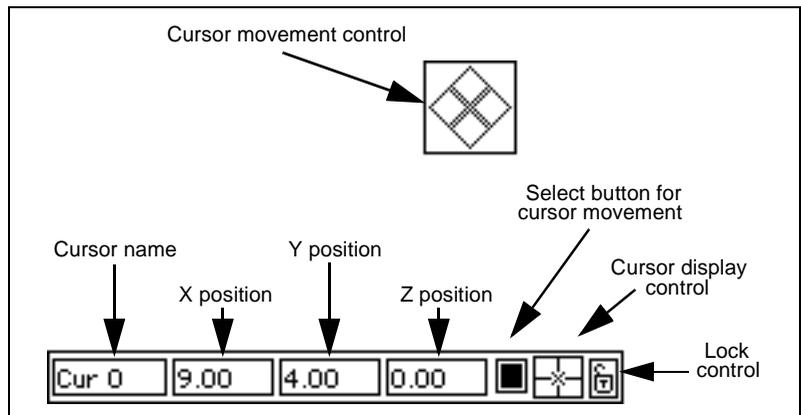
The **Cursor** palette behaves like an array. You can stretch it to display information on multiple cursors, and you can use the index control to view other cursor information in the palette. Use the **Show** items of the pop-up menu on the cursor display to show the index control.
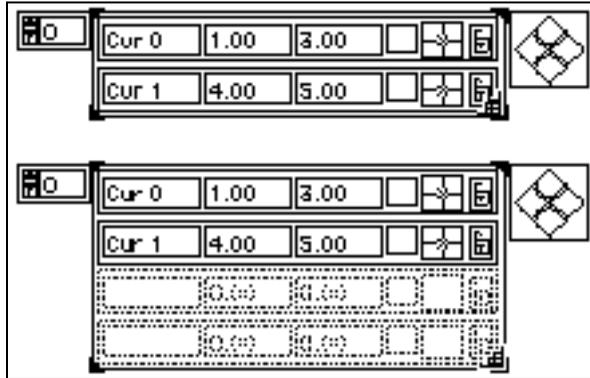
To delete a cursor, you must select it using the **Start Selection** and **End Selection** items on the **Data Operations** pop-up menu, and then cut the cursor with the **Cut** item on the same menu. See the *Selecting Array Cells* section of Chapter 14, *Array and Cluster Controls and Indicators*, for more information.

You can move a cursor on a graph by dragging it with the Operating tool, or by using the cursor movement control. To drag a cursor make sure the graph does not have the panning tool or zooming tool selected. 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 the select button for a particular cursor. In the following example, the top two cursors move vertically downwards.

cursor display
control
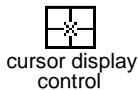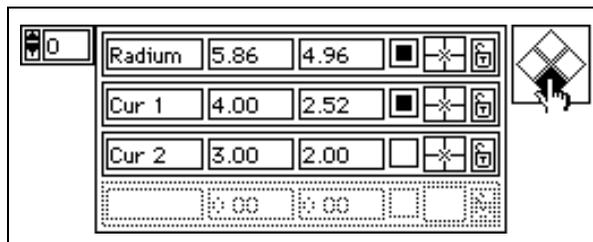
Clicking the cursor display control with the Operating tool displays a pop-up menu to control the look of the cursor and the visibility of the cursor name on the plot. This pop-up menu is shown in the following illustration.
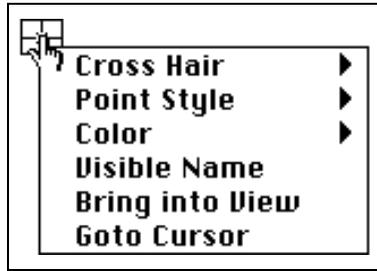
From this menu you select the style of the cross hairs. The cross hairs can consist of a vertical and/or horizontal line extending to infinity, a smaller vertical and/or horizontal line centered about the cursor location, or no cross hairs, as shown in the following illustration.
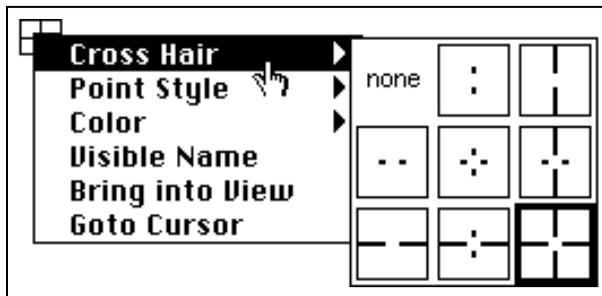
You also can choose the style of point to use for marking the cursor location and the color for the cursor as shown in the following illustration.
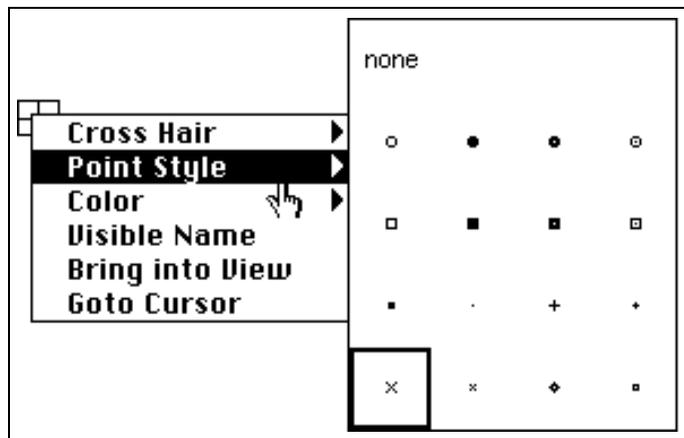
Select the **Visible Name** item from this menu to make the cursor name visible on the plot, as shown in the following illustration.
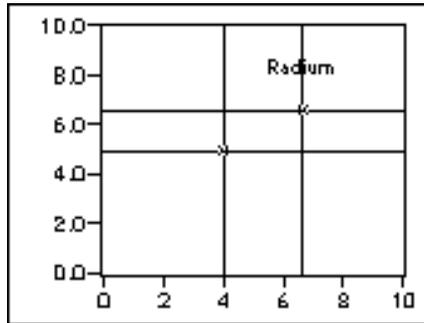
Selecting **Bring into View** moves the cursor back into the displayed region of the graph. This is helpful when the cursor moves out of visible range. Selecting this item changes the (x,y) coordinate position of the cursor.

Selecting **Goto Cursor** moves the displayed region of the graph so 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 mark a point of interest in the graph, such as a minimum or a maximum, and you want to see that point.

unlocked   locked
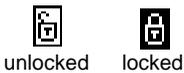
You can use the last button for each cursor to lock a cursor onto a particular plot. By clicking the lock button you see a pop-up menu you can use to lock the cursor to a specific plot. If you lock the cursor onto a plot, the button changes to a closed lock. This pop-up menu is shown in the following illustration.
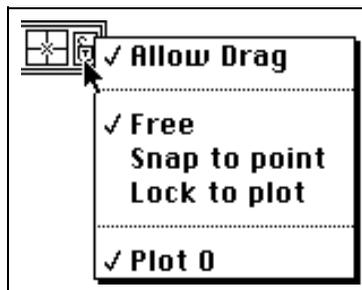
The **Drag** item determines if you can move the cursor with the mouse. If **Allow Drag** is selected, you can move, or drag the cursor. The items below the dotted line of the menu determine how you can move the cursor with the mouse. If **Allow Drag** is deselected, you cannot move the cursor around on the plot.

Select **Free** if you want to place or move the cursor anywhere on the graph. Select **Snap to Point** if you want the cursor to always attach itself to the nearest point on any plot. Select **Lock to Plot** to attach the cursor to a specific plot. The first time you select **Lock to Plot**, the cursor attaches itself to the first point on the plot. After freeing the locked cursor and moving it to any new position, selecting **Lock to Plot** moves the cursor to the last location of the locked cursor.

Below the second dotted line of this menu is a list of the plots you can lock to (for example, Plot 0, Plot 1, Plot 2, and so on).

A large number of items are available for creating, controlling, and reading cursors or markers programmatically using the attribute node for a graph. See Chapter 22, *Attribute Nodes*, for further information.

Intensity Charts

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

Column = y

		0	1	2
	0	10	50	13
Row = x	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

3						
	dk red	lt red	blue			
2						
	yellow	green	dk red			
1						
	lt red	orange	purple			
0	1	2	3	4	5	
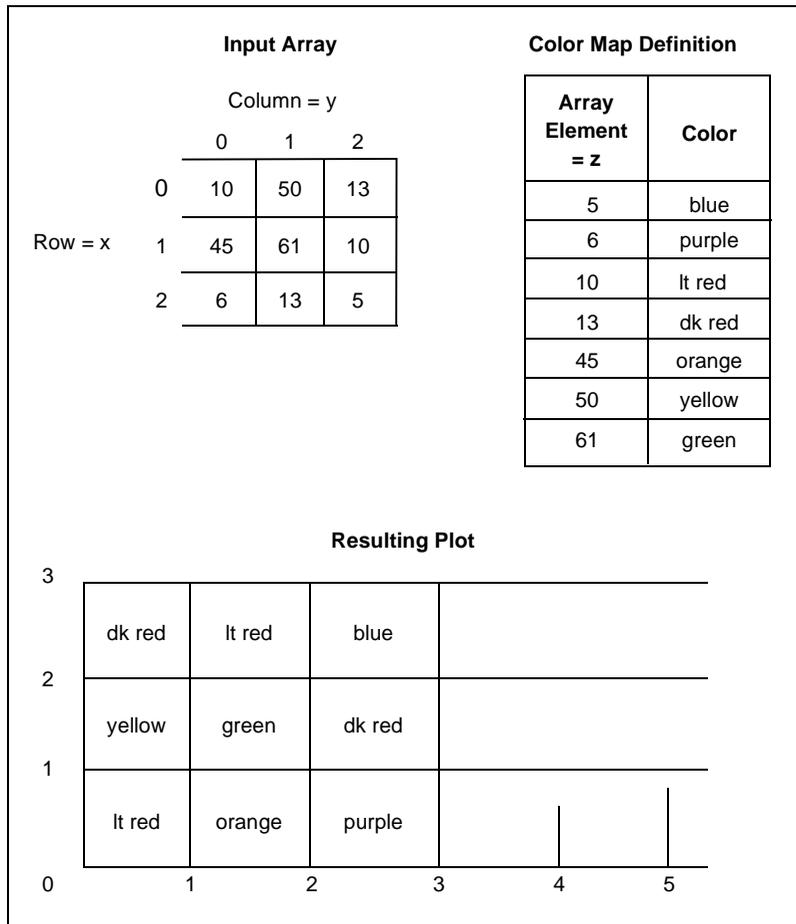
You can define the colors for the intensity chart interactively, by using the color scale, or you can define them programmatically through the chart attribute node. The *Color Mapping* section later in this chapter explains the procedure for assigning a color to a number.

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 discrete colors.

After a block of data is plotted, the origin of the Cartesian plane shifts 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.

New Input Array

Column = y

		0	1	2
	0	61	45	5
Row = x	1	45	5	61
	2			

Resulting Plot

3					
	dk red	lt red	blue	blue	green
2					
	yellow	green	dk red	orange	blue
1					
	lt red	orange	purple	green	orange
0	1	2	3 (0)	4 (1)	5 (2)
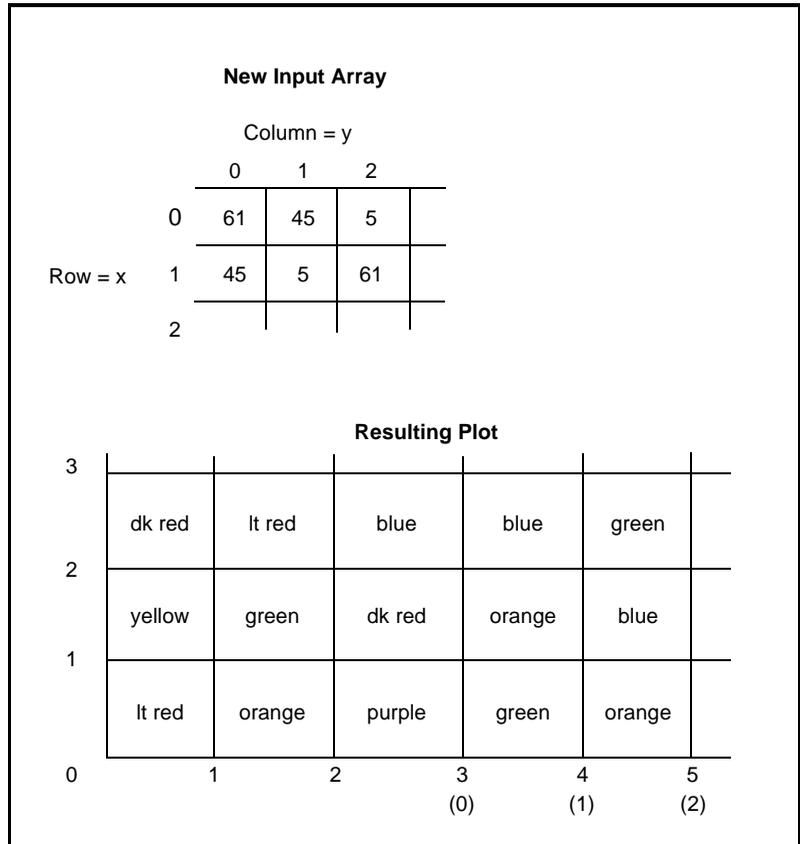
When the chart display is full, the oldest data scrolls off the left side of the chart.

See examples of intensity charts in
`examples\general\graphs\intgraph.llb`.

Intensity Chart Options

The intensity chart shares many of the optional parts of the other charts, most of which can be shown or hidden from the **Show** submenu of the graph pop-up menu. These items include a palette you use to change scaling and format while the VI is running. In addition, because the intensity chart has a third dimension (color), a scale similar to a color ramp control defines the range and mappings of values to colors.

Following is a picture of a chart showing all of these optional components.
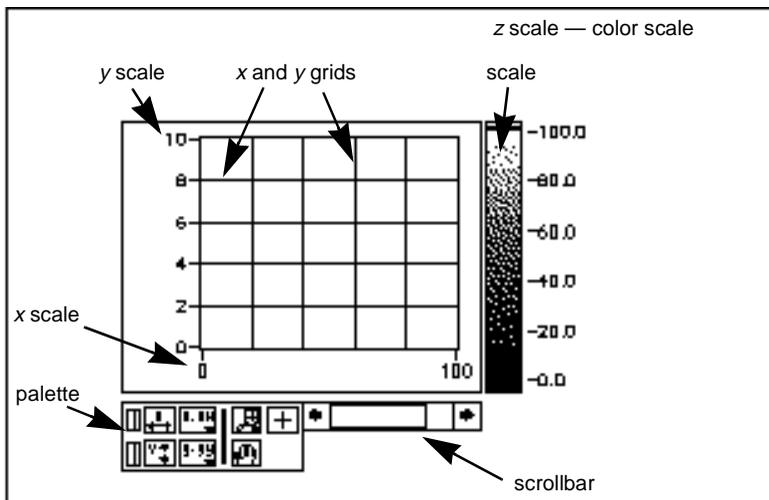
Intensity charts have a number of items you can use to customize your data display. The intensity chart pop-up menu is shown below in the following illustration.

Most of these items are identical to the menu items for the waveform chart. With the **Show** menu, you can show and hide the color scale for the z scale. The *X Scale* and *Y Scale* menus are identical to the corresponding menus for the waveform chart.

The intensity chart maintains a history of data from previous updates. You can configure this buffer by selecting **Chart History Length…** from the chart pop-up menu. The default size for an intensity chart is 128 points. Notice the intensity chart display can be very memory intensive. For example, to display a single precision chart with a history of 512 points and 128 *y* values requires 512 * 128 * 4 bytes (size of a single), or 256 kilobytes. If you want to display large quantities of data on an intensity chart, make sure enough memory is available.

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.

Color Mapping

You can set the color mapping interactively in the same way you define the colors for a color ramp numeric control. See the *Color Ramp* section of Chapter 9, *Numeric Controls and Indicators*, for more details.

Using the attribute node, there are two ways to set the colors programmatically. First, you can specify the value-to-color mappings to the attribute node in the same way you do it with the color scale. For this method, you specify the **Z Scale Info: Color Array** attribute. This attribute consists of an array of clusters, in which each cluster contains a numeric limit value, along with the corresponding color to display for that value. When you specify the color table in this manner, you can specify an upper out-of-range color using the **Z Scale Info: High Color** attribute, and a lower out-of-range color using the **Z Scale Info: Low Color**. The total number of colors is limited to 254 colors, with the lower and upper out of range colors bringing the total to 256 colors. If you specify more colors, then the 254 color table is created by interpolating between the specified colors.

The second way to set the colors programmatically is to specify a color table using the **Color Table** attribute. With this method you can specify an array of up to 256 colors. Data passed to the chart is mapped to indices in this color table based upon the color scale. If the color scale ranges from zero to 100, a value of zero in the data is mapped to index one, and a value of 100 is mapped to index 254, with interior values interpolated between one and 254. Anything below zero is mapped to the out of range below color (index zero), and anything above 100 is mapped to the out of range above color (index 255).

☞ **Note**   *The colors you want your intensity chart (or graph) to display are limited to the exact colors and number of colors your video card can display. You are also limited by the number of colors allocated for your display.*

Intensity Graphs

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

For an example of an intensity graph, see
`examples\graphs\intgraph.llb`.

Intensity Graph Data Type

The intensity graph accepts two-dimensional arrays of numbers, where each number is mapped by the chart to a color.

Rows of the data you pass in display as new columns on the chart. If you want rows to appear as rows, use the **Transpose Array** item from the chart pop-up menu.

Intensity Graph Options

The intensity graph works much like the intensity chart, except it does not have the chart update modes. Because each update replaces the previous data, it does not have a scrollbar, and it does not have history options.

The intensity graph can have cursors like other graphs. Each cursor displays the x, y, and z values for a specified point on the graph. See the

section of this chapter for more information on manipulating the cursors on the graph.

You set the color mapping the same way you set it for the intensity chart.

16

ActiveX Controls

This chapter describes the ActiveX Container capability, which enhances the interactions between G-based software and other applications.

ActiveX Front Panel Enhancements

The front panel includes the **ActiveX** subpalette, which contains two ActiveX controls, the ActiveX Container and ActiveX Variant, as shown in the following illustration.
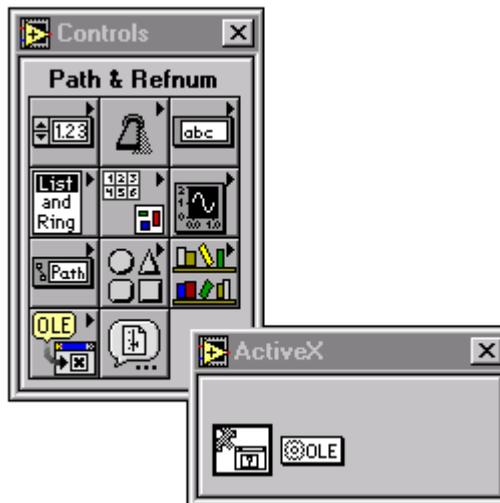
ActiveX Variant Control and Indicator

The ActiveX Variant control and indicator let you pass ActiveX Variant data into the software, so ActiveX client functionality is enhanced. When you place the ActiveX Variant control and indicator on the block diagram it appears as follows.

Use this front panel object when ActiveX Variant data is converted to display data.

ActiveX Container

The ActiveX container permits integration of ActiveX objects into a VI panel alongside built-in controls. You can use this container to display ActiveX controls and embedded documents on the front panel. The following illustration shows the container as it appears when you first place it on the front panel.

The ActiveX Container appears as an automation refnum terminal on the block diagram. You can wire this terminal to Automation functions and, therefore, control the object embedded in the container. If that object has no Automation interface, the terminal has an invalid refnum and cannot be used with the Automation functions.

To insert an object into the front panel container, pop up and select **Insert ActiveX Object**. The **Select ActiveX Object** dialog box appears.

There are two general types of objects that can be contained, Active X Documents and ActiveX Controls.

ActiveX Documents —These objects can be contained by the container object and edited by popping up and selecting the **Edit Object** option. This brings up a new window to edit the object. Some documents support an automation interface and can be used with automation functions on the diagram.

ActiveX controls—These objects can be activated and operated upon inside the container object itself. They have automation interfaces and can be controlled using Automation function on the diagram.

Objects can be dropped in three ways. This is enumerated by the selection at the top of the dialog.

Create Document—Choose a document type from those registered with the system.

Create Object from File—Choose a document located anywhere in the file system. The object can either remain linked to the file, or it can be copied statically into the panel.

Create Control—Choose an ActiveX control from those registered with the system.
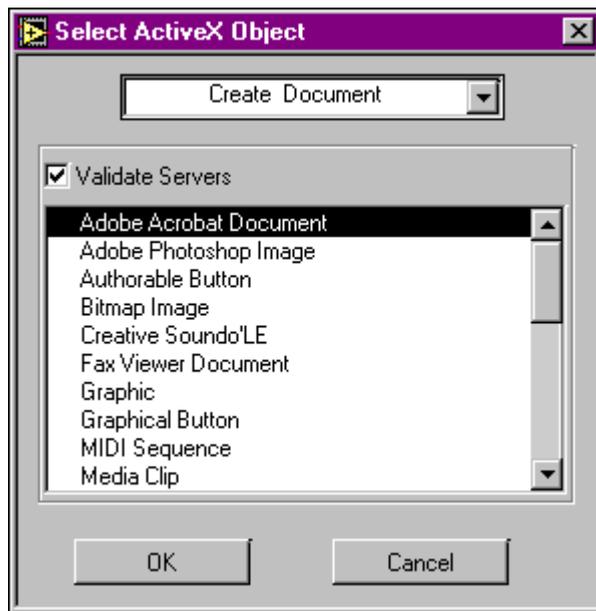
Two types of objects, controls and documents, can be placed in the container. You can create a new control or document or insert an existing document. To create a new document, select **Create Document** (as shown in the previous illustration) and select the object type from the items listed.

To insert an existing document or file, select **Create Object from File** and the dialog box changes as illustrated in the following illustration.
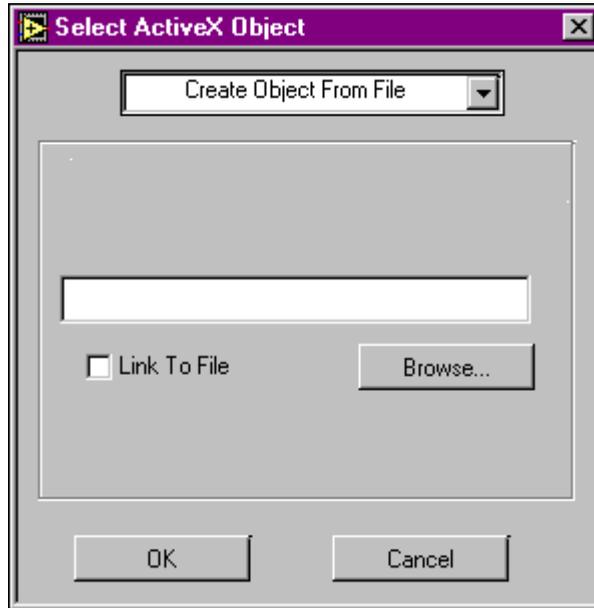
Use **Browse…** to find the document for insertion. If you select the **Link To File** option, the document updates when the front panel object is updated. If you do not select **Link To File**, a static version of the document is inserted.

To insert an existing control, select **Create Control** and the dialog box changes, as shown in the following illustration.
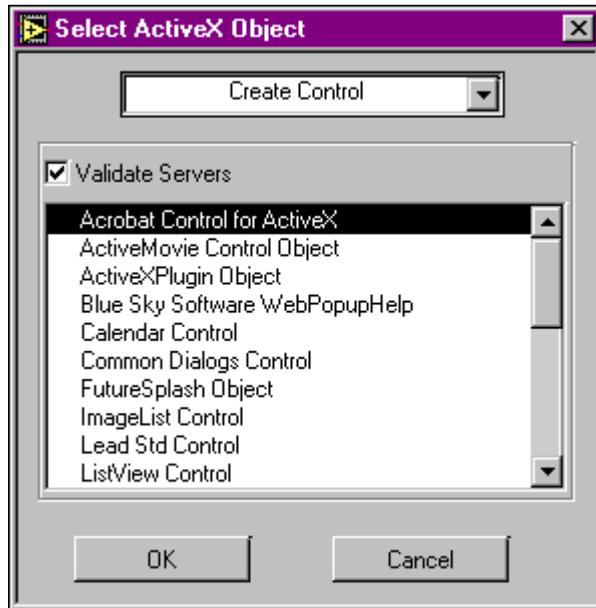
The available control types are registered with the system.

Building ActiveX Palettes

Under **Project»Import ActiveX Controls...** you also can convert a set of
ActiveX Controls into Custom Controls and add them to the palette menus.

When the menu item is selected, it brings up a list of Controls in the system.
Then you can select one or more controls. You are prompted to select a
directory or .llb file in which to store the custom controls. The default
destination is user.lib, because all files and directories in that directory
automatically appear in the palette menus.

Part III

Block Diagram Programming

This section contains information about the components required to build and manipulate a block diagram in G.

Part III, *Block Diagram Programming*, contains the following chapters.

- Chapter 17, *Introduction to the Block Diagram*, describes terminals and nodes—two of the three elements you use to build a block diagram. The third element, wiring, is covered in Chapter 18, *Wiring the Block Diagram*.

- Chapter 18, *Wiring the Block Diagram*, explains how to connect terminals on the block diagram by wiring them together.

- Chapter 19, *Structures*, describes how to use the For Loop, While Loop, Case Structure, and Sequence Structure. These structures are in the **Functions»Structures** palette.

- Chapter 20, *Formula Nodes*, describes how to use the Formula Node to execute mathematical formulas on the block diagram. The Formula Node is available from the **Functions»Structures** palette.

- Chapter 21, *VI Server*, describes the mechanism for controlling VIs and applications programmatically. This chapter also describes how to control VIs or applications remotely.

- Chapter 22, *Attribute Nodes*, describes how to use attribute nodes to set and read attributes of front panel controls programmatically. Some useful attributes include display colors, control visibility, menu strings for a ring control, graph or chart plot colors, and graph cursors.

- Chapter 23, *Global and Local Variables*, describes how to define and use global and local variables. Use global variables to access a particular set of values easily from multiple VIs. Local variables serve a similar purpose within a single VI.

17

Introduction to the Block Diagram

This chapter describes terminals and nodes—two of the three elements you use to build a block diagram. The thir5d element, wiring, is covered in Chapter 18, *Wiring the Block Diagram*.

Terminals and Nodes

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

Terminals are ports through which data passes between the block diagram and front panel, as well as between nodes on the block diagram. They also underlie the icons of functions and VIs. The following illustration shows an example of a terminal pattern on the left and its corresponding icon on the right. To display the terminals for a function or VI, pop up on the icon and select **Show»Terminals**.

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

Wires are the data paths between input and output terminals.

Terminals

G has many types of terminals. In general, a terminal is any point to which you can attach a wire. G 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, also are called *source terminals*. Node input terminals and front panel indicator terminals also are 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 G control and indicator terminals are shown in Table 17-1, *G Control and Indicator Terminal Symbols*. Each symbol encloses a picture and has a color that suggests the data type of the control or indicator and, in the case of numerics, the representation as well. Control terminals have a thicker border than indicator terminals. Because a terminal belongs to its corresponding control or indicator, you cannot delete a terminal; if you want to delete a control or an indicator, do it from the front panel.

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

Table 17-1. G Control and Indicator Terminal 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

Table 17-1. G Control and Indicator Terminal Symbols (Continued)

Control	Indicator	Description	Color
I32	**I32**	32-bit integer (long word)	Blue
I16	**I16**	16-bit integer (word)	Blue
I8	**I8**	8-bit integer	Blue
⊡Dẵ	⊡Dẵ	Cluster	Brown or Pink
[SGL]	**[SGL]**	Array	Varied
⊡⌐	⊡⌐	Path	Aqua
⌂	⌂	Refnum	Aqua
TF	**TF**	Boolean	Green
abc	**abc**	String	Pink
◀▶	◀▶	Enum	Blue
⚠	⚠	OLE Variant	Purple

Constants

Constants are terminals on a block diagram that supply data values directly to the block diagram. *User-defined constants* are defined by editing the constant prior to program execution, and you cannot change their values during execution. *Universal constants* have fixed values.

User-Defined Constants

The easiest way to create a user-defined constant is to pop up on an input or output and select **Create Constant**. These constants are also available from various palettes in the **Functions** palette, depending on their type. Most are located at the bottom or top of the relevant palette, such as the numeric, enumerated, and ring constants in the bottom row or the **Numeric** palette. Three constants—the color box, listbox symbol ring, and error ring constants—are located in the **Numeric»Additional Numerics** subpalette. The path constant is in the **File I/O»File Constants** subpalette.

The constants are shown in the following illustration.
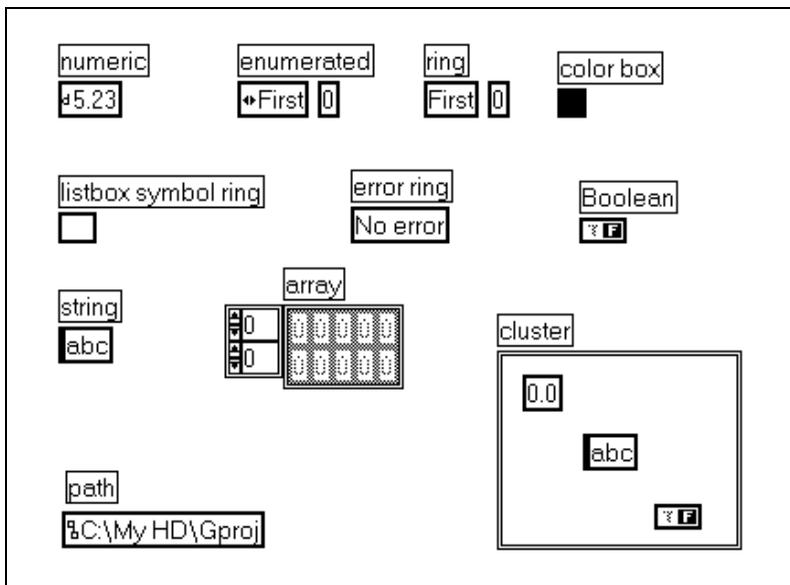
You can label your constant by popping up on the constant and selecting
Show»Label. A highlighted text box appears, indicating you can type into
it immediately without changing the tool.

The user-defined constants, like labels, resize automatically as you enter
information into them. If you resize or change the shape of a label or string
constant, you can select **Size to Text** from its pop-up menu, and the
constant or label resizes itself automatically to fit its contents.

You use the Operating tool to set the value of a user-defined constant the
same way you set the value of a digital control, Boolean slide switch, or
string control on the front panel. The *numeric* and *string constants*
resemble front panel numeric controls and have pop-up menus similar to
the pop-up menus of controls, as shown in the following illustration.

Numeric Constant

Show	▶
Data Operations	▶
Replace	▶
Representation	▶
Format & Precision...	
Create Constant	
Create Control	
Create Indicator	

String Constant

Show	▶
Data Operations	▶
Replace	▶
✓ Size to Text	
✓ Normal Display	
'\' Codes Display	
Password Display	
Hex Display	
Limit to Single Line	
Create Constant	
Create Control	
Create Indicator	
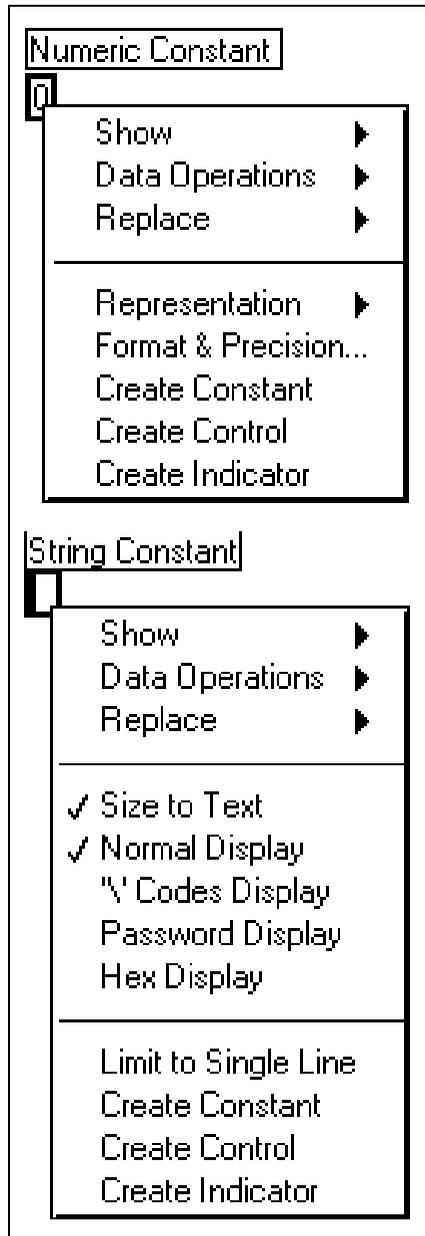
You can use the arrow keys to increment or decrement a new numeric constant or one with its value selected. This is particularly useful with programming control parameters that have low values like 1, 2, or 3.

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

The *enumerated constant* is similar to the ring constant except the mnemonics (strings associated with an integer value) are considered part of the type. When an enumeration is wired to the selection terminal of a Case Structure, cases are named according to the mnemonics of the enumeration rather than traditional numeric values. The numeric representation of an enumeration is always an unsigned byte, word, or long.

You can select colors from the *color box constant* to use with the color box control, a control whose values correlate to specific colors. Set the color box by clicking it with the Color tool or the Operating tool and choosing the color you want from the color palette.

The *listbox symbol ring* constant is used to assign symbols to items in a listbox control.

The *error ring constant* is a predefined ring. You click the constant with the Operating tool and select the error message you want from the dialog box that appears, as shown in the following illustration. This ring is useful with functions that return error codes or error clusters, such as the file I/O functions, because it makes the block diagrams more descriptive. For example, if you try to open a nonexistent file using the Open File function, the function returns an error code. 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).

No error 0

✓ No error
　Argument error
　Memory is full
　Out of zone
　End of file
　File already open
　Generic file I/O error
　File not found
　File permission error
　Disk is full
　Duplicate path
　Too many files open
　System feature not enabled
　▼
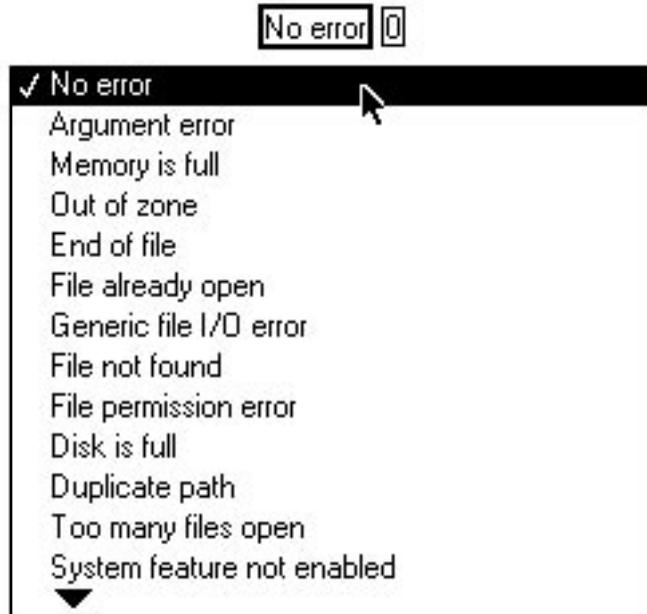
You can use the *path constant* to create a constant path value on the block
diagram.

The path, string, and color box constants are resizable, yet the Boolean,
numeric, ring, and enumeration constants are not. A numeric constant
adapts representation to whatever source you type. For example, if you type
1.1 the constant becomes DBL. This is controlled by the **Adapt to Source**
setting in the **Representation** pop-up menu.

The Adapt to Source representation indicates whether a numeric constant
automatically determines its representation when you type in a new value.
Block diagram numeric constants are set to Adapt to Source by default
when you create one. For example, when you type 2 into a DBL numeric
constant, it automatically changes its type to I32. If you change the
representation of a constant through the pop-up menu, it implicitly turns off
Adapt to Source so the constant remains the specified type. If you want to
restore the old behavior of automatically determining its type from any
value entered, you must select **Adapt to Source** from the representation
pop-up menu.

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** item from the constant pop-up menu.

Universal Constants

Universal constants are of two types: universal numeric constants and universal string constants.

- Universal numeric constants—Set of precise and commonly used mathematical and physical values, such as pi and the speed of light.

- Universal string constants—Set of commonly used nondisplayable string characters, such as line feed and carriage return.

For more information on these constants, see the LabVIEW or BridgeVIEW *Online Reference*, available by selecting **Help»Online Reference»Function and VI Reference**.

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. Functions are discussed in the *Functions* section of this chapter. SubVIs are discussed in Chapter 3, *Using SubVIs*.

Structures, which supplement the G dataflow programming model to control execution order, are described briefly in the *Structures* section of this chapter. For in-depth information, see Chapter 19, *Structures*.

CINs are interfaces between the block diagram and code you write in conventional programming languages, such as C or Pascal. For more information refer to the *LabVIEW Code Interface Reference Manual*, available in portable document format (PDF) on your software program disks or CD.

Formula nodes, which supplement G functions by using formulas on the block diagram, are described in Chapter 20, *Formula Nodes*.

Attribute nodes, which change control attributes programmatically, are described in Chapter 22, *Attribute Nodes*.

Functions

Functions are elementary nodes built into G. They perform elementary operations like adding numbers, file I/O, and string formatting. Functions do not have front panels or block diagrams. When compiled, they generate inline machine code. See **Online Reference»Function and VI Reference** for detailed descriptions of individual functions.

You select functions from the **Functions** palette, as shown in the following illustration.
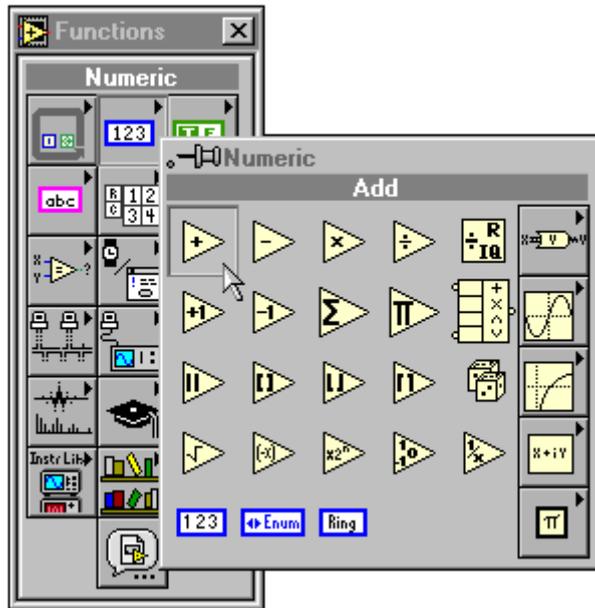
When you select a function, its icon appears on the block diagram. To display its label, pop up on the icon and select **Show»Label**. You can change the label by highlighting the text with the Labeling tool and retyping. You can use the label of the function to annotate its purpose in the block diagram.

You can use the Help window to see how to wire the function, or you can select **Show»Terminals** from the icon pop-up menu, as shown in the following illustration to see precisely where the terminals are located.
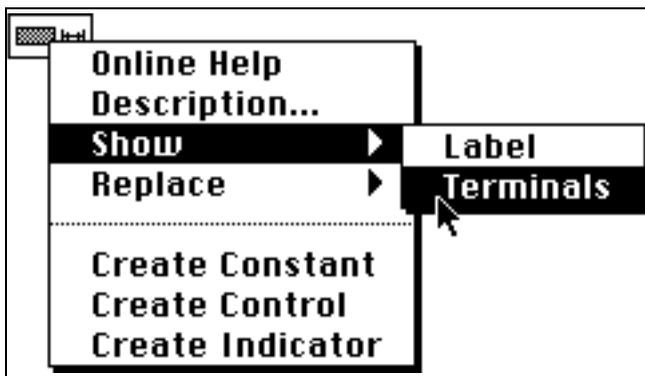
When terminals are showing, select **Show»Terminals** again to show the icon for the function instead.

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

Some 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 cursor in the same way you resize other objects, as shown in the following illustration. You can enlarge or reduce functions, but you cannot shrink a function if it causes any wired terminals to disappear.
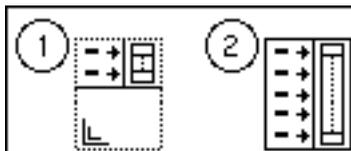
You also can change the number of terminals with the **Add Input** and **Remove Input** commands from the pop-up menu for a terminal, as shown in the illustration that follows. The **Remove Input** command removes the terminal on which you popped up and disconnects the wire, if the terminal is wired. The **Add Input** command adds a terminal immediately after the

terminal on which you popped up. The full names of these commands vary with the function.
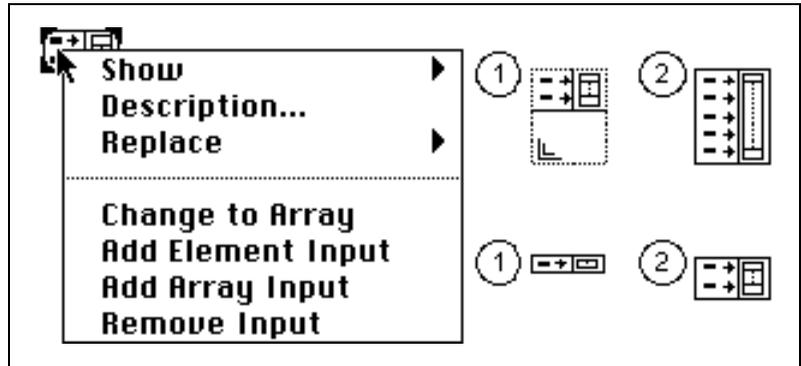
Structures

When you are programming, sometimes it is necessary to repeat sections of code a set number of times or while a certain condition is true. In other situations, it is necessary to execute different sections of code when different conditions exist or execute code in a specific order. G contains four special nodes, called *structures*. They help you to control code execution in ways that otherwise are not possible within the G dataflow framework.

Each structure has a distinctive, resizable border you use to enclose the code that executes under the structure's special rules. For this reason, the diagram inside the structure border is called a subdiagram. You can nest subdiagrams.

G uses the following structures to execute code.

- For Loop—Repeats execution a set number of times.

- While Loop—Repeats the execution of its subdiagram while a condition is TRUE.

- Case Structure—Contains multiple subdiagrams, only one of which executes depending on the value passed to its selector terminal.

- Sequence Structure—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 19, *Structures*, for more information.

Replacing and Inserting Block Diagram Objects

Suppose you use an Increment function in a block diagram and the Decrement function is actually the proper choice. You can delete the Increment function node and then select the Decrement node from the **Functions** palette and rewire it. You also can use the **Replace** item in the Increment node pop-up menu. Selecting **Replace** accesses the **Functions** palette, from which you can choose the Decrement function. The advantage of this method is the new node goes in place of the old node 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 might have broken wires.

You also can use **Replace** to replace a constant with another constant or a structure with another similar structure, such as a While Loop with a For Loop.

☞ **Note**   *When using* **Replace** *on a subVI, if you select a VI whose name is the same as one already in memory, the replaced node refers to the VI already in memory, not the VI you selected.*

Wire pop-up menus have an **Insert** item. Choosing **Insert** accesses the **Functions** palette, from which you can choose any function or VI on the palette. The editor 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 might not connect to the terminal you expected.

Adding Constants, Controls, and Indicators Automatically

Instead of creating a constant, control, or indicator by selecting it from a menu and wiring it manually to a terminal, you can pop up on the terminal and choose **Create Constant**, **Create Control**, or **Create Indicator** to create an object with an appropriate data type automatically. Assuming it makes sense to do so, the newly created constant, control, or indicator is wired automatically for you. If the terminal corresponds to a type definition, you create a type definition constant, control, or indicator. If the type definition is changed later, the control, indicator, or constant is updated.

For example, if you need a constant for the **select mode** input of a File Dialog function, you can pop up on the input and select **Create Constant**. The editor creates an enumerated type containing all of the permitted values for the **select mode** input for you. Assuming the mode input is not wired already to something else, the new constant is wired automatically.
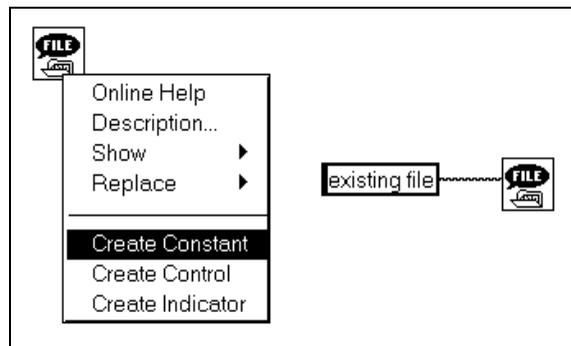
Other useful places where you can **Create Constant**, **Create Control**, or **Create Indicator** include the outputs of functions or VIs, wires, constants, and terminals for front panel controls or indicators.

18

Wiring the Block Diagram

This chapter explains how to connect terminals on the block diagram by wiring them together.

Wiring Techniques

You use the Wiring tool to connect terminals. The cursor point or hot spot of the tool is the tip of the unwound wire segment, as shown in the following illustration.
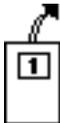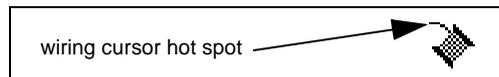
The symbol to the left represents the mouse. In the wiring illustrations in this chapter, the arrow at the end of this mouse symbol shows what area to click, and the number printed on the mouse button indicates how many times to click it.
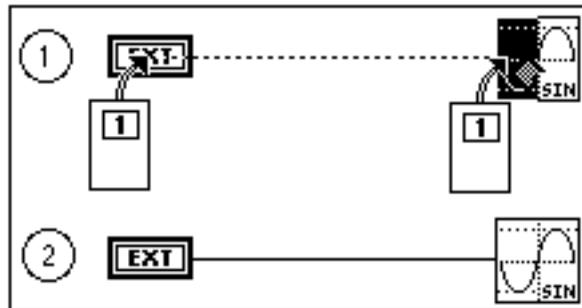
To wire from one terminal to another, click the Wiring tool on the first terminal, move the tool to the second terminal, and then click the second terminal as shown in the preceding illustration. Do not hold down the mouse button as you move the mouse from the first terminal to the second terminal. It does not matter which terminal you click first. The terminal area blinks when the hot spot of the Wiring tool is correctly positioned on the terminal. Clicking that terminal connects a wire to it. When you make the first connection, a wire is drawn between the terminal and the tool as

you move the cursor across the diagram, as if the wire were unwinding from a spool. You do not need to hold down the mouse button.

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

You can wire directly from a terminal outside a structure to a terminal within the structure using the basic wiring operation. A *tunnel* is automatically created where the wire crosses the structure boundary, as shown in the following illustration.
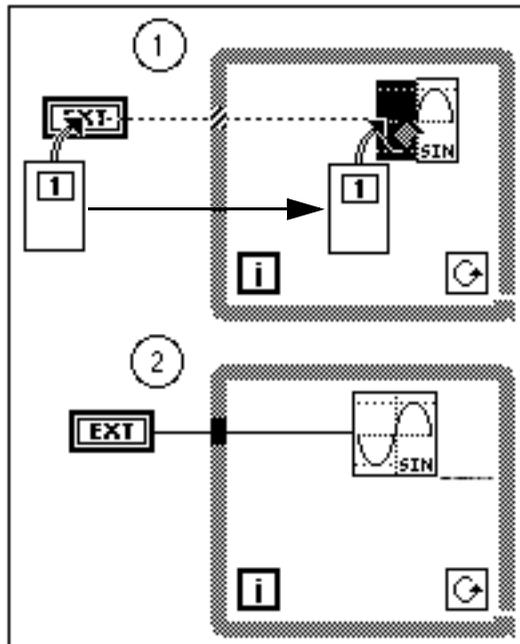
Wires unwind from terminals vertically or horizontally, depending on the direction in which you first move the Wiring tool. Wires move vertically if you move the tool up or down, and they move horizontally if you move the tool left or right. The connection is centered on the terminals, regardless of the exact position of the hot spot when you click the mouse button on it as shown in the following illustration.
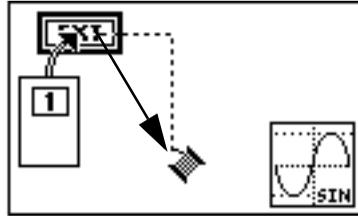
You can bend your wires at a ninety degree angle once without clicking it. You click an open area to tack the wire and change direction, as shown in the following illustration.
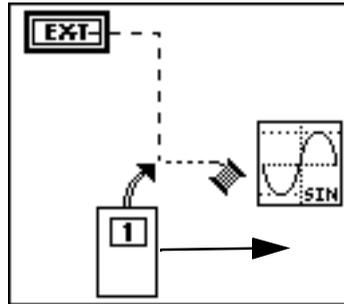
☞ **Note**   *You also can change between horizontal and vertical directions by pressing the spacebar. You can untack the last tack point by pressing* <Ctrl-click> **(Windows)**; <option-click> **(Macintosh)**; <middle button-click>**(UNIX).** *If the last tack point is the terminal or wire on which you first clicked, untacking removes the wire completely.*

You can double-click with the Wiring tool to begin or terminate a wire in an open area as shown in the following illustration.

| terminating in an open area | beginning in an open area |
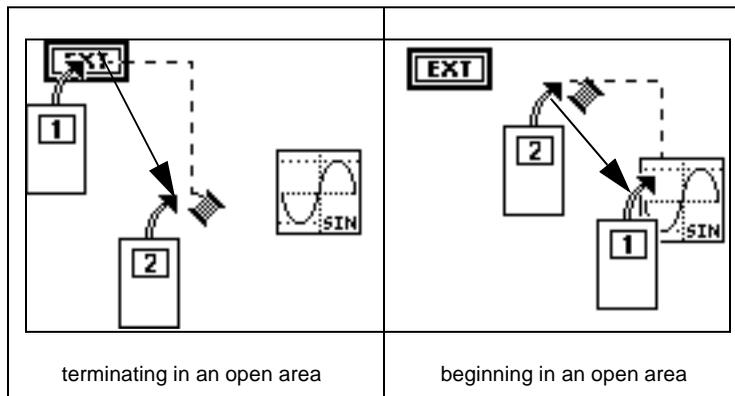
When wires cross, a small gap appears in the first wire drawn, as if it were underneath the second wire, as shown here.
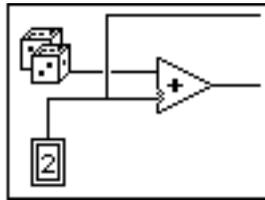
When you are wiring a complicated function or subVI, it is important to pay attention to the wire stubs and the tip strip that appears as the Wiring tool approaches the VI icon.

Wire stubs, the truncated wires shown around the VI icon to the left, indicate the data type of each terminal by their style, thickness, and color. (For details, refer to the *G Quick Reference Card*.) Dots at the end of the stubs indicate inputs. Outputs have no dots. The direction in which the stubs are drawn indicates the suggested wiring direction to produce clean diagrams. When you wire a terminal, the wire stub no longer appears for that terminal.

The tip strip, the box containing text shown to the left, indicates the name of the terminal to which you wire when you click the mouse button.

You also might want to take advantage of the **Help** window feature that highlights each connector pane terminal. With this feature, you can see exactly where wires must connect. The three connections of the Add function are shown as a simple example in the following illustration.

Block Diagram	Help Window
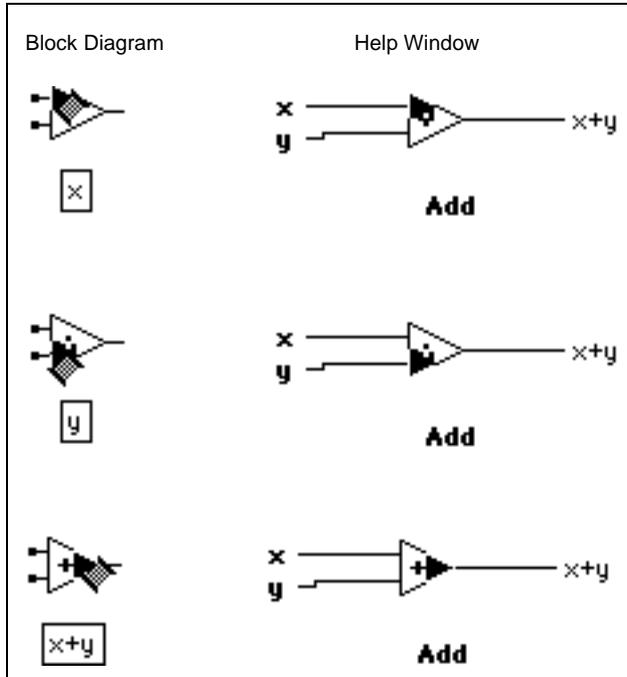
☞ **Note**   *The Help window feature illustrated above does not work for functions that can be grown. The Array Bundle function is one example.*

Wire Stretching

You can move wired objects individually or in groups by dragging the
selected objects to the new location using the Positioning tool. The wires
connected to the selected objects stretch automatically. The wire stretching
capability is demonstrated in the following illustration.
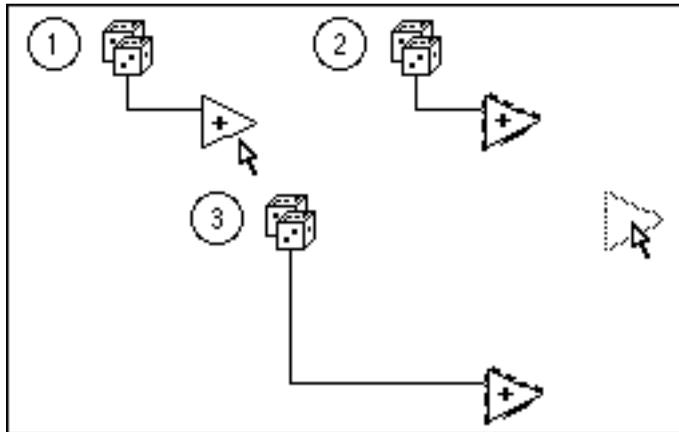
If you duplicate the selected objects or move them from one diagram
into another—for example, from the block diagram into a structure
subdiagram—the connecting wires remain behind, unless you select them
as well.

✑ **Note**   *Wire stretching occasionally creates wire stubs or loose ends, discussed in the*
*Wire Stubs and Loose Ends **subsections of the** Solving Wiring Problems **section of**
this chapter. You must remove the stubs or loose wires for the VI to execute. The
easiest way to do this is to select the Edit»Remove Bad Wires **command. This also**
removes unneeded loops in wires.*

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. Double-clicking it selects a branch.
Triple-clicking it selects an entire wire. Press the `<Delete>` key or
`<Backspace>` key to remove the selected portion of wire. This process is
shown in the following illustration.
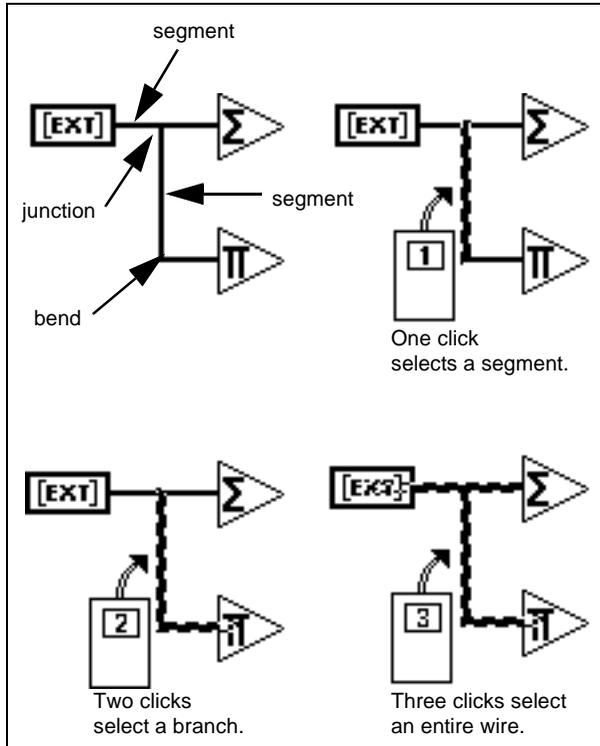
Repositioning a wired object might result in an extra segment or oddly
positioned wire segment. To reposition a wire segment, drag it to the new
location with the Positioning tool. You can reposition one or more segments
by selecting and dragging them. Use the <Shift> key to restrict the wire
drag in horizontal or vertical direction. The direction in which you initially
move determines if the wire is limited to horizontal or vertical translation.
You also can move selected segments one pixel at a time by pressing the
arrow keys on the keyboard. Hold down the <Shift> key while pressing
the arrow keys to move selected segments several pixels at a time. Adjacent,
unselected segments stretch to accommodate the change. You can select
and drag multiple wire segments, even discontinuous segments,
simultaneously.

When you move a tunnel, a wire connection normally is maintained
between the tunnel and the wired node, as shown in the following
illustration.
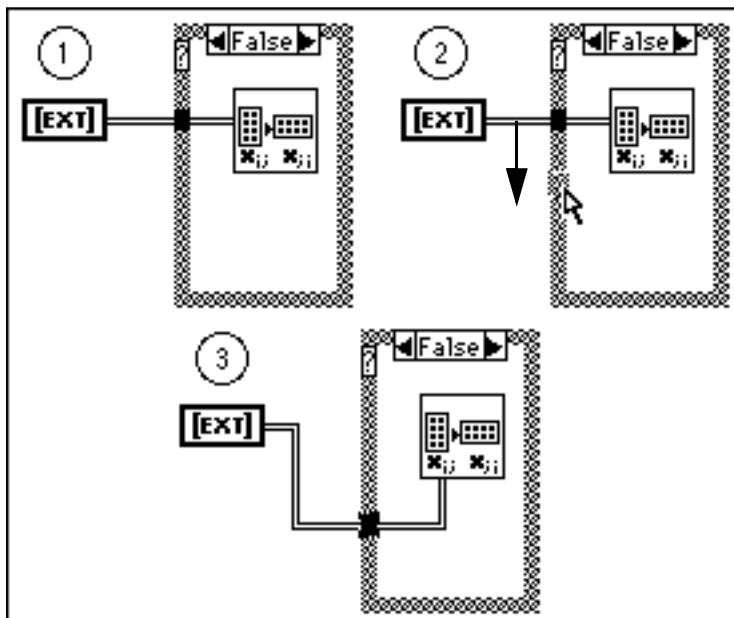
Moving a tunnel sometimes creates an extra wire segment that lies beneath
the structure border, however. You cannot select and drag this segment
because it is hidden, but it disappears if you drag the segment connected to
it, as shown in the following illustration. If you are unsure about which wire
is connected to a tunnel, triple-click the wire.
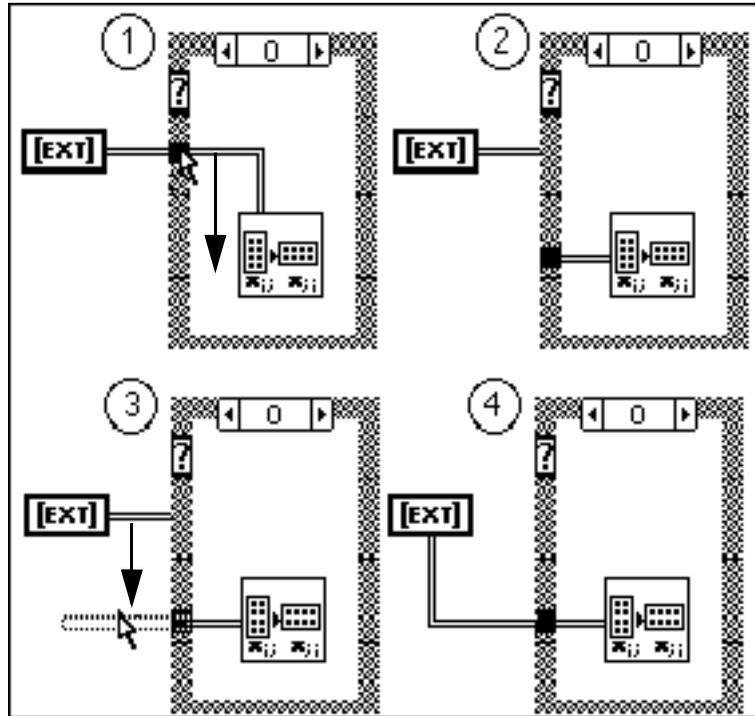
To select the parts of a wire inside and outside a loop structure at the same time, select the part of the wire on one side of the structure and hold down the <Shift> key while you select the part of the wire on the other side of the structure. You can add an object to a group of previously selected objects by holding down the <Shift> key while you select the new object. Also, you can drag a selection rectangle around both parts of the wire. The structure is not selected unless you completely surround it with a selection rectangle. Other nodes must only touch the rectangle to be selected.

Wiring Off-Screen

You might need to wire an object that is on-screen to one that is off-screen when, for example, the objects are too far apart to fit on-screen at the same time. To wire an on-screen object to and off-screen object, first you click with the wiring tool on a terminal of the on-screen object. Then you can scroll the diagram automatically while you are wiring by dragging the Wiring tool slightly past the edge of the block diagram When the second object is on-screen, click a terminal of the second object to complete wiring.

Solving Wiring Problems

Bad Wires

When you make a wiring mistake, a *broken wire*, indicated by a dashed line, might appear. Sometimes you have a faulty wiring connection that is not visible because the broken wire segment is very small or is hidden behind an object. If the **Run** button shows a broken arrow (as shown at the left) but you cannot see any problems in the block diagram, select **Edit»Remove Bad Wires** in case there are hidden, broken wire segments. If the Run button returns to its unbroken state (shown at the left), you corrected the problem. If not, click the **Broken Run** button to see a list of errors.

If you do not know why a particular wire is broken, pop up on the broken wire and choose **List Errors** from the pop-up menu, shown below. The **Error List** dialog box appears and lists the errors. Click one of the errors. Doing this selects the erroneous wire, which you then can delete or repair.

For more information, see the *Debugging Broken VIs* and *Debugging Executable VIs* sections in Chapter 4, *Executing and Debugging VIs and SubVIs*.

The following list shows some possible wire errors.

- Wire:type conflict
- Wire:dimension conflict
- Wire:element conflict
- Wire:refnum conflict
- Wire:class conflict
- Wire:enumeration conflict
- Wire:unit conflict
- Wire:has multiple sources
- Wire:has no source

- Wire:has loose ends

- Wire:is a member of a cycle

For information on wiring Structures, see the *Structure Wiring Problems* section in Chapter 19, *Structures*.

Wire Type Conflicts

A type mismatch occurs when you wire two objects of different data types together, such as a numeric and a Boolean as shown in the following illustration.

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 refnum conflict results when you wire two different kinds of refnums or two datalog file refnums having different record types.

The class conflict error results when you wire two refnums of the same kind that have classes that differ.

The enumeration conflict error results when you wire two incompatible enumerations. Two enumerations are compatible if they have the exact same set of strings or if one has a set of strings that is identical to the first strings in the other set of strings.

The unit conflict occurs when you wire together two objects that do not have commensurable units.

Any other type mismatch results in a type of conflict error. 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 might 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 following example you must disconnect one source.
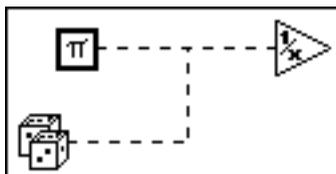
During front panel construction, you might have dropped a control when you meant to drop an indicator. If you try to wire an output value to the terminal of a front panel control, you see a multiple sources error. Pop up on the terminal and select the **Change To Indicator** command to correct this error.
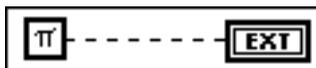
No Wire Source

Two examples of wires with no sources are shown in the following illustration. In one case, a tunnel supplies data to the Reciprocal function, but nothing supplies data to the tunnel. In the other case, two function inputs are wired together, but there is no data source for them. The solution to these problems is to wire a data source to the tunnel and to the Add and Increment functions, respectively. You also see this error if you wire together two front panel indicator terminals when one must be a control. In this case, pop up on one terminal and select the **Change To Control** command.
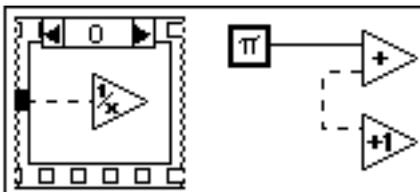
Loose Ends

Loose ends, shown in the following illustration, are wire branches 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 **Edit»Remove Bad Wires** disposes of loose ends.
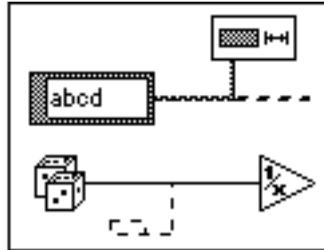
When you wire to a terminal, you seldom click the terminal when the cursor is exactly at its center. When you are off-center, a wire segment automatically attaches itself from the click point to the terminal center. If you then remove the wire going to the terminal, a loose end can remain, causing a Broken Run button. This problem is shown in the following illustration.
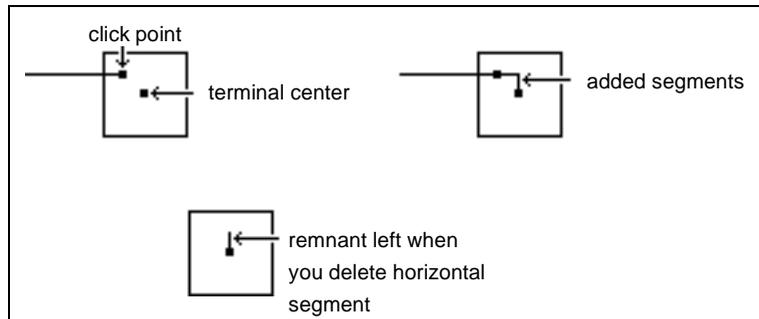
To avoid this situation, triple-click a wire to make sure all portions are selected before you delete it. You also can use **Edit»Remove Bad Wires** to delete the loose ends.

Wire Cycles

Wires must not form cycles. That is, wires must not form closed loops of icons or structures as shown in the following illustration. The execution system cannot execute cycles because each node waits on the other to supply it data before it executes.
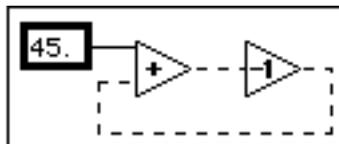
The *Shift Registers* section of Chapter 19, *Structures*, describes the correct way to feed back data in a repetitive calculation.

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 do.

☞ **Note**   *Remember, when you are unsure of what connects to a wire, you can double-click the wire to select the branch or triple-click to select the entire wire.*

Looping Wires

A loop of wire is not an error but is a poor design because it unnecessarily clutters the diagram. Double-click one of the branches to select it, then delete it. A wire loop is shown in the following illustration.
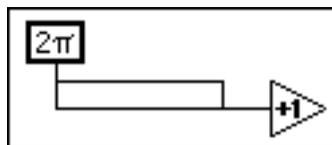
Edit»Remove Bad Wires removes one branch of these loops to clean up the wires.

Hidden Wire Segments

Try to avoid wiring under a structure border or between overlapped objects, because some segments of the resulting wire might be hidden. An example of hidden wire segments is shown in the following illustration.
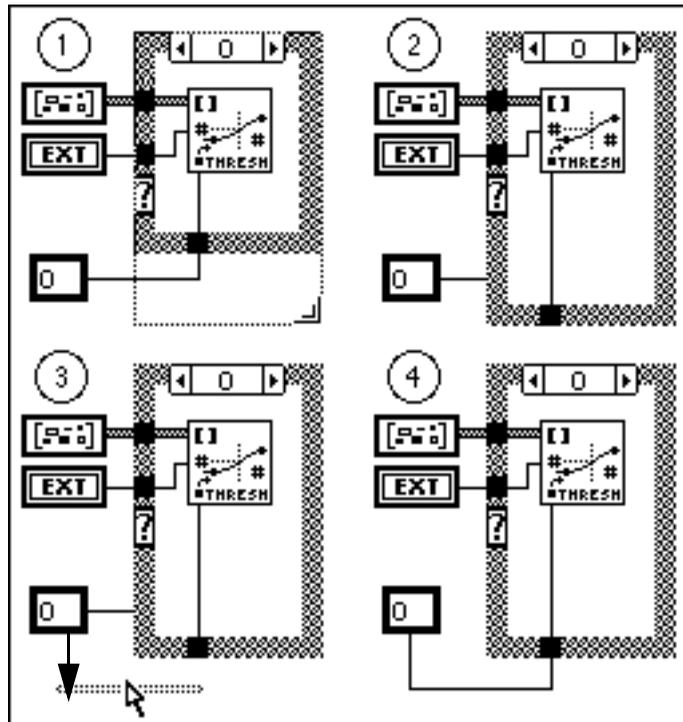
You can create hidden wire segments inadvertently, such as when you move a tunnel or enlarge a structure, as shown in parts 1 and 2 of the preceding example. Parts 3 and 4 of this example show one way you can make this diagram less confusing. You can drag the wire segment connected to the constant so 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 you click. Dragging a terminal or icon on top of a wire makes it appear as if a connection exists when it does not, as shown at the left of the following illustration (1, 2, 3 on the left).
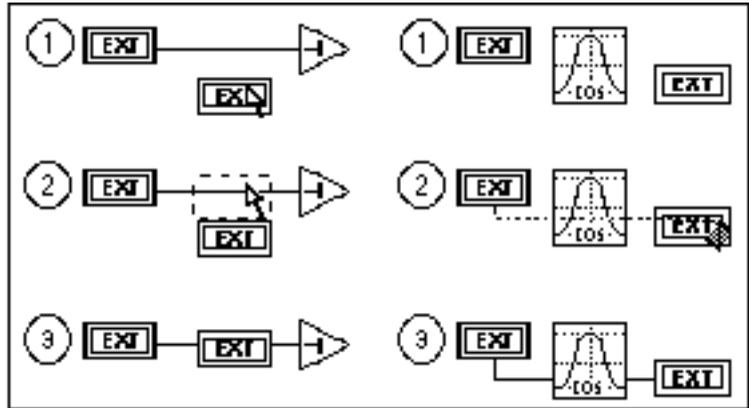
Dragging a wire through an icon or terminal also appears to make a connection, but the wire is actually behind the icon, as shown at the right of the illustration (1, 2, 3 on the right). Avoid these situations because they are confusing visually.

See the *Understanding Warnings* section in Chapter 4, *Executing and Debugging VIs and SubVIs*, for more information about this problem.

Structures

This chapter describes how to use the For Loop, While Loop,
Case Structure, and Sequence Structure. These structures are in the
Functions»Structures palette, as shown in the following illustration.
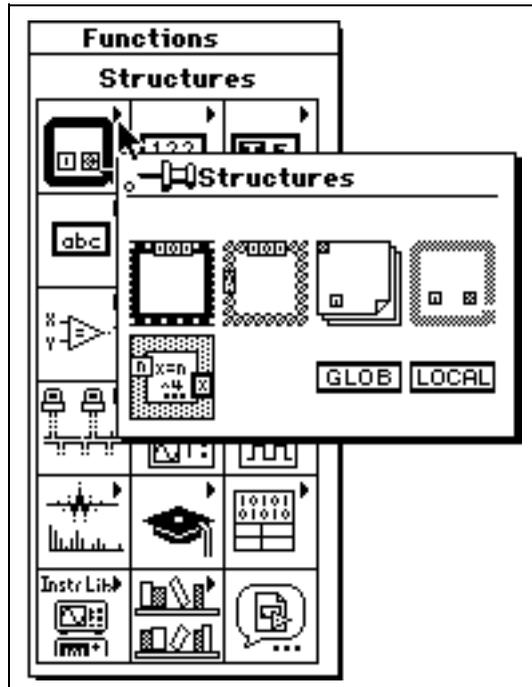
See `examples\general\structs.llb` for examples of how to use these
structures.

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 G structure is a resizeable box with a
distinctive border, as shown in the following illustration.
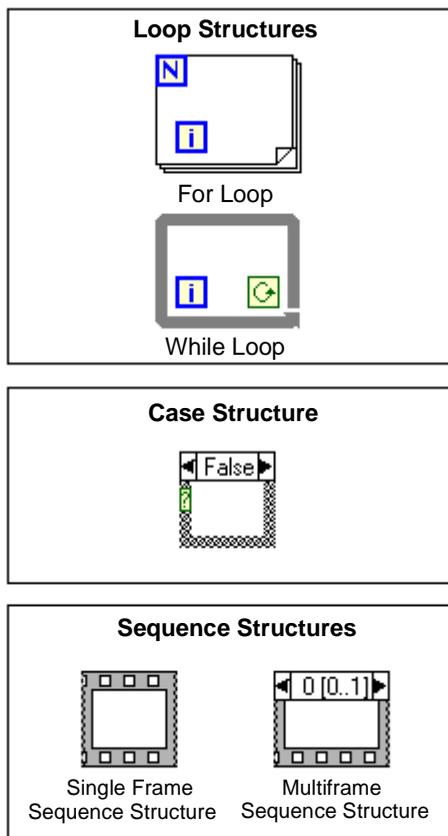
Structures behave like other nodes in that they execute automatically when
their input data is available, and they supply data to their output wires
only when execution completes. However, each structure executes its
subdiagram according to the rules described in the following sections.

A subdiagram is the collection of nodes, wires, and terminals residing
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 you construct the top-level
block diagram; subdiagrams can contain block diagram terminals, nodes
(including other structures), and wires.

You create terminals for passing data into and out of a structure automatically where wires 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.

Structures also have other terminals 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 completes (For Loop) or while a specified condition is true (While Loop).

For Loop

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 you can access the count internally.

iteration terminal

count 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, G rounds it off, if necessary, and coerces it to within the range. If you wire 0 to the count terminal, the loop does not execute.

The For Loop is equivalent to the following pseudocode.

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

While Loop
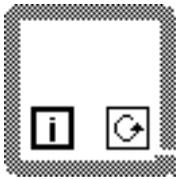
A While Loop executes its subdiagram until a Boolean value you wire to the *conditional terminal* is FALSE. G checks the conditional terminal value at the end of each iteration, and if the value is TRUE, another iteration occurs, so the loop always executes at least once. The default value of the conditional terminal is FALSE, so if it is unwired, the loop iterates only once.

conditional
terminal

iteration
terminal

The iteration terminal contains the number of completed iterations, just as it does in the For Loop.

The While Loop is equivalent to the following pseudocode.

```
Do

    Execute subdiagram (which sets condition)

While condition is TRUE
```

You can add terminals called *shift registers* to both loop structures. You use shift registers 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 a structure by dragging it inside, or by building the structure around the object.

You cannot put an object inside a structure by dragging the structure over the object. If you move a structure and it overlaps another object, the editor displays the object it considers to be in front. If you put the structure completely over another object, and the editor makes it the frontmost object, it displays a thick shadow to warn you the object is below, not inside the structure. Otherwise, the structure is hidden completely by the object. For information on which objects your application selects as frontmost, see the section *Positioning Objects* in Chapter 2, *Editing VIs*. Both situations are shown in the following illustration.
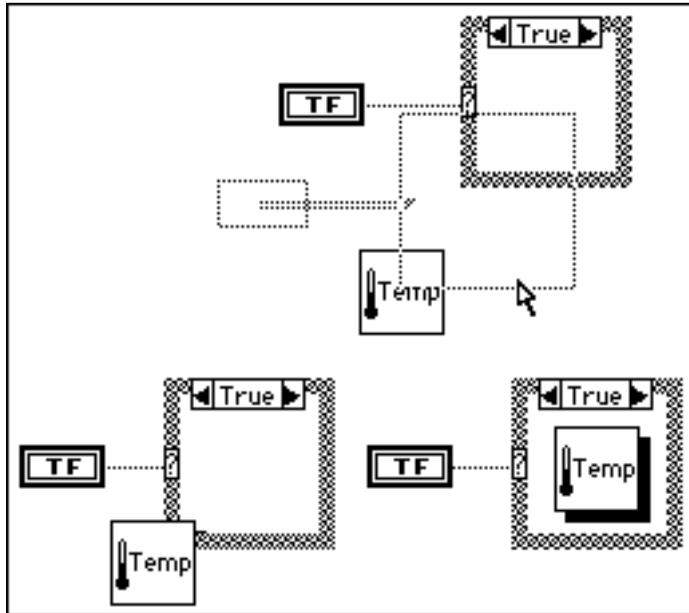
Placing and Sizing Structures on the Block Diagram

When you select a structure from **Functions»Structures** and prepare to put it on your diagram, you see a small icon of the structure in place of your cursor. This icon is ready to be placed on your block diagram. If you click and release without resizing, you are still in resizing mode until you click again. You can click the diagram while holding down the mouse and drag

the structure to the size you want. An example is shown in the following illustration. You also can size the structure *after* you drop it on the diagram by clicking any corner with the Resizing cursor and dragging.
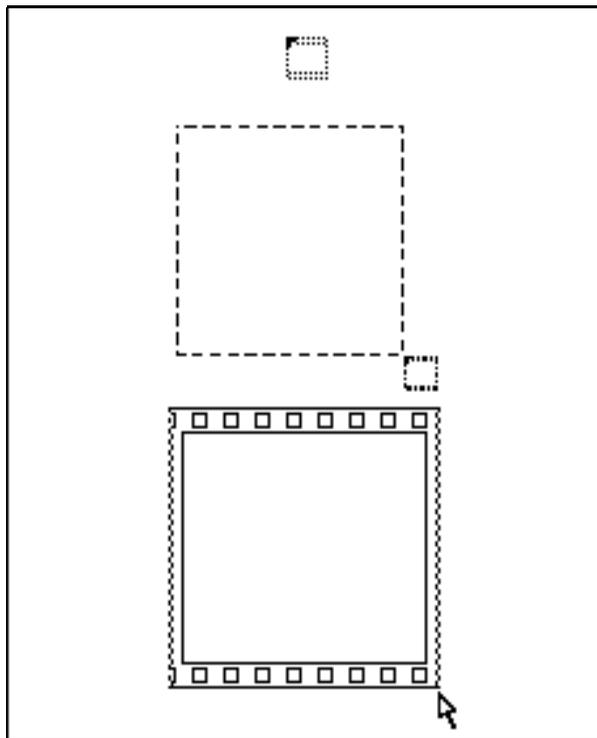
Placing Terminals inside Loops

Inputs pass data into a loop at the start of 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 value of the terminal 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 *every* iteration to determine whether it must iterate again. You can stop this While Loop, shown in the following figure, by changing the value of the front panel control to FALSE.
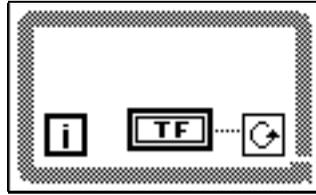
If you place the terminal of the Boolean control outside the While Loop, as shown below, you cause an infinite loop if the Boolean control is TRUE when the loop starts.
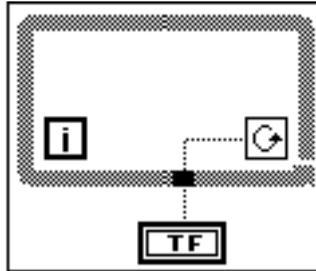
If the control was TRUE at the start, changing the value of the front panel control to FALSE does not stop the execution of this loop because the value is not propagated until the loop stops and the VI is re-run. If you inadvertently create an infinite loop, you can stop it by aborting the VI. Click the abort button to stop execution.

Auto-Indexing

For Loop and While Loop structures can index and accumulate arrays at their boundaries automatically. These capabilities collectively are called *auto-indexing*. When you wire an array from an external source to an input tunnel on the loop border and enable auto-indexing on the input tunnel, components of that array enter the loop one at a time, starting with the first component. The loop indexes scalar elements from one-dimensional arrays, one-dimensional arrays from two-dimensional arrays, and so on. The opposite action occurs at output tunnels– scalar 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.
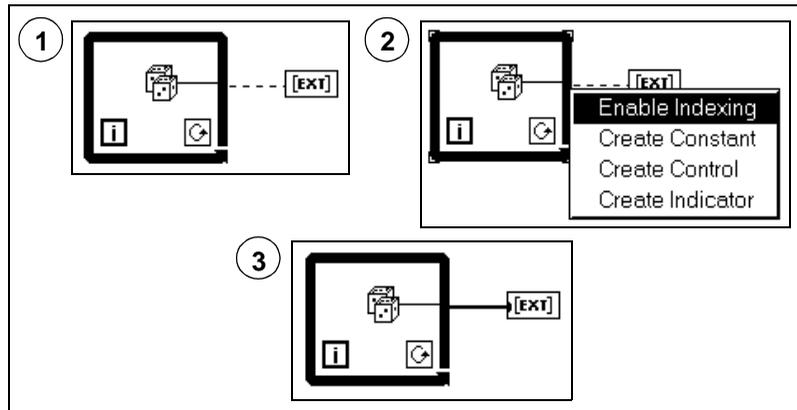
You often use For Loops to process arrays sequentially. For this reason, array tunnels have auto-indexing enabled by default when you wire an array into or out of a For Loop. While Loops are not commonly used for that purpose, so they are not enabled for auto-indexing by default. To enable or disable auto-indexing on a tunnel on the loop border, you must pop up on the tunnel at the loop border and choose **Enable Indexing** or **Disable Indexing**.

Auto-Indexing for Setting the For Loop Count

When you begin auto-indexing on an array entering a For Loop, it automatically sets the count to the size of the array, thus eliminating the necessity for you to wire to the count terminal explicitly. If you start auto-indexing for more than one tunnel, or if you also set the count explicitly, the count becomes the smallest of the choices. For example, if two auto-indexed arrays enter the loop, with 10 and 20 components respectively, and you wire a value of 15 to the count terminal, the loop executes 10 times, 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 on an output tunnel, only the element value from the last iteration of the loop is passed out.

Auto-Indexing with While Loops

When you enable auto-indexing for an array entering a While Loop, the While Loop indexes the array in the same way 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 input array, the default value for the array element type passes into the loop. Auto-indexed 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. If you are a LabVIEW user, see Chapter 5*, Arrays, Clusters, and Graphs**, of the** LabVIEW User Manual **for further information about building arrays with While Loops to prevent this problem.**

Executing a For Loop Zero Times

When you set the count to zero, a For Loop does not execute its subdiagram at all. The value of all scalar data leaving the For Loop conforms to the following rules.

- An output array created by auto-indexing at an output tunnel is empty.

- The output from an initialized shift register is the initial value. See the *Shift Registers* 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 auto-index an empty input array, or you can wire a zero or a negative number to the count terminal explicitly.

When you auto-index input arrays or set the loop count with a variable, analyze the diagram to determine whether a zero count occurs, and if so, ensure that the diagram can handle an empty array.

Shift Registers

Shift registers, which are available in For Loops and While Loops, are local variables that feed back or transfer values from the completion of one iteration to the beginning of the next. By selecting **Add Shift Register** from the loop border pop-up menu, you can create a register anywhere along the vertical boundary, as shown in the following illustration. This menu item is not available from the top or bottom edge of the structure. You can reposition a shift register along the vertical boundary by dragging it.
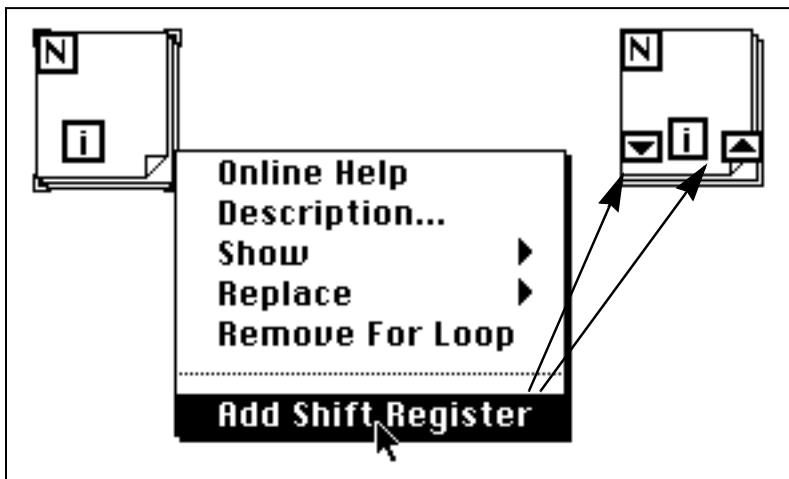
A shift register has a pair of terminals directly opposite each other on the vertical sides of the loop border. The *right terminal*, the rectangle with the up arrow, stores the data at the completion of an iteration. G shifts that data at the end of the iteration, and it appears in the *left terminal*, 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 the terminals of each registers must be of the same type. You can create multiple shift registers on a particular structure.

To initialize a shift register, wire a value from outside the loop to the left terminal. If you do not initialize the register, the loop uses the value written to the register when the loop last executed, or the default value for its data type if the loop has never executed. Use initialized shift registers to ensure consistent behavior. LabVIEW users can refer to Chapter 3, *Loops and Charts*, of the *LabVIEW User Manual* for further information on using uninitialized shift registers. BridgeVIEW users can refer to Chapter 10, *Loops and Charts*, of the *BridgeVIEW User Manual*.

When the loop finishes executing, the last value stored in the shift register remains at the right terminal. If the right terminal is wired outside of the loop, this last value passes out when the loop completes. This property is shown in the following illustration.
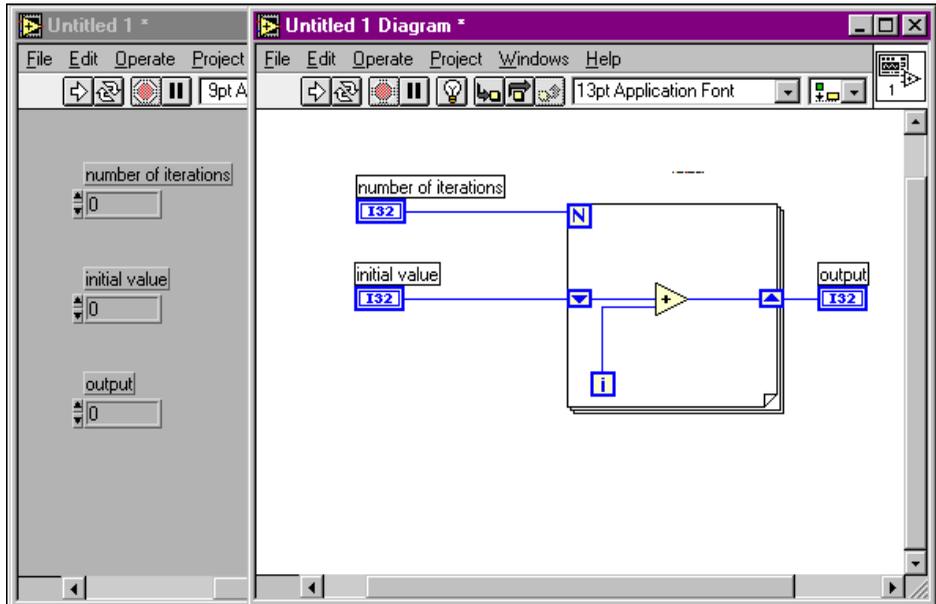
The left hand side of a shift register can have more than one terminal, so it can remember more than one previous value.

Resizing cursor

To add or remove terminals to a particular shift register, use the Resizing cursor to resize the left terminals. Alternatively, you can use the **Add Element** command from the shift register pop-up menu to add more left terminals to the shift register. Added terminals appear directly below the one on which you pop up. Use the **Remove Element** command to

remove the terminal on which you pop up. The following illustration shows
both methods. The only way to remove extra wired terminals is to choose
Remove Element from the shift register pop-up menu. Selecting **Remove
All** deletes the shift register.
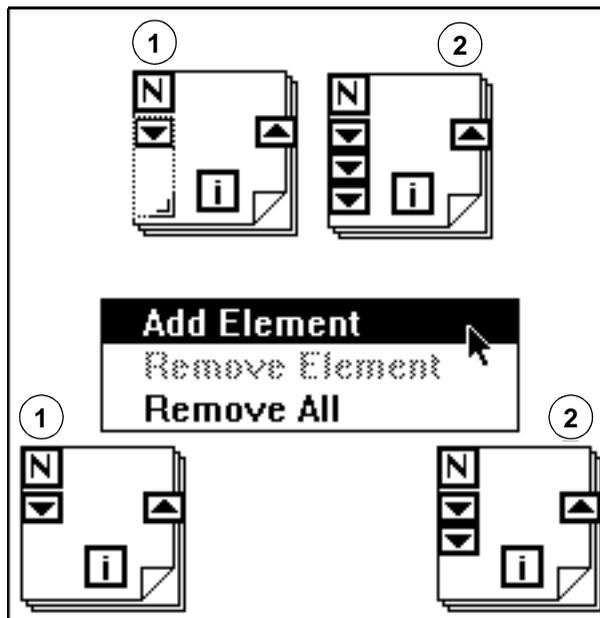
The left, topmost terminal holds the value from the previous iteration, $i - 1$.
The terminal immediately under the uppermost terminal contains the value
from iteration $i - 2$, and so on, with each successive terminal.

If you initialize one left terminal of a shift register, you must initialize all
of them.

The following pseudocode shows a three-value running average routine
equivalent to the G block diagram.

```
a = b = 0
for i = 0 to N - 1
    avg = (x[i] + a + b)/3
    b = a
    a = x[i]
```
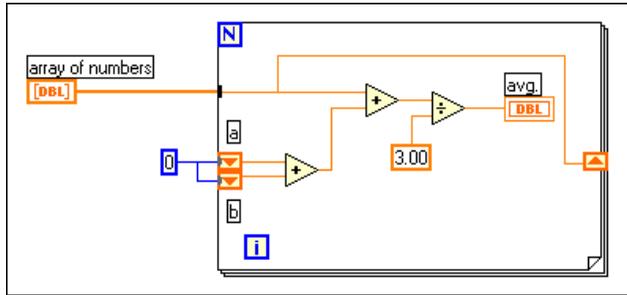
For an example of how a shift register is set up, see
`examples\general\structs.llb\Random Average.vi`.

Case and Sequence Structures

Both Case and Sequence structures can have multiple subdiagrams,
configured like a deck of cards, of which only one is visible at a time.
At the top of each structure border is the *subdiagram display window*,
which contains a *diagram identifier* in the center and decrement and
increment buttons at each side. For a Case structure, the diagram identifier
displays the values that cause the currently displayed subdiagram to
execute. For a Sequence structure, the diagram identifier indicates which
subdiagram is currently displayed. If the diagram identifier is numeric,
it is followed by a diagram identifier range, which shows the minimum
and maximum values for which the structure contains a subdiagram.
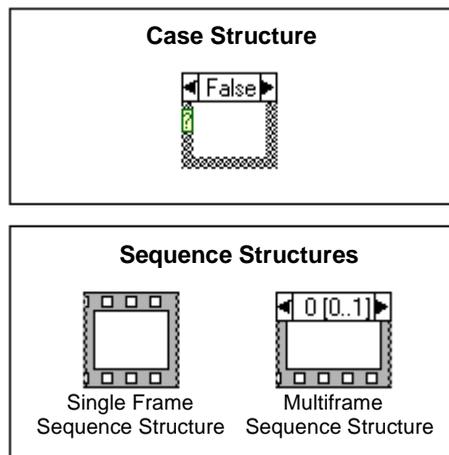
Clicking the decrement (left) or increment (right) button displays the previous or next subdiagram, respectively. Decrementing from the first subdiagram displays the last, and incrementing from the last subdiagram displays the first subdiagram. Other uses of the display window are explained in the *Case Structures* and *Sequence Structures* sections of this chapter.

See examples\general\structs.llb\SquareRoot.vi for an example of a Case Structure.

Case Structures

Case structures choose a single subdiagram, or case, to execute based on an input value, which is called the selector. With Case structures, you can do the following.

- Specify lists and ranges of selector values that correspond to a case.

- Use an integer, a boolean, a string or an enum type as a selector.

- Specify a default case (or action).

- Sort cases based on the first selector value.

When you place a Case structure on a block diagram, you type in the selector values directly into the Case structure selector label. You can also edit the selector values using the labeling tool. You can specify a single value, or lists and ranges of values that select the case. To indicate a list, separate the values by commas, such as –1, 0, 5, 10. A range is specified as 10..20, meaning all numbers from 10 to 20 inclusively. You also can use open-ended ranges such as ..0 (all numbers less than or equal to 0), or 100.. (all numbers greater than or equal to 100). Lists and ranges can be combined, such as ..5, 7..10, 12, 13, 14. When you type in a selector that contains overlapping ranges, the Case structure redisplays the selector in a more compact form. The previous example redisplays as ..10, 12..14.

You also can use string and enum values in a case selector. These always display in quotes, such as **"red"**, **"green"**, **"blue"**. However, you do not need to type in the quotes when entering the values unless the string or enum contains a comma or the symbol "..".

☞ **Note**   *If you type in a selector value that is not the same type as the object wired to the selector icon, then the value displays in red and your VI cannot run. Also, because of the possible round-off error inherent in floating point arithmetic, LabVIEW does not permit floating-point numbers to be used in case selector labels. If you wire a floating point type to the case, the type is rounded to the nearest integer as in previous versions of LabVIEW. If you try to type in a floating point value such as 1.2 into the Case selector, it displays in red.*

The Case shown below is a numeric type, so the string selector value displays in red.
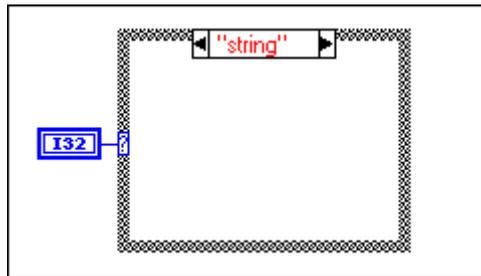
Figure 19-1. Case Structure Type Mismatch

To add more cases, follow the procedure outlined in the *Adding Subdiagrams* section of this chapter.

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

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*), G does not discard them, in case the change in data types is accidental. However, you must delete these extra cases before the structure can execute.

The same principle applies if you wire an enumeration to the selector and there are more cases than items in the enumeration. The diagram identifier for such cases is displayed as a red numeric identifier to indicate these cases must be deleted before the structure can execute.

When converting case selector values to a different type, consider the following. If you are converting numeric values like 23 to a string, then the string value is 23. If you convert a string to a numeric value, only those selector strings that represent a number are converted to numeric values. The other values remain strings. If you convert a number to a Boolean, then 0 is converted to False and 1 to True, while the other values become strings.

The data at all input terminals (tunnels and selection terminal) is available to all cases. Cases are not required to use input data or to supply output data, but *if any case supplies output data, all must do so*. If you do not wire data to an output tunnel from every case, the tunnel turns white, as in the top example in the following illustration, and the **Run** button shows a broken arrow.

When all cases supply data to the tunnel, it turns black, as in the bottom example in the following illustration, and the run button appears unbroken.
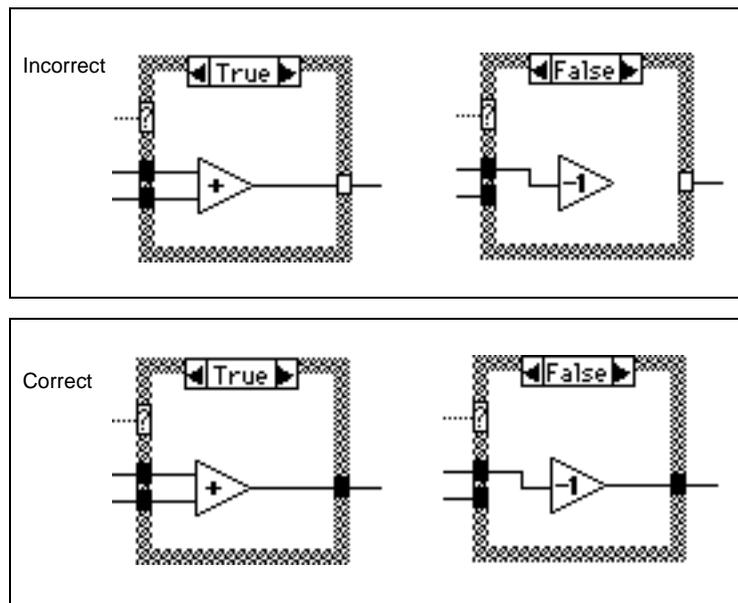
The pop-up menu items on the Case structure include items for non-Boolean cases, as shown in the following figure.
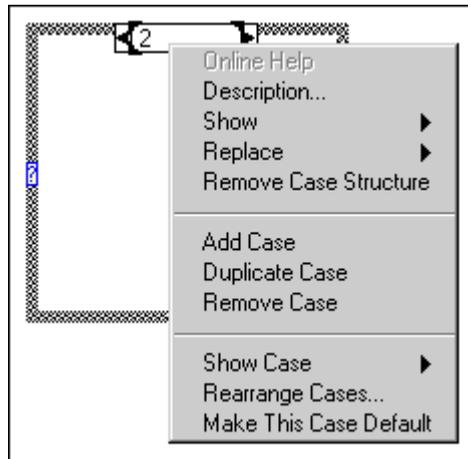
Figure 19-2. Pop-Up Menu Items for a Case Structure

The **Add Case** function adds a case after the visible case. You cannot add cases before the visible case. You can change the case order by selecting **Rearrange Cases…**. When you do so the following dialog box appears.
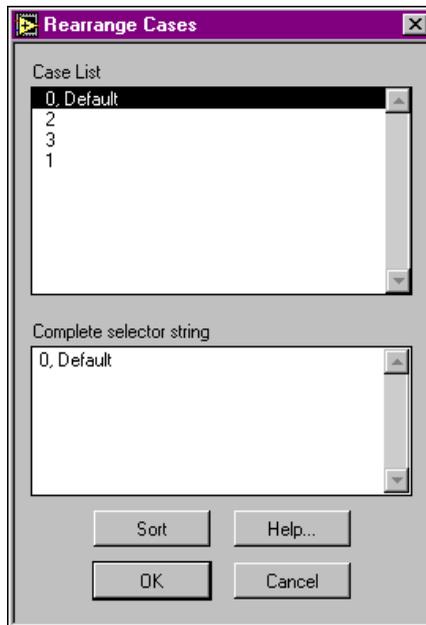
Figure 19-3. Rearrange Case Dialog Box

The **Sort** button sorts the case selector values based on the first selector value. To change the location of a selector, click the selector value you want to move and drag it to the location you want to move it to. **Complete selector string** shows the selected case selector in its entirety, in case it is too long to fit in the **Case List**. Context-sensitive help on any of the controls is available in this dialog box by clicking the **Help** button.

☞ **Note**   *This feature only changes the order in which the cases appear in the case structure. It does not affect the run-time behavior of the Case structure.*

The **Make This Case Default** item in the pop-up menu specifies a particular case to execute if the selector value is not listed in the Case structure. The word **Default** is listed in the selector value at the top of the Case structure. You must specify a default case for a Case structure if it does not contain selectors for every possible selector value.

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 Structures

The Sequence Structure, which looks like a frame of film, consists of one or more subdiagrams, or *frames*, that execute sequentially. For an example of a VI that uses a Sequence Structure, see examples\general\ structs.llb\TimingTemplate.vi.

Determining the execution order of a program by arranging its elements in sequence is called *control flow*. BASIC, C, and most other programming languages have inherent control flow, because statements execute in the order in which they appear in the program. The Sequence Structure is a way of obtaining control flow when data dependencies are not sufficient. 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 which are not data dependent. A node that receives its data directly or indirectly from another node has a data dependency on the other node and always executes after the other node completes. It is not necessary to use the Sequence Structure when data dependency exists or when the execution order is unimportant.

This situation occurs often when you want to determine how much time
a function takes. The Tick Count function returns the current time in
milliseconds. The time is not required to perform the function, so there
is no data dependency between time measurement and the function.
As shown in the following illustration, the Tick Count function to the left
of the sequence structure executes before the sequence structure. Then the
function executes in frame 0 and the time elapsed calculation is done in
frame 1. The sequence structure enforces the proper execution order.
(Frame 0 is not shown).
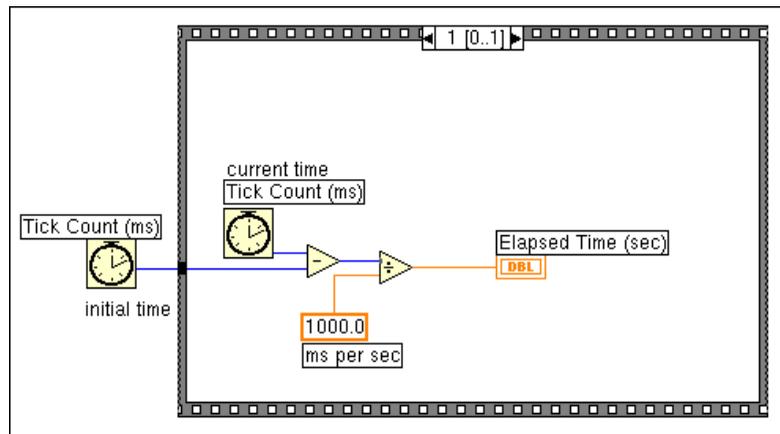
Output tunnels of Sequence Structures can have only *one* data source,
unlike Case Structures. The output can emit from any frame, but keep in
mind data leaves the structure only when it completes execution entirely,
not when the individual frames finish. Data at input tunnels is available to
all frames, 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 item is not available if you
pop up on a sequence local or over the subdiagram display window. You
can drag the terminal to any unoccupied location on the border. Use the
Remove command from the sequence local pop-up menu to remove a
terminal.

An outward-pointing arrow appears in the sequence local terminal of the
frame containing the data source. The terminal in subsequent frames
contains an inward-pointing arrow, indicating 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.
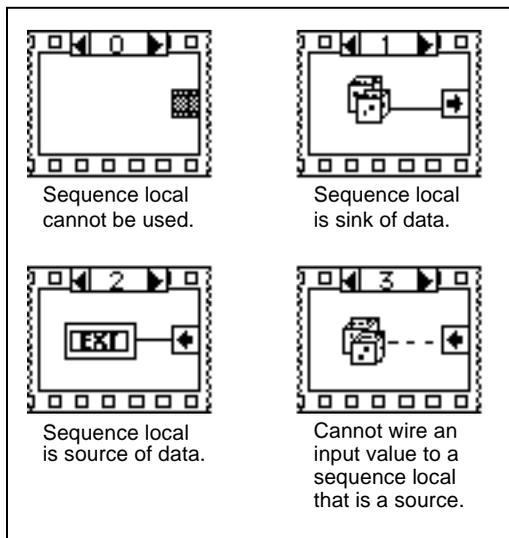
Editing Case and Sequence Structures

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 Structures, replace the word
case with the word *frame*. A new Case Structure has two cases, but can be
edited to have one case as well. A new Sequence Structure has one frame.

Moving between Subdiagrams

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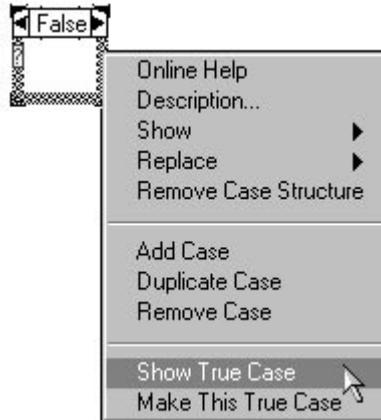
Adding Subdiagrams

If you select **Duplicate Case** from either the structure border pop-up menu or the diagram identifier pop-up menu, as shown below, a copy of the visible subdiagram is inserted after itself.
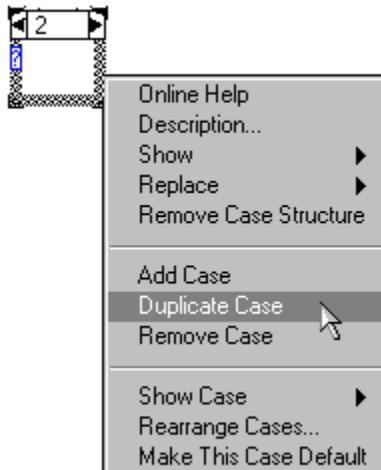
When you add or remove subdiagrams to a sequence structure, the editor automatically adjusts the diagram identifiers to accommodate the inserted or deleted subdiagrams. For case structures, the order of subdiagrams is irrelevant to program execution, but can be changed using the **Rearrange Cases...** item in the pop-up menu.

Deleting Subdiagrams

To delete the visible subdiagram, choose **Remove Case** from the structure border pop-up menu, shown below. This command is not available if only one subdiagram exists.
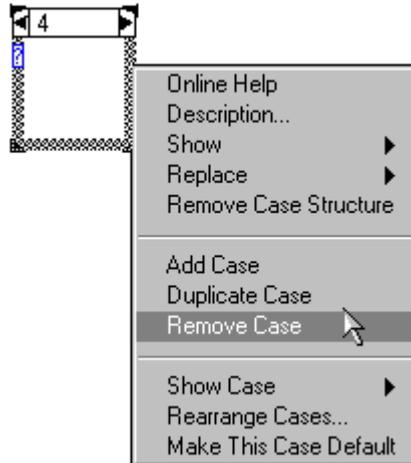
Structure Wiring Problems

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 produce a bad wire. This error is a variation of the multiple sources error.
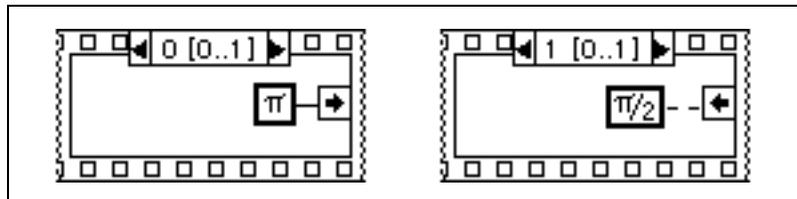
Failing to Wire a Tunnel in All Cases of a Case Structure

Wiring from a Case Structure to an object outside the structure results in a bad tunnel if you do not connect a source in all cases to the object, as shown in part 1 of the following example. This is a variation of the no source error because at least one case does not produce a data value if executed. Wiring to the tunnel in all cases, as shown in part 2 of this example, corrects the problem. This is not a multiple sources violation because only one case executes and produces only one output value per execution of the Case Structure.
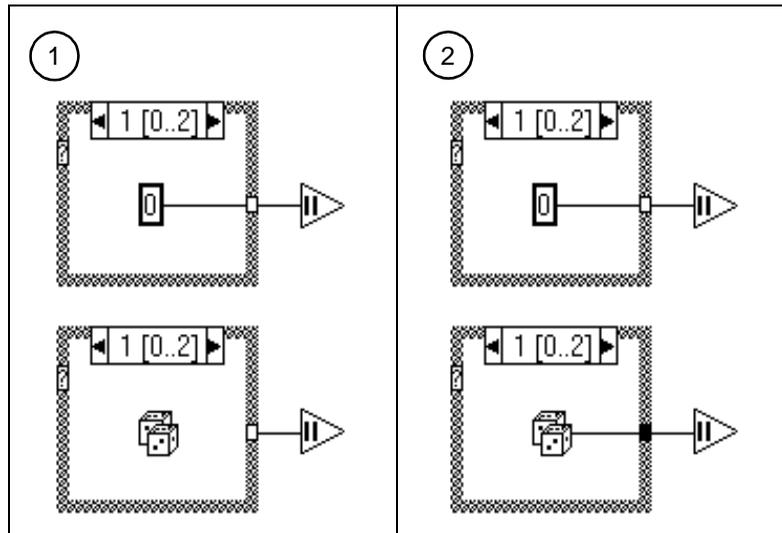
Overlapping Tunnels

Because G creates tunnels as you wire, tunnels sometimes overlap. Overlapping tunnels do not affect the execution of the diagram, but they can make editing difficult. Avoid creating overlapping tunnels. If they occur, drag one tunnel away to expose the other. Look at the following example.
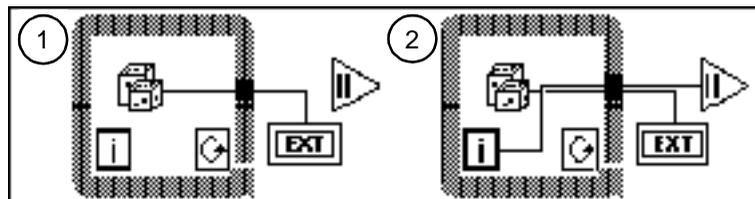
It is difficult to tell which tunnel is on top. You can make mistakes if you try to wire to one of them while they overlap, although you always remove the bad wires and try again.
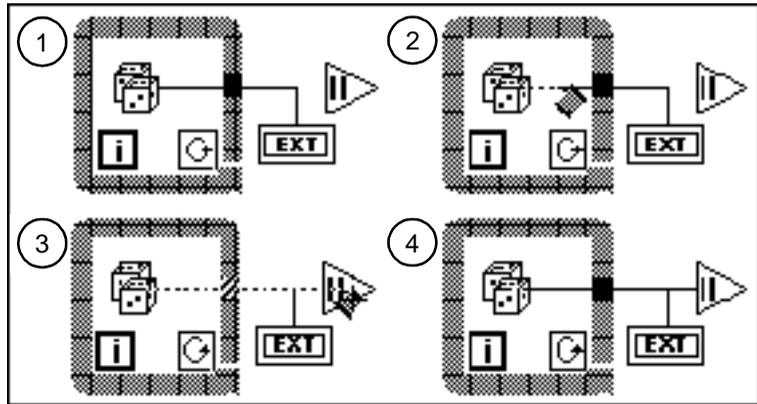
If you must wire from an object inside a structure to an object outside when one such wire already exists, *do not wire* through the structure again as shown in the above illustration. Instead, begin the second wire at the tunnel. In this example, two overlapping tunnels do not cause a problem. But if this was a Case Structure, two overlapping bad tunnels might appear wired in each case. You always can remove all the wires from a tunnel to make it vanish and then rewire correctly.

If your VI tunnels are completely overlapping, a warning appears in the Error List window if you select **Show Warnings**. See the *Understanding Warnings* section in Chapter 4, *Executing and Debugging VIs and SubVIs*, for more information on these problems.

Wiring from Multiple Frames of a Sequence Structure

This next illustration shows another variation of the multiple sources error. Two Sequence Structure frames attempt to assign values to the same tunnel. The tunnel turns white to signal this error.
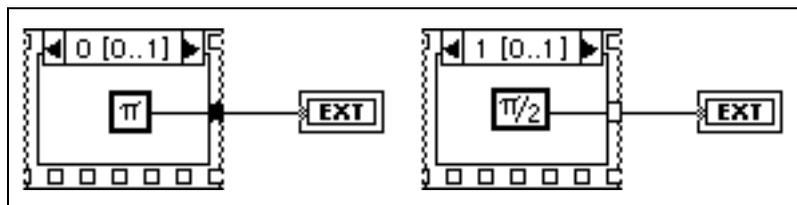
Wiring Underneath Rather Than Through a Structure

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

If you do not click in the interior or on the border of the structure, the wire passes underneath the structure, as shown below.
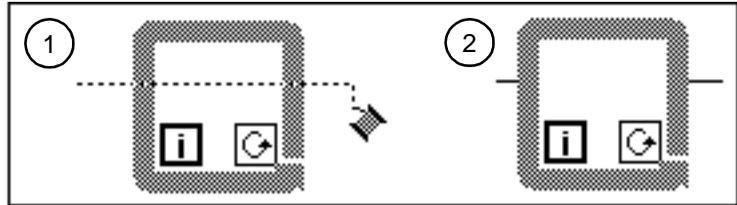
When the Wiring tool crosses the left border of the structure, a highlighted tunnel appears to indicate the editor creates a tunnel at that location as soon as you click the mouse button. If you continue to drag the tool through the structure without clicking the mouse until the tool touches the right 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 it, both tunnels disappear, and the wire passes underneath the structure rather than through it.

Examine the following illustration.
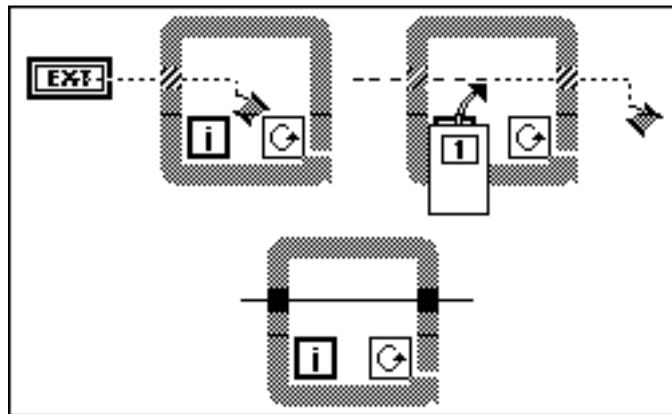
If you tack down the wire inside the structure, however, the wire goes through the structure even if you continue dragging the Wiring tool past the right border.

Removing Structures without Deleting Items in a Structure

You can remove a structure (For Loop, While Loop, Case Structure, or Sequence Structure) without losing the contents of the structure. If you pop up on any of these objects, you see an item for deleting the structure. In the case of For Loops and While Loops, the contents of the loop are copied to the underlying diagram. In addition, any wires connected by tunnels are connected together automatically.

In the case of a Case Structure or Sequence Structure, removing the structure only preserves the current frame or case. All other frames or cases are deleted. You are warned of losing hidden frames or cases and are given a chance to cancel the operation.

20

Formula Nodes

This chapter describes how to use the Formula Node to execute mathematical formulas on the block diagram. The Formula Node is available from the **Functions»Structures** palette.
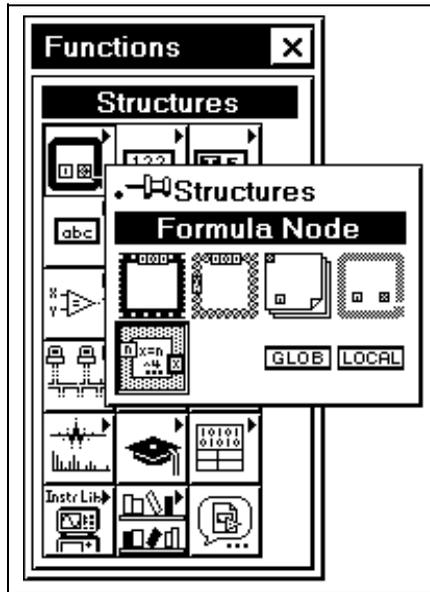
Using Formula Nodes

The Formula Node is a resizable box similar to the four structures (For Loop, While Loop, Case Structure, and Sequence Structure). Instead of containing a subdiagram, however, the Formula Node contains one or more formula statements delimited by a semicolon, as in the following example.

$$y = 3*x^2 + x*\log(x);$$
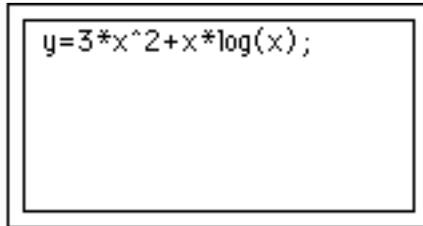
Formula statements use a syntax similar to most text-based programming languages for arithmetic expressions. You can add comments by enclosing them inside a slash–asterisk pair (`/*comment*/`).

See `examples\general\structs.llb\Equations.vi` for an example of a VI that uses a Formula Node.

The pop-up menu on the border of the Formula Node contains items to add input and output variables, as shown in the following illustration.
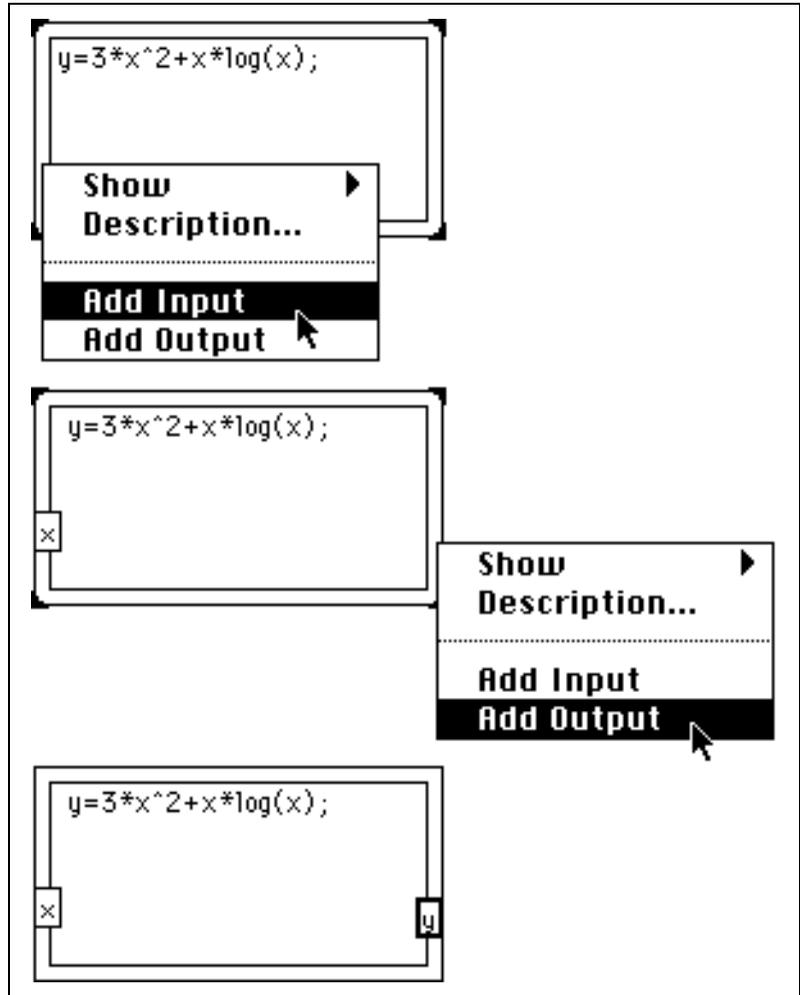
Output variables are distinguished by a thicker border.

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

Every variable used in the Formula Node for a calculation must be declared as an input or an output. Intermediate variable, that is, variables assigned to outcomes of operations calculated after input(s) to the node but before the final output from the node, must be declared as outputs. However, it is not necessary for intermediate variables to be wired to nodes external to the

Formula Node. In the following illustration, the *y* variable and the
z variable must both be declared as outputs, even if the value of *y* is not
wired to external nodes.
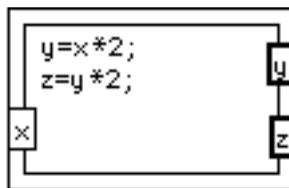
You can change an input to an output by selecting **Change to Output** from
the pop-up menu, as shown in the following illustration.
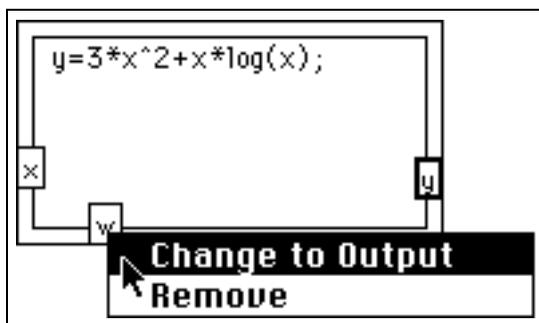
You can change an output to an input by selecting **Change to Input** from
the pop-up menu, as shown below.
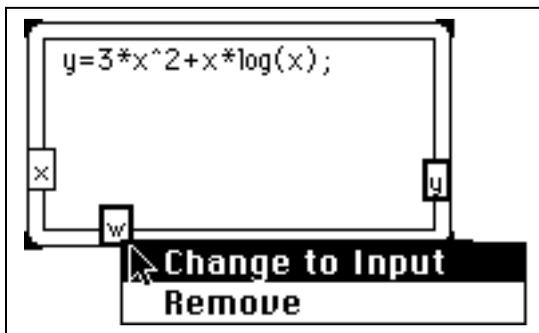
All variables are floating-point numeric scalars, whose precision depends
on the configuration of your computer. Variables cannot have units.
All input variables appearing in the formulas must be wired. All wired
output variables must be assigned in at least one statement; that is, they
must be on the left side of an equal sign. An output variable might appear

in an expression on the right side of an equal sign, but G does not verify
if it is assigned in a previous statement. When an assignment occurs as
a subexpression, the value of the subexpression is the value assigned;
for example

$x = \sin(y = \text{pi}/3)$

assigns $\frac{\pi}{3}$ to y, and then assigns $\sin\left(\frac{\pi}{3}\right)$ to x.

If a syntax error occurs, you can click the **Broken Run** button to see the
error listing. In the listing, the Formula Node displays a portion of the
formula with a # symbol marking the point at which the formula box
detected the error.

Formula Node Functions and Operators

All function names must be lowercase. Table 20-1 shows the names of the
Formula Node functions.

Table 20-1. Formula Node Functions

Function	Corresponding G Function Name	Description
abs(x)	Absolute Value	Returns the absolute value of x.
acos(x)	Inverse Cosine	Computes the inverse cosine of x in radians.
acosh(x)	Inverse Hyperbolic Cosine	Computes the inverse hyperbolic cosine of x in radians.
asin(x)	Inverse Sine	Computes the inverse sine of x in radians.
asinh(x)	Inverse Hyperbolic Sine	Computes the inverse hyperbolic sine of x in radians.
atan(x)	Inverse Tangent	Computes the inverse tangent of x in radians.

Table 20-1. Formula Node Functions (Continued)

Function	Corresponding G Function Name	Description
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 is greater than or equal to x).
cos(x)	Cosine	Computes the cosine of x in radians.
cosh(x)	Hyperbolic Cosine	Computes the hyperbolic cosine of x in radians.
cot(x)	Cotangent	Computes the cotangent of x in radians (1/tan(x)).
csc(x)	Cosecant	Computes the cosecant of x in radians (1/sin(x)).
exp(x)	Exponential	Computes the value of e raised to the x power.
expm1(x)	Exponential (Arg) – 1	Computes the value of e raised to the x power minus one ($e^x - 1$).
floor(x)	Round to –Infinity	Truncates x to the next lower integer (largest integer is smaller than or equal to x).
getexp(x)	Mantissa & Exponent	Returns the exponent of x.
getman(x)	Mantissa & Exponent	Returns the mantissa of x.
int(x)	Round To Nearest integer	Rounds its argument to the nearest even integer.
intrz(x)	Round Toward 0	Rounds x to the nearest integer between x and zero.

Table 20-1. Formula Node Functions (Continued)

Function	Corresponding G Function Name	Description
ln(x)	Natural Logarithm	Computes the natural logarithm of x (to the base e).
lnp1(x)	Natural Logarithm (Arg + 1)	Computes the natural logarithm of (x + 1).
log(x)	Logarithm Base 10	Computes the logarithm of x (to the base of 10).
log2(x)	Logarithm Base 2	Computes the logarithm of x (to the base 2).
max(x,y)	Max & Min	Compares x and y and returns the larger value.
min(x,y)	Max & Min	Compares x and y and returns the smaller value.
mod(x,y)	Quotient & Remainder	Computes the remainder of x/y, when the quotient is rounded toward –Infinity.
rand( )	Random Number (0–1)	Produces a floating-point number between 0 and 1 exclusively.
rem(x,y)	Remainder	Computes the remainder of x/y, when the 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).

Table 20-1. Formula Node Functions (Continued)

Function	Corresponding G Function Name	Description
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.
$x\text{^}y$	x^y	Computes the value of x raised to the y power.

Formula Node Syntax

The Formula Node syntax is summarized below using Backus–Naur Form
(BNF) notation. Square brackets enclose optional items.

```
<assignlst> := <outputvar> = <aexpr> ; [ <assignlst> ]

<aexpr>:=<expr> | <outputvar> = <aexpr>

<expr>:=<expr> <binaryoperator> <expr>

    |<unaryoperator> <expr>

    |<expr> ? <expr> : <expr>

    |( <expr>)

    |<inputvar>

    |<outputvar>

    |<const>

    |<function> ( <arglist> )

<binaryoperator>:= + | - | * | / | ^ | != | ==

    | > | < | >= | <= | && | ||

<unaryoperator>:= + | - | !

<arglist>:=<aexpr>[ , <arglist> ]

<const>:=pi | <number>
```

The precedence of operators is as follows, from lowest to highest.

`=`	assignment
`? :`	conditional
`||`	logical or
`&&`	logical and
`!= ==`	inequality, equality
`< > <= >=`	other relational: less than, greater than, less than or equal, greater than or equal
`+ –`	addition, subtraction
`* /`	multiplication, division
`+ – !`	unary: positive, negative, logical not
`^`	exponentiation

Exponentiation and the assignment operator are right-associative (groups right to left). All other binary operators are left-associative. The numeric value of TRUE is 1 and FALSE is 0 (for output). The logical value of 0 is FALSE, and any nonzero number is TRUE. The logical value of the conditional expression

```
<lexpr> ? <texpr>: <fexpr>
```

is `<texpr>` if the logical value of `<lexpr>` is TRUE and `<fexpr>` otherwise.

Formula Node Errors

Table 20-2 lists errors detected by the Formula Node.

Table 20-2. Formula Node Errors

Error Message	Error Message Meaning
syntax error	Misused operator, etc.
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.

21

VI Server

This chapter describes the mechanism for controlling VIs and applications programmatically. This chapter also describes how to control VIs or applications remotely.

G provides programmatic access to many of its features through the VI Server. You can access these features from any ActiveX client on the Windows 95/NT platforms. You also can access all the VI Server capabilities, on all platforms, through some G-language functions. These functions also make it possible to perform almost all VI Server operations in a local version of BridgeVIEW or LabVIEW as well any remote version across a TCP/IP network. See the *LabVIEW User Manual* or *BridgeVIEW User Manual* for more information on ActiveX server and client capabilities as well as **Online Reference** in your software.

Using the VI Server

The following are some of the tasks you can accomplish using the VI Server.

- You can dynamically load VIs into memory, rather than having them statically linked into your application, and call them just like a normal subVI call using VI server functions. This can be useful if you have a large application and wish to save memory or startup time. By making rarely used portions of your application, like a set of configuration dialogs, load and run only on demand, you can reduce the memory usage of your application as well as the amount of time it takes to load your application into memory. When the user finishes the operations, you can release the VIs and make the memory available again.

- You can control aspects of the user interface of a VI programmatically. For instance you might want to dynamically determine the location of a VI window, or scroll a panel so that a particular part of the panel is visible, or close or open the panel window. All these properties of a VI front panel are controllable through the VI Server.

- You can easily create a server application that exports functions that can be called from BridgeVIEW or LabVIEW on the Internet. For example, you might have a data acquisition application that acquires

and logs data at a remote site and you wish to sample that data occasionally from your local machine. Through a simple preference setting, you can make some VIs callable from across the Internet so that transferring the latest data is as easy as a subVI call. The VI Server takes care of all the networking details and makes it work no matter what platform the client or the server are running on.

- You can change properties of a VI programmatically and save those changes to disk . For example, during development of your application you might want VIs configured so that debugging is available, run-time pop-up menus are available, scroll bars are visible and windows are resizable. However, when you distribute your application, you might wish to turn off these features, as well as ensure that certain other properties are correctly set, such as whether a VI is reentrant, what execution system the VI is set to run within, and the path to the help file of a VI. All these properties can be programmatically set and queried via the VI Server, enabling you to write applications that edit VIs, rather than going through the VI Setup dialog box for each and every VI.

- You can create a plug-in architecture for your application, to add functionality to your application after it is distributed to customers. For example, you might have a set of data filtering VIs, all of which take the same parameters. By designing your application to dynamically load these filters from a plug-in directory, you can ship your application with a partial set of these filters and make more filtering options available to users by simply placing the new filtering VIs in the plug-in directory.

VI Server Capabilities

You access VI Server functionality through references to two main classes of objects: the Application object and the VI object. Once you create a reference to one of these objects, you can pass it to a function that operates on the object. When you are finished with it, you close the reference. This programming convention is similar to File I/O and network references.

An important aspect of both Application and VI references is their network transparency. This means you can open references to objects on remote machines in the same way you open references to those objects on your own machine. After you open a reference to a remote object, you can treat it in exactly the same way as a local object, with a few restrictions. For operations on a remote object, the VI Server takes care of sending the information about the operation across the network and sending the results

back. Your program looks virtually identical regardless of whether the operation is remote or local.

With a reference to a G application, you can obtain information about the G environment, such as what platform your application is running on, or the version of BridgeVIEW or LabVIEW, or the list of all VIs currently in memory. You can also set information such as the current user name, or the list of VIs exported to other applications.

When you have a reference to a VI, you load it into memory. Once you have the reference, the VI stays in memory until you close the reference. There is the possibility multiple references to a VI might be open at the same time. In that case, the VI stays in memory until all references to the VI are closed. With a reference to a VI, you can get and set all the properties of the VI available in **VI Setup**, as well as dynamic properties, such as front panel window position. You can also programmatically print the VI, save it to another location, and export and import its strings for translation purposes.

Application and VI References

In most applications, you create a reference to a VI by using the Open Application Reference function and the Open VI Reference function respectively. If you want to display the reference on the front panel, select **Path & Refnum** from the **Controls** palette. Choose the Application or VI Refnum. The data type of this refnum is displayed as an Application Refnum by default. If you want to change to a VI Refnum, pop up on the refnum and choose **Select VI Server Class»Virtual Instrument**.

Using the Property and Invoke Nodes with Application and VI References

Many of the available operations on Application and VI references are only available through the Property and Invoke nodes. These nodes are similar in that they both have two inputs and two outputs at the top of the node, and a variable list of inputs and outputs below that.
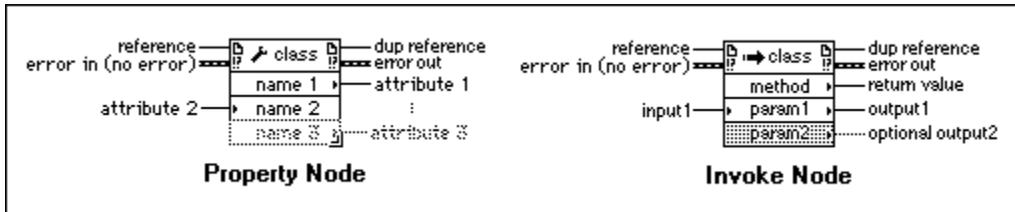
Reference is an Application or VI reference and **dup reference** is a copy of reference used to pass the value to other functions. **Error in** is the standard error cluster, which contains whether an error occurred, the code of the error, and a description of the error. If an error exists in **error in**, then the node does nothing and simply passes the error through to **error out**. Otherwise, if an error occurs during the operation of the node, **error out** contains information about the error.

Both nodes are polymorphic with respect to the reference input. When you wire an Application or VI refnum to this input, the node automatically adapts to that data type and makes available only those operations applicable to that type.

With the Property node you can get (read) and set (write) various properties on an application or VI. You can select the properties from the node pop-up menu by simply clicking a property terminal with the Operating tool, or pop up on the terminal.
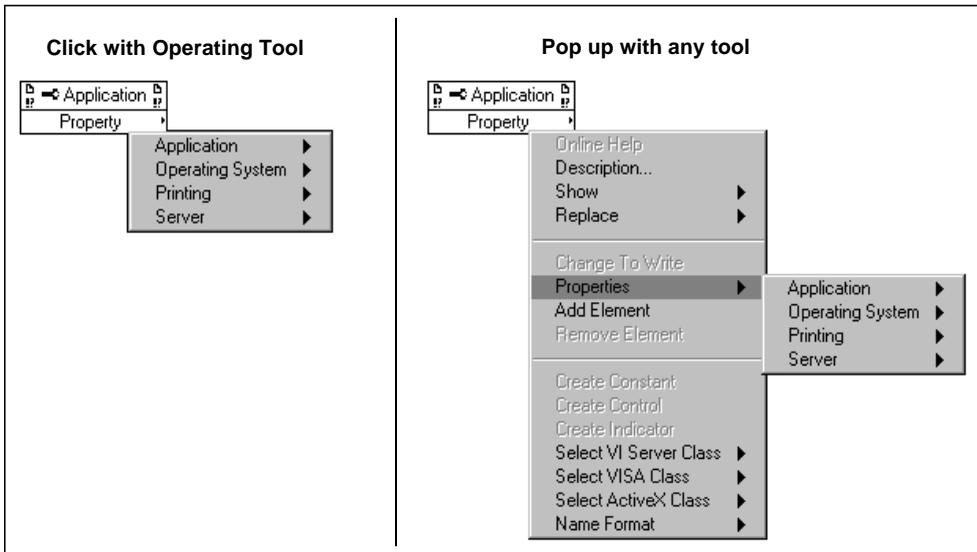
You can get or set multiple properties using a single node. As you enlarge a Property Node, new terminals are added.
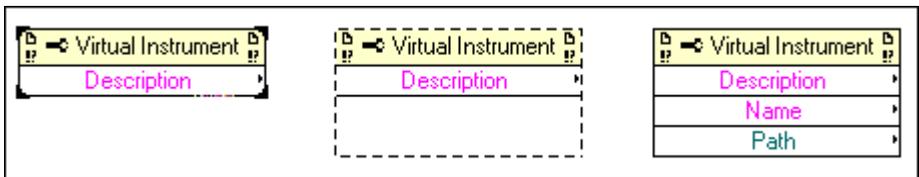
In one Property Node, you can read and write properties, as shown in the following illustration. The properties in which you read are designated by a small direction arrow on the right, while the properties you write to have a small direction arrow on the left. You choose to read or write properties by selecting **Change to Read** or **Change to Write** in the pop-up menu.
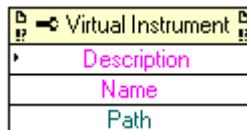
The node executes from top to bottom. If an error occurs midway down the node, the remaining properties are ignored and an error is returned. The error string contains information about which property caused the failure.

With the Invoke Node you can perform actions, or methods, on an application or a VI. Unlike the Property node, a single Invoke node executes only a single method on an application or VI. You access the available methods by clicking the method terminal with the Operating tool, or popping up on the terminal, which is the same as the Property Node.
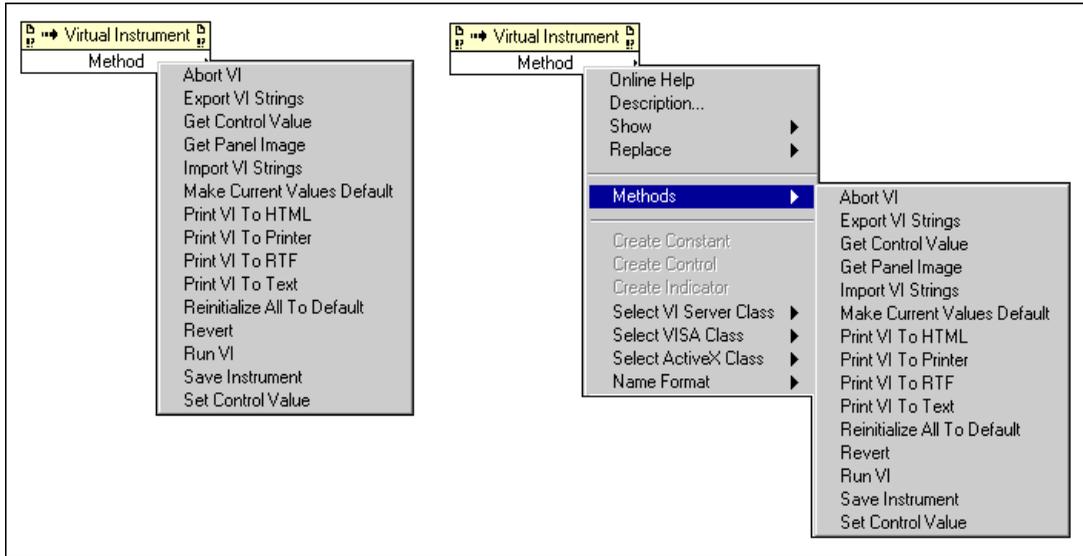
The name of the method is always the first terminal in the list of parameters. If the function returns a value, that result is the value of the method terminal. For example, a value is returned by the method **Get Control Value**. Otherwise, the method terminal has no value, like **Print VI To HTML**.
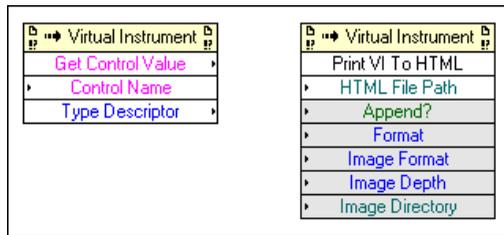
The Invoke node lists the parameters top to bottom with the optional parameters in gray at the bottom. In the previous illustration, **HTML File Path** is a required parameter, whereas **Append?**, **Format**, **Image Format**, **Image Depth**, and **Image Directory** are optional parameters.

Example of VI and Application Class Properties and Methods

In the following sections, common VI and Application Class applications are discussed. See the examples\general\viserver to explore other applications where VI and Application Class properties and methods are used.

Manipulating VI Class Properties and Methods

You can set or get properties on a VI independent of performing methods on a VI. In some applications you want to do both, access VI properties and perform a method on a VI. In the following diagram, the front panel objects in a VI are reinitialized to their default values then the front panel is opened, displaying those default values.
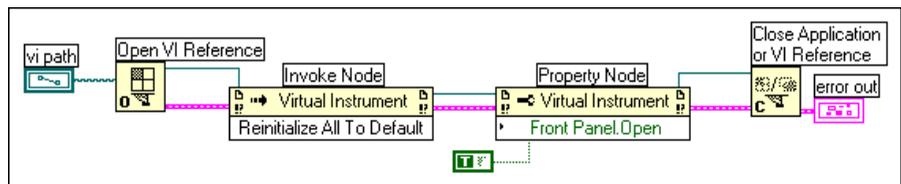
Before accessing properties and methods on a VI, you must create a reference to that VI by executing Open VI Reference. To invoke a method on a VI, use the Invoke Node. Once the wire is connected from Open VI Reference to the Invoke Node, you can access all the VI Class methods. You can also pop up on the node and choose **Select VI Server Class»Virtual Instrument** to gain access to the VI Class methods.
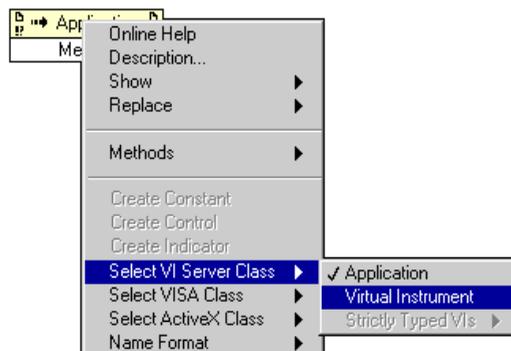
The Property Node operates in a similar way to the Invoke Node. Once you wire a VI reference to the Property Node, you can access all the VI Class properties. Also, you can pop up on the node and choose **Select VI Server Class»Virtual Instrument**.

To set property values, make sure the arrow is on the left-hand side by popping up and selecting Change to Write. Always check for the possibility of errors. The Invoke and Property Node does not execute if an error occurs before it executes. If the error occurs in one of those nodes, the property or method where the error occurred is noted in the error message.

Manipulating Application Class Properties and Methods

You can set or get properties on a local or remote LabVIEW independent of performing methods on LabVIEW. In the following diagram, the VIs in memory on a local machine are displayed on the front panel in a string array.
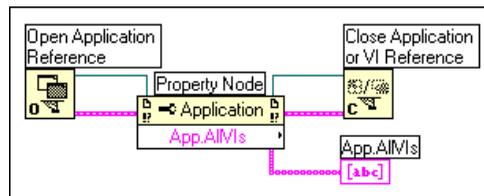
If you want to find the VIs in memory on a remote machine, wire a string control to the machine name input and enter the machine numeric IP address or domain name, as shown in the following illustration. You also must change the property to **Exported VIs in Memory** since the property **All VIs in Memory** used in the previous illustration only applies to local versions of LabVIEW and BridgeVIEW.
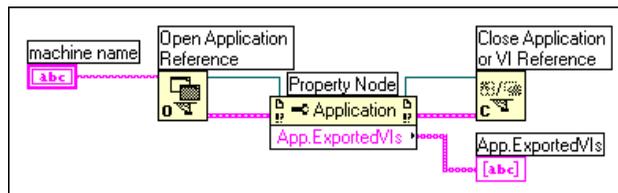
Always check for the possibility of errors. The Property Node does not execute if an error occurs before it executes. If the error occurs in the Property node, the property where the error occurred is noted in the error message.

Manipulating VI and Application Class Properties and Methods

You can use VI Class and Application Class properties and methods separately. In some applications, you must access properties and methods from both classes. In the following block diagram, the VIs in memory on a local machine are determined and the path to each of these VIs is displayed on the front panel. To find all the VIs in memory, you must access an Application Class property. To determine the paths to each of these VIs, you must access a VI Class property. The number of VIs in memory determines the number of times the For Loop executes. Open VI Reference and Close VI Reference need to be inside the For Loop because you need a VI reference for each VI in memory. It is best not to close the Application reference until all the VI paths are gathered.
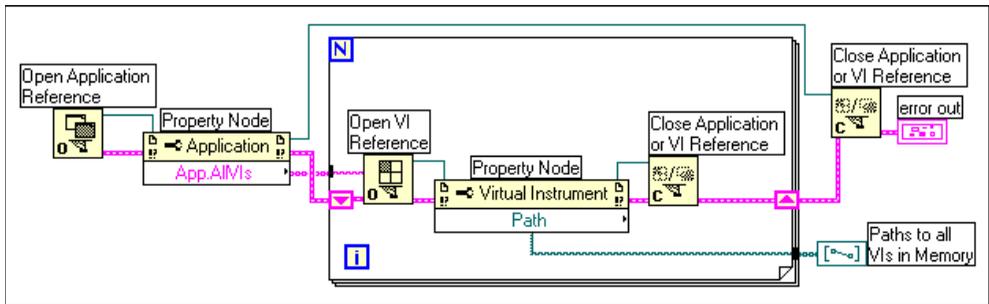
Strictly-Typed VI Refnums

Strictly-typed refnums need only be used in applications where you dynamically call a VI. In these applications there are two different situations in which they are used. The first and perhaps most typical is when you wish to pass a strictly-typed VI reference into a subVI as a parameter. In this case, you connect the strictly-typed VI refnum control to the connector pane of the VI and wire the refnum terminal to the input of a Call By Reference node. The value of the refnum is used in this case to determine which VI is called.

The second situation occurs when you wire a strictly-typed VI refnum to the type specifier input of the Open VI Reference function. In this case the value of the control is ignored—only the type of the refnum is used by this function. It is used to determine whether the VI that is opened has the same connector pane as that of the strictly-typed VI refnum.

To create a strictly-typed refnum, drop a VI refnum on the front panel by selecting **Controls»Path & Refnum»Application or VI Refnum**. Pop up and choose **Select VI Server Class»Browse….** The **Choose VI to open** dialog box appears prompting you for a VI.
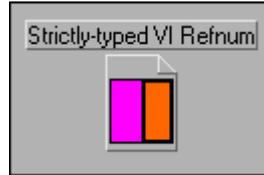
Remember even though you specify a VI for strictly-typed refnums, it only stores the connector pane information. That is, no permanent association is made between the refnum and the VI. In particular, do not confuse selecting the VI connector pane with getting a reference to the selected VI. You specify a particular VI through the **vi path** input on the Open VI Reference function.

After you select connector panes for strictly-typed refnums, the connector pane is retained in the VI refnum **Select VI Server Class» Strictly-Typed VIs** submenu. If you exit BridgeVIEW or LabVIEW, these connector pane selections are not retained the next time you launch the application.

Example of Strictly-Typed VI Refnums

The only application where you use strictly-typed VI refnums is dynamically calling a VI. The following block diagram shows an example.
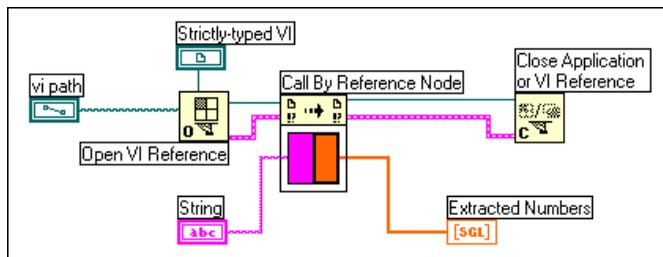
Once you wire **vi path** and **strictly-typed VI** to the Open VI Reference function, you can connect the Open VI Reference function to the Call By Reference Node. The connector pane is automatically displayed in the Call By Reference Node. Then you can wire any input values and output displays to the connector pane. The Call By Reference Node calls the VI specified by the VI reference input, using any input values specified. This node is helpful if you do not want to load all subVIs into memory at once.

22

Attribute Nodes

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

Creating Attribute Nodes

You create an attribute node by selecting the **Create»Attribute Node** item from the pop-up menu of a front panel control or from the terminal of the control.
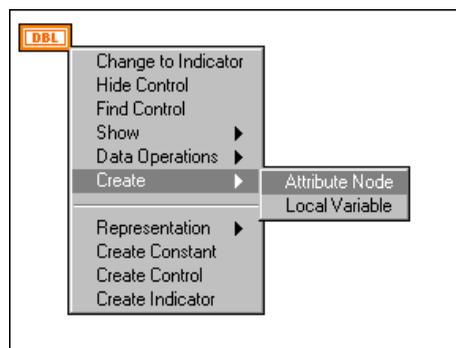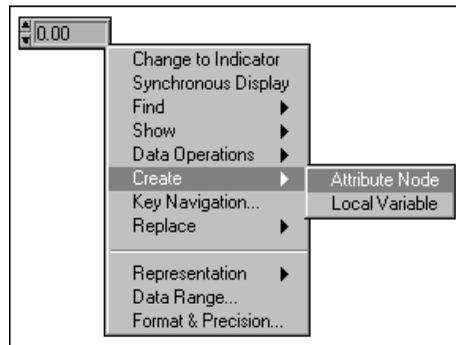
Selecting this item creates a new node on the diagram located near the terminal for the control, as shown in the following illustration. If the control has a label associated with it, the label for the control is used for the initial label of the attribute node. You can change the label after the node is created.

If you pop up on an attribute terminal, and then choose **Select Item**, you see a menu of attributes you can write to or read from the control, as shown in the following illustration. You can use a shortcut to the list of attributes by clicking the attribute node with the Operating tool.
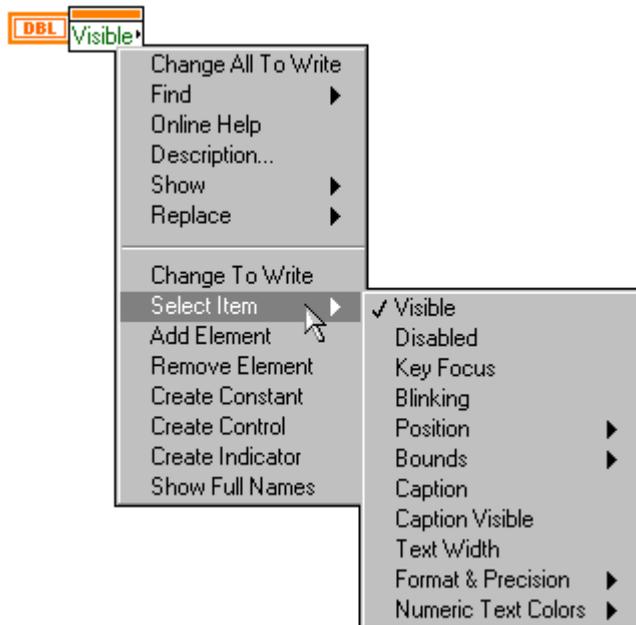
You choose to read or to write attributes by selecting either the **Change to Read** or **Change to Write** item from the attribute node pop-up menu, as shown in the illustration that follows.

You can write an attribute when the small direction arrow is located on the left side of the terminal. You can read an attribute when the arrow is located on the right side of the terminal.

direction arrow
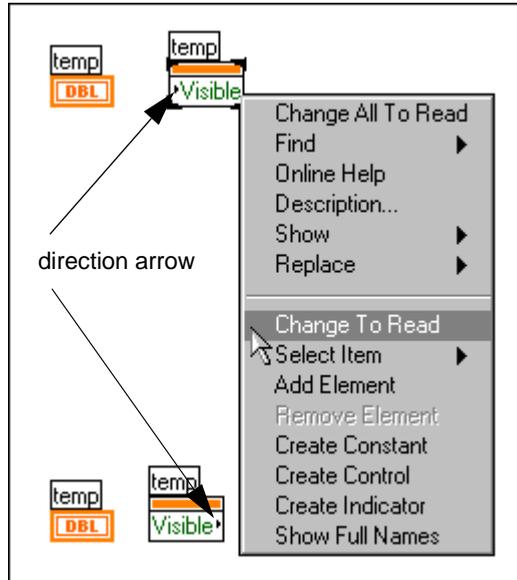
You can read or write more than one attribute with the same node by enlarging the attribute node. New terminals are added as the node enlarges. The execution order of an attribute node is from top to bottom.
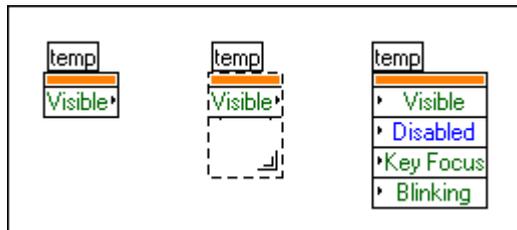
You associate a terminal with a given attribute by clicking the terminal with the Operating tool and selecting an attribute from the attribute node pop-up menu.
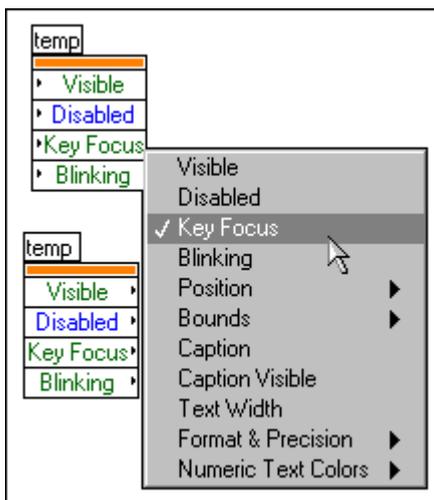
You can create more than one attribute node by cloning an existing node, or by selecting the **Create»Attribute Node** item again. To clone an existing node, click that node and drag it with the Positioning tool while holding down the <Ctrl> **(Windows)**; <option> **(Macintosh)**; <meta> **(Sun)**; or <Alt> **(HP-UX)** key. However, if you copy and paste an attribute node using the **Edit** menu commands, a new copy of both the attribute node and the control to which it refers is made.

Each copy of an attribute node can independently have attribute terminals for reading or writing.
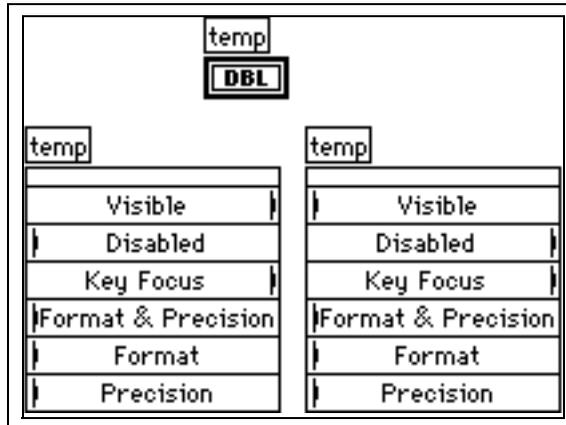
The attributes for a cluster and for a numeric control inside the cluster are shown in the following illustration.
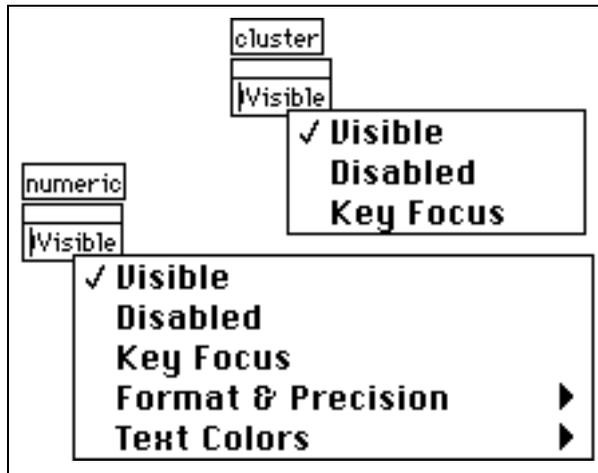
Some controls, such as the graph, have a large number of attributes you can read or set. Some of these attributes are grouped into categories and listed in submenus, such as the **Format & Precision** category for a numeric control. You can choose to set all of the attributes at once by selecting the **All Elements** item of the submenu. You also can set one or more of the elements individually by selecting the specific attribute(s). The **Format & Precision** item on a numeric control is shown in the following illustration as an example.
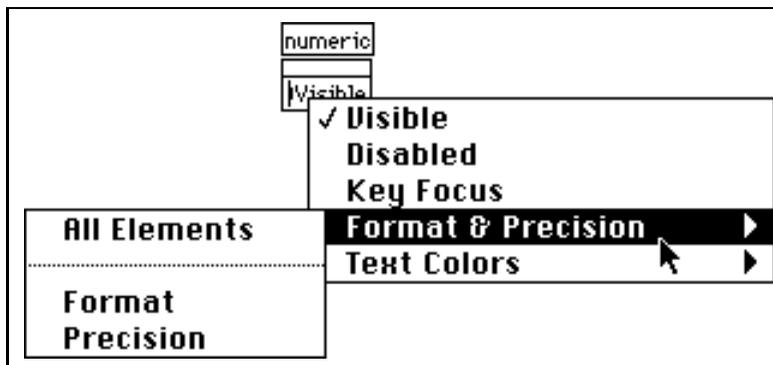
After you create an attribute node, the **Find Control** and **Find Terminal** items of the terminal and control pop-up menus change to submenus that help you find attribute nodes. In the same way, the attribute node has options to find the control and the terminal it is associated with.

See `examples\general\attribute.llb` for an example of how to use an attribute node.

Using Attribute Node Help

The Help Window and *Online Reference* (**Help** menu) are invaluable tools for using attribute nodes. You can use them to find descriptions, data types, and acceptable values for attributes. For more information, refer to the *Attribute Help* section in Chapter 1, *Introduction to G Programming*.

Base Attributes

There are many attributes available for modifying the various front panel objects in your application. This section discusses Visible, Disable, Key Focus, Blinking, Position, and Bounds attributes, which are common to nearly all front panel objects.

Visible Attribute

You can read or write the visibility of a front panel object with the Visible Attribute. The associated object is visible when TRUE, hidden when FALSE.

In the following illustration, the digital control is set to an invisible state. A Boolean TRUE value makes the control visible again, as shown.
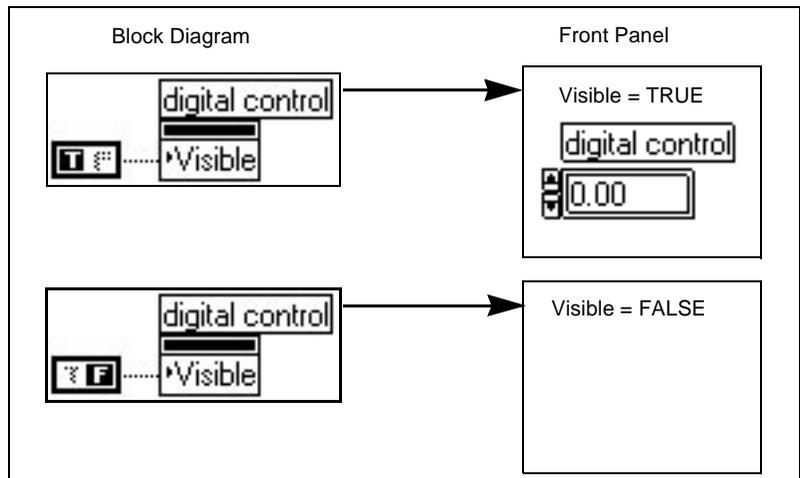
Disabled Attribute

You can control whether a user has access to an object, by implementing the Disabled Attribute. A value of zero enables an object so that the user can operate it. A value of one disables the object, preventing operation. A value of two disables and grays out the object.

You can disable user access to a digital control. The control does not change appearance when disabled in the first example. In the second example, the user access to the digital control is disabled and grayed out.
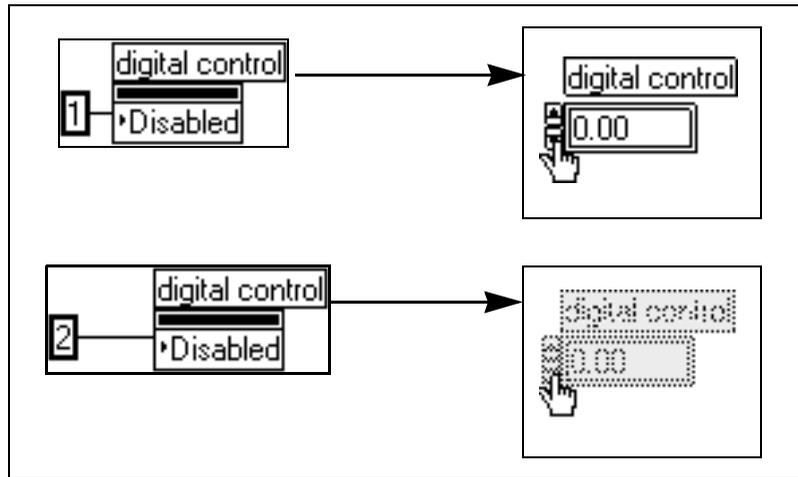
Key Focus Attribute

With the Key Focus Attribute, you can make a control the key focus or check to see if it currently has the focus. A key focus control behaves as though you tabbed to that control to make it active. On most controls, you can enter values into the control by typing them with the keyboard. You also can set the key focus on the front panel by pressing the <Tab> key while in run mode or by pressing the hot key associated with the control (assigned by using the **Key Navigation** item).

You can make a digital control the key focus. Then you can enter a new value in the control without selecting it with the mouse.
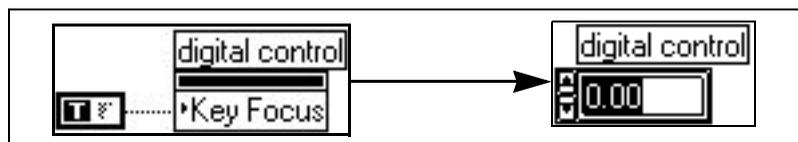
Blinking

Blinking Attribute

By using the Blinking Attribute, you can read or set the blink status of an object. If you set this attribute to TRUE, a front panel object blinks. The blink rate and colors are set in the **Preferences** dialog box. When you set this attribute to FALSE, the object stops blinking.

In the following illustration, the front panel indicator is set to blinking.
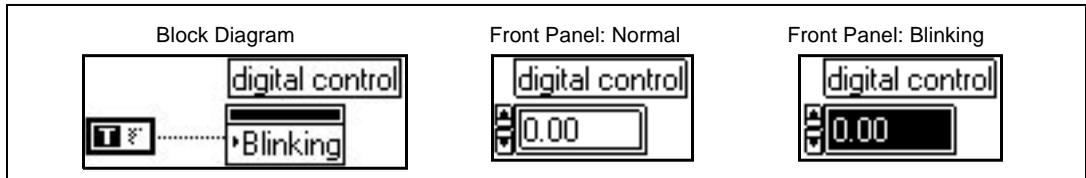
Position

Position Attribute

You can set or read the position of the upper left corner of an object on the front panel with the Position Attribute. The position is determined in units of pixels relative to the origin of the Panel window, which is initially the top left corner of the window. It might be somewhere else if you have scrolled the window. This attribute consists of a cluster of two unsigned long integers. The first item in the cluster (Left) is the location of the left edge of the control, and the second item in the cluster (Top) is the location of the top edge of the control relative to the origin of the Panel window.

Left
Top

By executing or using the Position attribute node, the digital control changes its location on the front panel in the following illustration.
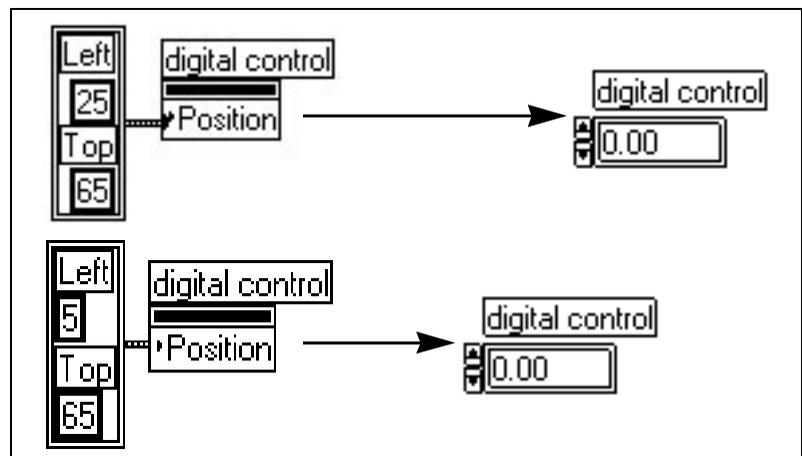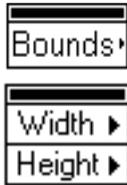
Bounds Attribute (Read Only)

The Bounds Attribute reads the boundary of an object on the front panel in units of pixels. The value includes the control and all of its parts, including the label, legend, scale, and so on. This attribute consists of a cluster of two unsigned long integers. The first item in the cluster (Width) is the width of objects in pixels, and the second item in the cluster (Height) is the height of the object in pixels.

The Bounds Attribute is a *read-only* attribute. It does *not* resize a control or indicator on the front panel. Most objects have other attributes for resizing, such as the Plot Area Size attribute for Graphs and Charts. This attribute is useful when you must read the overall size of a control, including all optional parts, so you can position other controls in relation to the control with which you are working.

You can determine the bounds of the digital control, as shown in the following illustration.
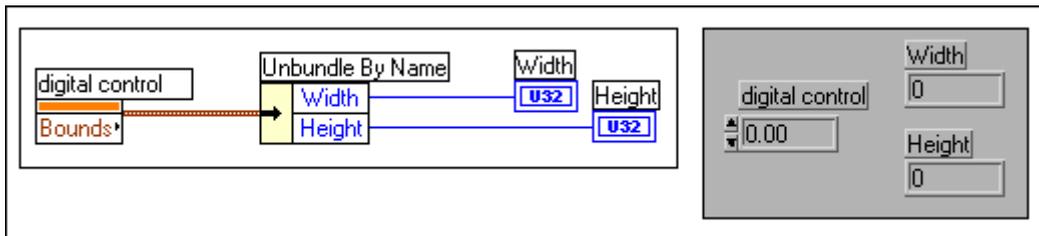
Examples of Attributes Specific to Controls or Indicators

In the following sections, you can learn about common applications of attribute nodes. See the general directory to explore other uses of the attribute node.
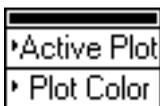
Changing Plot Color on a Chart

The attributes shown at left set or read the active plot (the trace for which subsequent trace-specific attributes are set or read) and the plot color for the active plot. Active Plot is an integer indicating which plot in a multi-plot chart you want active. Plot Color is an integer representing the desired color. You can access the Plot Color by selecting **Plot Info»Plot Color** from the attribute list.

This attribute node changes the plot color of the plot specified by the Active Plot attribute. In the following example, the color of the random

number plot changes color when the values go above a limit set by the user. Notice the active plot is specified before the plot color is changed.

Setting the Strings of a Boolean Attribute

This Boolean attribute sets or reads the labels on a Boolean control. The input is an array of up to four strings that correspond to the False, True, True Tracking, and False Tracking states.

- True and False: On and Off states of the Boolean.

- True and False Tracking: Temporary transition levels between the Boolean states. True Tracking is the transition when the Boolean is changed from True to False. False Tracking is the transition from False to True when the Boolean attribute is changed. The tracking applies only to Booleans with Switched When Released and Latch When Released mechanical action. These mechanical actions have a transitional state until you release the mouse. The text strings True Tracking and False Tracking are displayed during the transitional state.

You can set the display strings for Boolean controls to the string choices **Stop**, **Run**, and **Stop?** and **Run?**.

Setting the Strings of Ring Controls

The ring control is a pop-up menu control that holds the numeric value of the currently selected item. You can use it to present the user with a list of options. If the options cannot be determined until run time, you can use the attribute node to set the options. See Chapter 13, *List and Ring Controls and Indicators*, for more information about ring controls.

In the following example, the user is presented a panel with a ring control displaying a list of tests. The user selects a test and clicks the **Execute Test** button to continue.
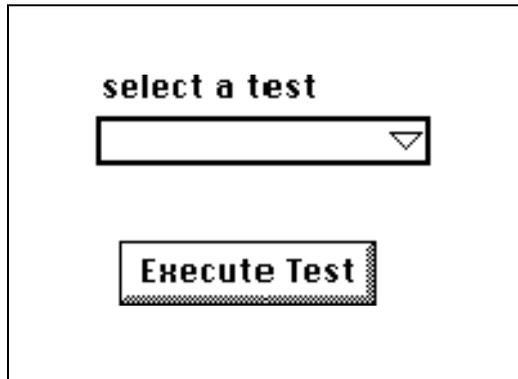
The block diagram shown in the following illustration reads a list of valid tests from a file and passes the list, represented as an array of strings, to an attribute node for the ring control. The diagram then loops, waiting for the user to click the **Execute Test** button. This gives the user a chance to select a test from the ring control, or to fill in information for other controls. When the user selects a test, the string corresponding to the numeric value of the ring control is read and then passed to a VI that executes the test.
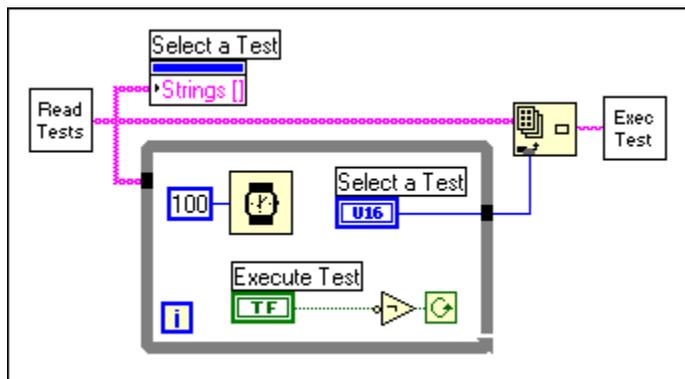
Using a Double-Clicked Listbox Item

The Double-Click attribute is a read-only attribute specific to listboxes. This attribute indicates which item from the listbox on the front panel you double-clicked. The value of the Double-Click attribute is set when you double-click an item or press <Enter> **(Windows** or **UNIX)** or

<Return> **(Macintosh** and **Sun)** after you select an item. The double-click value is –1 if nothing is double-clicked. It is reset to –1 after it is read using the attribute node. It also is reset to –1 if you select a different item, or if you set any of the other attributes.

The following illustration is an example of how to use the Double-Click attribute node to determine what test to execute.
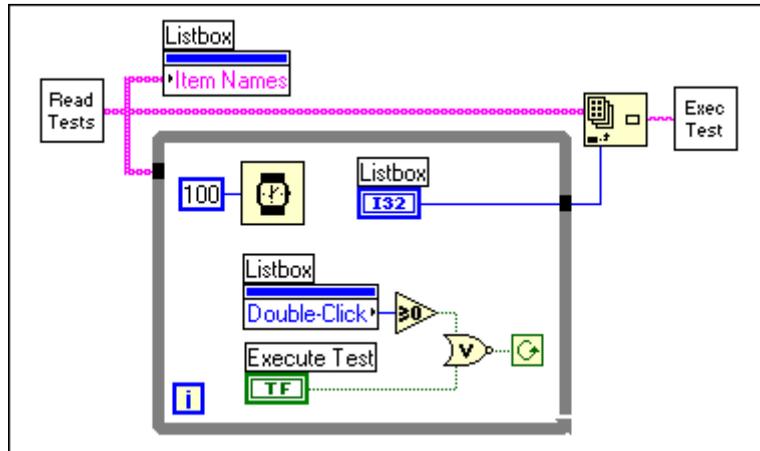
The loop stops when you click the Execute Test button or when you double-click an item, in which case the double-click attribute returns the item number instead of –1.

Selectively Presenting Users with Options

As users make selections, you might want to present them with other options. There are three possibilities.

1. One option is to use pop-up subVIs. You can create subVIs with the options you want to present to the user. By using the **Show Front Panel when Called** and **Close Afterwards if Originally Closed** items of **VI Setup»Execution Options** when you create your subVI, you can have one of these subVIs open when called.

2. Another method for presenting options is to use the Visible option of attribute nodes to selectively show and hide controls.

3. The third method for presenting options is to use the Disabled option of attribute nodes to disable controls selectively. When a control is disabled, the user cannot change the value.

Reading Graph Cursors Programmatically

You can use attributes to access information from one plot on a multi-plot graph, or one thumb on a multi-thumb slide, but you must indicate which item is being operated on. The multiple cursors on a graph are a good example of an attribute that must be activated before it can be accessed from the diagram.

The following block diagram shows a VI that displays data in a graph and programmatically reads the position of graph cursors.

Working through a While Loop, the VI first activates the Min Value cursor, and reads its numerical value. Next, the VI activates and reads the Max Value Cursor. Then the VI calculates and displays information about the cursor selection on the front panel. When you press the Confirm button, the VI exits the loop and confirms the cursor positions.

See the examples in `examples\general\graphs\zoom.llb` for an application of programmatically reading graph cursors.

Global and Local Variables

This chapter describes how to define and use global and local variables.
Use global variables to access a particular set of values easily from
multiple VIs. Local variables serve a similar purpose within a single VI.

Global and local variables are advanced G concepts. Be sure to study this
chapter carefully before using them.

For examples of how to use global and local variables, see
`examples\general\globals.llb` and
`examples\general\locals.llb`.

Global Variables

A global variable is a built-in G object. When you create a global variable
a special kind of VI is automatically created. You add front panel controls
to this VI that define the data types of the global variables it contains.

There are two ways to create multiple global variables. You can create
several VIs, each with one item, or create one multiple global VI by
defining multiple front panel controls on the one global variable front
panel. The multiple global VI approach is more efficient, because you
group related variables together.

You can create a global variable by selecting the global variable from the
Functions»Structures palette and placing it on the diagram.
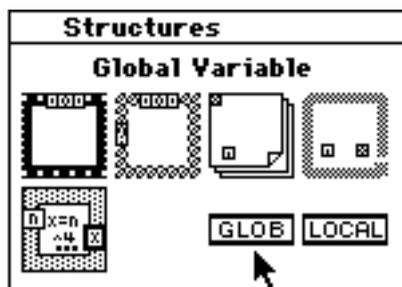
global node

A node for the global variable appears on the block diagram.

Double-click the node to open its front panel. You use this panel to define the data types for one or more global variables.

☞ **Note**   *Assign a name (label) to each control, because you must refer to a specific global variable by name. After you define the data types for the global variables, save the global VI.*

The following illustration is an example front panel describing three global variables—a number, a string, and a cluster containing two values.
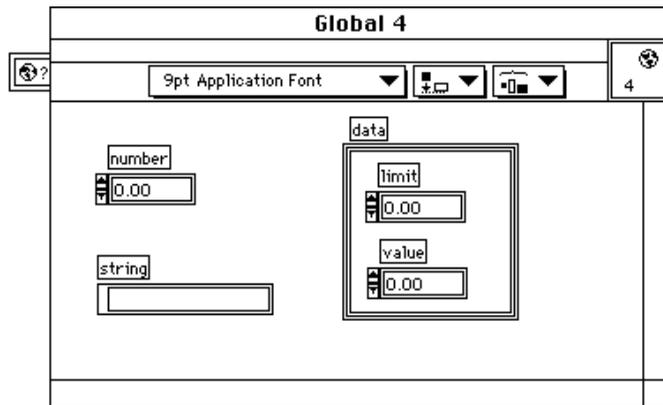
After you place a global variable on a block diagram and define a front panel for it, the node is associated with that front panel. Because a single global variable VI can define multiple global variables, you must select which global you want to access from a particular node. Select a global variable by popping up on the node and selecting the item by name from the **Select Item** menu as shown in the following illustration. Or, click the node with the Operating tool and select the item you want.
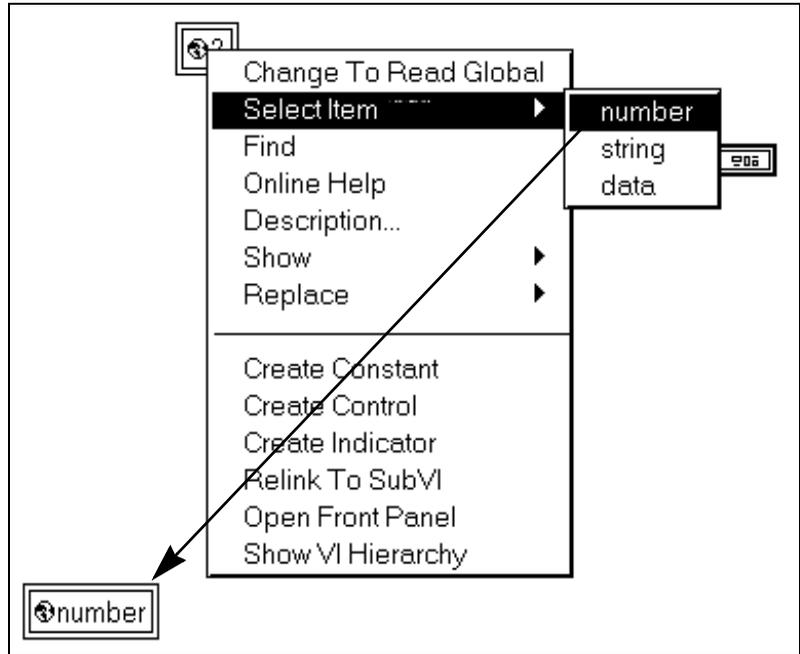
You either can write to a global variable or read from a global variable. Writing to a global variable means the value of the global changes; reading from a global means you access the global as a data source. If you want to write to or read from a global, select the **Change to Write Global** or the **Change to Read Global** item of the global variable pop-up menu.

Global variables can be written and read by any VI in memory. It is important to know where all the readers and writers in your application are, so that a global variable is not unexpectedly changed. G programs can have many activities going on at once, and it might not always be an easy thing to know when a global is accessed from various parallel diagrams. This problem with competing accesses to a shared resource like a global variable becomes even more pronounced in multithreaded software. There are many legitimate uses for global variables but indiscriminate use of them can lead to some difficult to debug situations. Use them carefully.

After you save a global variable VI, you can place its globals in other VIs using **Functions»Select a VI…**. If you select a global variable VI from the file dialog box, G places the global variable node on the diagram. You also can clone, copy and paste, or drag-copy a global from the Hierarchy window.

Local Variables

With a local variable you can write or read one of the controls or indicators on the front panel of your VI. Writing to a local variable has the same result as passing data to a terminal, except that you can write to it even though it is a control, or read from it even though it is an indicator. Also, you can have any number of local variable references to a particular 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.

☞ **Note**   *You must make sure the label you choose for your local variable associates with a front panel object. If your local variable does not associate with a front panel object, it does not work.*

LOCAL

local variable
icon

The easiest way to create a local variable is to pop up on the front panel control or terminal and select **Create»Local Variable**. A local variable appears automatically on the block diagram. Another way to create a local is to select the local variable from **Functions»Structures**, shown below.

local variable
node
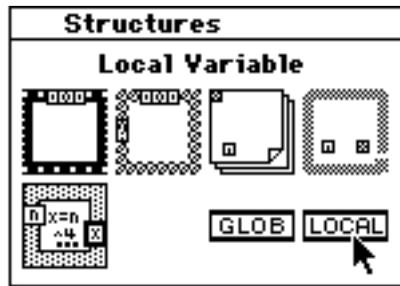
A node that looks similar to a global variable appears. You can pop up on the node, or click it with the Operating tool to select the control you want to read or set from a list of the top-level front panel controls, as shown in the following illustration. You also can determine whether you want to write to or read from the control by selecting either the **Change to Write Local** or the **Change to Read Local** items.
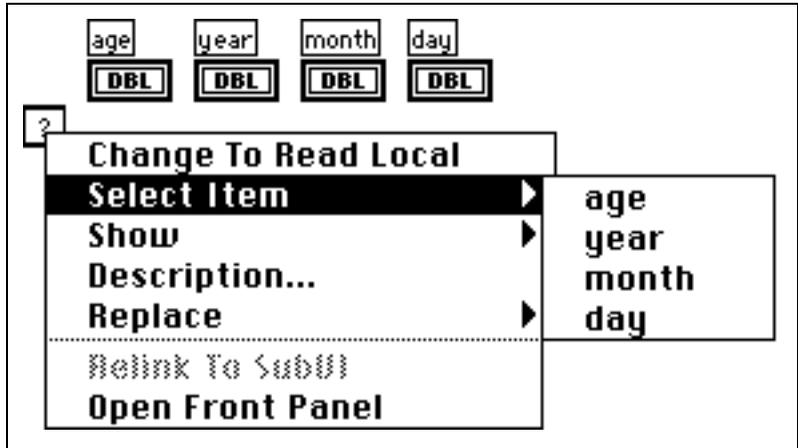
The following illustration shows how multiple local variables access the same control, each having a different sense, either read or write. You use the age variable twice in the diagram, once to write to, and another time to read from.
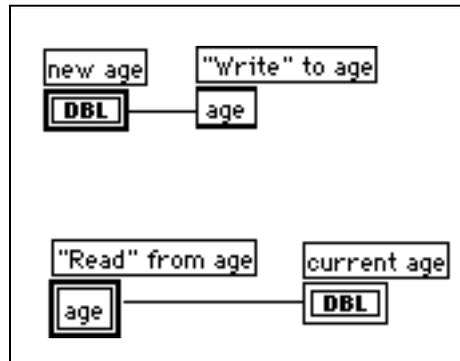
Be careful to sequence the access to local variables or global variables to produce the results you want, as in the preceding example. There is no guarantee the "Write" to age occurs before the "Read" from age if you do not create the proper sequencing yourself.

Part IV

Advanced Topics

This section contains information about advanced G features and techniques you can use to create virtual instruments.

Part IV, *Advanced G Topics*, contains the following chapters.

- Chapter 24, *Custom Controls and Type Definitions*, introduces custom controls and type definitions.

- Chapter 25, *Calling Code from Other Languages*, describes various methods of calling code written in other languages.

- Chapter 26, *Understanding the G Execution System*, describes VI multitasking and execution.

- Chapter 27, *Managing Your Applications*, describes how to manage files in your G applications.

- Chapter 28, *Performance Issues*, is in three sections. The first section describes the Performance Profiler, a feature that shows you data about execution time of your VIs and monitors single-threaded, multithreaded, and multiprocessor applications. The second section describes factors affecting run-time speed. The third section describes factors affecting memory usage.

- Chapter 29, *Portability and Localization Issues*, describes issues related to transporting VIs between platforms and VI localization.

24

Custom Controls and Type Definitions

This chapter introduces custom controls and type definitions.

You can customize a front panel control or indicator to make it better suited for your application. For example, you might want to make a Boolean switch that shows a closed valve when the switch is off and an open valve when the switch 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 you have customized in a directory or VI library, just as you do with VIs. Then, you can use this control on other front panels. You also can create an icon for your custom control, and have the name and icon of the control appear in the **Controls** palette.

If you require 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, you automatically update all the VIs that use it.

You can also use a customized control on a block diagram. This creates a constant with the same data type as the control. If you use a type definition on a block diagram, the resulting constant automatically updates when you make a change to the type definition.

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

Custom Controls

Using the Control Editor

Make certain you are in edit mode to customize a control. On the front panel, place a control most like the one you want to create. For example, to create a slide with its scale on the right, start by placing any vertical slide on the front panel.

With the Positioning tool, select the slide control and then choose **Edit»Edit Control…**. This option is available only when you select a control. You can edit only one control at a time from a front panel.
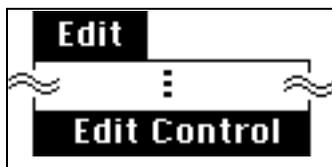
A window opens displaying a copy of the control. This window, shown in the following illustration, is called the Control Editor, and it is titled Control *N*, which is the name assigned to the Control Editor window until you save the control and assign it a permanent name.
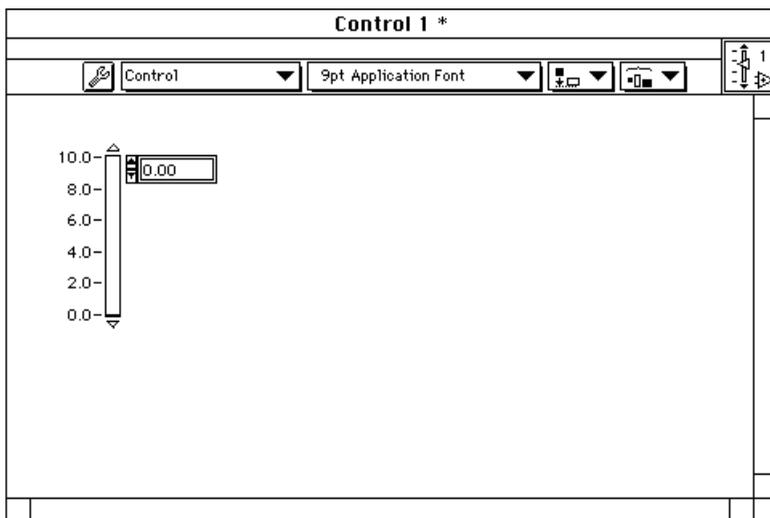
The Control Editor window looks like a front panel, but it is used only for editing and saving a single control; it has no block diagram and cannot run.

The Control Editor has an edit mode and a customize mode; the current mode is indicated by a button in the toolbar, as shown to the left of this text. The Control Editor is in edit mode when it first opens. In edit mode you can change the size or color of a control, and select options from its pop-up menu just as you do in edit mode of any front panel. In customize mode you can change the parts of a control individually. Customize mode is described in detail later in this chapter.

edit
mode

customize
mode

After you edit a control, you can use it in place of the original control on the front panel you were building when you opened the Control Editor. You also can save it to use on other front panels.

Applying Changes from a Custom Control

When you are ready to replace the original front panel control with your new custom control, select **File»Apply Changes** from the main menu of the Control Editor.
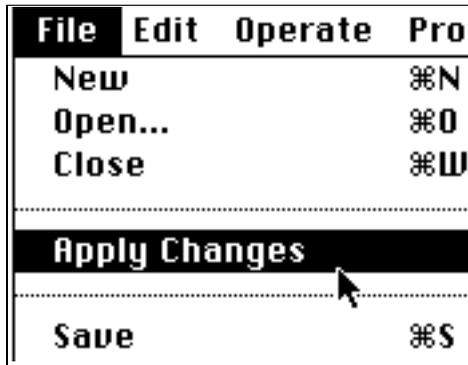
If your original front panel is the only place you use the custom control, you can close the Control Editor window without saving the control. Be sure to save the original VI with the custom control in place to preserve your work. If you want to use the custom control on other front panels in the future, you must save it as described in the *Saving Custom Controls* section of this chapter.

The **Apply Changes** option 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 open a custom control you saved earlier by selecting **File»Open**.

Valid Custom Controls

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 can be a cluster of other controls. The **not-OK** button appears temporarily while you move controls in and out of a cluster or array. To see an explanation for the error, click the **not-OK** button. If there is more than one control in the Control Editor the error message reads:

not-OK button

Possible errors include having more than one control or having no controls.

Saving Custom Controls

If you want to use your custom control on other front panels, choose **File»Save as…** from the main menu of the Control Editor. You save a control the same way you save a VI, in a directory or in a VI library. A directory or VI library contains controls, VIs, or both.

If you close the Control Editor window without saving your changes to the control, a dialog box asks you if you want to save the control.

Using Custom Controls

When you save the custom control, you can use it on other front panels by selecting **Controls»Select a Control…** from the front panel of any VI. You can also use it on block diagrams by selecting **Functions»Select a VI...** from the block diagram of any VI. If you use a custom control on a block diagram, you create a constant with the same data type as the custom control.

For more information about adding a custom control to the Controls palette, see the *Customizing the Controls and Functions Palettes* section of Chapter 7, *Customizing Your Environment*.

Making Icons
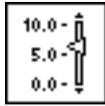
control icon

If you plan to add your control to the **Controls** palette, or if the control is a type definition, make an icon representing the control before you save it. Double-click the icon square in the top right corner of the Control Editor window or pop up on it and select **Edit Icon...** to create an icon for the control. This icon represents the control in the **Controls** palette, and if the control is a type definition, in the hierarchy window. See the *Type Definitions* section in this chapter and *Using the Hierarchy Window* in Chapter 3, *Using SubVIs*, for more information.

Independent Instances of Custom Controls

You can open any custom control you saved by selecting **File»Open**. A custom control always opens in the Control Editor window.

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

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

Customize Mode Option

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 the mode button in the toolbar belonging to the Control Editor, or by selecting **Change to Customize Mode** or **Change to Edit Mode** from the **Operate** menu while the Control Editor is the active window, as shown in the following illustrations.
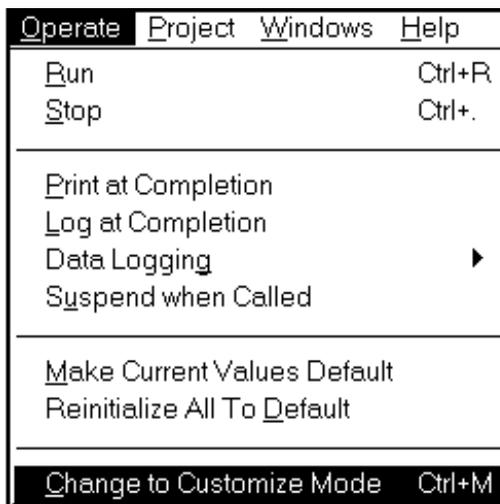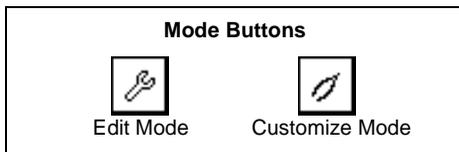
Independent Parts

All controls are built from smaller parts. For example, a slide control consists of a scale, a housing, a slider, the increment and decrement arrows, a digital display, and a name label. The parts of a slide are pictured in the following illustration.

increment arrow

name label

slider

3.50 kg

digital display

10.0—

8.0—

scale

6.0

4.0—

slider

2.0—

housing

0.0—

kg

unit label

decrement arrow
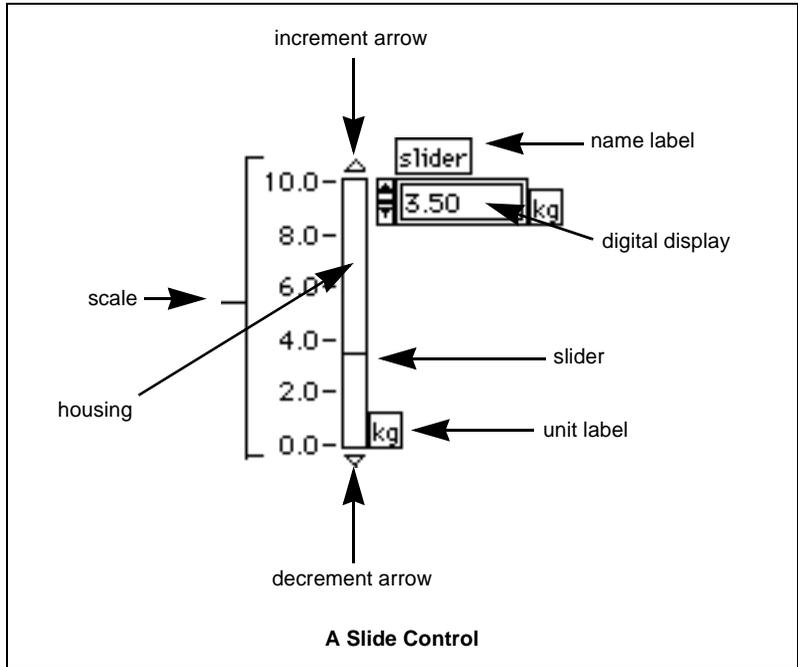
A Slide Control

When you switch to customize mode in the 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 scale
of the slide with the Positioning tool, only the scale moves. You can select
parts and align or distribute them using the **Align Objects** or **Distribute
Objects** rings from the toolbar; or change their layering order by using the
Reorder ring from the toolbar. Customize mode shows all parts of the
control, including any hidden in edit mode, such as the 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 the Operating tool is disabled. The Wiring tool is always disabled
in the Control Editor because you are not using the block diagram or
connecting controls to a connector pane.

Control Editor Parts Window

To help you size and position control parts, select **Windows»Show Parts Window**. The floating window that appears identifies the parts of the control and shows you the exact position and size of each part. The *Current Part* display in the Parts window displays 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 the current part display.

You also can scroll through the parts of the control by clicking 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 following illustration shows the Control Editor window on the left overlaid by the Control Parts window on the right. The name label of the slide is the current part, and is selected in the Control Editor window. The Control Parts window shows the menu of parts you see when you click the current part display. This example shows the name label is the current part, but that you are about to change to the scale part.
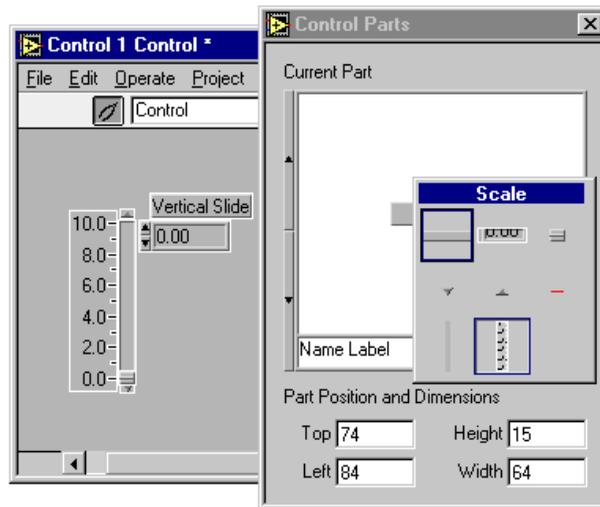
The Control Parts window shows you the exact position and size of the current part. These values are pixel values. When you move or resize a part in the Control Editor, the position and size in the Parts window are updated. You also can enter the position and size values directly in the

Control Parts window to move or resize the part in the Control Editor. This is useful when you must make two parts exactly the same size, or align one part with another. In the preceding illustration, the Parts window displays the position and size of name label of the slide—the upper left corner of the label is at the pixel coordinates (74, 84), and the label is 15 pixels high by 64 pixels wide.

The Control Parts window disappears if you switch to some other window. The Control Parts window reappears when you return to the Control Editor.

Customize Mode Pop-Up Menus for Different Parts

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 see 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 types of parts you can customize.

• Cosmetic parts, such as the slide housing, slider, and the increment and decrement arrows, are the most common. Cosmetic parts display a picture.

• Text parts, such as the name label of the slide. Text parts consist of a picture for the background (usually just a rectangle) and some text.

• Controls as parts, such as the numeric control used for the slide digital display. Knobs, meters, and charts also use a numeric control for a digital display. Some controls are even more complicated than that. For example, the graph uses an array of clusters for its cursor display part.

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

Cosmetic Parts

A cosmetic part has no dynamic user interaction or indication. 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 Control Parts window.

The following list describes the options available from the pop-up menu.

- **Copy to Clipboard**—Places a copy of the picture of the part 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** palette.

  When you require simple shapes like the housing rectangle for other parts, there are several advantages to using pictures copied from other parts, instead of making them in a paint program. Pictures taken from existing 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 built-in parts appear basically the same on both color and black-and-white monitors.

  In addition, you can color pictures taken from parts or decorations with the Color tool. Pictures imported from another source keep the colors they had when imported, because those colors are a part of the definition of that picture.

- **Import Picture**—Replaces the current picture of a cosmetic part with the picture currently on the Clipboard. Use this option to individually customize the appearance of your controls. For example, you can import pictures of an open and closed valve for a Boolean switch. If there is not a current picture on the Clipboard, **Import Picture** is disabled.

  There is a shortcut in the Control Editor for importing pictures into a Boolean control. When you are in edit mode you can select **Import Picture»True** or **Import Picture»False** from the pop-up menu of a Boolean. Doing this imports the picture into both the normal state and the corresponding transition state. See the section *Cosmetic Parts With More Than One Picture* later in this chapter for more information on transition states.

  You also can import different pictures for the transition states in customize mode. To do this, first pop up on the button and use **Picture Item** to change the third picture. With the **True»False** picture on the Clipboard, pop up again and select **Import Picture**. Repeat these steps for the fourth (**False»True)** picture.

- **Import at Same Size**—Replaces the current picture, but keeps the original size of the part, shrinking or enlarging the Clipboard picture to fit. If there is not a current picture on the Clipboard, **Import at Same Size** is disabled.

- **Revert**—Restores the part to its original appearance. **Revert** does not change the position of the part. If you open the Control Editor window by selecting **Edit Control** from a front panel, the editor reverts the part to the way it looks on that front panel. If you open the Control Editor window by selecting **File»Open...**, **Revert** is disabled.

- **Original Size**—Sets the picture of a part to its original size. This is useful for pictures you import from other applications, and then resize. Some of these pictures do not look as good as the original when resized, and you might want to restore their original size to fix them. If you do not import a picture, **Original Size** is disabled.

Cosmetic Parts with More than One Picture

Some cosmetic parts have more than one picture, which they display at different times. These different pictures are all the same size and use the same colors. For example, the increment arrow of the slide is a picture of a triangle, normally displayed as raised slightly from the background. It also has another picture, a recessed triangle, that appears while you are clicking it with the Operating tool to increment the value of the slide. The following illustration shows the two pictures of an increment arrow in action.
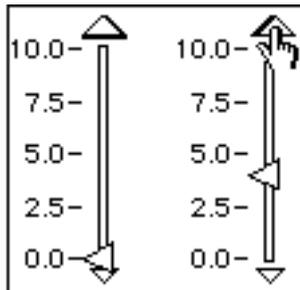
A cosmetic part with more than one picture has the **Picture Item** option on its pop-up menu, as shown in the following illustration.
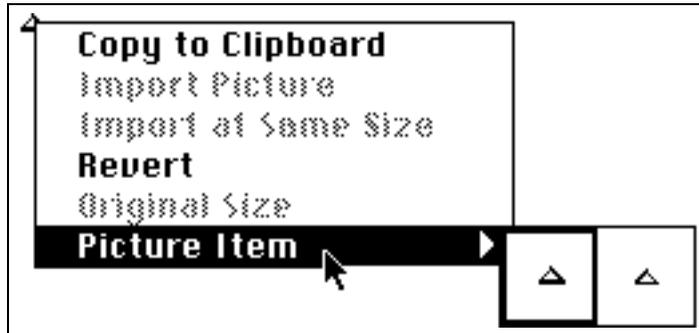
Picture Item displays all the pictures belonging to a cosmetic part. The picture item currently displayed has a dark border around it. When you import a picture, you change only the current picture item. To import a picture for one of the other picture items, first select that picture item and then import the new picture.

Cosmetic Parts with Independent Pictures

A cosmetic part with more than one picture can have pictures of different sizes, which each use different colors. The slide, for example, uses two pictures of different sizes to show which slider is active on a multi-value slide. The slide in the following example uses a bigger triangle to show the middle slider is the active one.
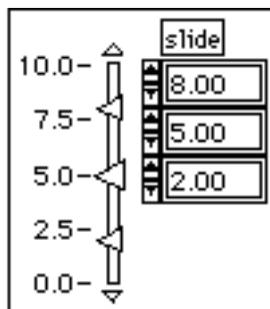
A Boolean switch also has more than one picture. Each picture can be a different size and have different colors. A Boolean switch has four different pictures—the first shows the false state; the second shows the true state. You use the third and fourth pictures when you set the mechanical action of a Boolean control to either **Switch When Released** or **Latch When Released**.

Until you release the mouse button, the value of the Boolean does not change with these two mechanical actions. Between the time you click the button and the time you release the click, the Boolean shows the third or fourth picture as a transition state. The third picture is for the true to false transition state, and the fourth is for the false to true transition state. In the following illustration of a toggle switch, the third and fourth pictures are the same, but this is not always the case.
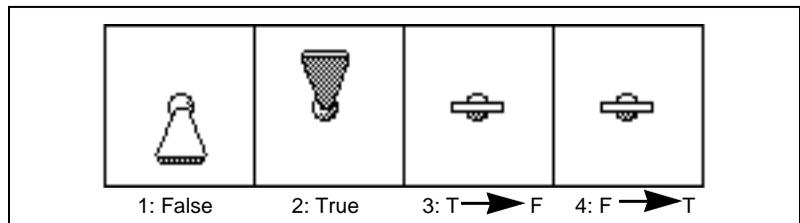
When a cosmetic part can have different sized pictures, the part has the **Independent Sizes** option on its pop-up menu, as shown in the following illustration.
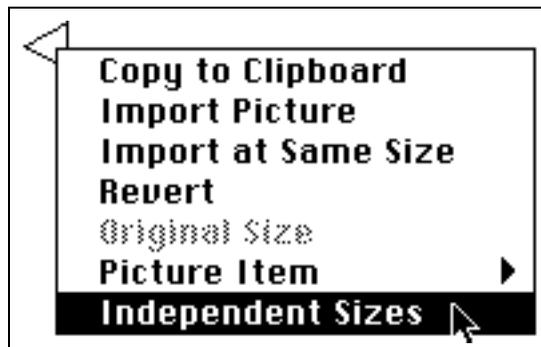
Independent Sizes is an option you can turn on only when you are in customize mode if you want to move and resize each picture individually without changing the other pictures of the cosmetic part. Normally, this option is not checked; and, when you move or resize the current picture of the cosmetic part, 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 the pop-up menu of a cosmetic part. The other items on this menu are the same as the text pop-up menu in front panel edit mode. An example pop-up menu for a text part is shown in the following illustration.

Only the background picture for the text part is shown in the Parts window, not the text itself. The background picture can be customized, not the text.

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 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 the Control Editor for the digital display and customize it there.

To customize a control that is a part of another control, open the Control Editor for it. You can open the Control Editor window for the part directly from the original front panel, if it can be selected separately from the main control in edit mode. The digital display can be selected separately from the slide control, for example. Then, you can choose **Edit»Edit Control...**.

You always can open the Control Editor window for the part from the Control Editor window of the main control, if it is in customize mode. Select the part in the Control Editor and choose **Edit»Edit Control...**. Control editors can be nested in this way indefinitely, but most controls use other controls as parts only at the top level. An exception is the graph,

which uses complicated controls as parts, which, in turn, use other controls as parts.

You cannot open a second Control Editor window for the main control already being customized.

The following illustration shows the Control Editor for the slide on the left, and a Control Editor window for the digital display on the right. You do not have to be in customize mode to open a nested Control Editor window unless you are unable to select the control part in edit mode.
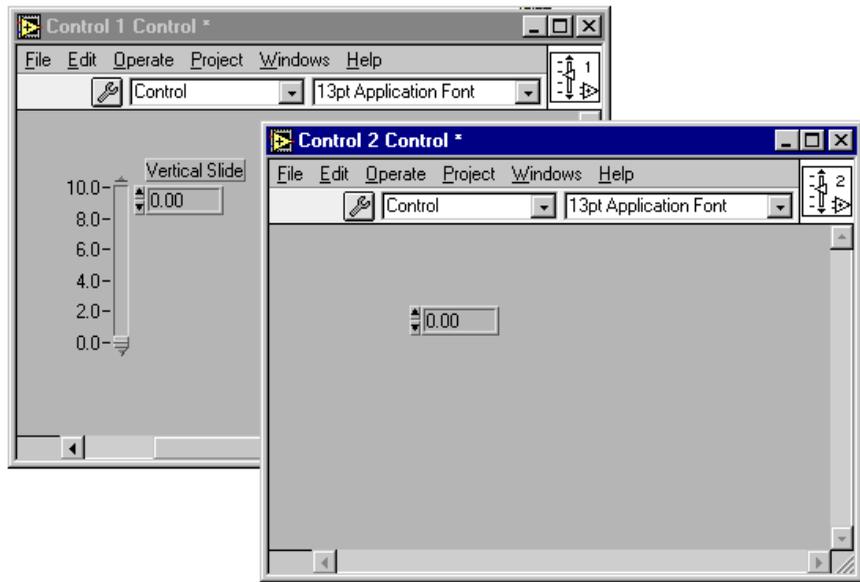
Adding Cosmetic Parts to Custom Controls

When you are making a custom control in the Control Editor, you can make even more changes to its appearance by adding cosmetic or text parts to it.

If you paste a picture or text from the Clipboard, create a label with the Labeling tool, or select a picture from **Controls»Decorations**, that picture or text becomes a part of your control and appears with the control when you place it on a front panel. You can do this in either edit or customize mode in the Control Editor. You can move, resize, or change the layering order of the new part, just like any other part. Your addition appears as a decoration part in the Control Parts window in customize mode.

You also can delete decoration parts when you are in the Control Editor.

The following illustration shows a custom graph with decoration parts, including the *Title of Graph* label, the *Legend:* label, and the box around the legend parts.
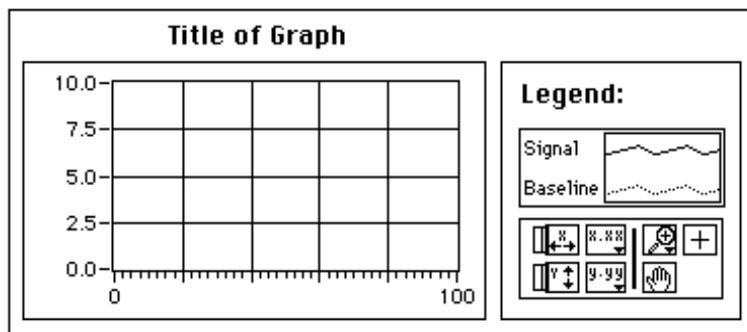
When you use a custom control on other front panels, you can change the size of any decoration parts you add, but you cannot move them.

Custom Control Caveats

There are some things you must be aware of when you make custom controls.

- Pictures created on one platform look slightly different when loaded on another platform (this applies to pictures imported into a Pict Ring or used as background on any front panel as well). For example, a picture with an irregular shape or a transparent background might have a solid white background on another platform. See the *Picture Differences* section in Chapter 29, *Portability and Localization Issues*.

- The Control Editor can change only the appearance of a control; it cannot change the behavior of a control. This has two implications.

    – You cannot change the way a control displays its data.

    – You cannot change the way a control behaves when you edit it, especially when you resize it.

    For example, when you make a ring control taller, the increment and decrement arrows also increase in height. If you move the increment and decrement arrows so they are side by side at the bottom of the ring, the ring continues to make them become taller when it becomes taller, and you produce some strange results.

- Custom controls often look correct, but occasionally behave oddly. If you like the control the way you made it but are not pleased with its irregular editing behavior, read about strict type definitions in the next section, *Type Definitions*, to learn how editing can be restricted.

Type Definitions

A *type definition* is a master copy of a control. You use the Control Editor to create the master copy, or type definition. Type definitions are useful when you use the same kind of control in many VIs. You can save the control as a type definition, and use that type definition in all your VIs. Then, if you change that control, you can update the single type definition file instead of updating the control in every VI using it.

General Type Definition: Matching Data Types

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 might 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 you subsequently use that type definition in many different VIs. Later, you change the type definition to a 16-bit integer digital control. When you change the type definition, you automatically can update every VI using that type definition. Only type definition instances you specifically set to not auto-update do no automatically update. See the section *Using Type Definitions* later in this chapter for more information.

You also can 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, you can update the type definition everywhere it is used.

As long as the data type matches the master copy, a type definition can have a different name, description, default value, size, color, or even a different style of control (for example, a knob instead of a slide).

Strict Type Definition: Everything Must Match

A type definition also can force almost *everything* about the control to be identical everywhere it is used, not just its data type but also its size, color, and appearance. This is called a *strict type definition*.

The only aspects of a control that can be different from the master copy of a strict type definition is the name, description, and default value. As an example, suppose you make a strict type definition that is a double-precision digital control with a red frame. Like the general type definition, if you change the strict type definition to an integer, you automatically update every VI using it. Unlike the general type definition, however, other changes to the strict type definition, such as changing the red frame color to blue, also requires VIs using the strict type definition to be updated. Furthermore, you can not disable automatic updating for a strict type-definition instance.

☞ **Note**   *Many attribute nodes are not available for strict type definitions. The only attributes available for strict type definitions are those affecting the appearance of the control, such as Visible, Disabled, Key Focus, Blinking, Position and Bounds.*

Type Definitions on the Block Diagram

When you use a type definition on a block diagram, it always has the appearance of a constant, not a control or indicator. Therefore, an instance of a strict type definition on a block diagram acts as an instance of a general type definition. It only automatically updates when the data type of the strict type-definition changes.

Creating Type Definitions

You make a type definition by setting the ring on the toolbar in a Control Editor window, as shown in the following illustration. Set up the control the way you want it, and choose **File»Save** in the Control Editor window.
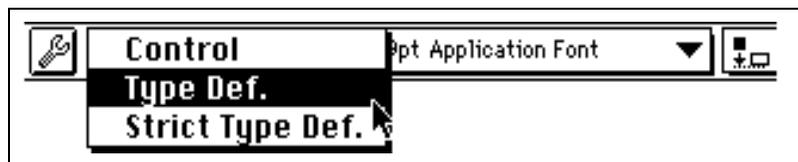
You can open any type definition you save by selecting **File»Open...**. A type definition always opens in a Control Editor window. Any changes you make to a type definition affect all VIs using it.

Using Type Definitions

Place general type definitions and strict type definitions on the front panel or block diagram of a VI as with any custom control. You can edit and operate a type definition on your front panel or block diagram as with any other control or constant.

☞ **Note**   *You cannot edit a strict type definition on your front panel except to change its name, description, or default value.*

You can tell a control is a type definition when you see the type definition options in its pop-up menu, as shown in the following illustration. You can recognize a strict type definition on your front panel or block diagram because you cannot edit it, and most of its pop-up menu options are missing.

```
Update from Type Def.
✓ Auto-Update Type Def.
  Disconnect from Type Def.
```

For each type definition you use on a front panel or block diagram, the VI 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 or block diagram and then select it and choose **Edit»Edit Control…**. The Control Editor that opens is the type definition you saved, with the name you gave it, instead of the generic Control *N*.

Updating Type Definitions

Your G development environment ensures the data type is the same everywhere a type definition is used. It also ensures everything about a strict type definition is the same in every front panel in which it is used. You can automatically correct any general type definitions or strict type definitions on your front panel, replacing incorrect ones on the front panel with an exact copy saved in the file or VI library and ones on the block diagram with a constant whose data type matches the control saved in the file or VI library.

If you edit an instance of a type definition on your front panel extensively, such as coloring and resizing it, you might not want this automatic update feature. You can pop up on the type definition on your front panel and turn off the **Auto-Update from Type Def.** option. Instead of automatically updating this type definition when necessary, the VI has a broken-run

arrow and the type definition on the front panel is disabled. You cannot run the VI until you fix the type definition, either by selecting the option **Update from Type Def.** from the pop-up menu, or by changing the data type to match the type definition. The **Auto-Update from Type Def.** option is not available in the pop-up menu of a strict-type definition since it always automatically updates.

When you use a type definition, you can assign it a different default value. However, if the data type of the type definition changes, all default data updates from the master copy sufficiently that the old default value cannot be converted to the new data type, such as when replacing a numeric with a string. Otherwise, the individual default values are preserved.

If you wire a type definition using **Create Constant**, **Create Control**, or **Create Indicator**, the type definition is updated from the master copy. For more information on **Create Constant**, **Create Control**, or **Create Indicator**, see Chapter 17, *Introduction to the Block Diagram*.

Searching for Type Definitions

Because a VI must keep a connection to each type definition, the file or VI library containing the type definition must be available to run a VI using it. If you open a VI, and if a type definition the VI needs is not found, the instances of that type definition in the VI are disabled and the run arrow is broken. To fix this problem, you must either find and open the correct type definition, or pop up on the disabled instance and select **Disconnect From Type Def.** Disconnecting from the type definition removes the restrictions on the data type of the instance, making it into an ordinary control or constant. You cannot re-establish that connection unless you find the type definition and replace the control with it.

Cluster Type Definitions

If you use a type definition or strict type definition that is a cluster, it is a good idea to use the Bundle By Name and Unbundle By Name functions on the block diagram to access the elements of the cluster, 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 you are referencing in Bundle by Name or Unbundle By Name, you must change your block diagram. Refer to the description of these functions in Chapter 14, *Array and Cluster Controls and Indicators*, for more information.

25

Calling Code from Other Languages

This chapter describes various methods of calling code written in other languages. These methods include using platform-specific protocols; creating a Code Interface Node to call code written specifically to link to VIs, and using a Call Library Function node to call Dynamic Link Libraries (.DLL files) in Windows, Code Fragments on the Macintosh, and Shared Libraries on UNIX. An additional method is using the LabWindows/CVI Function Panel converter to convert an instrument driver written in LabWindows/CVI.

Executing Other Applications from within Your VIs

You can execute other applications from within your VIs. The methods are different on Windows and UNIX than on the Macintosh.

(Windows, UNIX) You use System Exec to execute other applications from within your VIs. You can use the simple System Exec VI from the **Functions»Communication** palette to execute a command line from your VI. The command line can include any parameters supported by the application you plan to launch.

If you can access the application through TCP/IP (or DDE in Windows), you might be able to pass data or commands to the application. See the reference material for the application you plan to use to see the extent of its communication capability. If you are a LabVIEW user, you also can refer to Chapter 20, *Introduction to Communication*, and Chapter 21, *TCP and UDP* in the *LabVIEW User Manual*, for more information on techniques for using networking VIs to transfer information to other applications.

(Macintosh) You use Apple Event VIs to execute other applications from within your VIs. Apple Events are a Macintosh-specific protocol through which applications communicate with each other. They can be used to send commands between applications. You also can use them to launch other applications. If you are a LabVIEW user, see Chapter 21, *TCP and UDP,* in the *LabVIEW User Manual*, for details of different methods for using Apple Event VIs in G to launch and control other applications.

Using the Call Library Function

You can call most standard shared libraries (in Windows these are Dynamic Link Libraries or DLLs, on the Macintosh they are Code Fragments, and on UNIX they are Shared Libraries) using the Call Library Function node. The Call Library Function node includes a large number of data types and calling conventions. You can use it to call functions from most standard and custom-made libraries.

The Call Library Function node is most appropriate when you have existing code you want to call, or if you are familiar with the process of creating a DLL in Windows, Code Fragments on the Macintosh, or Shared Libraries on UNIX. Because a library uses a format standard among several development environments, you can use almost any development environment to create a library G can call. Check your compiler documentation to see if it can create standard shared libraries.

See the section *Call Library Function* later in this chapter for a detailed description of this node.

On a multithreaded operating system, you can make multiple calls to a dynamic link library (`.DLL` file) or shared library simultaneously. By default, all call library nodes, including nodes in previous versions, run in the user interface thread. Before configuring a call library node as reentrant, make sure the function called can be executed by multiple threads simultaneously. See the *Multithreading* section in Chapter 26, *Understanding the G Execution System*, for more information.

Using Code Interface Nodes

For applications in which you require the highest performance, or you want to pass arbitrary data structures to C code, you can create a Code Interface Node (CIN). By using CINs, you can call code written specifically to link to G-language VIs.

The CIN is a very general method for calling C code from G. You can pass arbitrarily complex data structures to and from a CIN. In some cases, you

might have a higher performance using CINs because data structures pass to the CIN in the same format they are stored in G.

To have this level of performance, however, you must learn to create a CIN. This requires you are a good C developer and take sufficient time to create the CIN you require. Also, because CINs are more tightly coupled with G, there are restrictions on which compilers you can use.

If you are a LabVIEW user and require information on how to create Code Interface Nodes, refer to the *LabVIEW Code Interface Reference Manual*, available in portable document format (PDF) on your software program disks or CD.

Call Library Function

Using the Call Library function you can call a 16-bit Windows 3.1 DLL, a 32-bit Windows 95/NT DLL, a Macintosh Code Fragment, or a UNIX Shared Library function directly.

(Macintosh) The Call Library function uses the Macintosh Code Fragment Manager (CFM). This is standard on all PowerMac machines. 680x0 Macintosh computers use the CFM extension. Additionally, the Call Library Node on 680x0 Macintosh computers cannot call variable argument functions. Shared libraries on the Macintosh operate differently than on other platforms. A file might contain more than one code fragment, each of which has a name.

The Call Library Function, shown at left, is available from **Functions»Advanced**.

If you double-click the Call Library Function or select **Configure…** from its node pop-up menu, G displays a dialog box you can use to specify from the library, function, parameters, and return value for the node as well as calling conventions in Windows. When you click the **OK** button, the node automatically adds the correct number of terminals, and sets the terminals to the correct data types.

The return value for the function returns to the right terminal of the top pair of terminals of the node. If there is no return value, this pair of terminals is unused. Each additional pair of terminals corresponds to a parameter in the functions parameter list. You pass a value to the function by wiring to the left terminal of a terminal pair. You read the value of a parameter after the function call by wiring from the right terminal of a terminal pair.

The Call Library Function dialog box is shown in the following illustration.
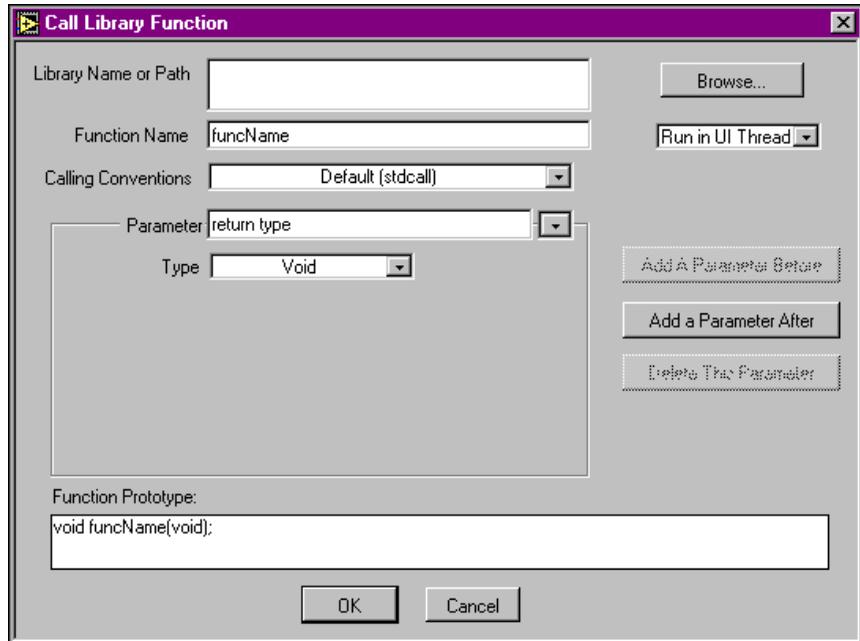
As you select items in the dialog box, an indicator at the bottom, called
Function Prototype, displays the C prototype for the selected function.

☞ **Note**   *Calling shared libraries written and compiled in C++ has not been tested and
might not work.*

*In Windows 3.1, an upper limit of 29, 4-byte arguments can be passed using
the Call Library Function. Double-precision floating-point parameters passed by
value are 8-byte quantities and count as two arguments. Thus, you are limited to
14 double-precision, floating-point parameters passed by value.*

*In Windows 3.1, you cannot have two or more Call Library Nodes in memory that
call the same function inside the same DLL with different arguments.*

*In Windows 3.1, DLLs must be 16-bit. In Windows 95/NT, DLLs must be 32-bit.
If you have a 16-bit DLL you want to call from Windows 95/NT, you either must
recompile it as a 32-bit DLL or create a "thunking" DLL. Refer to Microsoft
documentation for information on thunking DLLs.*

Calling Conventions (Windows)

Use the calling conventions of the ring to select the calling conventions for the function. The default calling convention for Windows 3.1 is Pascal, and Stdcall for Windows 95/NT. This default corresponds to the calling convention used by most DLLs. The alternative option is to use C calling conventions. Refer to the documentation for the DLL you are trying to call for the appropriate calling conventions.

Parameter Lists

Initially, the Call Library Function has no parameters and has a return value of void. You can click the **Add a Parameter Before** and **Add a Parameter After** buttons to add parameters to the function. You can click the **Delete this Parameter** button to remove a parameter.

You can use the parameter ring to select different parameters or the return value. When selected, you can change the parameter name to something more descriptive. The parameter name does not affect the call, but is propagated to output wires. Descriptive names make it easier to switch between parameters.

Indicate the type of each parameter using the type ring. The return type is limited to either **Void**, meaning the function does not return a value, **Numeric**, or **String**.

For parameters, you can select **Numeric**, **Array**, **String**, or **Adapt to Type**.

When you select an item from the type ring, you see more items you can use to indicate details about the data type and how to pass the data to the library function. The **Call Library Node** has a number of different items for data types, because of the variety of data types required by different libraries. Refer to the documentation for the library you call to determine which data types to use.

- **Void**—The type `void` is only accepted for the return value. This item is not available for parameters. Use it for the return value if your function does not return any values.

- **Numerics**—For numeric data types, you must indicate the exact numeric type using the data type ring. Items include the following.

  – Signed and unsigned versions of 8-bit, 16-bit, and 32-bit integers

  – Four-byte, single-precision numbers

  – Eight-byte, double-precision numbers

You cannot use extended-precision numbers and complex numbers. They generally are not used in standard libraries.

You also must use the format ring to indicate if you want to pass the value or a pointer to the value.

☞ **Note**   *In Windows 3.1, you cannot use either single-precision or double-precision data types as the return type. There is no standard method for DLLs to return single-precision and double-precision numbers. Floating-point return values are implemented differently by each compiler. If you must return a single-precision or a double-precision number, pass the data back as a parameter instead of as the return value.*

- **Arrays**—You can indicate the data type of arrays (using the same items as for numeric data types), the number of dimensions, and the format to use in passing the array. Use the **Format** item to select if you want to pass a pointer to the array data, or pass an **Array Handle** which is a pointer to a pointer to a four-byte value for each dimension followed by the data. If you select **Array Data Pointer**, pass the array dimension as separate parameter(s).

☞ **Note**   *In Windows 3.1, the Call Library Node uses only the array data pointer format for arrays. Additionally, in Windows 3.1 you can indicate the data is passed using a Huge pointer. You can use a Huge pointer if you must pass more than 64 kilobytes of data in the array. Only turn this item on if the DLL you are calling expects a Huge pointer of data. If you try to pass a Huge pointer to a function that expects a normal pointer, the application might crash.*

⚠ **Caution**   *Do not attempt to resize an array with system functions, such as realloc. Doing so might cause your system to crash.*

Strings—You can specify the string format for strings. The items for string format are C, Pascal, or G.

⚠ **Caution**   *Base your selection of the string format on the type of string the library function expects. Most standard libraries expect either a C string (string followed by a null character) or a Pascal string (string preceded by a length byte). If the library function you are calling is written specifically for G, you might want to use the String Handle format, which is a pointer to a pointer to four bytes for length information, followed by string data.*

☞ **Note**   *In Windows 3.1, you cannot use the String Handle format or return a string. On platforms that support a string return type, the string is immediately copied into a buffer. The string is not deallocated.*

⚠️ **Caution**   *Do not attempt to resize a string with system functions, such as realloc. Doing so can cause your system to crash.*

Adapt to Type—allows you to pass arbitrary G-language data types to DLLs. They are passed in the same way they are passed to a CIN. This means the following.

- Scalars are passed by reference (a pointer to the scalar is passed to the library).

- Arrays and strings are passed as a handle (pointer to a pointer to the data). See the *Code Interface Reference Manual,* available only in portable document format (PDF) on your software program disks or CD.

- Clusters are passed by reference.

- Scalar elements in arrays or clusters are in line. For example, a cluster containing a numeric is passed as a pointer to a structure containing a numeric.

- Cluster within arrays are in line.

- Strings and Arrays within clusters are referenced by a handle.

☞ **Note**   *In Windows 3.1, you cannot use* **Adapt to Type.**

Calling Functions that Expect Other Data Types

In some cases, you might encounter a function that expects a data type G does not use. For example, you cannot use the Call Library Function to pass an arbitrary cluster or array of non-numeric data.

Depending on the data type, you might be able to pass the data by creating a string or array of bytes that contains a binary image of the data you want to send. You can create binary data by typecasting data elements to strings and concatenating them.

Another option is to write a library function that accepts data types G does use, and parameters to build the data structures the library function expects, then calls the library function.

Finally, you can write a code interface node instead. Code interface nodes can accept arbitrary data structures, but take some time to master because you must understand data formats and storage in G.

LabWindows/CVI Function Panel Converter

☞ **Note**   *This feature is not available on the Macintosh.*

The LabWindows/CVI Function Panel converter automates the process of converting instrument drivers written in LabWindows/CVI so they can be used in G. A LabWindows/CVI instrument driver consists of a C source and header files, a Dynamic Link Library (DLL) **(Windows)** or shared library **(UNIX)** containing the compiled code, and a LabWindows/CVI-specific file called a Function Panel (FP) file. Function Panel files are used in LabWindows/CVI so that users can specify arguments and return values when editing a C function call statement by filling in values in a pop-up window.

The LabWindows/CVI Function Panel converter converts each function into a VI, using the information in the CVI FP file to determine the type, data representation, and placement of controls on the VI front panels. Each generated VI uses a Call Library Function node on its block diagram to call the appropriate C routine in the appropriate driver library.

Although many instrument drivers are written completely on the diagram without calling library functions, the LabWindows/CVI Function Panel converter has advantages in some cases. For instruments where a G driver is not available, this tool makes it relatively easy to take advantage of a CVI driver if one exists. Given the option, however, a true G driver consisting of G diagrams is preferable because it is easy for customers to view and modify, and because it multitasks well within the G environment. Library calls are synchronous, so any VIs running in parallel pause for the duration of the call.

☞ **Note**   *All CVI `.dll` files require run-time support. Install the CVI Run-Time engine to use VIs created by the conversion process.*

Conversion Process

If you select **File»Convert CVI FP File…**, a dialog box appears asking you to select a LabWindows/CVI Function Panel file. When you select an FP file, you see the following dialog box, in which you indicate where to save new VIs and what driver functions to convert.
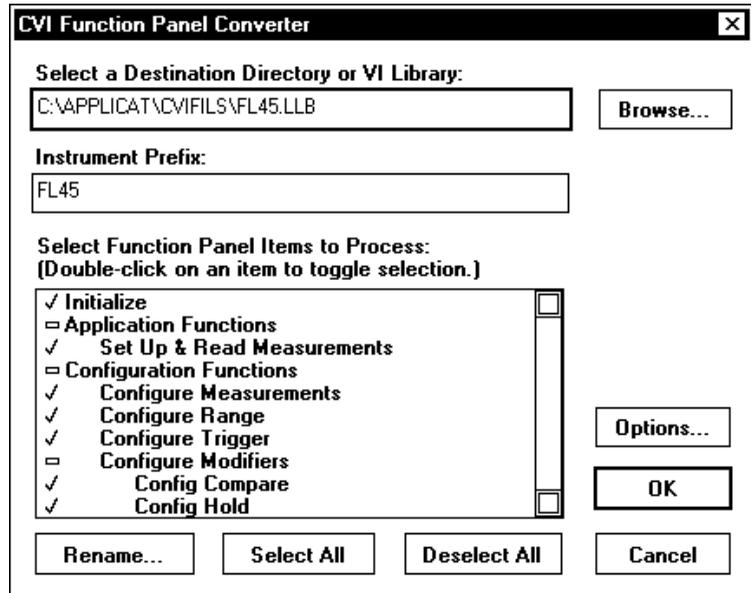
You specify the destination directory or VI library in the topmost text box. The suggested destination is shown, and you can change the path to place the new VIs anywhere you choose. You can use the **Browse…** button to specify the destination using a file dialog box in the usual way.

The **Instrument Prefix** text box shows the instrument prefix as shown by the function panel file. This prefix is prepended by CVI to all C function names in the Function Panel tree. The converter likewise prepends this prefix to the names of VIs it generates. The suggested prefix is shown, based on the DLL, but you can change or delete this string as you want.

The remainder of the dialog box contains the selection of function panels items to convert. A list box lists all items in the function panel tree, indented hierarchically by class, as in LabWindows/CVI. The class names are listed as well, but are grayed-out and not selectable. Initially, all nodes found in the Function Panel file are selected. Selected items are indicated by a checkmark symbol, non-selected items by no symbol, and class names by a rectangle.

Double-clicking an item toggles selection. The **Select All** and **Deselect All** buttons are used for convenience. You also can click an item once to move the highlighted bar to that item and click **Rename**. This pops up the following simple dialog box for renaming an FP item.

```
┌─────────────────────────────────────────┐
│ ▬         Rename FP Item                 │
├─────────────────────────────────────────┤
│ Enter a new name for the Function Panel item. │
│ This name will be used for the generated VI.  │
│ ┌─────────────────────────────────────┐ │
│ │Initialize                           │ │
│ └─────────────────────────────────────┘ │
│       ┌──────────┐     ┌──────────┐      │
│       │    OK    │     │  Cancel  │      │
│       └──────────┘     └──────────┘      │
└─────────────────────────────────────────┘
```
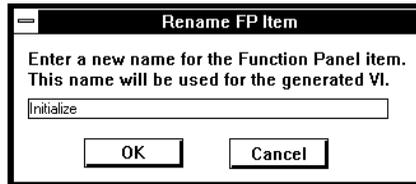
The **Options…** button brings up the following dialog box.

```
┌──────────────────────────────────────────────────┐
│ ▬              Conversion Options                 │
├──────────────────────────────────────────────────┤
│  ┌─ VI Generation Options: ──────────────────────┐│
│  │ ☐ Leave VI Front Panels Open                  ││
│  │ ☒ Create Library Call on Block Diagram        ││
│  │    ☒ Assume 16-bit DLL (INT type passed as INT16)││
│  │    ☒ Add Instrument Error I/O Checking        ││
│  │       ☒ Use SubVI for CVI Error Conversion    ││
│  │ ☐ Include CVI Class Names in VI Names         ││
│  │ ☐ Use C Function Names for VIs                ││
│  │    ☐ Capitalize and Remove Underscores in Names││
│  │ ☒ Convert All Control Names to Lowercase      ││
│  │ ☒ Assign Instrument Driver Icons Based on Name ││
│  │    ┌──────┐                                    ││
│  │    │ 2048 │  Default Size for Array Parameters ││
│  │    └──────┘                                    ││
│  └────────────────────────────────────────────────┘│
│       ┌──────────┐     ┌──────────┐               │
│       │    OK    │     │  Cancel  │               │
│       └──────────┘     └──────────┘               │
└──────────────────────────────────────────────────┘
```

The conversion options are as follows.

Leave VI Front Panels Open—Causes the converter to leave VIs in memory with their front panels open when conversion is complete rather than disposing of each VI after it is saved to disk. (Off by default.)

Create Library Call on Block Diagram—Causes the converter to place a Call Library Function node on the block diagram of each VI and wire up all front panel terminals appropriately. If this option is not on, only the front panel is created; nothing is dropped on the block diagram. (On by default.)

Assume 16-bit DLL—**(Windows 3.1)** Causes the converter to treat the Integer type from the LabWindows/CVI function panel as an INT16 rather than an INT32 when creating the type descriptor describing the arguments to the Call Library Function. This is necessary when calling DLLs created with some third-party compilers such as Borland C. (On by default.)

Add Instrument Error I/O Checking—Causes the converter to drop error-handling code on the block diagram of each VI created. If you select this option, Error In and Out clusters are dropped on the front panel, positioned below all other controls and indicators derived from the LabWindows/CVI function panel. The front panel terminals and Call Library Function on the block diagram are enclosed in a case structure, which is executed only if the status field of Error In is FALSE, indicating no error. (On by default.)

Use SubVI for CVI Error Conversion—Drops a subVI on the block diagram of each VI to map LabWindows/CVI-style error codes to G-style error clusters suitable for feeding to the General Error Handler.The integer return value from the Call Library Function node, the Error In cluster, and the name of the current VI is passed to the subVI. If an error is detected by the subVI, it is passed to Error Out, with status set to TRUE. If a warning is detected, it is passed to Error Out, but with status set to FALSE. Otherwise, Error In is passed to Error Out. If this option is not set, no subVI is dropped, and the block diagram is constructed so that an error is indicated in Error Out only if the Call Library Function return value is less than zero. (On by default.)

Include CVI Class Names in VI Names—Automatically creates a name for each VI based on either the function panel name or the C function name for each instrument driver option. If this option is checked, the Class name associated with the LabWindows/CVI function panel is prepended to the automatically generated name for the function.

Use C Function Names for VIs—Normally, the converter constructs the names of the VIs it generates directly from the function panel item names by merely prepending the instrument prefix and appending `.vi`. Unfortunately, this approach does not always produce unique names for each item, because LabWindows/CVI does not require the "leaves" of the function panel tree have unique names. Setting this option causes the converter to construct VI names from the actual C function names the FP items correspond to, thereby guaranteeing unique results.

To avoid problems caused by duplicate names, the converter checks to see if all items are unique when it builds the list of function panel items to be displayed. Non-unique items are flagged with a null symbol and are not selectable by double-clicking them. To prevent problems caused by duplicate names, the converter brings up a one-button alert when it first opens its dialog box if it detects such name conflicts, and automatically turns on the **Use C Function Names** option. If you prefer to use the function panel names instead, you can turn off this option manually, and then individually rename the items with name conflicts. Renamed items retain their user-supplied names even when the **Use C Function Names** option is changed.

Capitalize and Remove Underscores in Names—Because names built from C function names tend to be less "pretty" than those derived from the function panel item names, this option attempts to make them "prettier" by capitalizing initial letters and replacing underscores with spaces. Because it cannot expand abbreviations, the results still might not be as pleasing as the FP names, but this is inevitable with duplicate FP item names you do not want to rename individually.

Convert All Control Names to Lowercase—Converts control names to lowercase to conform to VXI *Plug&Play* standards.

Assign Instrument Driver Icon Based on Name—Assigns icons to VIs based on the name of the generated VIs. The converter searches for keywords such as initialize, close, self-test, reset, configure, or measure in the name of the function and uses the corresponding icon. If no keywords are found, a default icon is used. Additionally, the instrument prefix is stamped into the icon at the top left, up to a maximum of seven characters.

Default Size for Array Parameters—When an instrument driver DLL function outputs an array, the memory into which the function writes the array must be preallocated and passed into the DLL. With this option, you can select the default size, in elements, to allocate for arrays. When a Call Library Node function has an array as an argument, the converter drops an Initialize Array function to create an array to pass into the node. The initial size of such arrays is specified by the **Default Size for Array Parameters** option. A warning is generated in the .out file so that you can easily find VIs containing this construct and handle special cases individually.

Select **OK** at the main LabWindows/CVI Function Panel Converter dialog box. This brings up a file dialog box to select the library corresponding to the FP file being converted (if the **Create Library Call** option is set). Canceling this dialog box simply leaves the library path unspecified in the Call Library Function node. It does not abort conversion.

After you click **OK**, the converter brings up a working status dialog box to display the name of each new VI as it is created. A log file named *prefix*.out is created, listing all the VIs created, and any warnings or errors that occurred during conversion. If any warnings or errors do occur, you are notified to look at this file when conversion is complete.

26

Understanding the G Execution System

This chapter describes VI multitasking and execution.

With the G execution system, you can run multiple VIs simultaneously. In addition, within a given VI you might have several parallel branches, each of which also can execute simultaneously.

In normal use, it is not necessary to be concerned with the details of how multitasking takes place. You can think of portions of a diagram as executing in parallel, and multiple VIs as running in parallel. With the execution system multitasking capability, you can edit or single step through a VI while other VIs continue to run. Also, if you inadvertently build a VI with an infinite loop, it does not lock up the computer or prevent other VIs from executing.

In some applications where timing becomes more critical, a better understanding of how the multitasking system works is important. The following section covers this topic.

Multitasking Overview

Most computers have only one processor, meaning only one task executes at any given time. Multitasking is achieved by running one task for a short amount of time and then having other tasks run. As long as the amount of time each task has is small enough, there is the appearance of having multiple tasks running simultaneously.

The Windows 3.1 and Macintosh operating systems primarily use a form of multitasking called cooperative multitasking, which requires each program be written to voluntarily yield to other tasks periodically. Most applications spend a considerable amount of time waiting for events such as user interaction, so there are normally a number of opportunities for yielding to other tasks. However, if any task remains busy for too much time without yielding, other tasks do not get a chance to run. Cooperative multitasking only works well if all programs are written very carefully with it in mind.

In practice, most programs do not do a good job of cooperatively multitasking when performing common tasks such as printing, dragging windows, or saving files.

Many operating systems support another form of multitasking called preemptive multitasking. With preemptive multitasking, the operating system handles the switching and scheduling of tasks. Each task is given a limited amount of time to execute. When the time is up for a given task, one task is forced to pause while another starts execution. This switching is handled at arbitrary points without any special coding on the part of the developer.

Multithreading

When an application needs to perform a task, there are usually several ways to attack the problem. One way is to complete the task one step at a time. However, most tasks can be divided into smaller tasks that theoretically can be done in parallel. For instance, when you print a document, the application begins printing in the background, and you edit other documents. Multithreading is the term for applying multitasking to a single application, and breaking it into smaller tasks so they execute in parallel.

Just as with cooperative multitasking, you can design programs that cooperatively multithread within an application. Each thread must be written to periodically yield so other threads or tasks can run. Since all of the threads in a multithreaded application are developed as part of the same program, the developer has more control in ensuring cooperation takes place. However, it is difficult to do this uniformly throughout an application, and it is likely that most cooperatively multithreaded applications lose some potential parallelism.

Some operating systems provide a mechanism for preemptive multithreading. The operating system manages the context switches between the threads of the program in exactly the same manner as it manages the different programs in the preemptive multitasking system. The result can be a better distribution of the processor to the active threads of the program.

The G Execution System

The G execution system uses preemptive multithreading on operating systems that support it as well as cooperative multithreading. Even on systems with preemptive multithreading, a limited number of threads are used, so in certain cases it falls back to using cooperative multithreading.

(Windows 95/NT, Solaris 2 and **PowerMAX)** The application is multithreaded. The execution system preemptively multitasks VIs using threads. However, there are a limited number of threads available, so for highly parallel applications it falls back to using cooperative multitasking when available threads are used up. Also, the operating system takes care of preemptively multitasking between the application and other tasks.

(Macintosh and **Windows 3.1)** The application is single-threaded. The execution system cooperatively multitasks VIs using its own scheduling system. The application also cooperatively multitasks with other applications by periodically yielding a small amount of time.

(HP-UX and **Solaris 1)** The application is single-threaded. The execution system cooperatively multitasks VIs using its own scheduling system. The operating system takes care of preemptive multitasking between the application and other tasks.

Basic Execution System

The following description applies to both single-threaded and multithreaded execution systems.

The execution system maintains a queue of active tasks. For example, if you had three loops running in parallel, at any given time one task is executing and the other two are waiting in the queue. Assuming all tasks have the same priority, one of the tasks executes for a certain amount of time. That task is then put at the end of the queue, and the next task executes for a time. When a task completes execution, the execution system removes it from the queue.

The execution system executes the first element of the queue by calling the generated code of the VI. At some point, the generated code of that VI checks with the execution system to see if it assigns another task to execute. If not, the code for the VI continues to execute.

Managing the User Interface in the Single-Threaded Application

In addition to executing VIs, the execution system must coordinate interaction with the user interface. When you click a button, move a window, or change the value of a slide control, the execution system takes care of managing that activity and ensuring VI execution continues in the background.

The single-threaded execution system multitasks by switching back and forth between responding to user interaction and running VIs. When execution returns to the execution system from a VI, it checks to see if any user interface events require handling. If not, then the execution system either returns to the VI or takes the next task off the queue.

When you press buttons or pull down menus, the action you perform might take a while to complete. The execution system keeps running VIs in the background by switching back and forth between responding to your interaction with the control or menu and executing VIs.

Multithreaded Application and Multiple Execution Systems

Multithreaded versions have multiple execution systems. You can assign VIs to one of six different execution systems through the **Preferred Execution System** in **VI Setup»Execution Options.**
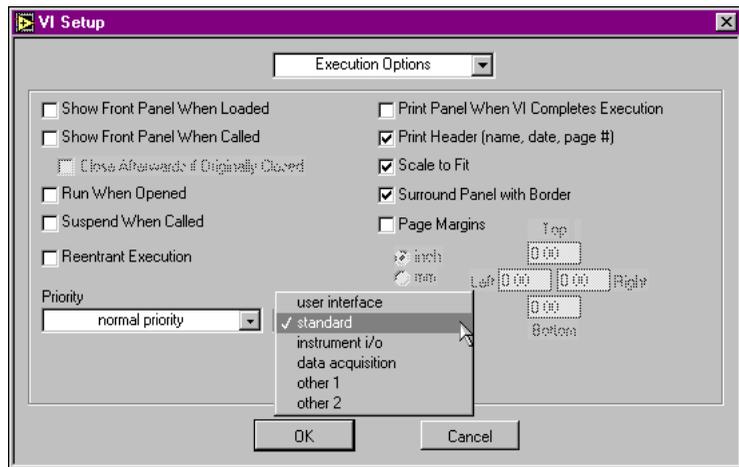
The execution systems you can choose are as follows.

- User Interface
- Standard
- Instrument I/O
- Data Acquisition
- Other 1
- Other 2

The purpose of having several execution systems is to provide some rough partitions for VIs that must run independently from other VIs. By default, VIs run in the Standard execution system, which runs in separate threads from the user interface. The Instrument I/O execution system is included to prevent VISA, GPIB, and Serial I/O from interfering with other VIs. Similarly, the Data Acquisition execution system is set for the data acquisition VIs.

The User Interface execution system behaves exactly the same in the multithreaded version as in the single-threaded version. The User Interface system is responsible for taking care of the user interface. VIs can execute in the user interface thread, but the execution system alternates between cooperatively multitasking and responding to user interface events.

Each of the other execution systems has its own queue. These execution systems are not responsible for managing the user interface. Whenever a VI in one of these queues needs to update a control, it passes responsibility to the User Interface thread.

Also, every execution system except for the User Interface system has two threads responsible for executing VIs from the queue. Each thread takes care of executing a task, so if a VI calls a CIN, for example, the second thread continues to execute other VIs within that execution system. Since each execution system has a limited number of threads, once the threads are full you end up with pending tasks just as you do in a single-threaded system.

While VIs you write run correctly if you leave them set to the Standard execution system, consider setting them to use another execution system as well. If you are developing instrument drivers, for example, you might want to set your VIs to use the Instrument I/O execution system.

Even if you leave VIs set to the Standard execution system, one important benefit of multithreading is the separation of the user interface into its own thread. Any activities conducted in the user interface (such as drawing on

the front panel, responding to mouse clicks, and so on) can take place without robbing the block diagram code of execution time. Displaying a lot of information on a graph does not prevent the block diagram code from executing. Likewise, executing a long computational routine does not prevent the user interface from responding to mouse clicks or keyboard data entry.

Computers with multiple processors benefit even more from multithreading. On a single-processor system, the operating system preempts the threads and distributes time to each thread on the processor. On a multi-processor computer, threads can run on the multiple processors simultaneously, so more than one activity can truly occur at the same time.

The names Instrument I/O, Data Acquisition, and so on are simply suggestions for the type of tasks you might want to place within these systems. I/O and DAQ work in other systems, but using these labels helps you to partition your application and understand the organization.

Other 1 and Other 2 are available if there are other tasks in your application requiring their own thread.

Synchronous/Blocking Nodes

As mentioned previously, a few nodes are synchronous, meaning they do not multitask with other nodes. In the multithreaded version, this means they execute to completion, and the thread in which they run is monopolized by that task until it completes.

Code Interface Nodes (CINs), DLL calls, and all computation functions execute synchronously. Most analysis VIs and DAQ VIs contain CINs and therefore execute synchronously. For example, a Fast Fourier Transform (FFT) executes to completion without the thread it runs in, regardless of how long the FFT executes.

Almost all other nodes are asynchronous. Structures, I/O functions, timing functions, and subVIs you build all execute asynchronously.

The Wait, Wait for Occurrence, Dialog Box, GPIB functions, VISA functions, and Serial VIs wait for the task to complete, but can do so without holding up the thread. These tasks are taken off of the queue until their task is complete. When it completes (for example, the user presses a button on a dialog displayed by the Dialog Box function), the task is put on end of execution queue.

Prioritizing Tasks

There are two methods for prioritizing parallel tasks. One is to change the priority setting in **VI Setup**. The other is to make strategic use of Wait functions.

In most cases, do not change the priority of a VI from the default. Using priorities to control execution order might not produce the results you expect. If used incorrectly, the lower priority tasks can be pushed aside completely.

Wait Functions for Prioritizing Tasks

You can use the Wait function to make less important tasks execute less frequently. For example, if several loops are in parallel and you want some of them to execute more frequently, put the Wait functions in the lower priority tasks. This relinquishes more time to other tasks.

As described in the *Synchronous/Blocking Nodes* section of this chapter, when a diagram waits, it is completely taken off of the queue, so other tasks can run.

It is particularly appropriate to use waits in loops polling the user interface. A wait of 100 to 200 milliseconds is not really noticeable, but it frees up the application to handle other tasks more effectively. In addition, it frees up the operating system so it has more time to devote to other threads or other applications.

VI Setup Priority

You change the priority of a VI using the **Priority** menu item in the **VI Setup** dialog box.
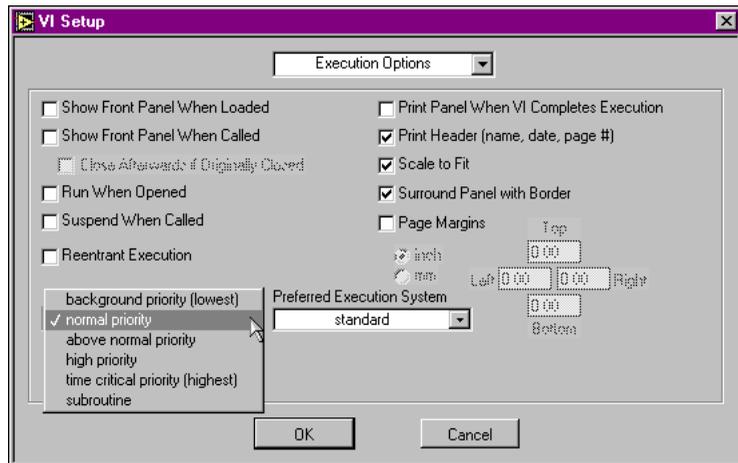
There are six levels of priority, listed below in order from lowest to highest.

- background priority (lowest)
- normal priority
- above normal priority
- high priority
- time critical priority (highest)
- subroutine priority

The first five priorities are similar in behavior (lowest to highest), while subroutine priority has some additional characteristics. The following applies to all of these priorities except the subroutine level.

Priorities in the User Interface Thread and Single-Threaded Execution System

Within the user interface thread, priority levels are handled in the same way for both the single-threaded and multithreaded execution system. The following paragraphs describe the way in which priorities are handled for these configurations.

In the single-threaded system and in the user-interface thread of the multithreaded system, the execution system queue described earlier has

multiple entry points. Higher priority VIs are placed on the queue in front of lower priority VIs. If a high priority task is executing and the queue only contains lower priority tasks, the high priority VI continues executing. For example, if the execution queue contains two VIs of each priority level, the time critical VIs share execution time exclusively until both of them finish. Then, the high priority VIs share execution time exclusively until both of them finish, and so on.

The exception to this occurs if the higher priority VIs calls a function that waits such as the Wait (ms) function (see the Synchronous/Blocking Nodes section earlier in this chapter for a list of asynchronous functions that wait). In this case, the higher priority VIs are taken off the queue until the wait or I/O is complete, assigning other tasks (possibly with lower priority) to execute. When the wait or I/O is complete, the execution system reinserts the pending task on the queue in front of lower priority tasks.

Also, if a high priority VI calls a lower priority subVI, that subVI is boosted to the same priority level as the caller for the duration of that call. Consequently, you do not need to modify the priority levels of the subVIs a VI calls in order to raise its priority level.

Use priorities cautiously, because lower priority tasks easily get starved for time by higher priority tasks. If the higher priority tasks are designed to run for long periods, lower priority tasks are not going to execute unless the higher priority task periodically waits or performs I/O so it is taken off the queue. When using priorities, consider adding waits to the less time-critical sections of your diagrams to free up time.

Priorities in Other Execution Systems of the Multithreaded System

Earlier, this chapter described six execution systems, corresponding to the Execution System menu item in **VI Setup**. In reality, for each of these six categories, there is a separate execution system for each priority level (not including subroutine priority level, nor the user interface execution system). Each of these prioritized execution systems has its own queue and two threads devoted to handling diagrams on that queue.

Rather than having six execution systems, there is 1 for the user interface system, regardless of the priority, and 25 for the other systems (5 systems multiplied by 5 for each priority level).

The threads for each of these execution systems are assigned operating system priority levels based upon your classification. What this means in normal execution is higher priority tasks get more time than lower priority

tasks. Just as with priorities in the user interface thread above, higher priority tasks might starve lower priority tasks if they execute for long periods of time without waiting periodically.

Some operating systems try to help avoid starvation of lower priority tasks by periodically giving an artificial boost to the priority level of lower priority tasks. Consequently, on operating systems with priority boosting, even if a high priority task wants to execute continuously, lower priority tasks periodically get a chance to operate. However, do not rely upon this since it varies from operating system to operating system, and on some operating systems, priorities of tasks and priority boosting behavior are adjusted by the user.

The user interface system has only a single-thread associated with it. This thread is set to the same operating system priority as Normal priority for other execution systems. Consequently, if you set a VI to run in the standard execution system with above normal priority, it might starve out the user interface. This might result in a sluggish or even non-responsive user interface. Likewise, if you assign a VI to run at background priority, it runs with lower priority than the user interface thread.

Just as in the user interface discussion above, when a VI calls a lower priority subVI, the execution system raises the subVI priority to the same level as the caller for the duration of the call. Consequently, if a VI and its subVI are both assigned the same execution system in VI setup, the call to the lower priority subVI stays in the same prioritized execution system. However, if the VI calls a higher priority subVI, then that subVI call runs in a different, higher priority execution system.

Subroutine Priority Level

If you set a VI to subroutine priority (the highest level), the behavior is slightly different. The idea of subroutine priority level is to permit a VI to execute as efficiently as possible. The compiler compiles Subroutine VIs so they do not share time with other VIs.

When a VI runs at subroutine priority, it effectively takes control of the thread in which it is running (and runs in the same thread as its caller). No other VI can run in that thread until the subroutine priority VI finishes executing, even if the other VI is also marked as a subroutine. In the single-threaded execution system, this means no other VI executes. In non-User Interface execution systems, the thread running the subroutine does not handle other VIs, but the execution-system second thread, along with other execution systems, can continue to run VIs.

In addition to not sharing time with other VIs, subroutine VI execution is streamlined so that front panel controls and indicators are not updated when the subroutine is called. Watching a subroutine VI front panel reveals nothing about its execution.

A subroutine VI can call other subroutine VIs, but it cannot call a VI with any other priority. Use subroutine priority in situations in which you want to minimize the overhead in a subVI that performs simple computations.

Also, since subroutines are not designed to interact with the execution queue, they cannot call any function that normally takes them off of the queue. This means they cannot call any of the Wait functions, I/O functions, or dialog box functions.

Subroutines have an additional item that can help in time-critical applications. If you pop up on a subVI and select Skip Subroutine Call if Busy, the execution system skips the call if the subroutine is currently running in another thread. This might be advantageous in time-critical loops where the operations the subroutine performs are skipped safely, and where you want to avoid the delay of waiting for the subVI to complete. If you skip the execution of a subVI, all outputs of the subVI are set to the default value for that data type. Consequently, numeric outputs are set to zero, string and array outputs are empty, and Booleans are set to False. Notice this is the default for the type, not the default value for the indicator on the subVI front panel. If you want to detect whether a subroutine actually executed, make it return True if it successfully ran. The default value of False is automatically returned otherwise.

Reentrant Execution Overview

Under normal circumstances, the execution system cannot execute multiple calls to the same subVI simultaneously. If you try to call a subVI that is not reentrant from more than one place, one call executes and the other call waits for the first to finish before executing. If you make a VI reentrant (using VI Setup), each instance of the call maintains its own state of information. Then, the execution system runs the same subVI simultaneously from multiple places. Reentrant execution is useful in the following situations.

- When a VI waits for a specified length of time or until a timeout occurs.

- When a VI contains data 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 begin reentrant execution through the **Execution Options** of the
VI Setup dialog box. If you select reentrancy, several other menu items
become unavailable.
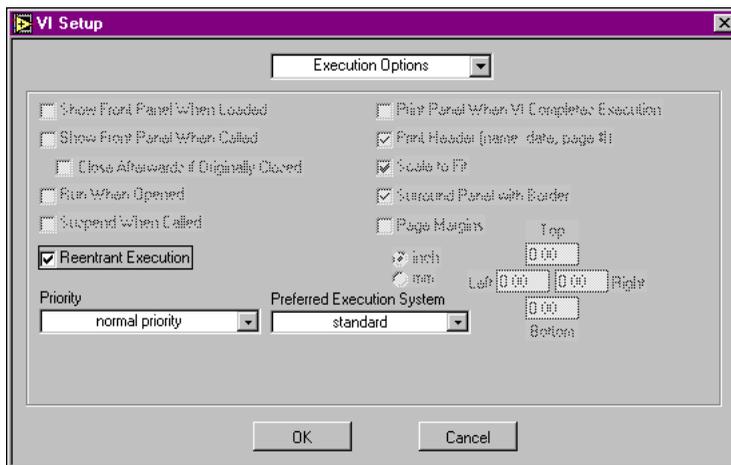
Notice the following items are not available when **Reentrant Execution** is
selected.

• Opening a front panel when loaded

• Opening a front panel when called

• Execution highlighting

• Single-Stepping

These menu items 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.

Examples of Using Reentrancy

The examples in the next two sections demonstrate reentrant VIs.

Using a VI That Waits

The following reentrant example describes a VI, called Snooze, which takes hours and minutes as input and waits until that time arrives. If you want to use this simultaneously in more than one location, the VI must be reentrant.
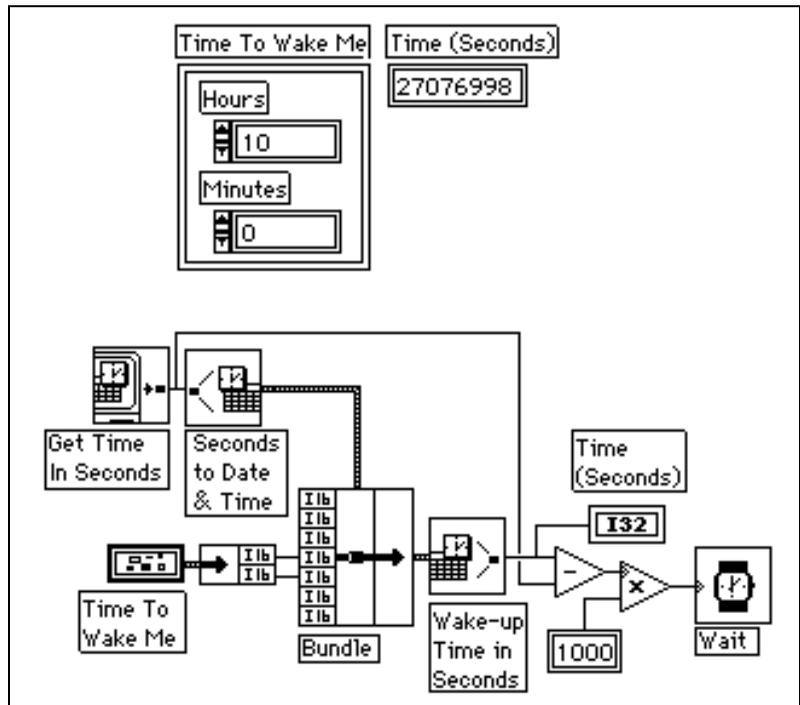
The Get Time In Seconds function reads the current time in seconds and the Seconds to Date/Time and converts this value to a cluster of time values (year, month, day, hour, minute, second, and day of week). A Bundle function replaces the current hour and minute with values representing a later time on the same day in the front panel Time To Wake Me cluster control. The adjusted record then is converted back to seconds, and the difference between the current time in seconds and the future time is multiplied by 1,000 to obtain milliseconds. The result passes to a Wait function.

Lunch and Break are two VIs that use Snooze as a subVI. The Lunch VI, whose front panel and block diagram are shown in the following illustration, waits until noon and pops up a front 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 the pop-up subVIs are different.

For Lunch and Break to execute in parallel, Snooze must be reentrant. Otherwise, if you start Lunch first, Break waits until Snooze wakes up at noon, which is two hours late.
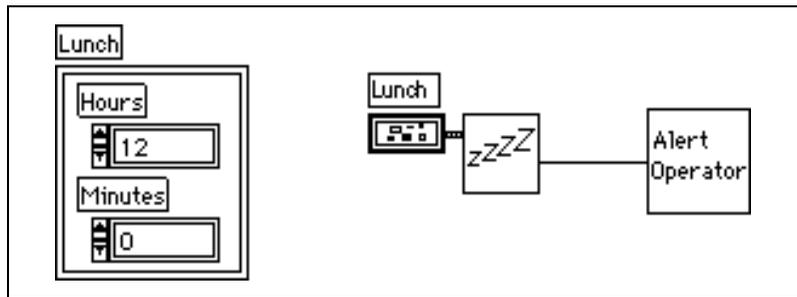
Using a Storage VI Not Meant to Share Its Data

Another situation that requires reentrancy is when you make multiple calls to a subVI that stores data. Suppose you create a subVI, ExpAvg, which calculates a running exponential average of four data points. To remember the past values, ExpAvg uses an uninitialized shift register with three left terminals. If you are a LabVIEW user, see Chapter 3, *Loops and Charts*, of your *LabVIEW User Manual* for more information on uninitialized shift registers.
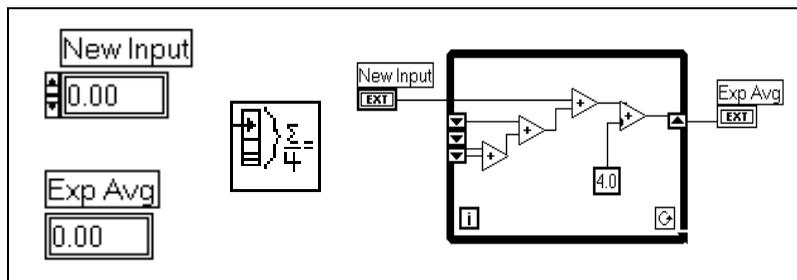
If you are a BridgeVIEW user, refer to Chapter 10, *Loops and Charts* of the *BridgeVIEW User Manual*.

Next, suppose a VI 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 the call for Channel 0 uses the average computed by the call for Channel 1. By making ExpAvg reentrant, each call can execute independently without danger of sharing the data.
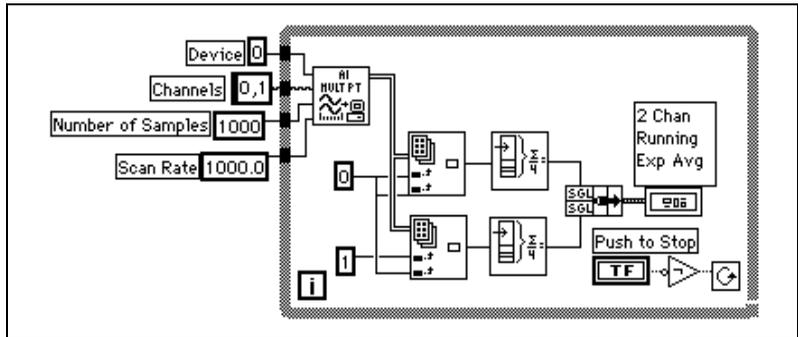
Synchronizing Access to Globals, Locals, and External Resources

Since the execution system can run several tasks in parallel, you must ensure global and local variables and resources are accessed in proper order.

Race Conditions

A *race condition* occurs when two or more pieces of code that execute in parallel are changing the value of the same shared resource, typically a global or local variable. The following two diagrams are an example of a race condition.
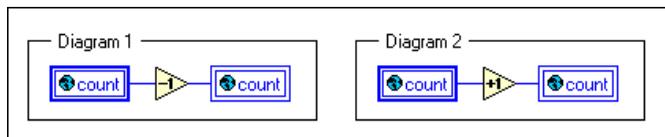
Diagram 1 decreases incrementally the value of count and Diagram 2 increases incrementally the value of count. Since there is no data dependency between the two diagrams, the execution of these two

diagrams might happen in this order (lets assume the initial value of count is 4).

Diagram 1: Read count (4)
Diagram 2: Read count (4)
Diagram 1: Increment 4 and write count (count is now 5)
Diagram 2: Decrement 4 and write count (count is now 3)

Although you might expect that running the two diagrams incrementally increases and then incrementally decreases count, in effect returning count back to its original value, the race condition can cause only one of the operations to have an effect.

Race conditions are prevented in one of several ways. The simplest one is to have only one place in the entire application through which a global variable is changed. For example, in the previous example all diagrams that incrementally increase or incrementally decrease the global variable count call a common subVI Change Count. They pass a Boolean parameter to indicate whether the global is incrementally increased or incrementally decreased.

Since a non-reentrant VI is executed only on behalf of one calling VI at a time and since Change Count is the only VI in the application that changes the global count, the race condition is avoided.

☞ **Note**   *In the single-threaded environment, it is possible to use a subroutine priority VI to read-modify-write a global variable without causing a race condition. This is because a subroutine priority VI does not share the execution thread with any other VIs. In the multithreaded environment, subroutine priority does not guarantee exclusive access to a global, since another VI, running in another thread, might be accessing the global at the same time.*

Functional Globals

Another way to avoid race conditions associated with global variables, is to not use global variables at all, but instead use VIs which use loops with uninitialized shift registers to hold global data. Such VIs are commonly referred to as Function Globals or LabVIEW 2-Style Globals (global variables did not exist in LabVIEW 2 and functional globals were used exclusively). A functional global usually has an action input parameter specifying which function the VI performs. The VI uses an uninitialized shift register in a While Loop to hold on to the result of the operation. The following is the diagram of a functional global implementing a simple count global. The actions in this example are **initialize**, **read**, **increment** and **decrement**.
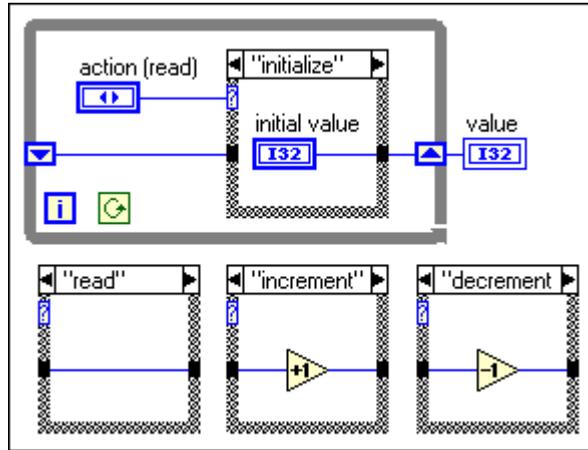
Every time the VI is called, the diagram in the loop is going to execute exactly once. Depending on the **action** parameter, the Case inside the loop either initializes, does not change, incrementally increases or incrementally decreases the value of the shift register.

Although functional globals can be used to implement simple global variables, as shown in the above example, they are especially useful when implementing more complex data structures, such as a stack or a queue buffer. Functional globals are also used to protect access to global resources, such as files, instruments and DAQ cards, that cannot be represented with a global variable.

Semaphores

Most synchronization problems dealing with global resources are easily solved with functional globals, since the functional global VI ensures the data it contains is changed on behalf of one caller at a time. One drawback of functional globals is that when the resource they hold is to be modified in a new way, the diagram of the global VI must be changed and a new action must be added. In some applications, where the usage of the global resources changes frequently, this might be inconvenient. In such instances, design your application to use a semaphore to protect access to the global resource.

A Semaphore, also known as a Mutex, is an object used to protect access to shared resources. The code where the shared resources are accessed is often referred to a critical section. In general, you only want one task at a time to be in a critical section protected by a common semaphore. It is possible for

semaphores to permit more than one task (up to a predefined limit) to be in a critical section.

A new semaphore is created with the **Create Semaphore** VI (found in the **Functions»Advanced»Synchronization»Semaphore** palette). The VI takes as an input the initial size of the semaphore. The size determines how many different tasks use the semaphore at the same time. Each time a task starts using the semaphore, the semaphore size is incrementally decreased. When the size reaches zero, any task trying to use the semaphore must wait until the semaphore is released by another task.
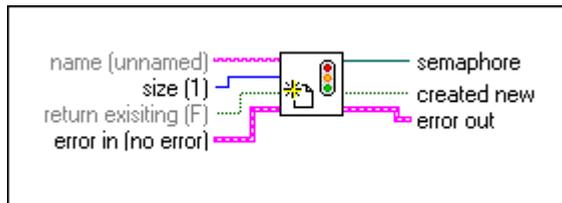
A task indicates it wants to use a semaphore by calling the **Acquire Semaphore** VI. When the size of the semaphore is greater than zero, the VI immediately returns timed out = FALSE and the task proceeds. If the semaphore size is zero, the task waits until the semaphore becomes available, or until the VI waits the specified amount of time. If the VI returns timed out = TRUE, it indicates the semaphore was not acquired, and the tasks do not execute the critical section. If the VI returns timed out = FALSE, it indicates the semaphore was successfully acquired.
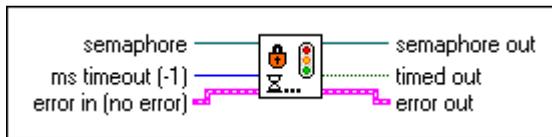
When a task successfully acquires a semaphore and is finished with its critical section, it releases the semaphore by calling the **Release Semaphore** VI. If, when the semaphore is released, there are any tasks waiting to acquire the semaphore, the first task is able to continue execution. Otherwise the size of the semaphore increases incrementally.
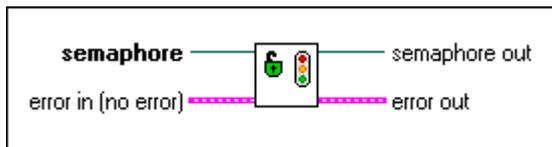
When a semaphore is no longer needed, it is destroyed by calling the
Destroy Semaphore VI. If there are any Acquire Semaphore VIs waiting on
the semaphore, they immediately return timed out = TRUE, as well as an
error.
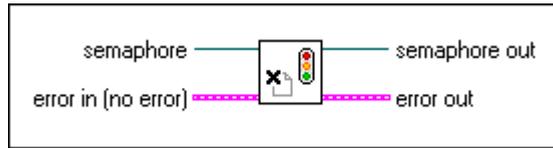
The following example shows how a semaphore is used to protect the
critical sections (increases incrementally or decreases incrementally a
global variable). The semaphore was created by passing size 1 to the
Create Semaphore VI.
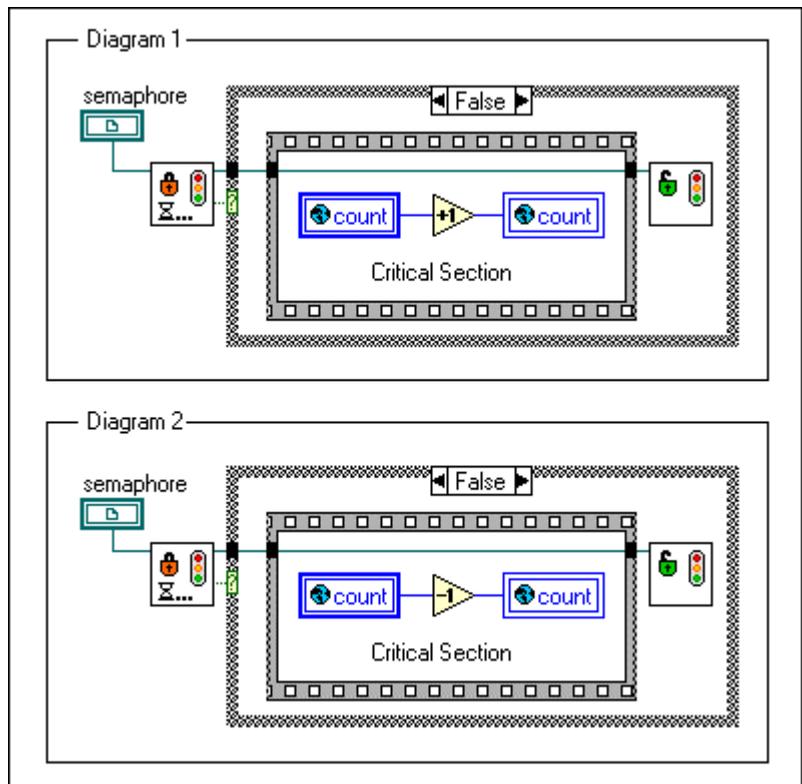
Each diagram wanting to execute a critical section must first call the
Acquire Semaphore VI. If the semaphore is busy (its size is zero) the VI
waits until the semaphore becomes available. When the Acquire

Semaphore VI returns timed out = FALSE, indicating the semaphore was acquired, the diagram starts executing the FALSE case. When the diagram is finished with its critical section (Sequence frame), the semaphore is released, permitting another waiting diagram to resume execution.

Other Synchronization Functions

In addition to the Semaphore VIs, there are a number of additional functions and VIs to synchronize parallel tasks. All of these functions are found in the **Functions»Advanced»Synchronization»Semaphore** palette.

Occurrence Functions — Occurrence functions give you a way to have one or more tasks wait until another task notifies them. You use the Generate Occurrence function to create an occurrence you pass to either a Wait on Occurrence or Set Occurrence function.

Notification VIs — Similar to Occurrences, but with the addition that when you send the notification you attach a message to it. If multiple VIs wait for the same notification, they all receive the message. Also, unlike occurrences, you can cancel a notification.

Queue VIs — Like Notifications, the Queue VIs let you send messages to other tasks. However, multiple messages can be queued up so that when a listener retrieves a message, he receives the oldest first. In addition, the Queue VIs are best used in situations with a single listener. If a listener removes an element from the queue, other listeners do not see that element.

A VI can add a message to a queue using `enqueue.vi`. To retrieve the oldest message from the queue, you use `dequeue.vi`. When you create a Queue using Open Queue Reference, you specify whether it has a bounded size or not; if you create a bounded queue and attempt to enqueue more data than the queue has room for, then the `Enqueue.vi` operation waits until a listener retrieves data using `dequeue.vi` before adding the additional element.

Rendezvous VIs — You can use the rendezvous VIs in situations where multiple parallel tasks must be synchronized at a common point. Each task that reaches the rendezvous waits until the specified number of tasks are waiting, at which point all tasks continue with execution.

General Suggestions for Using Execution Systems and Priorities

Following is a summary of some general suggestions about using the execution system options described in this chapter.

It is important to understand that while some of this seems fairly complicated, in most applications it is not necessary to use either priority levels or alternate execution system. The execution system automatically takes care of multitasking your VIs with each other.

By default, all VIs are set to run in the standard execution system at normal priority. In the multithreaded system, user interface activity is handled by a separate thread, so your VIs are insulated from user interface interaction. Even if you are using the single-threaded system, it tries to interleave handling of user interface interaction with execution of the VIs giving you similar results.

In general, the easiest and safest way to prioritize execution is to use Wait functions to slow down lower priority loops in your application. This is particularly useful in loops for user interface VIs, since delays of 100 to 200 milliseconds are not really noticeable to users.

If you use priorities, use them cautiously. If you design higher priority VIs that operate for a while, consider adding waits to those VIs in less time critical sections so they share time with lower priority tasks.

Be careful when manipulating globals, locals or other external resources which might be manipulated by other tasks. Use one of the synchronization techniques described earlier in this chapter to protect access to these resources.

The way priority use is designed in the single-threaded and multithreaded systems gives fairly similar results with the same VIs under both systems. There might be subtle timing differences, however. If you distribute VIs to customers running a different operating system, you might try your applications under those conditions. If you are using the multithreaded system, you can make it behave as a single-threaded version by turning off the **Run with Multiple Threads** preference in the **Performance** section of the **Preferences** dialog box.

27

Managing Your Applications

This chapter describes how to manage files in your G application.

File Arrangement Using VI Libraries

You can group multiple VI files by saving them as VI libraries (by convention, the names of these files end with the extension .llb). Or you can save VIs as individual files and group them within directories. For a comparison of the two methods, see the *Saving VIs* section of Chapter 2, *Editing VIs*.

If you use VI libraries, you might want to divide your application into multiple VI libraries. Put the high level VIs in one VI library, and set up other VI libraries to contain VIs separated by function. It is easier to manage your VIs if you organize your files in this manner.

Another reason to break your VIs into multiple VI libraries is because saving files takes longer as the size of your VI library increases. As you save a VI into a VI library, the library is copied to the temporary directory, the VI is compressed and saved into the VI library, and the VI library is copied back to the original directory. This copying process helps prevent accidental corruption of the original file, but it also takes longer to save VIs.

A good rule of thumb is to avoid creating VI libraries larger than 1 MB. Notice there is no real limit on the size of a library, and the time to load VIs from VI libraries does not noticeably change regardless of the size of the VI library. However, the time to save VIs into VI libraries does increase as the size of the library increases.

Another way to speed up the time it takes to save VIs, whether or not they are in VI libraries, is to make sure your temporary directory is on the same drive (partition, in UNIX) as your VIs. You can copy files faster if the source and destination are on the same drive (or partition).

If you use VI libraries, you can emphasize the top-level VIs by using **File»Edit VI Library…** from the menu option to mark certain VIs as top-level. G lists these files at the beginning of the file list in the file dialog box. There is no similar option for directories, but you can emphasize your top-level VIs by placing subVIs into subdirectories.

Backing Up Your Files

Accidents happen. Hard drives crash, and file systems become corrupted. It is best to make a backup copy of your VIs often, perhaps once a day. Copy them to another drive or to tape.

As an example of the problems that can occur, one user accidentally corrupted the VI library that contained several weeks of work. When prompted for a destination for a data file, this user had selected his VI library, at which point he wrote data over his own VI library. Fortunately he was able to recover most of the contents of his VI library, but only because he kept backups of his work.

Distributing VIs

When you are ready to distribute your VIs to other machines or to other users, consider whether you want to distribute editable block diagram source code, or whether you want to hide or remove the block diagram. VIs can be saved without their block diagrams. This reduces the size of the file on disk, and it prevents anyone from inadvertently changing the source code, but it also prevents the users from moving the VI to another platform or upgrading the VI to a future version of the development environment. If these features are important to you, you can consider password-protecting your diagrams. The source code is still available, but can only be viewed and modified if you enter a password.

Using the **Save with Options** dialog box, you can remove diagrams from your VIs. If you save your VIs without a diagram, the VI cannot be modified by other users. If you save your VIs without diagrams, be careful not to overwrite your original versions.

☞ **Note**   *VIs without diagrams cannot be converted to future versions of the application or to other platforms. During conversion, you need to recompile your VIs. Without a block diagram, a VI cannot be recompiled.*

Also consider the environment in which users run your VIs. Do you want end users to have a development system or a run-time application?

A run-time application is appropriate when you do not expect your users to make modifications to your VIs. A run-time application contains simplified menus, and users cannot edit VIs or view block diagrams.

Run-time applications are built using the LabVIEW Application Builder libraries. When you build a run-time application with these libraries, you must answer these three questions.

*   Do I want to embed a VI library in the application?
*   Do I want the application to have an **Open** menu item?
*   Do I want the application to have a **Quit** menu item?

If you choose to embed a VI library in the application, the library is combined with a run-time engine into a single file. When this file is launched, it automatically opens all top-level VIs in the library. If you do not embed a VI library, the application can open any VI when it is launched, assuming the VI was saved with a development system for that platform.

If you enable the **Open** menu item, you can use the application to open and run any VI in the file system, regardless of whether the application has an embedded VI library. By embedding a VI library, you can create a completely stand-alone application, one that prevents the user, or customer, from accessing the source VIs even if the user has the development system.

If you plan to ship multiple VI suites to the same customer, the single run-time application with the **Open** menu item enabled proves more efficient than separate embedded applications. This is true because each embedded application contains a copy of the run-time code, which is roughly 2 to 3 MB.

To create a run-time application, you must have the Application Builder libraries, which are sold separately. If you are a LabVIEW user, see the *LabVIEW Application Builder Release Notes* included with the Application Builder software for details on how to build an application.

☞ **Note**   *Because the BridgeVIEW execution system consists of several separate pieces, the Application Builder does not work for building BridgeVIEW run-time applications.*

Password-Protected VIs

When you distribute VIs to other people, you might want to prohibit others from viewing the source code you created in the block diagram, and you might want to prohibit others from editing the VI in any way.

You might save the VIs without the block diagrams, but doing so prevents the software from recompiling the VIs when you import them to another platform or a new version of G-language software.

To prohibit others from viewing block diagrams and editing VIs, you can protect VIs with passwords. In order to view the block diagram of a password-protected VI, or to edit that VI in any way, the user must enter the VI's correct password.

To password protect a VI, you select **Windows»Show VI Info…** or select **File»Save With Options**.

Through the **VI Information** dialog box, shown in the following illustration, you can add password protection to a single VI.
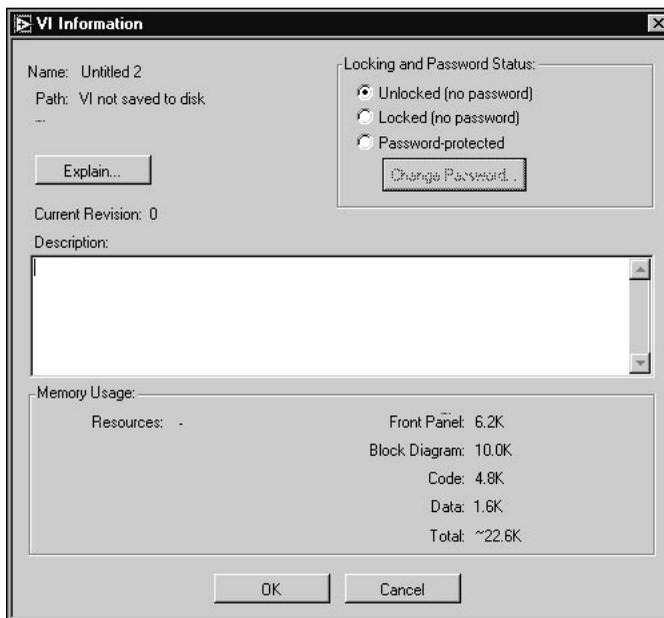
The following are descriptions of the three options available on the **VI Information** dialog box for protecting a VI.

Unlocked (no password)—Indicates that the VI is available for editing and is not password-protected. If you switch from Locked (no password) to this option, the VI immediately switches to the unlocked state. If the previous state was Password-Protected, and if the password for this VI is not already in memory, a dialog prompts you to enter the password.

Locked (no password)—VI is locked so that you cannot edit it, but you can run it and view its diagram. If you switch from Unlocked (no password), the transition proceeds immediately. However, if previously Password-Protected was selected, you must enter the password if it is not in memory.

Password-Protected—This state indicates that the VIs are password protected. Password protected VIs are similar to locked VIs, but you cannot access the diagram without entering a password. When you want to switch from this state, a dialog prompts you for a password if it is not in memory.

For more information on passwords, see the following section, *Save with Options Dialog Box.*

Save with Options Dialog Box

The **Save with Options** dialog box makes it simple for you to save an entire hierarchy of VIs for distribution. You can use this dialog box to selectively save an entire hierarchy, without the VIs in `vi.lib`, and save all external subroutines referenced by VIs in your hierarchy. The **Save with Options** dialog box is shown in the following illustration.
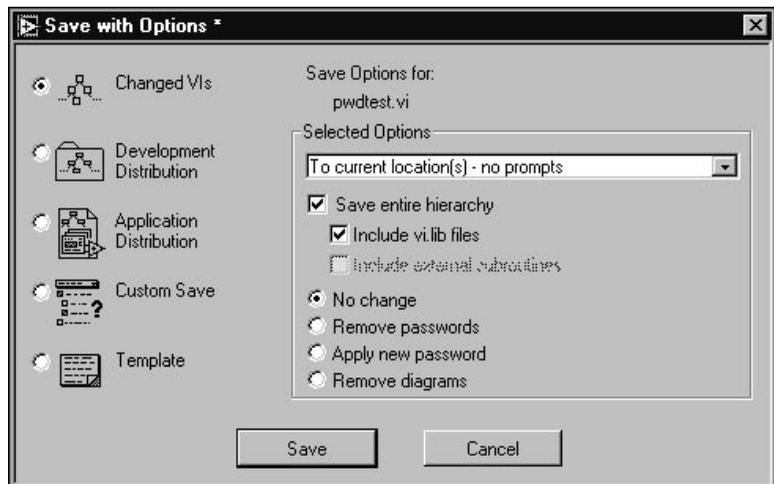
By selecting from the options on the left of the **Save with Options** dialog box, you can select from a set of predefined save options. As you select options, the area at the right shows the behavior of your current selection. You can customize the behavior by selecting specific options from the right section of the dialog box.

- **Changed VIs**—Saves any changes to the frontmost VI and its subVIs. This option is useful as a quick way to save changes.

- **Development Distribution**—Saves all non-`vi.lib` VIs, controls, and external subroutines to a single location (either a directory or library). This option is an easy way to save a hierarchy that is to be transported to another G development system, because the other development system has its own `vi.lib`.

- **Application Distribution**—Saves all VIs, controls, and external subroutines, including the ones in `vi.lib` to a single location and removes the diagrams from all of the VIs. If you want to create a run-time application with an embedded library, you can use this option. To actually create a run-time application, you must have the Application Builder libraries, which are sold separately.

- **Custom Save**—Selects the specific options you want from the **Selected Options** area of the dialog box, if none of the predefined options fit your needs.

- **Template**—Saves the VI as a VI template with the `.vif` file extension. When a template VI is loaded, it creates a copy of itself that must be saved under a new name.

The **Change Password...** button in the **Show Info** dialog box is not available if you do not have password protection on the VI. When you decide to change a password, a dialog box prompts you for a password if it is not in memory.

Through the **Save With Options** dialog box, shown in the previous illustration, you can add password protection to one VI or multiple VIs. The following descriptions detail the locked and password status in the Save with Options dialog box.

- **No change**—Selecting this option results in no change.

- **Remove passwords**—If selected, all the VIs you have not saved are saved without password protection. If you encounter password-protected VIs while saving, then for each password (not necessarily for each VI) associated with a VI or a group of VIs, a dialog prompts you for the password of each. To remove the passwords you must know the password for each VI or group of VIs.

- **Apply new passwords**—Selecting this option password protects VIs. If, while saving, you encounter VIs that are already password-protected, then for each password (not necessarily for each VI) associated with a VI or a group of VIs, a dialog prompts you for the password of each. To apply new passwords to VIs or groups of VIs you must know the password for each VI or group of VIs.

- **Remove diagrams**—This option saves your VIs without block diagrams so the VIs do not require as much storage. The VIs do not require as much disk space, but you cannot move them to another version of your software or to other platforms.

Designing Applications with Multiple Developers

If multiple developers work on the same project, spend some design time defining responsibilities to ensure the project works well. You might also consider using the Professional G Developers Toolkit, which provides source code control tools that assist in environments with multiple developers.

Start by considering the top-level design of your application and create an outline. Create *stub VIs* for the major components of your application. A stub VI is a prototype of a subVI. It has inputs and outputs, but is incomplete. It serves as a place holder for future VI development at which time you add functionality. Creating stub VIs for LabVIEW users is described in Chapter 28, *Program Design*, in the *LabVIEW User Manual*. BridgeVIEW users refer to Chapter 15, *Program Design* of the *BridgeVIEW User Manual*.

Once you verify the stub VIs provide the functionality you expect, you can determine which components individual developers work on. To simplify matters, you might place the major stub VIs in separate directories or VI libraries which can then be managed by different developers.

Keeping Master Copies

It is best to keep the *master* copies of your project VIs on a single computer. You can institute a check-in/check-out policy to ensure control of changes in your VIs. With this policy, a developer can check out a copy of the VIs, similar to checking a book out of the library. During that time, others do not touch the files the developer checks out. The developer then checks in the finished VIs after making any changes.

The Professional G Developers Toolkit contains Source Code Control (SCC) tools that can make this sharing of files and check in/out policy fairly easy. It also contains a utility you can use to compare VIs and view their differences. This can be extremely useful in verifying exactly which changes were made between versions of VIs.

VI History Window

In addition to its front panel and block diagram, a VI has a History window that displays the development history of the VI, including revision numbers. Each user who changes a VI can record any changes made in the history. This feature helps users keep track of changes to a VI as they are made. However, the VI history does not provide a method for comparing two VIs to detect differences (see the description of the Professional G Developers Toolkit for information on a utility for doing this). G saves the history as part of the VI. It consists of a series of comments entered by those users who make changes to the VI.

☞ **Note**   *Unless you select the* **Record comments generated by LabVIEW** *option in either the* **VI Setup»Documentation** *or the* **Edit»Preferences»History** *dialog box, the comments in a VI history are not created automatically. Anyone who makes changes to the VI must type in the information and keep the history up to date.*

The History window is available when you are editing the VI by selecting **Windows»Show History** or pressing the key equivalent, `<Ctrl-y>` **(Windows)**; `<command-y>` **(Macintosh)**; `<meta-y>` **(Sun)**; or `<Alt-y>` **(HP-UX)**. The following illustration shows an example History window.

Widget Calculator.vi History

User Name: johann   **Next Revision:** 4

Comment:

Added code for the new widgets.

▼**History**   [Reset...]   [**Add**]

rev. 3 Mon, Mar 14, 1994 11:52:23 AM johann
Added descriptions to all VI's in the Widget library.
rev. 2 Mon, Mar 14, 1994 11:51:40 AM johann
Added support for multiple foos.
rev. 1 Mon, Mar 14, 1994 11:50:29 AM johann
Converted all WidgetVI's to LV 3.1.
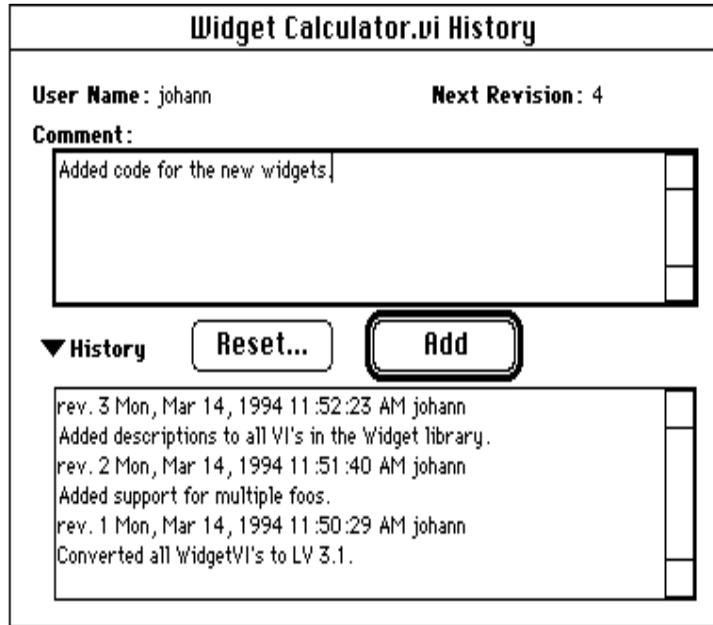
As you edit the VI, you can type a description of important changes into the Comment box near the top of the window. When you finish making changes, add the comment to the history by clicking the **Add** button. If there is a comment in the Comment box when you save the VI, it is added to the history automatically. Once you add a comment, it is a permanent part of the history. You cannot rewrite the history, so be sure to check your comments before adding them.

The box at the bottom of the window displays the history of the VI. The dialog box shows a header for each comment in the history that includes the revision number of the VI, the date and time, and the name of the person who made the change. You can see three comments added to this history, and each is displayed with a header. Because the preceding illustration is only an example, the comments are short; you can make them as long as you want when editing your own VIs.

The History window is fairly large with the history showing, but you can hide the history by clicking the **History** arrow (the small black triangle to the left of the word). This makes the window small enough that it stays out of your way while you are editing the VI. When you want to see the history

again, click the same arrow and the history reappears. An example is shown in the following illustration.

```
┌──────────────────────────────────────────────────────┐
│              Widget Calculator.vi History            │
├──────────────────────────────────────────────────────┤
│                                                      │
│   User Name: johann              Next Revision: 4    │
│   Comment:                                           │
│   ┌────────────────────────────────────────────┐ ┌─┐ │
│   │ Added code for the new widgets.            │ │ │ │
│   │                                            │ └─┘ │
│   │                                            │ ┌─┐ │
│   │                                            │ │ │ │
│   │                                            │ └─┘ │
│   └────────────────────────────────────────────┘     │
│                                                      │
│   ▶ History    ┌──────────┐  ┌──────────────┐        │
│                │ Reset... │  │     Add      │        │
│                └──────────┘  └──────────────┘        │
└──────────────────────────────────────────────────────┘
```
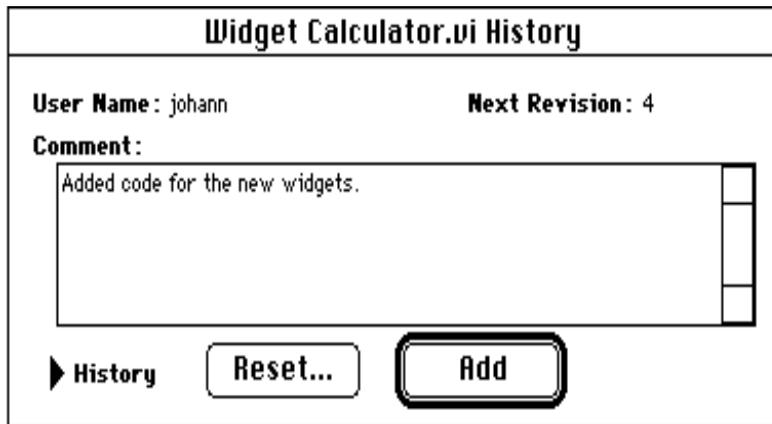
You can also change the size of the history and comment boxes by resizing the window.

Revision Numbers

The revision number is also a part of the history of the VI and is an easy way to see if the VI changed (and how it changed, if you commented on your changes). The revision number starts at zero and increases incrementally every time you save the VI. The current revision number (the one on disk) is displayed in the **Get Info...** dialog box. It is also displayed in the titlebar of the VI if the option to do so is checked in the dialog box of **Edit»Preferences»History**.

The number displayed in the History window or VI window is the next revision number; that is, the number saved on disk if you save the VI. It is the current revision number plus one. When you add a comment to the history, the next revision number is included in the header of the comment. For example, if you click the **Add** button in the example, your comment adds to the history with the revision number 4. If you save the VI, the current revision number is 4, and the comment that applies to the changes you just made is labeled with the same number.

☞ **Note**   *The revision number is not increased incrementally when you save the VI if the only changes you made were changes to the VI History.*

In many cases, there are gaps in revision numbers between comments because you saved the VI without entering a comment. This is not a problem. In fact, the revision number is useful because it is independent. Suppose you receive a copy of a VI from another user. Later, you want to know if the other user updated the VI since you received your copy. You can easily tell by looking at the revision numbers, even if the other user made a change without adding comments.

If the revision number only changes when you add a comment, it is a comment number, not a revision number. The revision number is even more effective when combined with the history. If users add comments to the history, you can tell what changes were made since you last received a copy.

Resetting History Information

Under the comment box there is a button labeled **Reset**. Pressing **Reset** erases the history and optionally resets the revision number to zero. This is useful when you copy a VI and want to start the new VI with a clean slate (no history).

Because the history is strictly a developer tool, the history is removed automatically when you remove the block diagram of a VI. The History window is not available in the run-time version, but the revision number is available in the VI Information dialog box even for VIs with their block diagrams removed. You can remove the revision number using the **Reset** button to reset the history of your VI before you remove the block diagram.

Printing History Information

When you print a VI, you can include the history and revision number by selecting **Complete Documentation** from the Print Documentation dialog box. If you want to change the format of the printout to include only the history information, you can use the **Custom Print Settings** button to indicate exactly what you want to print. Additionally, you can export the history information to a file by selecting **Save Text Info** from the Print Documentation dialog box.

Setting Related VI Setup and Preference Dialog Box Options

The **VI Setup** and the **Preferences** dialog boxes both have options you can use to indicate how VI history behaves.

You can use the **VI Setup** options to indicate for a given VI if and when history information is automatically recorded. See the *Documentation Options* section of Chapter 6, *Setting up VIs and SubVIs*, for information on these settings.

You can use the **Preferences** dialog box options to indicate what the default history settings are for new VIs. You can also use this dialog box to indicate a user name entered in history information. See the *History Preferences* section of Chapter 7, *Customizing Your Environment*, for information on these settings.

28

Performance Issues

This chapter is in three sections. The first section describes the Performance Profiler, a feature that shows you data about execution time of your VIs and monitors single-threaded, multithreaded, and multiprocessor applications. The second section describes factors affecting run-time speed. The third section describes factors affecting memory usage.

VI Performance Profiling

The VI Performance Profiler is a powerful tool for determining where your application is spending its time, as well as how it is using memory. This information is invaluable in finding the hot spots of your application so that you can make the best use of the VIs that take up the most time. It has an interactive tabular display of time and memory usage for each VI in your system. Each row of the table contains information for a specific VI. The time spent by each VI is broken down into several categories as well as summarized in a few statistics. The Performance Profile window calculates the minimum, maximum, and average time spent per run of a VI.

You can use the interactive tabular display to view all or parts of this information, sort it by different categories, and look at the performance data of subVIs when called from a specific VI.

Select **Project»Show Profile Window** to access the Profile window. The following illustration shows an example of the window already in use.

	VI Time	Sub VIs Time	Total Time	Avg Bytes
Frequency Response.vi	**3234.7**	**1091.6**	**4326.2**	**10.99k**
... Demo Tek FG 5010.vi	430.6	140.2	570.8	5.44k
... Demo Fluke 8840A.vi	350.5	170.2	520.7	2.85k
Cut Length.vi	2553.7	0.0	2553.7	2.90k
Situation at Time t.vi	1932.8	4526.5	6459.3	132.46k
Simulation of Tomography	1422.0	6469.3	7891.3	742.63k
Interior of a Circle.vi	691.0	0.0	691.0	1.86k
Demo Tek FG 5010.vi	430.6	140.2	570.8	5.44k
... Demo Send.vi	130.2	0.0	130.2	2.47k
... Frequency.gbl : Frequency	0.0	0.0	0.0	0.00k

There are several things to notice about the window. First, the collection of memory usage information is optional. This is because the collection process can add a significant amount of overhead to the running time of your VIs. You must choose whether to collect this data before starting the Profiler by checking the **Profile Memory Usage** checkbox appropriately. This checkbox cannot be changed once a profiling session is in progress. The following buttons are available on the Profile window.
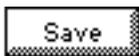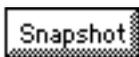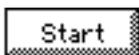
- **Start**—Enables the collection of performance data. It is best to start a profiling session while your application is not running. This way, you can ensure that you measure only complete runs of VIs, and not partial runs.

- **Snapshot**—Views the data currently available. This gathers data from all the VIs in memory and displays it in the tabular display.

- **Save**—Saves the currently displayed data to disk as a tab-delimited text spreadsheet file. This data can then be viewed in a spreadsheet program or by VIs.

Viewing the Results

You can choose to display only parts of the information in the table. Some basic data is always visible, but you can choose to display the statistics, details, and (if enabled) memory usage by checking or unchecking the appropriate checkboxes in the Profile window.

Performance information also is displayed for Global VIs. However, this information sometimes requires a slightly different interpretation, as described in the category-specific sections below.

Performance data for subVIs when called from a specific VI can be viewed by double-clicking the name of that VI in the tabular display. When you do this, new rows appear directly below the name of the VI and contain performance data for each of its subVIs. When you double-click the name of a Global VI, new rows appear for each of the individual controls on its front panel.

You can sort the rows of data in the tabular display by clicking in the desired column header. The current sort column is indicated by a bold header title.

Timings of VIs do not necessarily correspond to the amount of elapsed time that it takes for a VI to complete. This is because a multithreaded execution system can interleave the execution of two or more VIs. Also, there is a certain amount of overhead taken up not attributed to any VI, such as the amount of time taken by a user to respond to a dialog box, or time spent in a Wait function on a block diagram, or time spent to check for mouse clicks.

The basic information that is always visible in the first three columns of the performance data consists of the following items.

- **VI Time**—Total time spent actually executing the code of this VI and displaying its data, as well as time spent by the user interacting with its front panel controls (if any). This summary is broken down into sub-categories in the **Timing Details** described below. For Global VIs, this time is the total amount of time spent copying data to or from all of its controls. To see timing information on individual controls in a Global VI, you can double-click the name of the Global VI.

- **SubVIs Time**—Total time spent by all subVIs of this VI. This is the sum of the VI time (described above) for all callees of this VI, as well as their callees, etc.

- **Total Time**—Sum of the above two categories, calculating the total amount of time.

Timing Information

When the **Timing Statistics** checkbox is checked, the following columns become visible in the tabular display.

• **# Runs**—Number of times that this VI completed a run. For Global VIs, this time is the total number of times any of its controls were accessed.

• **Average**—Average amount of time spent by this VI per run. This is simply the VI time divided by the number of runs.

• **Shortest**—Minimum amount of time the VI spent in a run.

• **Longest**—Maximum amount of time the VI spent in a run.

When the **Timing Details** checkbox is checked, you can view a breakdown of several timing categories that sum up the time spent by the VI. For VIs that have a great deal of user interface, these categories can help you see what operations take the most time.

• **Diagram**—Time spent only executing the code generated for the diagram of this VI.

• **Display**—Time spent updating front panel controls of this VI with new values from the diagram.

• **Draw**—Time spent drawing the front panel of this VI. Draw time includes the following.

– The time it takes simply to draw a front panel when its window just has been opened, or when it is revealed after being obscured by another window.

– The time that is conceptually Display time, caused by new values coming in from the diagram but occurring because the control is transparent and/or overlapped. These controls must invalidate their area of the screen when they receive new data from the diagram so everything in that area can redraw in the correct order. Other controls can draw immediately on the front panel when they receive new data from the diagram. More overhead is involved in invalidating and redrawing—most (but not all) of which shows up in the Draw timings.

• **Tracking**—Time spent tracking the mouse while the user was interacting with the front panel of this VI. This can be significant for some kinds of operations, such as zooming in or out of a graph, selecting items from a pop-up menu, or selecting and typing text in a control.

- **Locals**—Time spent copying data to or from local variables on the diagram. Experience with users shows this time can sometimes be significant, especially when it involves large, complex data.

Memory Information

When the **Memory Usage** checkbox is checked (remember this is only available if the **Profile Memory Usage** checkbox was selected before you began the profiling session), you can view information about how your VIs are using memory. These values are a measure of the memory used by the data space for the VI and do not include the support data structures necessary for all VIs. The data space for the VI contains not just the data explicitly being used by front panel controls, but also temporary buffers implicitly created by the compiler.

The memory sizes are measured at the conclusion of the run of a VI and might not reflect its exact, total usage. For instance, if a VI creates large arrays during its run but reduces their size before the VI finishes, the sizes displayed do not reflect the intermediate larger sizes.

Two sets of data are displayed in this section—data related to the number of bytes used, and data related to the number of blocks used. A block is a contiguous segment of memory used to store a single piece of data. For example, an array of integers might be multiple bytes in length, but it occupies only one block. The execution system uses independent blocks of memory for arrays, strings, paths, and pictures (from the Picture Control Toolkit). Large numbers of blocks in the memory heap of your application can cause an overall degradation of performance (not just execution).

The categories of Memory Usage as follows.

- **Average Bytes**—Average number of bytes used by the data space of this VI per run.

- **Min Bytes**—Minimum number of bytes used by the data space of this VI for an individual run.

- **Max Bytes**—Maximum number of bytes used by the data space of this VI for an individual run.

- **Average Blocks**—Average number of blocks used by the data space of this VI per run.

- **Min Blocks**—Minimum number of blocks used by the data space of this VI for an individual run.

- **Max Blocks**—Maximum number of blocks used by the data space of this VI for an individual run.

VI Execution Speed

Although the compiler produces code that generally executes very quickly, when working on time critical applications you might want to do all you can to obtain the best performance out of your VIs. This section discusses factors that affect execution speed and suggests some programming techniques to help you obtain the best performance possible.

Examine the following items to determine the causes of slow performance.

- Input/Output (files, GPIB, data acquisition, networking)
- Screen Display (large controls, overlapping controls, too many displays)
- Memory Management (inefficient usage of arrays and strings, inefficient data structures)

Other factors, such as execution overhead and subVI call overhead can have an effect, but these are usually minimal and not the most critical source of slow execution.

Input/Output

Input/Output calls generally incur a large amount of overhead. They often take an order of magnitude more time than the time it takes to perform a computational operation. For example, a simple serial port read operation might have an associated overhead of several milliseconds. This amount of overhead is true for any application that uses serial ports. The reason for this overhead is an I/O call involves transferring information through several layers of an operating system.

The best method for addressing too much overhead is to minimize the number of I/O calls you make. Your performance improves if you can structure your application so that you transfer a large amount of data with each call, instead of making multiple I/O calls using smaller amounts of data.

For example, if you are creating a data acquisition (NI-DAQ) VI, you have a couple of options for reading data. You can use a single-point data transfer function such as the AI Sample Channel VI, or you can use a multi-point data transfer function such as the AI Acquire Waveform VI. If you must acquire 100 points, use the AI Sample Channel VI in a loop with a Wait function to establish the timing. Or you can use the AI Acquire Waveform VI with an input indicating you want 100 points.

You can produce much higher and more accurate data sampling rates by using the AI Acquire Waveform VI, because it uses hardware timers to manage the data sampling. In addition, overhead for the AI Acquire Waveform VI is roughly the same as the overhead for a single call to the AI Sample Channel VI, even though it is transferring much more data.

Screen Display

Updating controls on a front panel frequently can be one of the most time expensive operations in an application. This is especially true if you use some of the more complicated displays, such as graphs and charts.

This overhead is minimized to a certain extent because most of the controls are intelligent. They do not redraw when they receive new data if the new data is the same as the old data. Graphs and charts are exceptions to this rule.

If redraw rate becomes a problem, the best solutions are to reduce the number of controls you use, and keep the displays as simple as possible. In the case of graphs and charts, you can turn off autoscaling, scale markers, and grids to speed up displays.

If you have controls overlapped with other objects, their display rate is cut down significantly. The reason for this is that if a control is partially obscured, more work must be done to redraw that area of the screen. Unless you have the **Smooth Updates** preference on, you might see more flicker when controls are overlapped.

As with other kinds of I/O, there is a certain amount of fixed overhead in the display of a control. You can pass multiple points to an indicator at one time using some controls, such as charts. You can minimize the number of chart updates you make by passing more data to the chart each time. You can see much higher data display rates if you collect your chart data into arrays to display multiple points at a time, instead of displaying each point as it comes in.

When you design subVIs whose front panels are closed during execution, do not be concerned about display overhead. If the front panel is closed, you do not have the drawing overhead for controls, so graphs are no more expensive than arrays.

Controls and indicators have a synchronous display pop-up that, in multithreaded systems, controls in multithreaded systems whether or not updates can be deferred. In single-threaded execution, this item has no effect. If you turn this item on or off within VIs in the single-threaded

version, however, those changes affect the way updates behave if you load
those VIs into a multithreaded system.

By default, this item is off, which means that when the execution system
passes data to front panel controls and indicators, it can pass the data
to the control or indicator and then immediately continue execution.
At some point thereafter, the user interface system notices that the control
or indicator needs to be updated, and it then redraws to show the new data.
If the execution system attempts to update the control multiple times in
rapid succession, you might not see some of the intervening updates.

In many applications, this can significantly speed up execution without
affecting what the user sees. For example, even with this item off, you
can update a Boolean hundreds of times in a second, with more updates
than the human eye can actually discern. Asynchronous displays permits
the execution system to spend more time executing VIs, with updates
automatically reduced to a slower rate by the user interface thread.

If you want to avoid this type of display, you can turn the synchronous
display on.

Other Issues

Parallel Diagrams

When you have multiple diagrams running in parallel, the execution
system switches between them periodically. If some of these loops are
less important than others, use the Wait function to ensure the less
important loops use less time.

For example, consider the following diagram.
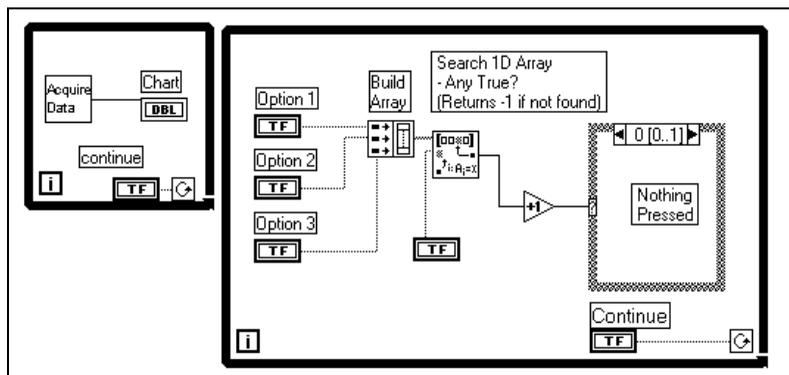
There are two loops in parallel. One of the loops is acquiring data, and needs to execute as frequently as possible. The other loop is monitoring user input. The loops receive equal time because of the way this program is structured. The loop monitoring the user's action has a chance to run several times a second.

In practice, it is usually acceptable if the loop monitoring the button executes only once every half second, or even less often. By calling the Wait (ms) function in the user interface loop, you allot significantly more time to the other loop.
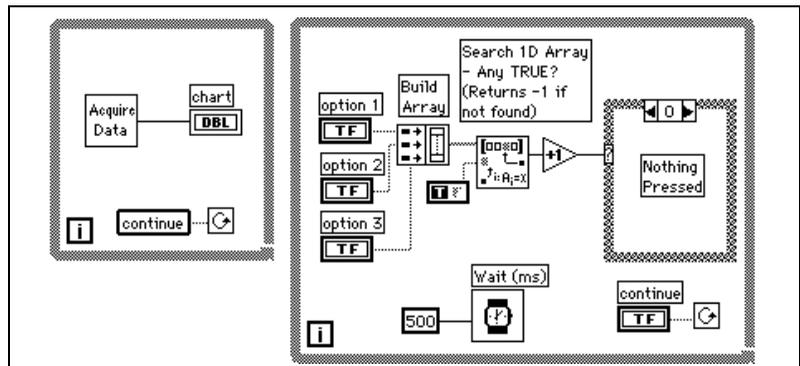
SubVI Overhead

When you call a subVI, there is a certain amount of overhead associated with the call. This overhead is fairly small (on the order of tens of microseconds), especially in comparison to I/O overhead and display overhead, which can range from milliseconds to tens of milliseconds.

However, this overhead can add up in some cases. For example, if you call a subVI 10,000 times in a loop, this overhead might take up a significant amount of time. In this case, you might want to consider whether the loop can be embedded in the subVI.

Another option you might consider is turning certain subVIs into subroutines (using the VI Setup **Priority** item). When a VI is marked as a subroutine, the execution system minimizes the overhead to call a subVI. There are a few trade-offs, however. Subroutines cannot display front panel data (G does not copy data from or to the front panel controls of subroutines), they cannot contain timing or dialog box functions, and they do not multitask with other VIs. Subroutines are short, frequently executed tasks and are generally most appropriate when used with VIs that do not

require user interaction. See Chapter 26, *Understanding the G Execution System*, for more information.

Unnecessary Computation in Loops

Avoid putting calculations in loops if the calculation produces the same value for every iteration. Instead, move the calculation out of the loop and pass the result into the loop.

For example, examine the following diagram.
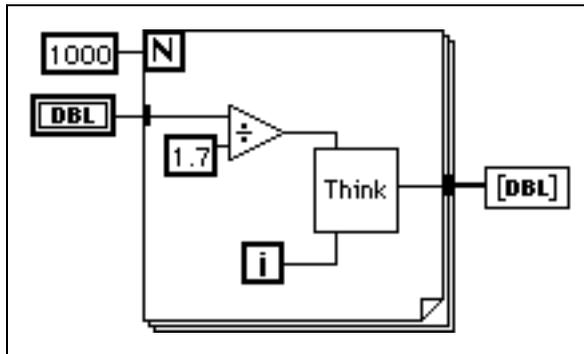
The result of the division is the same every time through the loop; therefore you can increase performance by moving the division out of the loop, as shown in the following illustration.
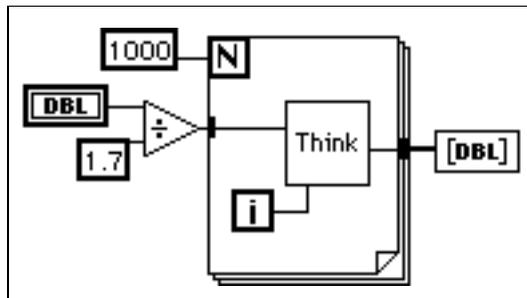
Now, refer to the diagram in the following illustration.
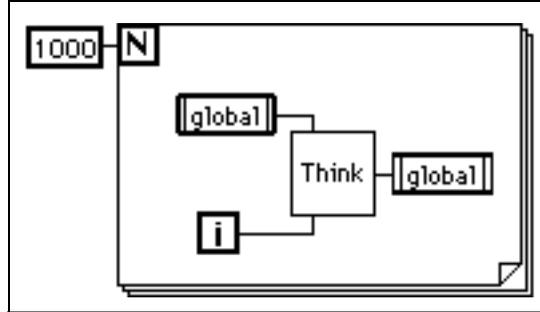
If you know the value of the global variable is not going to be changed by another concurrent diagram or VI during this loop, this diagram wastes time by reading from the global variable and writing to the global every time through the loop.

If you do not require the global variable to be read from or written to by another diagram during this loop, you might use the following diagram instead.
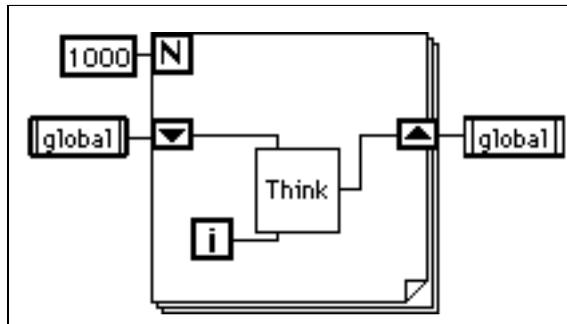
Notice that the shift registers must pass the new value from the subVI to the next iteration of the loop. The following diagram is a common mistake some beginning users make. Since there is no shift register, the results from the subVI never pass back to the subVI as its new input value.
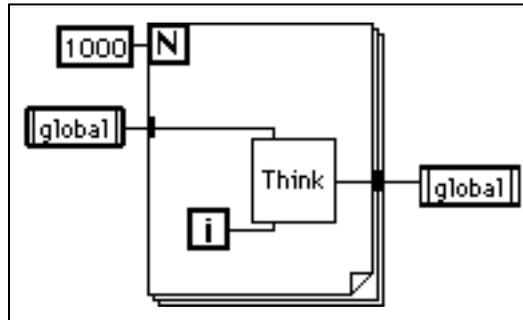
VI Memory Usage

G handles many of the details which you normally worry about in a conventional programming language. One of the main challenges of a conventional language is memory usage. In a conventional language, you, the programmer, have to take care of allocating memory before you use it, and deallocating it when you finish. You also must be particularly careful not to accidentally write past the end of the memory you allocated in the first place. Failure to allocate memory or to allocate enough memory is one of the biggest mistakes programmers make in conventional, text-based languages. Inadequate memory allocation is also a difficult problem to debug.

The dataflow paradigm for G removes much of the difficulty of managing memory. In G, you do not allocate variables, nor assign values to and from them. Instead, you create a diagram with connections representing the transition of data.

Functions that generate data take care of allocating the storage for that data. When data is no longer being used, the associated memory is deallocated. When you add new information to an array or a string, enough memory is allocated automatically to manage the new information.

This automatic memory handling is one of the chief benefits of G. However, because it is automatic, you have less control of when it happens. If your program works with large sets of data, it is important to have some understanding of when memory allocation takes place. An understanding of the principles involved can result in programs with significantly smaller memory requirements. Also, an understanding of how to minimize memory usage can also help to increase VI execution speeds because memory allocation and copying data can take a considerable amount of time.

Virtual Memory

If you have a machine with a limited amount of memory, you might want to consider using virtual memory to increase the amount of memory available for applications. Virtual memory is a capability of your operating system by which it uses available disk space for RAM storage. If you allocate a large amount of virtual memory, applications perceive this as memory generally available for storage.

For applications, it does not matter if your memory is real RAM or virtual memory. The operating system hides the fact that the memory is virtual. The main difference is speed. With virtual memory, you occasionally might notice more sluggish performance, when memory is swapped to and from the disk by the operating system. Virtual memory can help run larger applications, but is probably not appropriate for applications that have critical time constraints.

Macintosh Memory

When you launch an application on the Macintosh, the system allocates a single block of memory for it, from which all memory is allocated. When you load VIs, components of those are loaded into that memory. Similarly, when you run a VI, all the memory it manipulates is allocated out of that single block of memory.

You configure the amount of memory the system allocates at launch time using the **File»Get Info** command from the Finder. Keep in mind if your application runs out of memory, it cannot increase the size of this memory pool. Therefore, set up this parameter to be as large as is practical. If you have a 16 MB machine, consider the applications you want to run in parallel. If you do not plan to run any other applications, set the memory preference to be as large as possible.

VI Component Memory Management

A VI has four major components.

- Front panel

- Block diagram

- Code (diagram compiled to machine code)

- Data (control and indicator values, default data, diagram constant data, and so on.)

When a VI loads, the front panel, the code (if it matches the platform), and the data for the VI are loaded into memory. If the VI needs to be recompiled because of a change in platform or a change in the interface to a subVI, then the diagram is loaded into memory as well.

The VI also loads the code and data space of its subVIs into memory. Under certain circumstances, the front panel of some subVIs might be loaded into memory as well. This can occur, for example, if the subVI uses attribute nodes, because attribute nodes manipulate state information for front panel controls. More information on subVIs and their panels is discussed later in this chapter.

An important point in the organization of VI components is that you generally do not use much more memory when you convert a section of your VI into a subVI. If you create a single, huge VI with no subVIs, you end up with the front panel, code, and data for that top-level VI in memory. However, if you break the VI into subVIs, the code for the top-level VI is smaller, and the code and data of the subVIs reside in memory. In some cases, you might see lower run-time memory usage. This idea is explained later in this chapter in the section, *Monitoring Memory Usage*.

You also might find massive VIs take longer to edit. You do not see this problem as much if you break your VI into subVIs, because the editor can handle smaller VIs more efficiently. Also, a more hierarchical VI organization is generally easier to maintain and read.

☞ **Note**   *If the front panel or block diagram of a given VI is much larger than a screen, you might want to break it into subVIs to make it more accessible.*

Dataflow Programming and Data Buffers

In dataflow programming, you generally do not use variables. Dataflow models usually describe nodes as consuming data inputs and producing data outputs. A literal implementation of this model produces applications that can use very large amounts of memory and have sluggish performance. Every function produces a copy of data for every destination to which an output is passed. The G compiler improves on this implementation by attempting to determine when memory can be reused and by looking at the destinations of an output to determine whether it is necessary to make copies for each individual terminal.

For example, in a more traditional approach to the compiler, the following diagram uses two blocks of data memory, one for the input and one for the output.
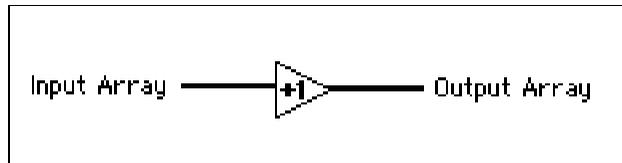
The input array and the output array contain the same number of elements, and the data type for both arrays is the same. Think of the incoming array as a buffer of data. Instead of creating a new buffer for the output, the compiler reuses the input buffer. This saves memory and also results in faster execution, because no memory allocation needs to take place at run time.

The compiler cannot reuse memory buffers in all cases, however, as shown
in the following illustration.
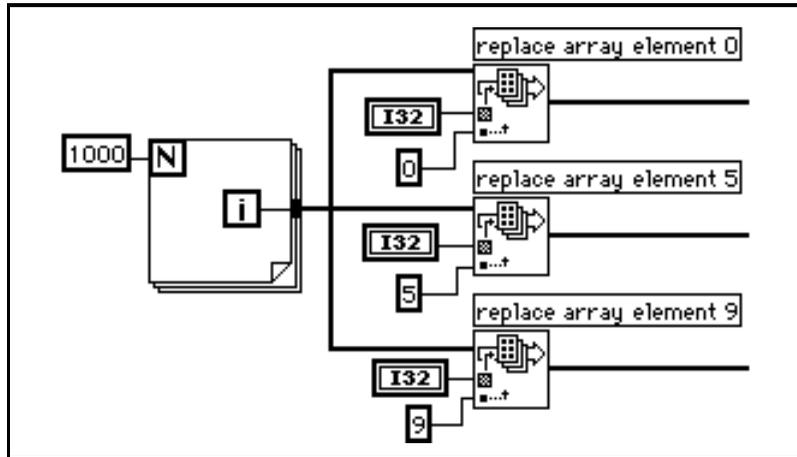
A single source of data is being passed to multiple destinations.
The destinations want to modify the data to produce resulting arrays. In this
case, the compiler creates new data buffers for two of the functions and
copies the array data into the buffers. Thus, one of the functions reuses the
input array, and the others do not. This diagram uses about 12 KB (4 KB for
the original array and 4 KB for each of the extra two data buffers).

Now, examine the following diagram.
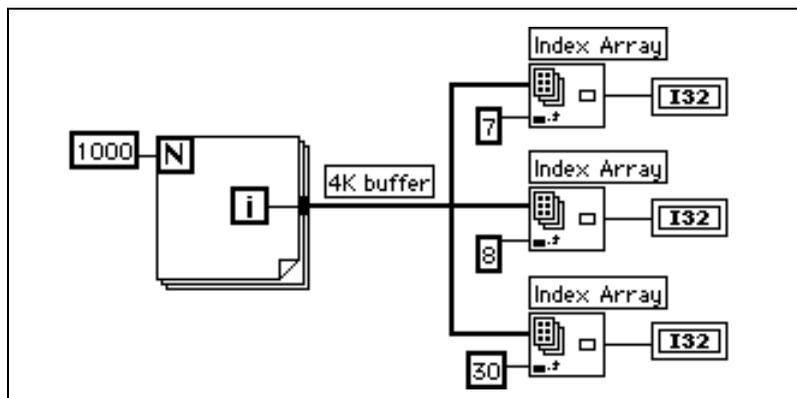
As before, the input branches to three functions are the same. However, in
this case none of them need to modify the data. If you pass data to multiple
locations, all of which read the data without modifying it, G does not make
a copy of the data. This diagram uses up about 4 KB of data.

Finally, consider the following diagram.
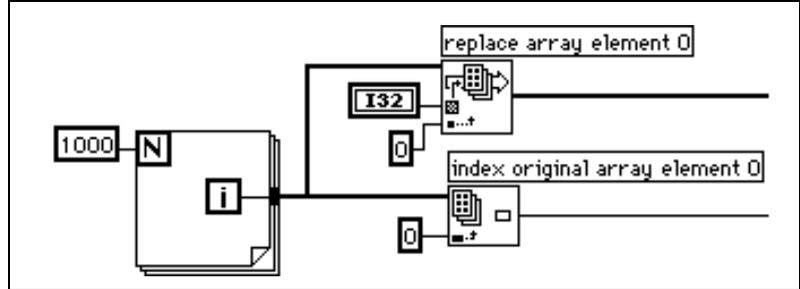
In this case, the input branches to two functions, one of which wants to modify the data. There is no dependency between the two functions. Therefore, you can predict at least one copy needs to be made so the replace array element function can safely modify the data. In this case, however, the compiler schedules the execution of the functions in such a way that the function that wants to read the data executes first, and the function that wants to modify the data executes last. This way, the Replace Array Element function reuses the incoming array buffer without generating a duplicate array. If the ordering of the nodes is important, make the ordering explicit by using either a sequence or an output of one node for the input of another.

In practice, the analysis of diagrams by the compiler is not perfect. In some cases, the compiler might not be able to determine the optimal method for reusing diagram memory.

Monitoring Memory Usage

There are a couple of methods for determining memory usage.

If you select **Help»About LabVIEW…** (if you are a LabVIEW user) or **Help»About BridgeVIEW…** (if you are a BridgeVIEW user), you see statistics that summarize the total amount of memory used by your application. This memory includes memory for VIs as well as memory the application uses. You can check this amount before and after execution of a set of VIs to obtain a rough idea of how much memory the VIs are using.

You can obtain a view of the dynamic usage of memory by your VIs with the Performance Profiler. It keeps statistics on the minimum, maximum, and average number of byes and blocks used by each VI per run. See the section *VI Performance Profiling*, at the beginning of this chapter, for more details.

As shown in the following illustration, you can use **Windows»
Show VI Info…** to have a breakdown of the memory usage for a given VI.
The left column of this information summarizes disk usage, and the right
column summarizes how much RAM currently is being used for various
components of the VI. Notice these statistics do not include memory
usage of subVIs.
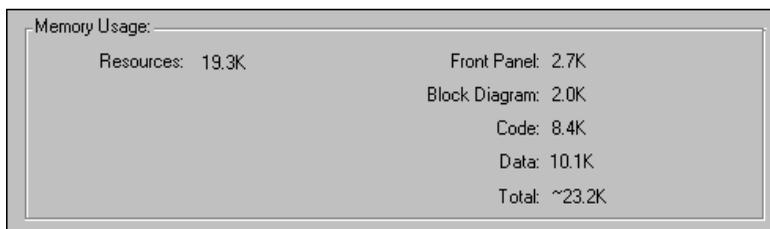
A fourth method for determining memory usage is to use a VI called the
Memory Monitor VI (in memmon.llb inside of the examples directory).
This VI uses the VI Server functions to determine memory usage for all
VIs in memory.

Rules for Better Memory Usage

The main point of the previous discussion is that the compiler attempts
to reuse memory intelligently. The rules for when the compiler can reuse
memory and when it cannot are fairly complex and are discussed at the
end of this chapter. In practice, the following rules can help you to create
VIs that use memory efficiently.

- Breaking a VI into subVIs generally does not hurt your memory usage.
  In fact, in many cases, memory usage improves, because the execution
  system can reclaim subVI data memory when the subVI is not
  executing.

- Do not worry too much about copies of scalar values; it takes a lot of
  scalars to negatively effect memory usage.

- Do not overuse global and local variables when working with arrays or
  strings; reading a global or local variable causes a copy of the data of
  the variable to be generated.

- On open front panels, do not display large arrays and strings unless it
  is necessary. Indicators on open front panels retain a copy of the data
  they display.

- If the front panel of a subVI is not going to be displayed, do not leave unused attribute nodes on the subVI. attribute nodes cause the front panel of a subVI to remain in memory, which can cause unnecessary memory usage.

- Do not use the suspend data range feature on time/memory critical VIs. The front panel for the subVI needs to be loaded to check range checking, and extra copies of data are made for the subVIs controls and indicators.

- In designing diagrams, watch for places where the size of an input is different from the size of an output. For example, if you see places where you are increasing the size of an array or string frequently using the Build Array or Concatenate Strings functions, you are generating copies of data.

- Use consistent data types for arrays and watch for coercion dots when passing data to subVIs and functions; when you change data types, the execution system makes a copy of the data.

- Do not use complicated, hierarchical data types (for example, arrays of clusters containing large arrays or strings, or clusters containing large arrays or strings). You might end up using more memory, and performance suffers. See the section *Developing Efficient Data Structures* later in this chapter for more details and suggestions for tactics in designing your data types.

Memory Issues in Front Panels

When a front panel is open, controls and indicators keep their own, private copy of the data they display.

The following illustration shows the increment function, with the addition of front panel controls and indicators.
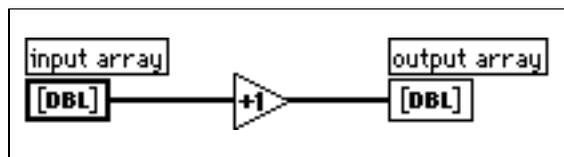
When you run the VI, the data of the front panel control is passed to the diagram. The increment function reuses the input buffer. The indicator then makes a copy of the data for display purposes. Thus, there are three copies of the buffer.

This data protection of the front panel control prevents the case in which you enter some data into a control, run the associated VI, and see the data change in the control as it is passed in-place to subsequent nodes. Likewise, data is protected in the case of indicators so that they can reliably display the previous contents until they receive new data.

With subVIs, you can use controls and indicators as inputs and outputs. The execution system makes a copy of the control and indicator data of the subVI in the following conditions.

- The front panel is in memory—this can occur for any of the following reasons.
    - The front panel is open.
    - The VI has been changed, but not saved (all components of the VI remain in memory until the VI is saved).
    - The panel uses data printing.
    - The diagram uses attribute nodes.
- The VI uses local variables.
- The panel uses data logging.
- A control uses suspend data range checking.

A few of these reasons are not intuitive and require further explanation.

One reason concerns attributes such as chart history. For an attribute node to be able to read the chart history in subVIs with closed panels, the control or indicator needs to display the data passed to it. Because there are numerous other attributes like this, the execution system keeps subVI panels in memory if the subVI uses attribute nodes.

If a front panel uses front panel datalogging or data printing, controls and indicators maintain copies of their data. In addition, panels are kept in memory for data printing so the panel can be printed.

If a VI uses suspend data range checking, data is copied to and from all front panel controls and indicators. The front panel values are retained so that the front panel can be displayed if any data goes out of range. Do not use Suspend range checking if memory and speed are a major concern.

Remember, if you set a subVI to display its front panel when called using VI Setup or SubVI Setup, the panel is loaded into memory when the subVI is called. If you set the **Close if Originally Closed** item, the panel is removed from memory when the subVI finishes execution.

SubVIs Can Reuse Data Memory

In general, a subVI can use data buffers from its caller as easily as if the diagrams of the subVI were duplicated at the top level. In most cases, you do not use more memory if you convert a section of your diagram into a subVI. For VIs with special display requirements, as described in the previous section, there might be some additional memory usage for front panels and controls.

Local Variables Cannot Reuse Data Memory

When you create subVIs, you create a connector pane that describes how data is passed to and from the subVI. The data buffers that come from terminals connected to a connector pane can reuse data buffers from calling VIs. Local variables cannot do this. Instead, when you read from a local variable, you create a new buffer for the data from its associated control.

If you use local variables to transfer large amounts of data from one place on the diagram to another, you generally use more memory, and consequently have slower execution speed than if you can transfer data using a wire.

Global Variables Always Keep Copies of Their Data

When you read from a global variable, you create a copy of the data stored in that global variable.

When manipulating large arrays and strings, the time and memory required to manipulate global variables can be considerable. This is especially inefficient when dealing with arrays if you only want to modify a single array element, then store the entire array. If you read from the global variable in several places in your diagram, you might end up creating several buffers of memory.

One technique for minimizing memory usage in this case is to use an uninitialized shift register in a subVI to store the data. This technique can combine the compactness of a global variable with the efficiency of a shift register. See *Case Study 2: Global Table of Mixed Data Types* later in this chapter for more information.

Understanding When Memory Is Deallocated

Consider the following diagram.
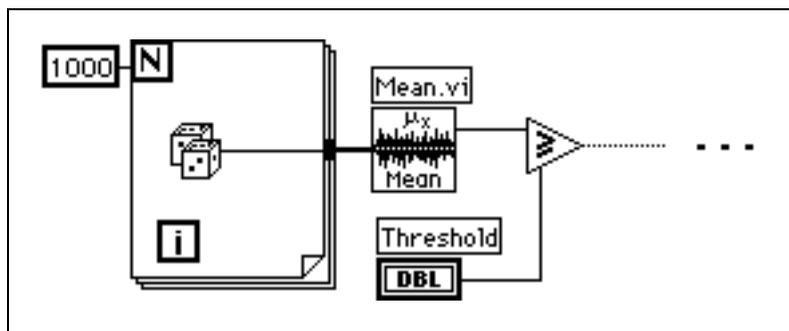
After the Mean VI has executes, the array of data is no longer needed. Because determining when data is no longer needed can become very complicated in larger diagrams, the execution does not deallocate the data buffers of a particular VI during its execution.

On the Macintosh, if the execution system is low on memory, it deallocates data buffers used by any VI not currently executing. The execution system does not deallocate memory used by front panel controls, indicators, global variables, or uninitialized shift registers.

Now, consider the same VI described previously as a subVI to a larger VI. The array of data is created and used only in the subVI. On the Macintosh, if the subVI is not executing and the system is low on memory, it might deallocate the data in the subVI. This is a case in which using subVIs can save on memory usage.

On Windows and Unix platforms, data buffers are not normally deallocated unless a VI is closed and removed from memory. Memory is allocated from the operating system as needed, and virtual memory works well on these platforms. Due to fragmentation, the application might appear to use more memory than it really does. As memory is allocated and freed, the application tries to consolidate memory usage so it can return unused blocks to the operating system.

On all platforms, you can optionally set a preference "Deallocate memory as soon as possible." When this item is on, as subVIs complete execution, they deallocate memory immediately. This might help out with memory usage, but it can slow down performance significantly.

Determining When Outputs Can Reuse Input Buffers

If an output is the same size and data type as an input, and the input is not required elsewhere, the output can reuse the input buffer. As mentioned previously, in some cases even when an input is used elsewhere, the compiler and the execution system can order code execution in such a way that it can reuse the input for an output buffer. However, the rules for this are complex. Do not depend upon them.

Consistent Data Types

If an input has a different data type from an output, the output cannot reuse that input. For example, if you add a 32-bit integer to a 16-bit integer, you see a coercion dot that indicates the 16-bit integer is being converted to a 32-bit integer. The 32-bit integer input can be usable for the output buffer, assuming it meets all of the other requirements (for example, the 32-bit integer is not being reused somewhere else).

In addition, coercion dots for subVIs and many functions imply a conversion of data types. In general, the compiler creates a new buffer for the converted data.

To minimize memory usage, use consistent data types wherever possible. Doing this produces fewer copies of data because of promotion of data in size. Using consistent data types also makes the compiler more flexible in determining when data buffers can be reused.

In some applications, you might consider using smaller data types. For example, you might consider using four-byte, single-precision numbers instead of eight-byte, double-precision numbers. However, carefully consider which data types are expected by subVIs you can call, because you want to avoid unnecessary conversions.

How to Generate Data of the Right Type

Refer to the following example in which an array of 1,000 random values is created and added to a scalar. The coercion dot at the Add function occurs because the random data is double precision, while the scalar is single precision. The scalar is promoted to a double precision before the addition. The resulting data is then passed to the indicator. This diagram uses up 16 KB of memory.
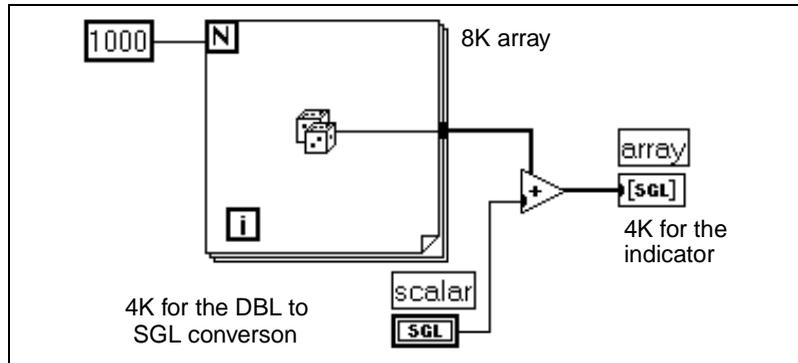
The following illustration incorrectly attempts to correct this problem by converting the array of double-precision random numbers to an array of single-precision random numbers. It uses the same amount of memory as the previous example.
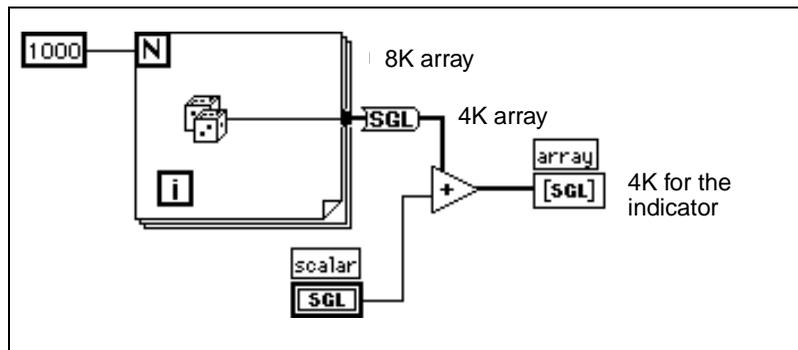
The best solution, shown in the following illustration, is to convert the random number to single precision as it is created, before you create an array. Doing this avoids the conversion of a large data buffer from one data type to another.
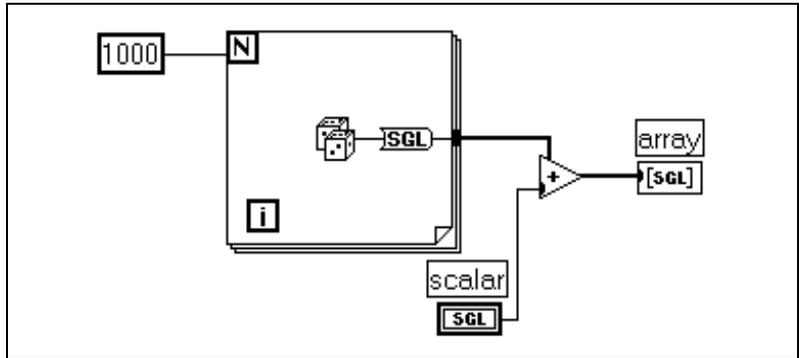
Avoid Constantly Resizing Data

If the size of an output is different from the size of an input, the output does not reuse the input data buffer. This is the case for functions such as Build Array, String Concatenate, and Array Subset which increase or decrease the size of an array or string. When working with arrays and strings, avoid constantly using these functions, because your program uses more data memory, and executes more slowly because it is constantly copying data.

Example 1: Building Arrays

As an example, consider the following diagram which is used to create an array of data. This diagram creates an array in a loop by constantly calling Build Array to concatenate a new element. The input array is not reused by Build Array. Instead, the VI continually resizes the output buffer to make room for the new array, and copies data from the old array to the new array. The resulting execution speed is very slow, especially if the loop is executed many times.
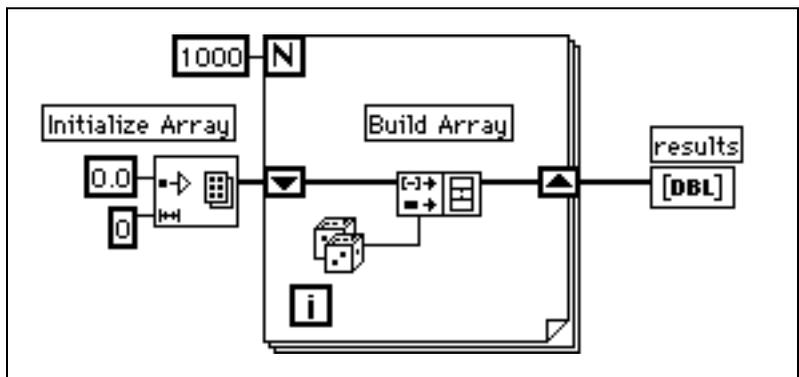
If you want to add a value to the array with every iteration of the loop, you can see the best performance by using auto-indexing on the edge of a loop. With For Loops, the VI can predetermine the size of the array (based on the value wired to N), and resize the buffer only once.
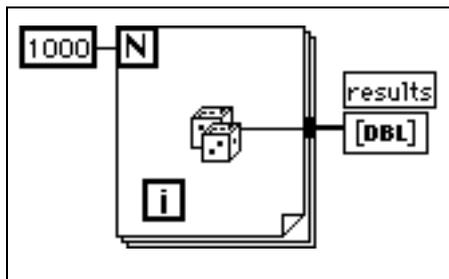
With While Loops, auto-indexing cannot be quite as efficient, because the end size of the array is not known. However, While Loop auto-indexing avoids resizing the output array every iteration by increasing the output array size in large increments. When the loop is finished, the output array is resized to the correct size. The performance of While Loop auto-indexing is nearly identical to For Loop auto-indexing.
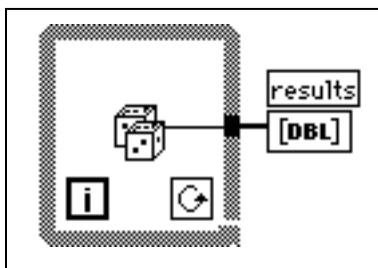
Auto-indexing assumes you are going to add a value to the resulting array with each iteration of the loop. If you must conditionally add values to an array, but can determine an upper limit on the array size, you might consider preallocating the array and then using Replace Array Element to fill up the array.

When you finish filling up the array values, you can resize the array to the correct size. The array is created only once, and Replace Array Element can reuse the input buffer for the output buffer. The performance for this is very similar to the performance of loops using auto-indexing. If you use this technique, be careful the array in which you are replacing values is large enough to hold the resulting data, because Replace Array Element does not resize arrays for you.

An example of this process is shown in the following illustration.
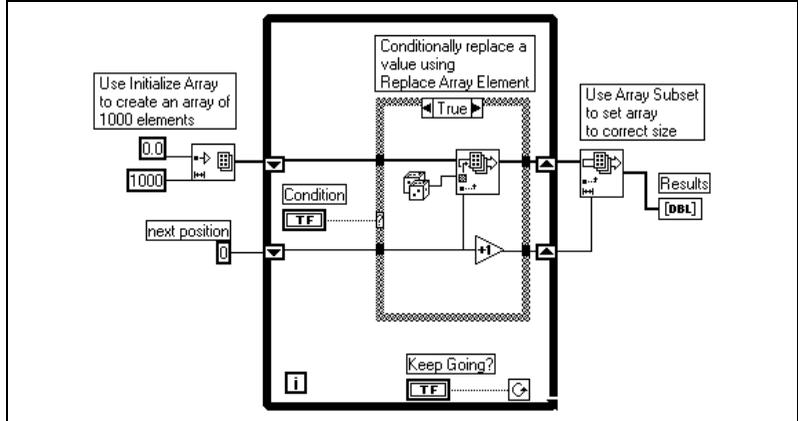
Example 2: Searching through Strings

You can use the Match Pattern function to search a string for a pattern. Depending on how you use it, you might slow down performance by unnecessarily creating string data buffers. The following illustration shows the Help window for Match Pattern.
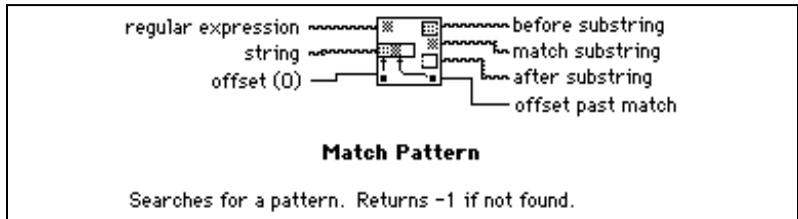
Assuming you want to match an integer in a string, you can use [0–9]+ as the regular expression input to this function. To create an array of all integers in a string, use a loop and call Match Pattern repeatedly until the offset value returned is –1.

The following diagram is one method for scanning for all occurrences
of integers in a string. It initially creates an empty array, and then each
time through the loop, searches the remaining string for the numeric
pattern. If the pattern is found (offset is not –1), this diagram uses Build
Array to add the number to a resulting array of numbers. When there are
no values left in the string, Match Pattern returns –1 and this diagram
completes execution.
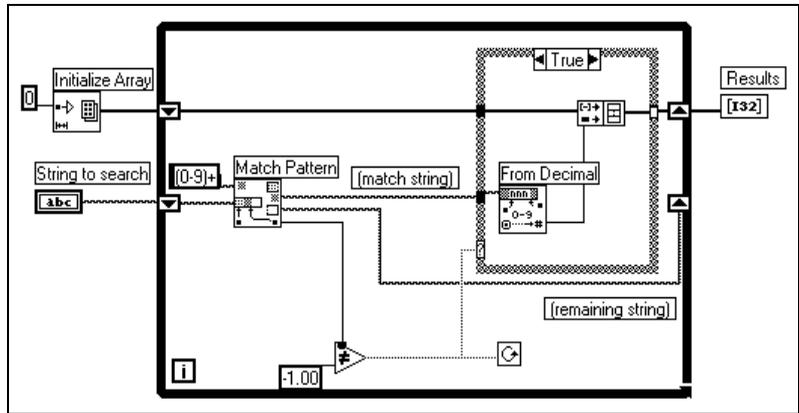
One problem with this diagram is that it uses Build Array in the loop to
concatenate the new value to the previous value. Instead, you can use
auto-indexing to accumulate values on the edge of the loop. Notice you
end up seeing an extra, unwanted value in the array from the last iteration
of the loop where Match Pattern fails to find a match. A solution is to use
array subset to remove the extra unwanted value. This is shown in the
following illustration.
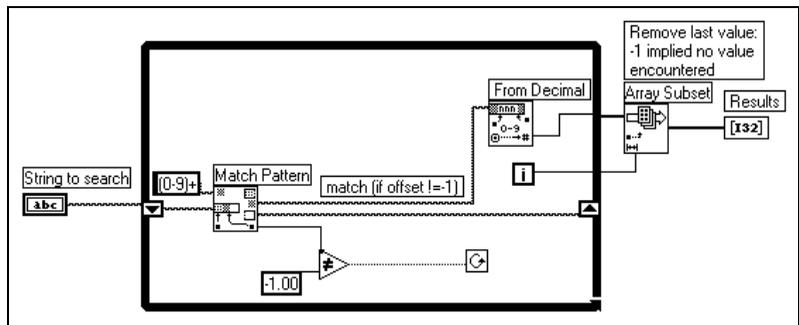
The other problem with this diagram is that you create an unnecessary copy of the remaining string every time through the loop. Match Pattern has an input you can use to indicate where to start searching. If you remember the offset from the previous iteration, you can use this number to indicate where to start searching on the next iteration. This technique is shown in the following illustration.
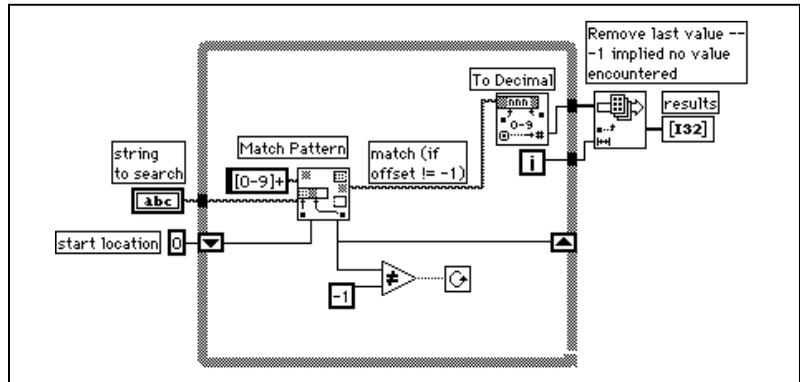
Developing Efficient Data Structures

One of the points made in the previous section is that hierarchical data structures such as arrays of clusters containing large arrays or strings, or clusters containing large arrays or strings, cannot be manipulated efficiently. This section explains why this is so and describes strategies for choosing more efficient data types.

The issue with complicated data structures is that it is difficult to access and change elements within a data structure without causing copies of the elements you are accessing to be generated. If these elements are large, as in the case where the element itself is an array or string, these extra copies use more memory and the time it takes to copy the memory.

You can generally manipulate scalar data types very efficiently. Likewise, you can efficiently manipulate small strings and arrays where the element

is a scalar. In the case of an array of scalars, the following code shows what you do to increment a value in an array.
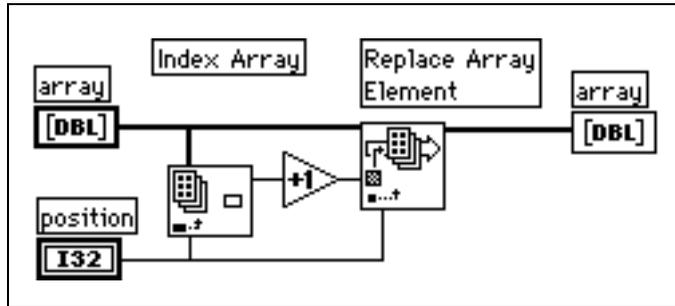
This is quite efficient because it is not necessary to generate extra copies of the overall array. Also, the element produced by the Index Array function is a scalar, which can be created and manipulated efficiently.

The same is true of an array of clusters, assuming the cluster contains only scalars. In the following diagram, manipulation of elements becomes a little more complicated, because you now must use Unbundle and Bundle. However, because the cluster is probably small (scalars use very little memory), there is no significant overhead involved in accessing the cluster elements and replacing the elements back into the original cluster.
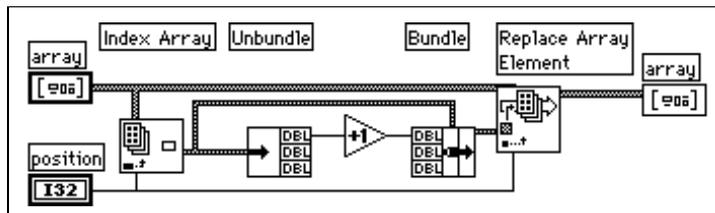
If you have an array of clusters where each cluster contains large sub-arrays or strings, indexing and changing the values of elements in the cluster can be more expensive in terms of memory and speed.

When you index an element in the overall array, a copy of that element is made. Thus, a copy of the cluster and its corresponding large subarray or string is made. Because strings and arrays are of variable size, the copy process can involve memory allocation calls to make a string or subarray of the appropriate size, in addition to the overhead actually copying the data of a string or subarray. This might not be significant if you only plan to do it a few times. However, if your application centers around accessing this data structure frequently, the memory and execution overhead might add up quickly.

The solution is to look at alternative representations for your data. The following three case studies present three different applications, along with suggestions for the best data structures in each case.

Case Study 1: Avoiding Complicated Data Types

Consider an application in which you want to record the results of several tests. In the results, you want a string describing the test and an array of the test results. One data type you might consider using to store this information is shown in the following illustration.
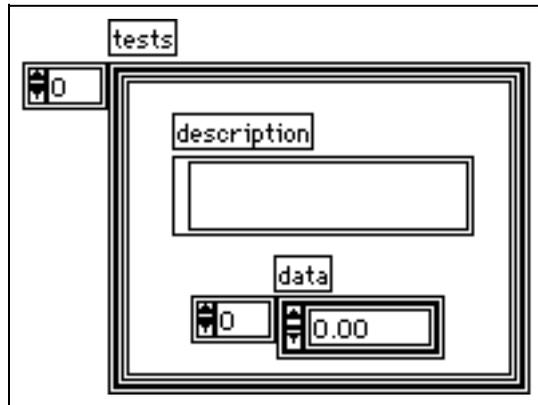
To change an element in the array, you must index an element of the overall array. Now, for that cluster you must unbundle the elements to reach the array. You then replace an element of the array, and store the resulting array

in the cluster. Finally, you store the resulting cluster into the original array. An example of this is shown in the following illustration.
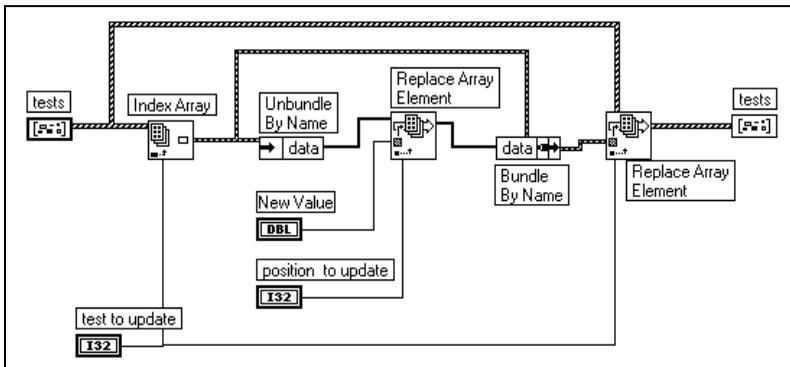
Each level of unbundling/indexing might result in a copy of that data being generated. Notice a copy is not necessarily generated. Copying data is costly in terms of both time and memory. The solution is to try to make the data structures as flat as possible. For example, in this case study break the data structure into two arrays. The first array is the array of strings. The second array is a 2D array, where each row is the results of a given test. This result is shown in the following illustration.
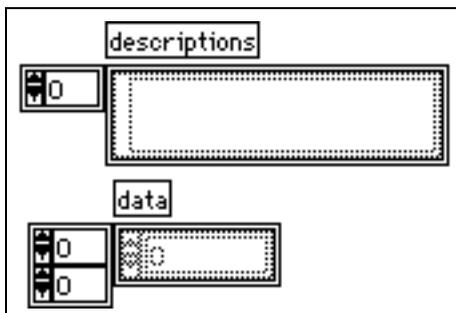
Given this data structure, you can replace an array element directly using
the Replace Array Element function, as shown in the following illustration.
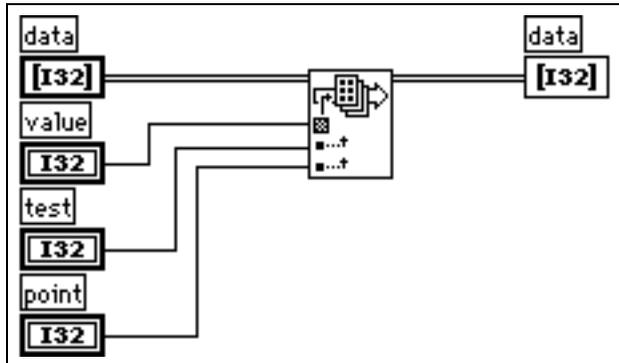
Case Study 2: Global Table of Mixed Data Types

Here is another application in which you want to maintain a table of
information. In this application, you decide you want the data to be
globally accessible. This table might contain settings for an instrument,
including gain, lower and upper voltage limits, and a name used to refer
to the channel.

To make the data accessible throughout your application, you might
consider creating a set of subVIs to access the data in the table, such as
the following subVIs, the Change Channel Info VI and the Remove
Channel Info VI.
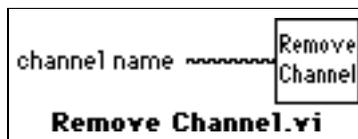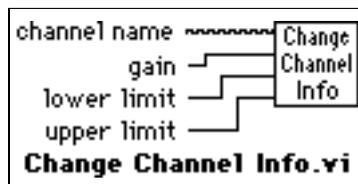
The following sections present three different implementations for these VIs.

Obvious Implementation

With this set of functions, there are several data structures to consider for the underlying table. First, you might use a global variable containing an array of clusters, where each cluster contains the gain, lower limit, upper limit, and the channel name.

As described in the previous section, this data structure is difficult to manipulate efficiently, because generally you must go through several levels of indexing and unbundling to access your data. Also, because the data structure is a conglomeration of several pieces of information, you cannot use the Search 1D Array function to search for a channel. You can use Search 1D Array to search for a specific cluster in an array of clusters, but you cannot use it to search for elements that match on a single cluster element.

Alternative Implementation 1

As with the previous example, choose to keep the data in two separate arrays. One contains the channel names. The other array contains the channel data. The index of a given channel name in the array of names is used to find the corresponding channel data in the other array.

Notice that because the array of strings is separate from the data, you can use the search 1D Array function to search for a channel.

In practice, if you are creating an array of 1,000 channels using the Change Channel Info VI, this implementation is roughly twice as fast as the previous version. This change is not very significant because there is other overhead affecting performance.

A note in the *Memory Usage* section of this chapter cautions against overusing local and global variables. When you read from a global variable, a copy of the data of the global variable is generated. Thus, a complete copy of the data of the array is being generated each time you access an element. The next method shows an even more efficient method that avoids this overhead.

Alternative Implementation 2

There is an alternative method for storing global data, and that is to use an uninitialized shift register. Essentially, if you do not wire an initial value, a shift register remembers its value from call to call. If you are a LabVIEW user and are not familiar with uninitialized shift registers, see Chapter 3, *Loops and Charts*, of the *LabVIEW User Manual* before continuing with this discussion. If you are a BridgeVIEW user, see Chapter 10, *Loops and Charts* of the *BridgeVIEW User Manual* for similar information.

The G compiler handles access to shift registers efficiently. Reading the value of a shift register does not necessarily generate a copy of the data. In fact, you can index an array stored in a shift register and even change and update its value without generating extra copies of the overall array. The problem with a shift register is only the VI that contains the shift register can access the shift register data. On the other hand, the shift register has the advantage of modularity.

What you can do is make a single subVI with a mode input that specifies whether you want to read, change, or remove a channel, or whether you want to zero out the data for all channels, as shown in the following illustration.
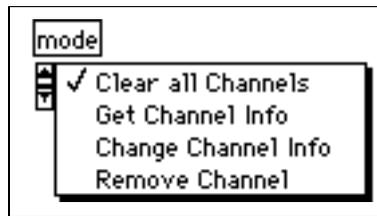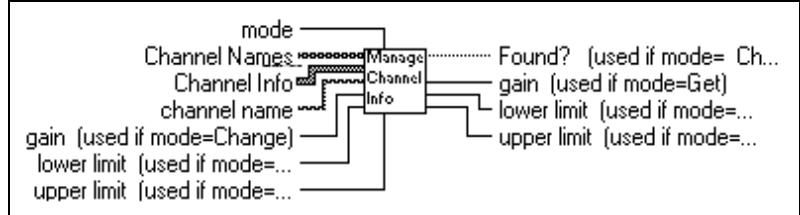
The subVI contains a While Loop with two shift registers—one for the channel data, and one for the channel names. Neither of these shift registers is initialized. Then, inside the While Loop you place a Case structure connected to the mode input. Depending on the value of the mode, you might read and possibly change the data in the shift register.

Following is an outline of a subVI with an interface that handles these three different modes. Only the Change Channel Info code is shown.
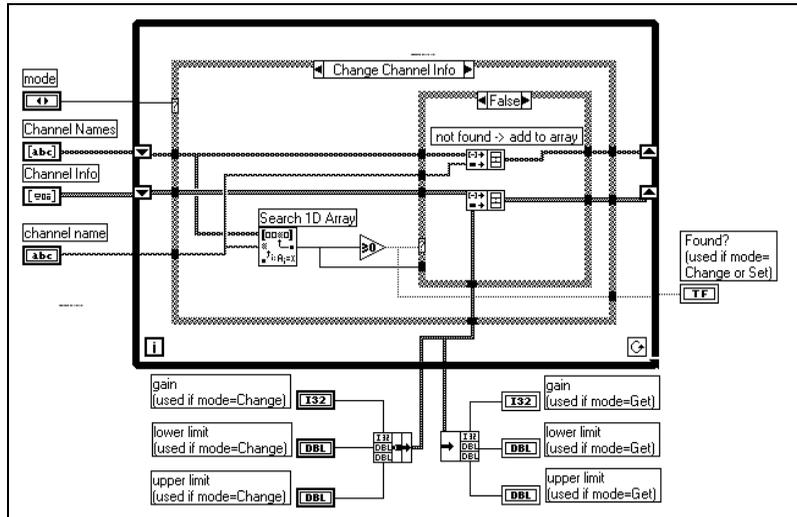
For 1,000 elements, this implementation is twice as fast as the previous implementation, and four times faster than the original implementation.

Case Study 3: A Static Global Table of Strings

The previous example looked at an application in which the table contained mixed data types, and the table might change frequently. In many applications, you have a table of information that is fairly static once created. The table might be read from a spreadsheet file. Once read into memory, you mainly use it to look up information.

In this case, your implementation might consist of the following two functions, Initialize Table From File and Get Record From Table.
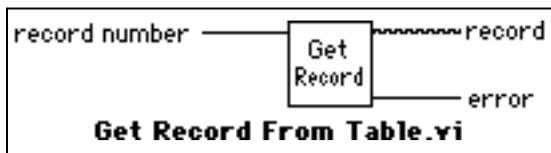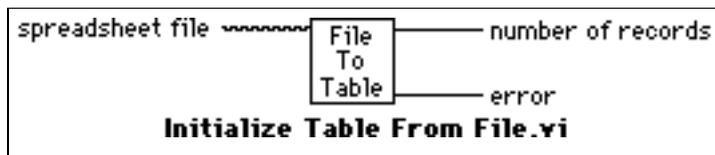
One way to implement the table is to use a two-dimensional array of strings. Notice the compiler stores each string in an array of strings in a separate block of memory. If there are a large number of strings (for example, more than 5,000 strings), you might put a load on the memory manager. This load can cause a noticeable loss in performance as the number of individual objects increases.

An alternative method for storing a large table is to read the table in as a single string. Then, build a separate array containing the offsets of each record in the string. This changes the organization so that instead of having potentially thousands of relatively small blocks of memory, you have instead one large block of memory (the string) and a separate smaller block of memory (the array of offsets).

This method might be more complicated to implement, but it can be much faster for large tables.

29

Portability and Localization Issues

This chapter describes issues related to transporting VIs between platforms and localization.

Portable and Nonportable VIs

VIs are portable among all the platforms your application runs on, as long as the versions of the application are the same. VIs containing CINs or platform-specific features such as DDE are not portable. In this case, a VI ports, but it is broken.

G uses the same file format on all platforms. You can transfer VIs from one system to another system, either by disk or through a network.

After the file is on the new system, you can open the VI. Your application detects the VI is from another platform and recompiles the VI to use the correct instructions for the current processor. If you transfer VIs on a disk formatted for a different platform, you might require a utility program, such as Apple File Exchange on the Macintosh, to read the disk.

You cannot port the following VIs.

- VIs distributed in the `vi.lib` directory. Each distribution contains its own `vi.lib`, so do not move VIs in `vi.lib` across platforms.

- VIs containing CINs. You see an `object code not found` error if the CIN is from a different platform. If you write your CIN source code in a platform independent manner, you can recompile it on another platform and relink it to the ported VI.

- VIs containing the Call Library function. The VI can port, but is broken unless it can find a library of the same name.

- Platform-specific communication VIs such as AppleEvents on the Macintosh and DDE on Windows.

- In Port and Out Port Utility VIs for Windows and the Peek and Poke Utility VIs for Macintosh.

Porting between Platforms

There are several things you can do to ease porting between platforms. Portability issues include differences in filenames, separator characters, resolutions and fonts, possible overlapping of labels, and differences in picture formats.

One consideration is the filename. Filenames are limited to eight characters plus an optional three-character extension, typically .vi in DOS/Windows 3.*x* and in FAT volumes in Windows NT. Macintosh filenames can have 31 characters. Windows 95/NT and UNIX filenames can have 255 characters, including the .vi extension.

To avoid complications, either save VIs with short names, or save them into a VI library. A VI library is a single file that can contain multiple VIs. A library name must conform to platform limits, but VIs in libraries can have names up to 255 characters, regardless of platform. Thus, VI libraries are the most convenient format for transferring VIs, because libraries eliminate most file system dependencies. See the *Saving VIs* section in Chapter 2, *Editing VIs,* for more information on creating VI libraries.

Another issue concerning file names is that they are case-sensitive on UNIX, and not case-sensitive elsewhere. If you are referencing a VI by name, for example, you must ensure that you capitalize the name consistently wherever it is used.

Separator Character Differences

If you are developing a VI for use on multiple platforms, do not use any of the platform-specific path separator characters [\ , / , and :] in your filenames. Avoid any special characters in your filenames because these can be interpreted differently by different file systems. For example, hidden files in UNIX begin with a period.

Resolution and Font Differences

Another portability issue concerns differences in screen resolution and fonts. Fonts can vary from platform to platform, so after porting a VI, it might be necessary to choose new fonts to obtain an appealing display. When designing VIs, keep in mind there are three fonts which best map between platforms—the Application font, the System font, and the Dialog font.

These predefined fonts and the real fonts they map to are as follows.

- The Application font is the default font. It is used in the **Controls** palette, the **Functions** palette, and for new controls.

    - **(Windows)** The U.S. version of Windows usually uses the Arial font. The size depends on the settings of the video driver, because you frequently can set up higher resolution video drivers to use Large Fonts or Small Fonts. In the Japanese version of Windows, the application uses the font that Windows uses for file names in the program manager.

    - **(Macintosh)** The application uses the same font used in the Finder for file names for the Application font. For example, on the U.S. Macintosh system, the application uses Geneva, although on the Japanese Macintosh system, the application uses Osaka.

    - **(UNIX)** The application uses Helvetica by default.

- The System font is the font used for menus.

    - **(Windows)** The application uses Helvetica, with the size dependent upon the video driver.

    - **(Macintosh)** The application normally uses Chicago in the U.S. system software, Osaka in the Japanese system software, and so on.

    - **(UNIX)** The application normally uses Helvetica for this font.

- The Dialog font is the font G uses for text in dialog boxes.

    - **(Windows)** In the U.S. version of Windows, this font is a bold version of the Application font. In the Japanese version of Windows, this font is the same as the System font.

    - **(Macintosh)** The application uses the same font as the System font.

    - **(UNIX)** The application normally uses Helvetica for this font.

When you take a VI containing one of these fonts to another platform, your application ensures the font maps to something similar on that platform.

If you do not use the predefined fonts, but instead select a specific font such as Geneva or New York, the font can change size on the new platform because of the differences in the fonts available and differences in the resolution of the display. If you select Geneva or New York on the Macintosh, the application cannot match it on the Sun or HP-UX, and the application uses the font named *fixed*. If you take a VI with unrecognized fonts to Windows, you might not have a good mapping to a new font.

If you use a predefined font on a section of text, National Instruments recommends you do not change the size of that text. If you change the font to something other than the default size and then take the VI to a different platform, the application tries to match the font with the new size, which might be inappropriate given the resolution of the screen. For example, an application font with size 10 (one pixel bigger than the default) looks good on the Macintosh (Geneva 10) but looks very tiny on Windows with a high-resolution video driver, where it is three pixels smaller than the default.

Overlapping Labels

When you move a VI to a new platform, controls and labels might change size, depending on if the fonts are smaller or larger. G tries to keep labels from overlapping their owners by moving them away from the owning control. Also, every label and constant has a default attribute called **Size to Text**. When you first create a label or constant, this attribute is set, so the bounds of the object resize as necessary to display all of the enclosed text.

If you ever manually resize the object, the application turns off this attribute (the item in the pop-up menu is no longer checked). With **Size to Text** turned off, the bounds of the object stay constant, and the application clips (or crops) the enclosed text as necessary. If you do not want the application to clip text when you move between systems or platforms, leave this attribute on for labels and constants.

Most Sun and HP monitors are much larger and have a higher resolution than PC and Macintosh monitors. If you are a Sun or an HP-UX user, do not make your front panels very large if you want them to port well.

For best results, avoid overlapping controls and leave extra space. If a label even partially overlaps another object and the font is enlarged, it might end up overlapping the control.

Picture Differences

The most basic type of picture contains a bitmap, a series of values specifying the color of each pixel in the picture. More complex pictures might contain any number of commands executed every time the picture is drawn. Pictures containing drawing commands are created by drawing programs or in the draw layer of a graphics application. Bitmap-based pictures are created by paint programs or in the paint layer of a graphics application.

Bitmaps are common storage formats for pictures on all platforms. If you use pictures that contain bitmaps on your front panels, the pictures usually look the same when you load your VIs on another platform. However, pictures containing drawing commands might include some commands not supported on other platforms, for example clipping and pattern filling. These pictures might look strange on other platforms. Check how your VIs look on another platform if you expect them to be used there.

You still can use a drawing program or the draw layer of a graphics application to create your pictures. But to make them more portable, paste the final picture into a paint program or into the paint layer of a graphics application before importing the picture.

(Windows 95/NT and Macintosh) With some applications on Windows 95/NT and many graphics applications on Macintosh, you can cut or copy a picture with a non-rectangular shape. In Macintosh, for example, you can use a Lasso tool like the one shown at the left to select the outline of a circle, triangle, or a more complicated shape like a musical note. Other platforms might draw these irregular shapes on a rectangular white background. In Windows 95/NT, look for applications supporting enhanced metafiles if you want pictures with irregular shapes and pictures that scale well.

VI Localization

You can localize the strings on the front panel by exporting those strings to a tagged text file, translating the text file, and importing that file back into your LabVIEW panel. Refer to Chapter 5, *Printing and Documenting VIs*, for more information. When you translate that tagged text file, you can translate the VI window title, as shown in the following illustration, or you can change the window title interactively through VI Setup. Refer to the *Editing VI Window Titles* section later in this chapter and the *Window Options* section in Chapter 6, *Setting Up VIs and Sub VIs*, for more information.

Figure 29-1. VI Localization Example

All the front panel objects can include caption labels. Name labels cannot be translated because the G-language engine uses that label to identify the object. You cannot change name labels without affecting the diagram, but you can translate the caption labels and hide the name label. For more information refer to the section *G Environment* in Chapter 2, *Building VIs*.

Besides translating strings on the front panel, you can use localized decimal separators when converting numbers to strings. For more information refer to the *Period and Comma Decimal Separators* section, later in this chapter. The Format Date/Time String function displays the date and time according to your specification.

Importing and Exporting VI Strings

The VI string export/import tool writes out all the localizable strings contained in the front panel of a VI to a tagged text file, which is called the VI string file. You select **Project»Export VI strings** or **Project»Import VI strings** and select a text file to export or import within the File dialog box. The software does not import a VI string file if the file contains unknown tags or if there is a missing tag.

You can localize the following strings.

- VI window titles and descriptions
- object caption labels and descriptions
- free labels
- default data (string, table, path, and array default data)
- private data (list box item names, table row and column headers, graph plot names, and graph cursor names).

Importing and exporting strings also creates a log file with a list of any errors that occurred during the operation.

Syntax of the VI String File

The format of the VI string file is much like an HTML file. Every element is marked by a start-tag and an end-tag. A start-tag begins with < and ends with >, and an end-tag starts with </ and ends with >. White-space characters are ignored except within text. Since the character < indicates the beginning of a tag, << is used for the less than character in text and for the > character, >> is used for the greater than character. The double quote character is replaced by " ". Also, end-of-line characters are denoted as <CR>, <CRLF>, or <LF>, which are treated as a carriage return, a carriage return followed by a line feed, and a line feed respectively. This format is intended to be machine readable, so do not be concerned if you find it difficult to read. If tags change or are deleted, errors are issued when the file is imported into the software.

Table 29-1 lists the VI tag types and their VI tag syntax.

Table 29-1. VI Tag Descriptions

VI Tag Type	VI Tag Syntax
`[VI string file]`	`<VI [vi attributes] > [vi info] </VI>`
`[vi attributes]`	`syntaxVersion=1 lvVersion=nnn revision=nnn name="text"`
`[vi info]`	`[vi title] [description] [content]`
`[vi title]`	`<TITLE>text</TITLE> | <TITLE><NO_TITLE></TITLE>`
`[description]`	`<DESC>text</DESC>`
`[content]`	`<CONTENT>[objects]</CONTENT>`

VI attributes are separated by a space, and no space is permitted between the attribute name and the following equal sign, and between the equal sign and the attribute value.

For example:

```
<VI syntaxVersion=1 LVversion=4502007 revision=10
name="AO Generate Waveform.vi">

<TITLE>AO Generate Waveform.vi</TITLE>

<DESC>This VI generates a timed, simple-buffered
waveform for the given output channel at the specified
update rate.</DESC>

<CONTENT>

.....................

</CONTENT>

</VI>
```

Table 29-2 lists the tags which describe the content of the front panel, which are the free labels and object owned labels, caption labels, and attributes.

Table 29-2.   Contents of the Front Panel

Content Tag Type	Content Tag Syntax
[content]	<CONTENT>[objects]</CONTENT>
[objects]	([control] | [label]) *
[control]	<CONTROL [control attributes]> [control info] </CONTROL>
[label]	<LABEL>[style text] </LABEL>
[style text]	<STEXT>text with font information </STEXT>

Between <STEXT> and </STEXT>, you can type font specifications. Font information is encoded using the following format: <FONT name="font name" size='3' style='BIUSO' color=00ff00>. The font attributes can be listed in any order. Font specification is different from other elements because it has no end-tag. For example, a caption with the text "**Bold** *label*" might be described as the following.

```
<LABEL><STEXT><FONT name="times new roman" size=12
style='B'>Bold <FONT style='I'>label</STEXT></LABEL>
```

Fonts can be defined as predef to specify one of the predefined fonts (application font, dialog font, or system font).

Table 29-3 lists the tags which describe the [control].

Table 29-3.   Tags for the [control]

Content Tag Type	Content Tag Syntax
[control]	<CONTROL [control attributes]> [control info] </CONTROL>
[control attributes]	ID=xxx type="Boolean" name="switch"
[control info]	[description] [parts] [privData section] [defData section] [content]
[parts]	<PARTS> [part]*</PARTS>

Table 29-3.   Tags for the [control] (Continued)

Content Tag Type	Content Tag Syntax
`[part]`	`<PART [part attributes]> [part info] </PART>`
`[part attributes]`	`partID=nnn partOrder=nnn`
`[part info]`	`[control]	[label]	[multiLabel]`

The following is an example of a ring control description with a caption of "Ring" and options Load, Unload, Open, and Close.

```
<CONTROL ID=87 type="Ring" name="RING control">

<DESC>ring control</DESC>

<PARTS>

<PART ID=12 order=0 type="Ring
Text"><MLABEL><STRINGS><STRING>Load</STRING><STRING>Unl
oad</STRING><STRING>Open</STRING><STRING>Close</STRING>
</STRINGS></MLABEL></PART>

<PART ID=82 order=0 type="Caption"><LABEL><STEXT><FONT
color=FF0033 size=12>RING</STEXT></LABEL></PART>

</PARTS>

</CONTROL>
```

The MLABEL (multilabel) tag above is used to designate the option string on a ring control or the strings on Boolean buttons, a string for each of the four states. The following is a generic description of the MLABEL tag syntax.

```
[multiLabel]<MLABEL> [mlabel info] </MLABEL>

[mlabel info][font][strings]
```

Table 29-4 lists the tags which describe the default data for strings, tables, arrays, and paths.

Table 29-4.   Default Data for Strings

Content Tag Type	Content Tag Syntax
`[defData section]`	`<DEFAULT> [defData] </DEFAULT>`
`[defData]`	`[str def]	[table def]	[arr data]	[path data]`
`[str def]`	`[string]	<SAME_AS_TEXT>`
`[table def]`	`[strings]`
`[arr data]`	`<ARRAY nElems=n> [arr element data] </ARRAY>`
`[arr element data]`	`clust data]	[str data]	[non-str data]`
`[str data]`	`[string]`
`[non-str data]`	`<NON_STRING>`
`[clust data]`	`<CLUSTER nElems=n> [clust element data] </CLUSTER>`
`[clust element data]`	`[clust data]	[str data]	[non-str data]	[arr data]	[path data]`
`[path data]`	`<PATH type ="absolute"> a<SEP> SYSTEM </PATH>`

For `[arr data]`, n `[arr element data]` must follow the `<ARRAY>` tag. Likewise, for `[clust data]`, there must be n `[clust element data]`.

For string control's default data, use a special tag, `<SAME_AS_TEXT>`, which indicates the string's default data is the same as the `kStrTextID` label on the string's partsList. Use of this tag eliminates the necessity of repeating the same text for both the `kStrTextID` label and the string's default data.

For path control's default data, the `<PATH>` start tag can have an attribute that specifies the path type. The possible attribute values are `"absolute"`, `"relative"`, `"not-a-path"`, and `"unc"`. The path segments that come between the `<PATH>` and `</PATH>` tags are

delimited by a <SEP> tag. For example, on the Windows platform, an absolute path c:\windows\temp\temp.txt is written as follows.

```
<PATH type="absolute">c<SEP>windows<SEP>temp<SEP>temp.
txt</PATH>
```

Table 29-5 lists the tags which describe private data, such as list box item names, table row and column headers, the font used for table cells, and graph plot names and cursor names.

Table 29-5. Tag Descriptions for Table Cells, Graph Plot Names, and Cursor Names

Content Tag Type	Content Tag Syntax
[privData section]	<PRIV> [privData] </PRIV>
[privData]	([items] | [col header] | [row header] | [cell fonts] | [plots] | [cursors])
[items]	<ITEMS> [string]* </ITEMS>
[col header]	<COL_HEADER> [string]* </COL_HEADER>
[row header]	<ROW_HEADER> [string]* </ROW_HEADER>
[cell fonts]	<CELL_FONTS> [cell font]* </CELL_FONTS>
[plots]	<PLOTS> [string]* </PLOTS>
[cursors]	<CURSORS> [string]* </CURSORS>
[cell font]	[row# col#][font]
[font]	<FONT name="font name" size='x' style='BIUSO' color=000000>

[strings] and [string] have the following format:

[strings]<STRINGS> [string]* </STRINGS>

[string]<STRING> text </STRING>

Editing VI Window Titles

You can customize the VI window title to make it more descriptive than the VI file name. This feature is important for localized VIs. The VI window title can be translated to the local language, is not governed by file-system naming constraints and is still recognized by the VIs that might call it. To change the VI window title while you are editing the VI, select **VI Setup** and **Window Options** from the top ring, as shown in the following illustration.
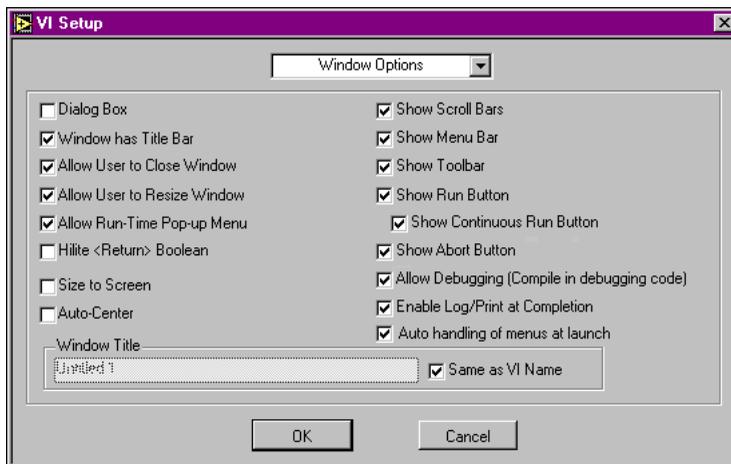
To change the **Window Title**, deselect **Same as VI Name** and type in the desired VI window title. To change the VI window title programmatically, refer to Chapter 21, *VI Server*.

Period and Comma Decimal Separators

You can specify the use of the system decimal separator or the use of a period, by selecting **Front Panel** from the drop down menu in the **Edit»Preferences** dialog box. Alternately, you can force a period as a decimal separator when converting numbers to strings, and vice versa, using the following functions.

• To Engineering

• To Fractional

• To Exponential

• From Exponential/Fract/Eng

Format Date/Time String

You can set the display format of date and time by using the Format
Date/Time String Function. For information on this function, refer to the
Online Reference, available online by selecting **Help»Online Reference**.

Porting across G-Language Applications

Any LabVIEW VI that still contains its block diagram can be converted
to BridgeVIEW, provided the version of the G language in LabVIEW
and BridgeVIEW are compatible. Converting VIs from BridgeVIEW
to LabVIEW is also possible, but not all VIs convert.

LabVIEW to BridgeVIEW

The most important factor to note when converting VIs from LabVIEW
to BridgeVIEW is the version of the G-programming language used to
create the VI. Consult the following table. All LabVIEW VIs created with
LabVIEW 3.x through LabVIEW 4.x can be loaded into any version of
BridgeVIEW, provided the VIs still have diagrams. VIs created with
LabVIEW 5.0.x can be loaded only by BridgeVIEW 2.0 or later.

LabVIEW version	G Version	BridgeVIEW version	G Version
any up to 4.0.1	any up to 4.0.1	1.0	4.1
4.1	4.0.2	1.0.1	4.1.1
4.1.1	4.0.3	1.1	4.1.2
5.0	5.0	2.0	5.0

BridgeVIEW to LabVIEW

There are two factors that determine whether a BridgeVIEW VI can be
loaded and run by LabVIEW. First, the versions of the G language
must be compatible. Therefore, BridgeVIEW VIs must be loaded by
LabVIEW 5.0 or later at this time, because G 4.0.x cannot load VIs
created by G 4.1.x or later. The second factor that determines if a VI can
be converted from BridgeVIEW to LabVIEW is whether the VI contains
VIs specific to BridgeVIEW. Any BridgeVIEW VI using features
specific to BridgeVIEW might not load, or operate correctly in LabVIEW.

For example, the tag data type is specific to BridgeVIEW. Any VI that uses the tag data type or accesses the BridgeVIEW engine does not work in LabVIEW.

If you plan to use BridgeVIEW to develop VIs you intend to convert to LabVIEW, select the Basic G or LabVIEW palette set. This palette set displays the same functions as the LabVIEW palette set. VIs built using only the functions in these palettes are portable between BridgeVIEW and LabVIEW.

A

Data Storage Formats

This appendix describes the formats in which you can save data. This information is most useful to advanced users, such as those using code interface nodes (CINs) and those reading from or writing to files used by the file I/O functions. This 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 Formats for Front Panel Controls and Indicators

Booleans

Booleans are stored as 8-bit values. If the value is zero, the Boolean is FALSE. Any non-zero value represents TRUE.

```
7              0
┌──────────────┐
│              │
└──────────────┘
```
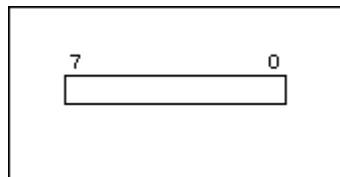
Numerics

Extended

When extended-precision numbers are saved to disk, they are stored in a platform-independent 128-bit format, which is the same as the UNIX in-memory format. In memory, the size and precision varies depending on the platform.

(Windows) Extended-precision floating-point numbers have 80-bit format (80287 extended-precision format).

```
79        64              0
┌─┬────────┬──────────────┐
│s│15 exp 0│63  mantissa  0│
└─┴────────┴──────────────┘
```
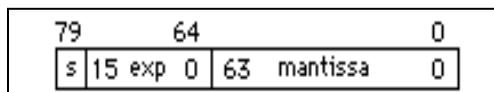
(68K Macintosh) Extended-precision floating-point numbers have 96-bit format (MC68881-MC68882 extended-precision format).
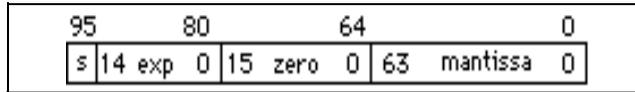
(Power Macintosh) Extended-precision floating-point numbers are represented as two double-precision floating-point numbers combined, that is, the Apple double-double format.
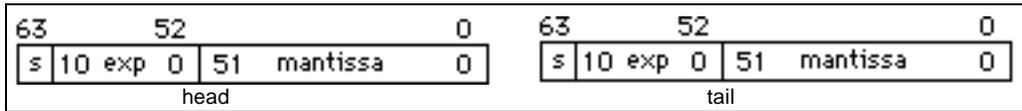
(Sun) Extended-precision floating-point numbers have 128-bit format.
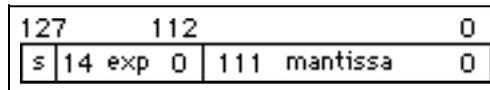
(HP-UX) Extended-precision floating-point numbers are represented as double-precision floating-point numbers, as shown in the next illustration.

Double

Double-precision floating-point numbers have 64-bit IEEE double-precision format (format default).
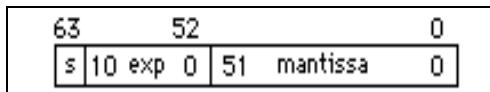
Single

Single-precision floating-point numbers have 32-bit IEEE single-precision format.
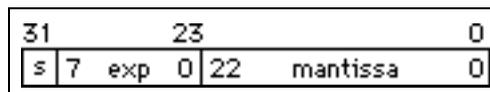
Long Integer

Long integer numbers have 32-bit format, signed or unsigned.

```
31                                  0
```

Word Integer

Word integer numbers have 16-bit format, signed or unsigned.

```
15              0
```

Byte Integer

Byte integer numbers have 8-bit format, signed or unsigned.

```
7      0
```

Arrays

Arrays are stored with the size of each dimension of the array in long integers, followed by the data. Because of alignment constraints of certain platforms, the dimension size might be followed by a few bytes of padding so that the first element of the data is correctly aligned. If you are a LabVIEW user, see the *Alignment Considerations* section of Chapter 2, *CIN Parameter Passing*, of the *LabVIEW Code Interface Reference Manual* for further information. This document is available only in portable document format (PDF) on your software program disks or CD. The following example shows a one-dimensional array of single-precision floating-point numbers. The decimal numbers to the left represent the byte offsets of locations in memory from which the array begins.

```
0:        dimSize = n
4:        float_32 [0]
8:        float_32 [1]
             ⋮
          float_32 [n-2]
          float_32 [n-1]
```
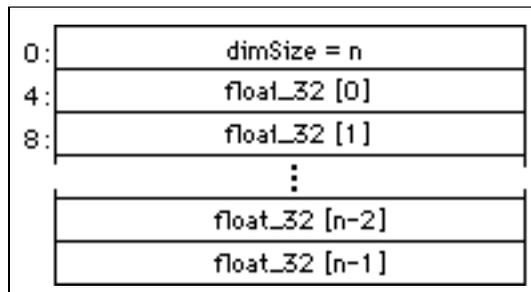
This illustration shows a four-dimensional array of word integers.

0:	1st dimSize = i
4:	2nd dimSize = j
8:	3rd dimSize = k
12:	4th dimSize = 1
16:	Int_16 [0,0,0,0]
18:	Int_16 [0,0,0,1]
	⋮
	Int_16 [i-1,j-1,k-1,1-2
	Int_16 [i-1,j-1,k-1,1-1
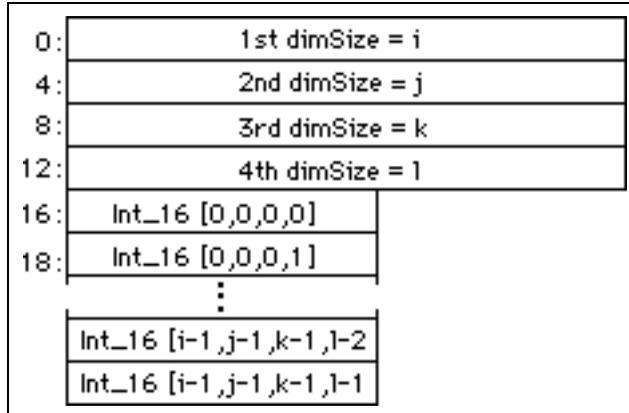
Strings

Strings are stored as if they were one-dimensional arrays of byte integers (8-bit characters).

0:	dimSize = n
4:	char[0]
5:	char[1]
	⋮
	char[n-2]
	char[n-1]
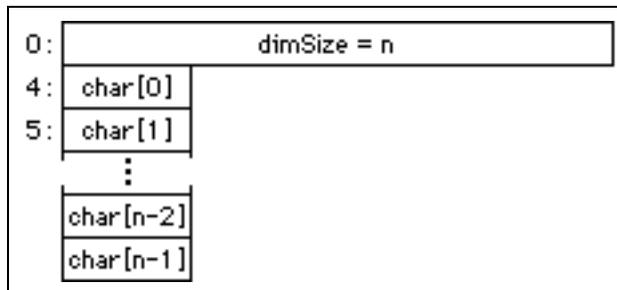
Paths

Paths are stored with 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 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*. You store scalar data directly in the cluster. Arrays, strings, handles, and paths are stored indirectly. The cluster stores a handle that points to the memory area in which the data actually is stored. Because of alignment constraints of certain platforms, the dimension size might be followed by a few bytes of padding so that the first element of the data is correctly aligned. If you are a LabVIEW user, see the *Alignment Considerations* section of Chapter 2, *CIN Parameter Passing*, of the *LabVIEW Code Interface Reference Manual* for further information. This document is available only in portable document format (PDF) on your software program disks or CD.

The following illustrations show a cluster containing a single-precision floating-point number, an extended-precision floating-point number, and a handle to a one-dimensional array of unsigned word integers, presented in that order. For more information, see Chapter 14, *Array and Cluster Controls and Indicators*.

- **(Windows)**

0:	SGL float
4:	EXT float
14:	Handle to Array

- **(Macintosh)**

0:	SGL float
4:	EXT float
16:	Handle to Array
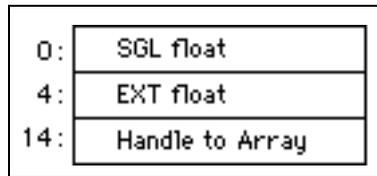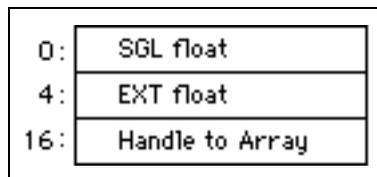
- **(Sun)**
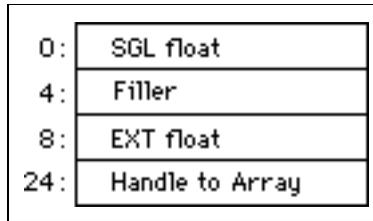
- **(HP-UX)**
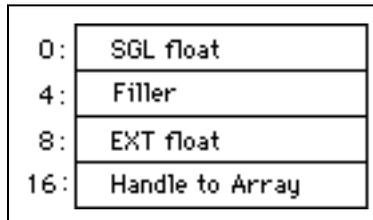
In the next example, the embedded clusters are not stored indirectly. Instead, the data is stored inside the embedded clusters directly, as if the data were not embedded in the subcluster. Only arrays, strings and handles are stored indirectly.
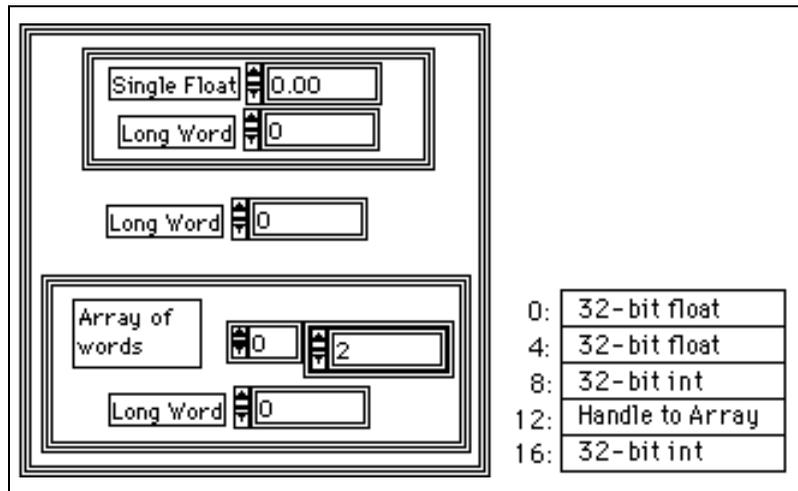
Type Descriptors

Each wire and terminal in the block diagram is associated with a data type. You can keep 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 G. Numeric values are written in hexadecimal, unless otherwise noted.

The generic format of a type descriptor is: `<length> <type code>`.

Some type descriptors have additional information following the type code. Arrays and clusters are structured or aggregate data types because they include other types. For example, the cluster type contains additional information about the type of each of its elements.

The first word (16 bits) in any type descriptor is the length, in bytes, of that type descriptor (including the length word). The second word (16 bits) is the type code. The high-order byte of the type code is reserved for internal use. When comparing two type descriptors for equality, it is a good idea to ignore this byte; two descriptors are equal even if the high-order bytes of the type codes are not.

The type code encodes the actual type information, such as single-precision or extended-precision floating-point number, as listed in the following table. These type code values might change in future versions. The *xx* in the type descriptor columns represent reserved values. Ignore them.

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 (numbers in hexadecimal)	Type Descriptor (numbers in hexadecimal)
Byte Integer	01	0004 *xx*01
Word Integer	02	0004 *xx*02
Long Integer	03	0004 *xx*03
Unsigned Byte Integer	05	0004 *xx*05
Unsigned Word Integer	06	0004 *xx*06
Unsigned Long Integer	07	0004 *xx*07
Single-Precision Floating-Point Number	09	0004 *xx*09
Double-Precision Floating-Point Number	0A	0004 *xx*0A
Extended-Precision Floating-Point Number	0B	0004 *xx*0B
Single-Precision Complex Floating-Point Number	0C	0004 *xx*0C
Double-Precision Complex Floating-Point Number	0D	0004 *xx*0D
Extended-Precision Complex Floating-Point Number	0E	0004 *xx*0E
Enumerated Byte Integer	15	\<nn> *xx*15 \<k> \<k pstrs>
Enumerated Word Integer	16	\<nn> *xx*16 \<k> \<k pstrs>
Enumerated Long Integer	17	\<nn> *xx*17 \<k> \<k pstrs>

Table A-1. Scalar Numeric Data Types (Continued)

Data Type	Type Code (numbers in hexadecimal)	Type Descriptor (numbers in hexadecimal)
Single-Precision Physical Quantity	19	<nn> *xx*19 <k> <k base-exp>
Double-Precision Physical Quantity	1A	<nn> *xx*1A <k> <k base-exp>
Extended-Precision Physical Quantity	1B	<nn> *xx*1B <k> <k base-exp>
Single-Precision Complex Physical Quantity	1C	<nn> *xx*1C <k> <k base-exp>
Double-Precision Complex Physical Quantity	1D	<nn> *xx*1D <k> <k base-exp>
Extended-Precision Complex Physical Quantity	1E	<nn> *xx*1E <k> <k base-exp>

Table A-2. Non-Numeric Data Types

Data Type	Type Code (numbers in hexadecimal)	Type Descriptor (numbers in hexadecimal)
Boolean	21	0004 *xx*21
String	30	0008 *xx*30 <len>
Handle	31	0006 *xx*31 <kind>
Path	32	0008 *xx*32 <len>
Pict	33	0008 *xx*33 <len>
Array	40	<nn> *xx*40 <k> <k dims> <element type descriptor>
Cluster	x50	<nn> *xx*50 <k> <k element type descriptors>

The minimum value in the size field of a type descriptor is 4 (as shown in Table A-1). However, any type descriptor can have a name appended (a Pascal string) in which case the size is larger by the length of the name (rounded up to a multiple of 2).

The string, path, and pict data types have a 32-bit length (similar to the array dim size), although the only value currently encoded is FFFFFFFF (-1) which indicates variable-sized. Currently, all strings, paths, and picts are variable-sized. The actual length is stored with the data.

Notice the array and cluster data types each have their own type code. They also contain additional information about their dimensionality (for arrays) or number of elements (for clusters), as well as information about the data types of their elements.

In the following example of an enumerated byte integer for the items am, fm, fm stereo, each group of characters represents a 16-bit word. The space enclosed in quotes (" ") represents an ASCII space.

```
0016 0015 0003 02a m02 fm 09f m" " st er eo
```

0016 indicates 22 bytes total. 0015 indicates an enumerated byte integer. 0003 indicates there are three items.

In the following example of a double-precision physical quantity with units m/s each group represents a 16-bit word.

```
000E 001A 0002 0002 FFFF 0003 0001
```

000E indicates 14 bytes total. 0x1A indicates this is a double-precision number with units. 0002 indicates two base-exponent pairs. 0002 denotes the seconds base index. FFFF (-1) is the exponent of seconds. 0003 denotes the meters base index. 0001 is the exponent of meters.

☞ **Note**   *All physical quantities are stored internally in terms of base units, regardless of what unit they are displayed as. Table 9-2, Base Units, shows the nine bases which are represented by indices 0-8 for radians-candela.*

Array

The type code for an array is 0x40. Immediately after the type code is a word containing the number of dimensions of the array. Then, for each dimension, a 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 might be any type except an array. The dimension size for any dimension might be FFFFFFFF (–1). This means the array dimension size is variable. Currently, all arrays are variable-sized. The actual dimension size is stored 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 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 0021

Cluster

The type code for a cluster is 0x50. Immediately after the type code is a word containing the number of items in the cluster. After this word is the type descriptor for each element in *cluster order*. For example, consider a cluster of two integers: a signed-word integer and an unsigned long integer:

000E 0050 0002 0004 0002 0004 0007

000E is the length of the type descriptor including the element type descriptors.

The following is a type descriptor for a multiplot graph (the numeric types can vary):

```
0028 0040 0001 FFFF FFFF...1D array of
001E 0050 0001...1 component cluster of
0018 0040 0001 FFFF FFFF...1D array of
000E 0050 0002...2 component cluster of
0004 000A... double-precision floating-point number
0004 0003... long integer
```

Flattened Data

Two internal functions convert data from the format in memory 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 is stored in *tree* form. For example, a cluster of a double-precision floating-point number and a string is stored as an 8-byte floating-point number, followed by a 4-byte handle to the string. The string data is not stored adjacent in memory to the extended-precision floating-point number. Therefore, if you want to write the cluster data to disk, you must get the data from two different places. Of course, with an arbitrarily complex data type, the data might be stored in many different places.

When data is saved 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 stored in several pieces. When your G development environment loads such a file from disk, it must perform the reverse operation—it must read a single string and *unflatten* it into its internal, possibly discontiguous form.

The flattened data is *normalized* to a standard form so the data can be used unaltered by VIs running on any platform. It stores numeric data in *big endian* (most-significant byte first) form, and it stores extended precision floating-point numbers as 16-byte quantities using the Sun extended-precision format described earlier in this section.

☞ **Note**   *When writing data to a file for use by an application not created using G, you might want to transform your data after flattening it. Similarly, when reading data from a file produced by an application not created using G, you might want to transform your data before unflattening. Other Windows applications typically expect numeric data to be in little endian form (least-significant byte first). Other Windows and Macintosh applications typically expect extended-precision floating-point numbers to be in the 80-bit and 96-bit formats described previously, respectively.*

The Flatten to String and Unflatten from String block diagram functions (described in the **Online Help**, available online by selecting **Help»Online Reference**) let you flatten and unflatten data just as your G development environment does internally when saving and loading data.

Scalars

The flattened form of any numeric type, as well as the Boolean type, contains only the data in big endian format. For example, a long integer with value –19 is encoded as FFFF FFED. A double-precision floating-point number with a value equal to ¼ is 3FD0 0000 0000 0000. A Boolean TRUE is any non-zero value. A Boolean FALSE is 00.

The file form for extended-precision numbers is based on the SPARC quadruple-precision form, which is a 16-bit biased exponent, followed by a mantissa which is 112 bits long. In this form, the bit left of the binary point in the mantissa is an implicit bit (assumed always to be one) which does not appear in the representation.

Strings, Handles, and Paths

Because strings, handles, and paths have variable sizes, the flattened form is preceded by a normalized long integer that records their length in bytes. For paths, this length is preceded by four characters: PTH0. For instance, a string type with value ABC is flattened to 0000 0003 4142 43.

The flattened format is similar to the format the string takes in memory. However, handles and paths do not have a length value preceding them when they are stored in memory, so this value comes 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 record 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. Also, this data is flattened if necessary. The following is an example of a two-dimensional array of six 8-bit integers.

{ {1, 2, 3}, {4, 5, 6} } is stored as 0000 0002 0000 0003 0102 0304 0506.

The following is an example of a flattened one-dimensional array of Boolean variables:

{T, F, T, T} is stored as 0000 0004 0100 0101. (01 is the preferred value for TRUE.)

Clusters

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 is `0004 0000 000C`.

A flattened cluster of a string `ABC` and a word integer of value 4 is `0000 0003 4142 4300 04`.

A flattened cluster of a word integer of value 7, a cluster of a word integer of value 8, and a word integer of value 9 is `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; the type descriptor is required 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.

B

Common Questions about G

This appendix provides answers to some of the questions commonly asked by G users.

Charts and Graphs

How do I wire data to a chart or graph?

Show the Help window and move the Wiring tool across a graph terminal in the block diagram to see a brief description of how to wire basic graph types. There are excellent examples on graphs and charts in the `examples\general\graphs` directory.

What is the basic difference between charts and graphs?

Charts and graphs differ in the way they display and update data. VIs with graphs usually collect the data in an array and then plot it on the graph, similar to a spreadsheet that first stores the data then generates a plot of it. In contrast, a chart appends new data points to those already in the display. In this manner, you can see the current reading or measurement in context with data previously acquired. The length of the chart history buffer can be set by a pop-up option on the chart. Of course, it is possible to implement a history buffer with the graph indicators as well; however, this needs to be done in the block diagram. These features are already built into the chart.

How do I flip the scales on my chart or graph?

To flip the scales for the *x*-axis or *y*-axis, use the attribute X Flipped or Y Flipped. If the X Flipped attribute is set to TRUE, then the minimum of the *x*-axis scale is set to the right and the maximum to the left. Similarly, if the Y Flipped attribute is set to TRUE, the minimum of the *y*-axis scale is at the top and the maximum is at the bottom.

How do I make a chart or graph display information in a time format?

Graphs and charts can display numeric information in a relative or absolute time format. Use the **X Scale»Formatting…** or **Y Scale»Formatting…** pop-up option to set the *x*-axis or *y*-axis scale format to **Relative Time**.

How do I make the x-axis of a chart display real time?

See the example VI called `Real-Time Chart.vi`, located in `examples\general\graphs\charts.llb`. It unbundles the date time cluster from the Seconds to Date/Time function into hours, minutes, and seconds. It then converts these numbers into the number of seconds elapsed since midnight. Finally, this number is used as an input to the Xo and delta X attribute of a chart with its *x*-scale formatting set to **Relative Time**. Or you can use the **Absolute Time** Format & Precision option.

How do I clear a chart programmatically?

Wire an empty array to the History Data attribute. The data type of this empty array is the same as the data type wired to the chart. An excellent example illustrating this technique is found in `examples\general\graphs\charts.llb\How to Clear Charts & Graphs.vi`.

How do I avoid the flashing from the graph each time it is updated?

Use the **Smooth Updates** option to keep graphs from flashing. This option is located in the **Data Operations** submenu of the graph pop-up menu. **Smooth Updates** redraws the graph to an off-screen buffer before copying the image to the display. While producing a smoother update to the indicators, this option is generally slower and requires more memory.

Smooth Updates can be set globally through the **Edit»Preferences»Front Panel** dialog box. It can be turned on or off for individual graphs from their pop-up menus.

How do I update a graph without clearing it?

All the graphs (waveform, *XY,* and intensity) always clear before writing new data; however, it is a simple procedure to keep track of the data previously written to the graph and append new data with each write. The `examples\general\arrays.llb\Separate Array Values.vi` demonstrates the technique of using the Build Array function to append new values to an array. If you are a LabVIEW user, refer to Chapter 5, *Arrays, Clusters, and Graphs*, of the *LabVIEW User Manual*.

How can I create a bar graph?

The legend menu item, **Common Plots** lists bar plots as one style.
The other plot attributes allow for a different baseline and for horizontal bar plots. Also, see examples\general\graphs\bargraph.llb. This VI library contains an example bar graph and two VIs to create bar graphs out of an array input. The G Picture Control Toolkit also can create bar graphs.

How can I create polar plots and Smith charts?

The G Picture Control Toolkit contains examples with routines to create polar plots and Smith charts. The toolkit is a versatile graphics package for creating arbitrary front panel displays. The Picture VI Library, which ships with the toolkit, implements a common set of two-dimensional drawing commands for building graphic images.

How can I make a label for the y-axis that is rotated 90 degrees?

BridgeVIEW and LabVIEW do not rotate text. To use rotated text, first create it in an application with this capability, Windows Paintbrush for example, and save it in a suitable format (see the *Customizing Controls Using Imported Graphics* section of Chapter 8, *Introduction to Front Panel Objects*, for more information on importing graphics). You can then import this graphic and place it near the y-axis of the graph or chart.

How do I place a text label on a graph cursor?

For programmatic access you first use the Cursor Name Visible attribute to display the cursor name. You then use the Cursor Name attribute to assign a name to the cursor. For interactive use, the cursor style menu permits control of the name in the cursor display of the graph.

After I've added a cursor to my graph, how do I remove it?

You must empty the array that contains the clusters holding the information of each cursor. Select **Data Operations»Empty Array** from the pop-up menu of the cursor display, or write an empty array to the Cursor List Attribute Node programmatically.

Error Messages and Crashes

What do I do when a dialog box appears stating that my application is out of memory?

LabVIEW and BridgeVIEW allocate memory as needed, but arrays and strings must be stored in a contiguous block of memory. If your LabVIEW or BridgeVIEW application is unable to find a block of unused memory (physical or virtual) large enough for the string or array, a dialog box appears to indicate it was not able to allocate the required memory.

If you want to save changes to your VIs after the application runs out of memory, you might save your VIs to another location or make sure you have a backup copy. After you restart the software with more memory, you can load these VIs into memory, check to see if the save was successful, and proceed in your VI development.

What do I do if LabVIEW or BridgeVIEW crashes when printing? (Windows)

This crash might be related to the video driver or printer driver. See the question below concerning random crashes on Windows for information on how to proceed.

While running LabVIEW or BridgeVIEW, what do I do if I receive a failure message which includes a source code file and line number?

This indicates an error has occurred and the application cannot proceed. Please notify National Instruments of this message and explain what action triggered the message.

What do I do if LabVIEW or BridgeVIEW tends to crash randomly? (Windows)

The crashes might be general protection faults or failure messages which include a source code file and line number. Often random crashes involve problems with the video driver on the computer. To help determine if this is the case, use standard VGA as the video driver. To change video drivers, select VGA from the list of video drivers in Windows Setup. LabVIEW and BridgeVIEW use the standard Windows API for its graphical calls; however, many video drivers do not fully conform to this standard. If the crashes do not occur with the standard VGA driver, then it is a good idea to

get an update to your video driver. You probably can obtain the latest release of the video driver software from both the manufacturer of your PC and the manufacturer of the video card.

If the situation is not resolved by updating the video driver, please contact National Instruments.

(Windows) What do I do if I receive a memory parity error, followed by a General Protection Fault?

Memory parity errors are the result of bad memory in your machine. This memory might be virtual memory, indicating a problem on your hard disk, or physical memory, indicating a bad SIMM. To check for problems with your hard disk, use a standard disk utility package such as Norton Utilities. To check for problems with physical RAM, physically rotate the SIMMs in your machine, rebooting the PC after each rotation. When the bad SIMM is placed in the lowest bank of memory, the machine does not boot at all. It is a good idea to replace the SIMM.

Platform Issues and Compatibilities

What is required to transfer VIs between platforms?

When a VI file is brought to another platform, no conversion is necessary for LabVIEW or BridgeVIEW to read it. When a VI is opened, it is recompiled for the new platform. This means you *must* include the block diagram of a VI if you want to take it to another platform. For VIs containing CINs, you must recompile the CINs on the new platform and then relink them to the ported VI. Eliminate platform-specific functions (for example, Apple Events for Macintosh, DDE for Windows, and so on) before you port a VI to another platform.

You can move VIs between platforms using networks, modems, and disks. VIs are saved in the same file format on all platforms. If you transfer VIs across a network using ftp or modem, make sure you specify a binary transfer.

If disks are the method of transfer, disk conversion utilities are required to read the disks from other platforms. Conversion utilities change the format of files stored on disk because each platform (Macintosh, Windows, Sun) saves files to disk in a different format. Most file conversion utilities not only *read* files from another platform, but also *write* files in the disk format of that platform. For example, there are utilities such as MacDisk and

TransferPro available for the PC that transfer Macintosh disks to the PC format and vice versa. On the Macintosh, DOS Mounter and Apple File Exchange are two utilities that convert files on DOS-formatted disks to the Macintosh format and vice versa. For the Sun and HP, there is PC File System (PCFS) that helps SunOS and HP-UX to read and write DOS-formatted disks.

To ease porting between platforms, you can save your VIs into a VI library. To facilitate saving all your VIs into a library, you can select the **Development Distribution** option from the Save with Options dialog box. In this way, you can save all non-`vi.lib` VIs, controls, and external subroutines to a single library.

Printing

How do I print a single control from the front panel (for example, a graph)?

To print only a graph from the front panel, create a subVI with a graph on its front panel.

1. Change the graph from an indicator to a control.

2. Open the subVI and select **Operate»Print at Completion**.

3. Assign the subVI a connector and pass the data from the graph on the main VI to the graph on the subVI.

Every time your main VI calls the subVI, it automatically prints the graph.

How can I print a string?

Use the `Serial Port Init.vi` to initialize the port where the printer is connected (`LPT1`, `LPT2`, and so on, on the PC, or the printer port on the Mac) and then the `Serial Port Write.vi` to write the string to the initialized port. The printer sees the data at the port and prints it. Doing this generally requires some knowledge of the command language of your printer, but works well for a number of applications developed by LabVIEW and BridgeVIEW users. If you are a LabVIEW user, see Appendix B, *Common Questions*, of the *LabVIEW User Manual* for more details.

(**Windows** and **UNIX**) Use the System Exec VI to print a file through a command line function. The VI is located in **Functions» Communication»AppleEvent**. For example, under Windows NT/95, if you want to print a text document, use the System Exec VI with the following command:

```
notepad /p <document name>
```

Notepad has a limit of 32K. For larger documents use a different text editor.

(**Macintosh**) You can use the AESend Print Document VI to direct the other application to print a document. The VI is located in **Functions» Communication»Apple Event**. For more details see the section *Using Alternative Printing Methods*, in Chapter 5, *Printing and Documenting VIs*. You also can use the programmatic print option as described in the previous question.

How do I print all the data on a waveform chart?

The Waveform Chart only prints the displayed data when you print the front panel. To print all the data in a chart, including that in the history buffer, first autoscale the *x*-axis of the chart by popping up on the chart, showing the chart palette and locking the *x*-axis, or setting the *x*-axis programmatically through a chart Attribute Node.

Why does LabVIEW or BridgeVIEW crash when printing?

On Windows, this crash might be related to the video driver or printer driver.

See the question concerning random crashes on your platform in the section *Error Messages and Crashes* in this appendix for information on how to proceed.

How do I use PostScript printing?

If your printer works with PostScript language, first install and select the PostScript printer driver for your printer. In LabVIEW or BridgeVIEW go to **Edit»Preferences»Printing** and select **PostScript printing**.

If you have a PostScript printer, you can take advantage of the following benefits:

- PostScript printouts reproduce the image of the screen more accurately.

- PostScript reproduces patterns and line styles more accurately.

When I select PostScript printing, why does my printout yield a pile up of printed text at the top of the page?

Printing translates the VI print data into PostScript (`.ps`) format and sends it to the Windows printer driver as PostScript text. If your printer driver does not work with PostScript printing, the printout consists of the actual text of the PostScript file instead of a graphics image. See the previous PostScript question for more information.

What do I do if the text on labels and front panel controls is clipped when the VI prints?

This occurs when the size of the font for the printer does not match the size of the font on the monitor.

*   If your printer works with PostScript language, use the PostScript printer driver for your printer to maintain higher quality printouts. See the question above for more information on PostScript printing.

*   Enlarge the labels and front panel controls so that, when the text expands because of a font mismatch, the text still fits in the control or label boundary.

*   Find a font which is the same size on both the monitor and the printer. Many printer drivers have font substitution algorithms to assist this process.

*   **(Windows)** Use bitmap printing, which makes the printout exactly match what you see on the monitor.

(Windows) How do I select bitmap printing?

Go to **Edit»Preferences** and select **Printing**. The last option on the screen is **Bitmap printing**.

Miscellaneous

What is Info-LabVIEW and how do I subscribe?

`Info-LabVIEW` is a user-sponsored and supported network of LabVIEW and users. Users can post information to a specific e-mail address (see below); the posting is then broadcast to all users subscribing to the list. Other users then might respond back to the group or perhaps directly to the individual who wrote the post.

Several power users, as well as some National Instruments employees, subscribe to the list and respond to various issues. Overall the Info-LabVIEW group has been received extremely well. Although the group is not connected with National Instruments, official responses are made on occasion to the group. Most of the time the conversations take place strictly between users.

If you want to subscribe to the list send e-mail to this address:

`info-labview-request@pica.army.mil`

There are two ways for users to receive the postings—the standard format and digest format. Standard format means you receive the postings as they occur. Digest format means you receive one package at the end of each day containing all postings for the day. Indicate digest format in the e-mail messages you send to `info-labview-request` if you want to receive one bulk message daily.

After you subscribe to the list, you receive postings from other subscribers. Traffic in the group is fairly active, from 10 to 30 postings per day. To post information to the list, send e-mail to the following address.

`info-labview@pica.army.mil`

How can I load and run VIs dynamically?

When a subVI exists in the block diagram of another VI, the subVI loads into memory as soon as the calling VI loads into memory. For memory considerations, you might want to load and unload VIs dynamically from memory during the execution of your program. You can use the VI Server feature to do this. For more information see Chapter 21, *VI Server,* in this manual.

(Macintosh) In addition to the platform-independent VI Server, you can use Apple Events on the Macintosh to dynamically load and run VIs. The VIs are located in the palette **Functions»Communication»AppleEvent** or its subpalette **LabVIEW Specific Apple Events,** and are called:

- AESend Finder Open
- AESend Open, Run, Close VI
- AESend Run VI
- AESend Close VI

How do I replace a subVI with another VI that has the same name?

LabVIEW and BridgeVIEW reference all VIs by name, and thus never load two VIs with the same name, regardless of the paths to those VIs. This includes subVIs. When you select **Replace** on a subVI to replace it with a VI of the same name, G realizes it already has a copy of the VI in memory, and replaces the VI with itself.

To load the new copy of the subVI, you must first remove the old copy from memory. The VIs currently are listed in the **Windows** menu. The easiest way to remove the subVI from memory is to close the subVI and any other VIs that call the subVI. Open the new copy of the subVI into memory, check **File»Show VI Info…** to confirm this is the version you want, and then re-open the main VI. When the VI tries to link to the subVI, it finds a copy in memory (the new version) and uses it rather than search for the previous version.

The solution to many of these problems is always to give subVIs unique names.

Loading two VIs that call two distinct subVIs with the same name causes problems.

For example, `main1.vi` and `main2.vi` both call distinct subVIs which are both named `subVI.vi`.

1. `main1.vi` is loaded into memory, `subVI.vi` is loaded in because it is called by `main1.vi`.

2. `main2.vi` is loaded into memory, linker information signals LabVIEW or BridgeVIEW to load `subVI.vi`. The application recognizes it already has `subVI.vi` loaded into memory, tries to link `main2.vi` with this `subVI.vi`. If the connector pane is exactly the same, it probably makes the link (but the behavior might be very peculiar). If the connector pane is different, you receive a bad linkage error.

Solution: Always give subVIs unique names.

How does priority affect the execution of VIs?

The G execution system can execute multiple VIs or subdiagrams in parallel. VIs might be assigned one of five priorities: background (lowest), normal, above normal, high and time critical (highest). Another priority, subroutine, forces a VI to run until completion without sharing its execution system thread with other VIs. Each execution system has a queue

of sections of code from the VIs in the system. The queue is ordered by priority such that higher priority VIs run before lower priority VIs. As long as high priority items are on the queue, lower priority items can not work their way to the head of the queue and can not execute. This means that starvation of low priority VIs is possible if higher priority VIs run continuously without pausing for a timeout or other asynchronous activity. Because of this, take care when elevating the priority of a VI.

The G execution queue has entry points for each of the five priorities. There is no entry point for subroutine priority. A subroutine priority VI begins running when called from a VI, and dominates its execution system thread until it returns to the caller VI of another priority. It might call other subroutine priority VIs, but not VIs of the other priorities. Subroutine VIs do not update their indicators, cannot perform Waits or other asynchronous operations and therefore can never be put on a queue. In this sense, once they begin running, they are the highest priority VI in an execution system.

In multithreaded systems, there are several execution systems each with a run queue and one or more threads. The threads are set to run at priorities such that the operating system can preempt a low priority thread when a higher priority thread is ready to run. The VI priority is used to select which execution system queue it is best placed upon. A subroutine VI called in a multithreaded system dominates the thread in which it is called but cannot prevent other VIs in other threads from running. This means you cannot use subroutine VIs to block out all other execution of VIs on systems with other threads at equal or higher priority.

On multithreaded systems, the user interface is a thread at normal priority. On some systems, it can be starved by VIs running at higher priority which never pause for a timeout or perform other asynchronous activity. It might not be possible to change a control to stop a high priority VI which never permits the user interface to notice the control. Because of this, take care when elevating the priority of a VI.

How can interrupts be serviced by G without using polling?

Occurrences are a method for G to make a section of code wait for a particular event. For example, you can write a CIN that spawns a process (in Windows through a DLL, for example) that generates an occurrence when an event takes place in the DLL. A particular section of diagram code is executing, and another section or VI is waiting for the event. When the event happens, it logs the occurrence in LabVIEW or BridgeVIEW, and the appropriate code begins to execute as soon as its turn on the queue comes up.

Internally, LabVIEW and BridgeVIEW look at the queue when it finishes a block of atomic code. This means if something else is executing it must complete before the occurrence, and hence the interrupt, is handled.

☞ **Note**   *There is no definite time limit on how long this might be, particularly if the VI contains CINs, which might take a very long time to complete execution.*

With nested While Loops, how do I stop both loops without anything in the outer loop executing after the inner loop stops?

Put the code you do not want to execute on the last iteration into a Case structure. It is best if the condition terminal of the inner loop is connected to the selection terminal of the Case structure.

What is Panel Order?

Edit»Panel Order determines the order of the objects on the front panel. In other words, if you were using <Tab> on the keyboard to move between objects, the key focus changes from object to object in the order determined by the panel order. Only controls can receive key focus; the <Tab> key does not switch focus to indicators. By default, the panel order is the order in which you create objects on the front panel.

With front panel data logging, the panel order determines the order in which the different objects are compacted into a cluster within the file.

How can I instruct LabVIEW or BridgeVIEW to launch a particular VI automatically?

By default, LabVIEW and BridgeVIEW launch and create an Untitled1.vi. You might want your application to launch a particular VI automatically. You also can set the VI to run when loaded, so launching LabVIEW or BridgeVIEW not only opens your VI but also starts running it. Set the appropriate **Execution Options** in **VI Setup…** to make a VI run when opened. If you specify a library (.llb) to be opened when your application is launched, it opens all VIs marked as **Top Level**. Use **File»Edit VI Library…** to mark VIs with **Top Level**. Finally, if you specify a library and no VIs are marked **Top Level**, when launched, a file dialog box prompts the user to select a VI within the specified .llb.

To tell LabVIEW or BridgeVIEW to launch a particular VI, follow the instructions below for the relevant platform.

(Windows) Select the LabVIEW or BridgeVIEW icon, select
File»Properties (in Program Manager), and change the pathname in
Command Line to point towards your VI. For example, to make LabVIEW
load `test.vi` automatically, set the **Command Line** to:

```
c:\labview\labview.exe test.vi
```

If `test.vi` is inside a library called `test.llb`, then:

```
c:\labview\labview.exe "test.llb\test.vi"
```

It might be necessary to specify the full path to the VI.
If you want to launch a VI from the command line (**Start»Run...**), the
syntax for this is

```
c:\labview\labview.exe <path to VI relative to the
labview directory>
```

For example to launch the Readme.vi inside the c:\labview\examples
directory, the command is

```
c:\labview\labview.exe examples\readme.vi
```

If the VI is located in a different path, you need to specify the full path to
the VI:

```
c:\labview\labview.exe c:\coolapp\mycool.vi
```

If a directory in the path contains spaces, you must enclose the path in
quotes:

```
c:\labview\labview.exe "c:\cool application\mycool.vi"
```

If the VI is within an LLB, you can do one of the following:

1. Specify the full path in quotes:

```
c:\labview\labview.exe "c:\coolapp\eagle.llb\mycool.vi"
```

or

2. Change the VI Setup for the VI to **Run When Opened**. Then, select
    File»Edit VI Library... and mark the VI as **Top Level**.

In this case, you only need to specify the path to the llb:

```
c:\labview\labview.exe "c:\coolapp\eagle.llb"
```

If you want LabVIEW to automatically launch a VI, modify the Shortcut
properties and specify the path in Target as shown above.

(Macintosh) There is no way to instruct the LabVIEW application to launch a particular VI on the Macintosh automatically; however, you can launch a VI by double-clicking its icon in the Finder. If you have multiple copies of LabVIEW installed on your machine, the Finder determines which copy launches when you double-click the VI (you cannot control which copy is launched). For example, the Finder might launch the Run-Time System if both the Run-Time System and Full Development System are installed. If you are running System 7 or later, you can use Drag & Drop to launch one or more VIs, or VI libraries.

(UNIX) LabVIEW responds to command-line options to launch a particular VI when opened. Therefore, you can type:

```
labview /usr/home/test.vi
```

or

```
labview /usr/home/test.llb/test.vi
```

to launch the Test VI, depending on the correct path. You can use a simple script to make a command that launches LabVIEW with a particular VI.

How do I programmatically change the entries of an enumerated type?

You cannot programmatically change the type (the strings) of an enumerated data type, just as you cannot programmatically change an integer control into a double or a string control into a path control. The strings in an Enum are a part of its data type and thus can only be changed during edit time. It is possible to read the strings of the Enum through an Attribute Node, but you cannot write them using an Attribute Node.

If you want to programmatically change the text values in the Enum, use a text ring control instead. You can use the ring control to programmatically read and write the strings through the Strings[] attribute.

How do I hide the menu bars of a VI?

All VI attributes, including whether the menu bar and the toolbar are displayed, are set through the dialog box in **VI Setup…**. To access the VI Setup dialog box, pop up on the VI icon in the upper-right corner of the front panel and select **VI Setup…**. See Chapter 6, *Setting up VIs and SubVIs*, for more details.

How do I disable mouse interrupts to improve performance?

Interrupts caused by moving the mouse or clicking it on an item take CPU time. In some extremely time-critical applications, this interrupt activity can result in a loss of data or some other problem with the application. Other than unplugging the mouse, there is no recommended method for disabling mouse interrupts. By making sure the mouse does not move during the time-critical portions of your application you can minimize its effects.

C

Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a fax-on-demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Electronic Services

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as `anonymous` and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.

Fax-on-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at 512 418 1111.

E-Mail Support (Currently USA Only)

You can submit technical support questions to the applications engineering team through e-mail at the Internet address listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

`support@natinst.com`

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call 512 795 6990. You can access these services at:

United States: 512 794 5422
Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 01 48 65 15 59
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

Country	Telephone	Fax
Australia	03 9879 5166	03 9879 6277
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Brazil	011 288 3336	011 288 8528
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 725 725 11	09 725 725 55
France	01 48 14 24 24	01 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Israel	03 6120092	03 6120095
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
United Kingdom	01635 523545	01635 523154
United States	512 795 8248	512 794 5678

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name _____

Company _____

Address _____

_____

Fax (___) _____Phone (___) _____

Computer brand_____ Model _____Processor_____

Operating system (include version number) _____

Clock speed _____MHz RAM _____MB   Display adapter _____

Mouse ___yes ___no   Other adapters installed _____

Hard disk capacity _____MB Brand_____

Instruments used _____

_____

National Instruments hardware product model _____ Revision _____

Configuration _____

National Instruments software product _____ Version _____

Configuration _____

The problem is: _____

_____

_____

_____

_____

List any error messages: _____

_____

_____

The following steps reproduce the problem: _____

_____

_____

_____

_____

Hardware and Software Configuration Form

Record the settings and revisions of your hardware and software on the line to the right of each item. Complete a new copy of this form each time you revise your software or hardware configuration, and use this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

National Instruments Products

DAQ hardware _____

Interrupt level of hardware _____

DMA channels of hardware _____

Base I/O address of hardware _____

Programming choice _____

National Instruments software version(s) _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

Other Products

Computer make and model _____

Microprocessor _____

Clock frequency or speed _____

Type of video board installed _____

Operating system version _____

Operating system mode _____

Programming language _____

Programming language version _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: *G Programming Reference Manual*

Edition Date: January 1998

Part Number: 321296B-01

Please comment on the completeness, clarity, and organization of the manual.

_____

_____

_____

_____

_____

_____

_____

If you find errors in the manual, please record the page numbers and describe the errors.

_____

_____

_____

_____

_____

_____

_____

Thank you for your help.

Name _____

Title _____

Company _____

Address _____

_____

E-Mail Address _____

Phone (___) _____ Fax (___) _____

Mail to: Technical Publications   **Fax to:**   Technical Publications
           National Instruments Corporation   National Instruments Corporation
           6504 Bridge Point Parkway   512 794 5678
           Austin, Texas 78730-5039

Glossary

Prefix	Meaning	Value
m-	milli-	10^{-3}
μ-	micro-	10^{-6}
n-	nano-	10^{-9}

Symbols

∞   Infinity.

π   Pi.

Δ   Delta. Difference. Δx denotes the value by which x changes from one index to the next.

A

absolute path   File or directory path that describes the location relative to the top level of the file system.

active window   Window that is currently set to accept user input, usually the frontmost window. The titlebar of an active window is highlighted. Make a window active by clicking it, or by selecting it from the Windows menu.

ANSI   American National Standards Institute.

array   Ordered, indexed list 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   Condition in a dataflow programming language in which the arrival of data,
dependency   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	Automated test equipment.
Attribute Node	Special block diagram nodes you can use to control the appearance and functionality of controls and indicators.
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.
autosizing	Automatic resizing of labels to accommodate text that you enter.

B

block diagram	Pictorial description or representation of a program or algorithm. In G, the block diagram, which consists of executable icons called nodes and wires that carry data between the nodes, is the source code for the VI. The block diagram resides in the block diagram window of the VI.
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	A pause in execution used for debugging. You set a breakpoint by clicking a VI, node, or wire with the Breakpoint tool from the **Tools** palette.
Breakpoint tool	Tool used to set a breakpoint on a VI, node, or wire.
broken VI	VI that cannot be compiled or run; signified by a broken arrow in the run button.

Bundle node	Function that creates clusters from various types of elements.
byte stream file	File that stores data as a sequence of ASCII characters or bytes.

C

case	One subdiagram of a Case Structure.
Case Structure	Conditional branching control structure, which executes one and only one of its subdiagrams based on its input. It is similar to 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.
caption label	Label on a front panel object used to name the object in the user interface. This name can be translated to other languages without affecting the block diagram source code. *See* also Name Label.
chart	*See* scope chart, strip chart, and sweep chart.
CIN	*See* Code Interface Node.
cloning	To make a copy of a control or some other G object by clicking the mouse button on it while pressing the `<Ctrl>` **(Windows)**; `<option>` **(Macintosh)**; `<meta>` **(Sun)**; or `<Alt>` **(HP-UX)** key and dragging the copy to its new location.
	(UNIX) You also can clone an object by clicking the object with the middle mouse button and then dragging the copy to its new location.
cluster	A set of ordered, unindexed data elements of any data type including numeric, Boolean, string, array, or cluster. The elements must be all controls or all indicators.
cluster shell	Front panel object that contains the elements of a cluster.
Code Interface Node (CIN)	Special block diagram node through which you can link conventional, text-based code to a VI.
coercion	The automatic conversion G performs to change the numeric representation of a data element.

coercion dot	Glyph on a node or terminal indicating that the numeric representation of the data element changes at that point.
Color Copy tool	Copies colors for pasting with the Color tool.
Color tool	Tool you use to set foreground and background colors.
compile	Process that converts high-level code to machine-executable code. VIs are compiled automatically before they run for the first time after creation or alteration.
conditional terminal	The terminal of a While Loop that contains a Boolean value that determines whether the VI performs another iteration.
connector	Part of the VI or function node that contains its input and output terminals, through which data passes to and from the node.
connector pane	Region in the upper right corner of a front panel window that displays the VI terminal pattern. It underlies the icon pane.
constant	*See* universal constant and user-defined constant.
continuous run	Execution mode in which a VI is run repeatedly until the operator stops it. You enable it by clicking the Continuous Run button.
control	Front panel object for entering data to a VI interactively or to a subVI programmatically.
control flow	Programming system in which the sequential order of instructions determines execution order. Most conventional text-based programming languages, such as C, Pascal, and BASIC, are control flow languages.
Controls palette	Palette containing front panel controls and indicators.
conversion	Changing the type of a data element.
count terminal	The terminal of a For Loop whose value determines the number of times the For Loop executes its subdiagram.
CPU	Central Processing Unit.
current VI	VI whose front panel, block diagram, or Icon Editor is the active window.
custom PICT controls	Controls and indicators whose parts can be replaced by graphics and indicators you supply.

D

data acquisition	DAQ. Process of acquiring data, typically from A/D or digital input plug-in boards.
data dependency	Condition in a dataflow programming language in which a node cannot execute until it receives data from another node. *See also* artificial data dependency.
data flow	Programming system consisting of executable nodes in which nodes execute only when they have received all required input data and produce output automatically when they have executed. G is a dataflow system.
data logging	Generally, to acquire data and simultaneously store it in a disk file. G 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 G object.
destination terminal	*See* sink terminal.
dialog box	An interactive screen with prompts in which you describe 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

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	Debugging feature that animates VI execution to illustrate the data flow in the VI.
external routine	*See* shared external routine.

F

FFT	Fast Fourier transform.
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 too cumbersome to build in block diagram form.
frame	Subdiagram of a Sequence Structure.
free label	Label on the front panel or block diagram that does not belong to any other object.
front panel	The interactive user interface of a VI. Modeled from the front panel of physical instruments, it is composed of switches, slides, meters, graphs, charts, gauges, LEDs, and other controls and indicators.
function	Built-in execution element, comparable to an operator, function, or statement in a conventional language.
Functions palette	Palette containing block diagram structures, constants, communication features, and VIs.

G

G	Graphical programming language used in LabVIEW and BridgeVIEW.
glyph	A small picture or icon.
GPIB	General Purpose Interface Bus is the common name for the communications interface system defined in ANSI/IEEE Standard 488.1-1987 and ANSI/IEEE Standard 488.2-1987. Hewlett-Packard, the inventor of the bus, calls it the HP-IB.
graph control	Front panel object that displays data in a Cartesian plane.

H

handle	Pointer to a pointer to a block of memory; handles reference arrays and strings. An array of strings is a handle to a block of memory containing handles to strings.
Help window	Special window that displays the names and locations of the terminals for a function or subVI, the description of controls and indicators, the values of universal constants, and descriptions and data types of control attributes. The window also accesses the **Online Reference** for G.
hex	Hexadecimal. A base-16 number system.
hierarchical palette	Menu that contains palettes and subpalettes.
Hierarchy window	Window that graphically displays the hierarchy of VIs and subVIs.
housing	Nonmoving part of front panel controls and indicators that contains sliders and scales.
Hz	Hertz. Cycles per second.

I

icon	Graphical representation of a node on a block diagram.
Icon Editor	Interface similar to that of a paint program for creating VI icons.

icon pane	Region in the upper right corner of the front panel and block diagram that displays the VI icon.
IEEE	Institute for Electrical and Electronic Engineers.
indicator	Front panel object that displays output.
Inf	Digital display value for a floating-point representation of infinity.
inplace execution	Ability of a function or VI to reuse memory instead of allocating more.
instrument driver	VI that controls a programmable instrument.
I/O	Input/Output. The transfer of data to or from a computer system involving communications channels, operator input devices, and/or data acquisition and control interfaces.
iteration terminal	The terminal of a For Loop or While Loop that contains the current number of completed iterations.

L

label	Text object used to name or describe other objects or regions on the front panel or block diagram.
Labeling tool	Tool used to create labels and enter text into text windows.
LED	Light-emitting diode.
legend	Object owned by a chart or graph that display the names and plot styles of plots on that chart or graph.

M

marquee	A moving, dashed border that surrounds selected objects.
matrix	Two-dimensional array.
MB	Megabytes of memory.
menu bar	Horizontal bar that contains names of main menus.
mnemonic	A string associated with an integer value.

modular programming	Programming that uses interchangeable computer routines.
multithreading	A multithreaded application is one which executes several different threads of execution independently. On a multiple processor computer, the different threads might be running on different processors simultaneously.

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).
Name Label	The label of a front-panel object used to name the object as well as distinguish it from other objects. Used on the terminal, local variables, and attribute nodes that are part of the object. *See* also caption label.
nodes	Execution elements of a block diagram consisting of functions, structures, and subVIs.
nondisplayable characters	ASCII characters that cannot be displayed, such as null, backspace, tab, and so on.
not-a-path	A predefined value for the path control that means the path is invalid.
not-a-refnum	A predefined value that means the refnum is invalid.
numeric controls and indicators	Front panel objects used to manipulate and display or input and output numeric data.

O

object	Generic term for any item on the front panel or block diagram, including controls, nodes, wires, and imported pictures.
Object pop-up menu tool	Tool used to access the pop-up menu for an object.
one-dimensional	Having one dimension, as in the case of an array that has only one row of elements.
Operating tool	Tool used to enter data into controls as well as operate them. Resembles a pointing finger.

P

palette	Menu of pictures that represent possible options.
pixmap	A standard format for storing pictures in which each pixel is represented by a color value. A bitmap is a black and white version of a pixmap.
platform	Computer and operating system.
plot	A graphical representation of an array of data shown either on a graph or 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 an object with the right mouse button (on Windows and UNIX platforms) or while holding down the command key (Macintosh).
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, select, and resize objects.
probe	Debugging feature for checking intermediate values in a VI.
Probe tool	Tool used to create probes on wires.
programmatic printing	Automatic printing of a VI front panel after execution.
pseudocode	Simplified language-independent representation of programming code.
pull-down menus	Menus accessed from a menu bar. Pull-down menu options are usually general in nature.

R

race condition	Occurs when two or more pieces of code that execute in parallel are changing the value of the same shared resource, typically a global or local variable.
reentrant execution	Mode in which calls to multiple instances of a subVI can execute in parallel with distinct and separate data storage.

refnum	An identifier that G associates with a file when you open it. You use the file refnum to indicate that you want a function or VI to perform an operation on the open file.
representation	Subtype of the numeric data type, of which there are signed and unsigned byte, word, and long integers, as well as single-precision, double-precision, and extended-precision floating-point numbers, both real and complex.
resizing handles	Angled handles on the corner of objects that indicate resizing points.
ring control	Special numeric control that associates 32-bit integers, starting at 0 and increasing sequentially, with a series of text labels or graphics.

S

scalar	Number capable of being represented by a point on a scale. A single value as opposed to an array. Scalar Booleans and clusters are explicitly singular instances of their respective data types.
scale	Part of mechanical-action, chart, and graph controls and indicators that contains a series of marks or points at known intervals to denote units of measure.
scope chart	Numeric indicator modeled on the operation of an oscilloscope.
Scroll tool	Tool used to scroll windows.
sequence local	Terminal that passes data between the frames of a Sequence Structure.
Sequence Structure	Program control structure that executes its subdiagrams in numeric order. Commonly used to force nodes that are not data-dependent to execute in a desired order.
shared external routine	Subroutine that can be shared by several CIN code resources.
shift register	Optional mechanism in loop structures used to pass the value of a variable from one iteration of a loop to a subsequent iteration.
sink terminal	Terminal that absorbs data. Also called a destination terminal.
slider	Moveable part of slide controls and indicators.
source terminal	Terminal that emits data.

string controls and indicators	Front panel objects used to manipulate and display or input and output text.
strip chart	A numeric plotting indicator modeled after a paper strip chart recorder, which scrolls as it plots data.
structure	Program control element, such as a Sequence, Case, For Loop, or While Loop.
stub VI	A nonfunctional prototype of a subVI. It has inputs and outputs, but is incomplete. It is used during early planning stages of a VI design as a place holder for future VI development.
subdiagram	Block diagram within the border of a structure.
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.
tip strip	Small yellow text banners that identify the terminal name and make it easier to identify function and node terminals for wiring.
tool	Special cursor you can use to perform specific operations.
toolbar	Bar containing command buttons that you can use to run and debug VIs.
Tools palette	Palette containing tools you can use to edit and debug front panel and block diagram objects.
top-level VI	VI at the top of the VI hierarchy. This term distinguishes the VI from its subVIs.
two-dimensional	Having two dimensions, such as an array having several rows and columns.

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, π.
user-defined constant	Block diagram object that emits a value you set.
UUT	Unit under test.

V

VI	*See* virtual instrument.
VI class	A reference to a virtual instrument that allows access to VI properties and methods.
VI application class	A reference to an application capable of loading virtual instruments that allows access to the application properties and methods.
VI library	Special file that contains a collection of related VIs for a specific use.
VI Server	Mechanism for controlling VIs and G-applications programmatically. Can also be used to control VIs or G-applications remotely.
VI string file	A tagged text file containing all localization strings found in the front panel of a VI.
virtual instrument	Program in LabVIEW or BridgeVIEW; so-called because it models the appearance and function of a physical instrument.

W

While Loop	Loop structure that repeats a section of code until a condition is met. Comparable to a Do loop or a Repeat-Until loop in conventional programming languages.
wire	Data path between nodes.
Wiring tool	Tool used to define data paths between source and sink terminals.

Index

Numbers and Symbols

A