LabVIEW™

Advanced Programming Techniques

SECOND EDITION

Rick Bitter
Taqi Mohiuddin
Matt Nawrocki

CD-ROM INCLUDED

CRC Press
Taylor & Francis Group

LabVIEW™
Advanced Programming Techniques

SECOND EDITION

LabVIEW™ Advanced Programming Techniques

SECOND EDITION

Rick Bitter
Motorola, Schaumburg, Illinois
Taqi Mohiuddin
Mindspeed Technologies, Lisle, Illinois
Matt Nawrocki
Motorola, Schaumburg, Illinois

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an informa business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-10: 0-8493-3325-3 (Hardcover)
International Standard Book Number-13: 978-0-8493-3325-5 (Hardcover)

Library of Congress Cataloging-in-Publication Data

Bitter, Rick.
LabVIEW : advanced programming techniques / Richard Bitter, Taqi Mohiuddin, Matthew R. Nawrocki. -- 2nd ed.
p. cm.
ISBN 0-8493-3325-3 (alk. paper)
1. Computer programming. 2. LabVIEW. 3. Computer graphics. I. Mohiuddin, Taqi. II. Nawrocki, Matt. III. Title.

QA76.6.B5735 2006
005.1--dc22 2006044686

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Preface and Acknowledgments

As the power of the standard personal computer has steadily evolved, so have the capabilities of LabVIEW. LabVIEW has simplified the working lives of thousands of scientists, engineers, and technicians, and has increased their productivity. Automation has reduced the costs and increased the manufacturing outputs of factories around the world. Cycle times for product development have been shortened and quality of many products has steadily improved. LabVIEW does not get credit for all of these improvements, but has without question played a valuable role in many organizations for accomplishing these goals.

In our earlier experiences with LabVIEW, we found that adequate coverage of key topics was lacking. Subjects that are useful to users without a formal background in computer science such as approaches to software development, exception handling, and state machines were very difficult to find. In addition, newer areas such as multi-threading and ActiveX are even harder to locate and sometimes documentation is non-existent. Part of our intent in this book is to cover these topics that are difficult to find in other books on LabVIEW.

The chapters in this book are written in a manner that will allow readers to study the topic of interest without having to read the contents in sequential order. Users of LabVIEW with varying levels of expertise will find this book beneficial.

Proficiency with a programming language requires an understanding of the language constructs and the tools needed to produce and debug code. The first two chapters provide an overview of LabVIEW's Integrated Development Environment, programming constructs, and main features. These chapters are meant to supplement LabVIEW's documentation, and provide some good background information for programmers new to the language.

Effective programmers have an understanding of programming techniques that are applicable to a large number of programming problems. Programming tools such as state machines that simplify logic of handling various occurrences and the use of instrument drivers are two such programming tools. Exception handling is left out of more applications than we want to discuss (including some of our own), but we have included a chapter specifically on exception handling in LabVIEW.

Advanced programmers understand the operation of the language they are working with and how it interacts with the system. We present a chapter on multi-threading's impact on LabVIEW. Version 5.0 was LabVIEW's debut into the world of multi-threaded capable programming languages. A number of the issues that occur with multi-threading programming were abstracted from the programmer, but a working knowledge of muti-threaded interactions is needed.

Object Oriented Programming (OOP) is commonly employed in languages such as C++ and Java. LabVIEW programmers can realize some of the benefits to such an approach as well. We define key terms often used in OOP, give an explanation of object analysis and introduce you to applying these concepts within a LabVIEW environment.

We also present two chapters on ActiveX and .NET. An explanation of related technologies such as Component Object Model (COM) and Object Linking and Embedding (OLE) is provided along with the significance of ActiveX. A description on the use of ActiveX in LabVIEW applications is then provided. We follow this up with several useful examples of ActiveX/.NET such as embedding a browser on the front panel, use of the tree view control, and automating tasks with Microsoft Word, Excel, and Access.

This book would not have been possible without the efforts of many individuals. First, we want to thank our friends at National Instruments. Ravi Marawar was invaluable in his support for the completion of this book. We would also like to thank Norma Dorst and Steve Rogers for their assistance.

Our publishers at CRC Press, Nora and Helena have provided us with guidance from the first day we began working on this edition until its completion. We haven't forgotten about the first edition publishing support of Dawn and Felicia. If not for their efforts, this book may not have been successful enough to warrant a second edition.

A special thanks to Tim Sussman, our colleague and friend. He came through for us at times when we needed him. Also thanks to Greg Stehling, John Gervasio, Jeff Hunt, Ron Wegner, Joe Luptak, Mike Crowley, the Tellabs Automation team (Paul Mueller, Kevin Ross, Bruce Miller, Mark Yedinak, and Purvi Shah), Ted Lietz, and Waj Hussain (if it weren't for Waj, we would have never written the papers which got us to writing this book).

Finally, we owe many thanks for the love and support of our families. They had to put up with us during the many hours spent on this book. We would like to begin by apologizing to our wives for the time spent working on the second edition that could not be spent on the households! A special appreciation goes out to the loving wives who dealt positively with our absences — Thanks to Claire, Sheila, and Jahanara! Thank you moms and dads: Auradker and Mariam Mohiuddin, Rich and Madalyn Bitter, Barney and Veronica Nawrocki. For moral support we thank Jahanara, Mazhar, Tanweer, Faheem, Firdaus, Aliyah and Asiya, Matt Bitter, Andrea and Jerry Lehmacher; Sheila, Reilly, Andy, Corinne, Mark, and Colleen Nawrocki, Sue and Steve Fechtner.

The Authors

Rick Bitter graduated from the University of Illinois at Chicago in 1994. He has presented papers at Motorola and National Instruments-sponsored symposia. Rick currently develops performance testing applications as a Senior Software Engineer.

Taqi Mohiuddin graduated in electrical engineering from the University of Illinois at Chicago in 1995. He obtained his MBA from DePaul University. He has worked with LabVIEW since 1995, beginning with version 3.1, ranging in various telecommunications applications. He has presented papers on LabVIEW at Motorola and National Instruments conferences.

Matt Nawrocki graduated from Northern Illinois University in 1995. He has written papers and has done presentations on LabVIEW topics at Motorola, National Instruments, and Tellabs.

Contents

1 Introduction to LabVIEW

Programmers develop software applications every day in order to increase efficiency and productivity in various situations. LabVIEW, as a programming language, is a powerful tool that can be used to help achieve these goals. LabVIEW (Laboratory Virtual Instrument Engineering Workbench) is a graphically-based programming language developed by National Instruments. Its graphical nature makes it ideal for test and measurement (T&M), automation, instrument control, data acquisition, and data analysis applications. This results in significant productivity improvements over conventional programming languages. National Instruments focuses on products for T&M, giving them a good insight into developing LabVIEW.

This chapter will provide a brief introduction to LabVIEW. Some basic topics will be covered to give you a better understanding of how LabVIEW works and how to begin using it. This chapter is not intended to teach beginners LabVIEW programming thoroughly. Those wishing to learn LabVIEW should consider attending a National Instruments LabVIEW Basics course. Relevant information on the courses offered, schedules, and locations can be found at www.ni.com/training. If you have prior experience with LabVIEW, you can skip this chapter and proceed to the advanced chapters.

First, VIs and their components will be discussed, followed by LabVIEW's dataflow programming paradigm. Then, several topics related to creating VIs will be covered by explaining the front panel and block diagram. The chapter will conclude with descriptions of icons and setting preferences.

1.1 VIRTUAL INSTRUMENTS

Simply put, a Virtual Instrument (VI) is a LabVIEW programming element. A VI consists of a front panel, block diagram, and an icon that represents the program. The front panel is used to display controls and indicators for the user, and the block diagram contains the code for the VI. The icon, which is a visual representation of the VI, has connectors for program inputs and outputs.

Programming languages such as C and BASIC use functions and subroutines as programming elements. LabVIEW uses the VI. The front panel of a VI handles the function inputs and outputs, and the code diagram performs the work of the VI. Multiple VIs can be used to create large scale applications, in fact, large scale applications may have several hundred VIs. A VI may be used as the user interface or as a subroutine in an application. User interface elements such as graphs are easily accessed, as drag-and-drop units in LabVIEW.

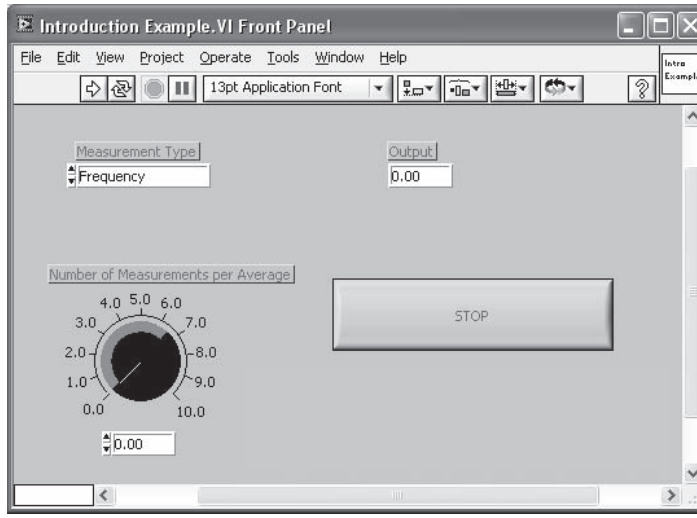


FIGURE 1.1

1.1.1 The Front Panel

Figure 1.1 illustrates the front panel of a LabVIEW VI. It contains a knob for selecting the number of measurements per average, a control for selecting the measurement type, a digital indicator to display the output value, and a stop button. An elaborate front panel can be created without much effort to serve as the user interface for an application. Front panels and LabVIEW's built-in tools are discussed in more detail in Section 1.5.

1.1.2 Block Diagram

Figure 1.2 depicts the block diagram, or source code, that accompanies the front panel in Figure 1.1. The outer rectangular structure represents a While loop, and the inner one is a case structure. The icon in the center is a VI, or subroutine, that takes the number of measurements per average as input and returns the frequency value as the output. The orange line, or wire, represents the data being passed from the control into the VI. The selection for the measurement type is connected, or wired to the case statement to determine which case is executed. When the stop button is pressed, the While loop stops execution. This example demonstrates the graphical nature of LabVIEW and gives you the first look at the front panel, block diagram, and icon that make up a Virtual Instrument. Objects and structures related to the code diagram will be covered further in Section 1.6.

LabVIEW is not an interpreted language; it is compiled behind the scenes by LabVIEW's execution engine. Similar to Java, the VIs are compiled into an executable code that LabVIEW's execution engine processes during runtime. Every time a change is made to a VI, LabVIEW constructs a wire table for the VI. This wire table identifies elements in the block diagram that have inputs needed for that element

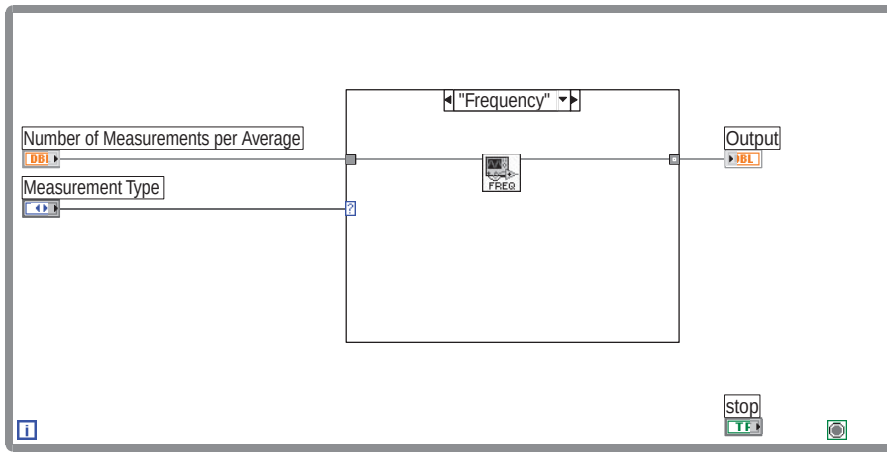


FIGURE 1.2

to run. Elements can be primitive operators such as addition, or more complex such as a subVI. If LabVIEW successfully constructs all the wire tables, you are presented a solid arrow indicating that the VIs can be executed. If the wire table cannot be created, then a broken arrow is presented for the VIs with a problem, and also for each VI loaded in memory that requires that VI for execution. LabVIEW runs in several subsystems, which will be described throughout this book. All that we need to understand now is that the main execution subsystem compiles diagrams while you write them. This allows programmers to write code and test it without needing to wait for a compiling process, and programmers do not need to worry about execution speed because the language is not interpreted.

The wire diagrams that are constructed do not define an order in which elements are executed. This is an important concept for advanced programmers to understand. LabVIEW is a dataflow-based language, which means that elements will be executed in a somewhat arbitrary order. LabVIEW does not guarantee which order a series of elements is executed in if they are not dependent on each other. A process called arbitrary interleaving is used to determine the order elements are executed in. You may force an order of execution by requiring that elements require output from another element before execution. This is a fairly common practice, and most programmers do not recognize that they are forcing the order of execution. When programming, it will become obvious that some operations must take place before others can. It is the programmer's responsibility to provide a mechanism to force the order of execution in the code design.

1.1.3 Executing VIs

A LabVIEW program is executed by pressing the arrow or the Run button located in the palette along the top of the window. While the VI is executing, the Run button changes to a black color as depicted in Figure 1.3. Note that not all of the items in the palette are displayed during execution of a VI. As you proceed to the right along

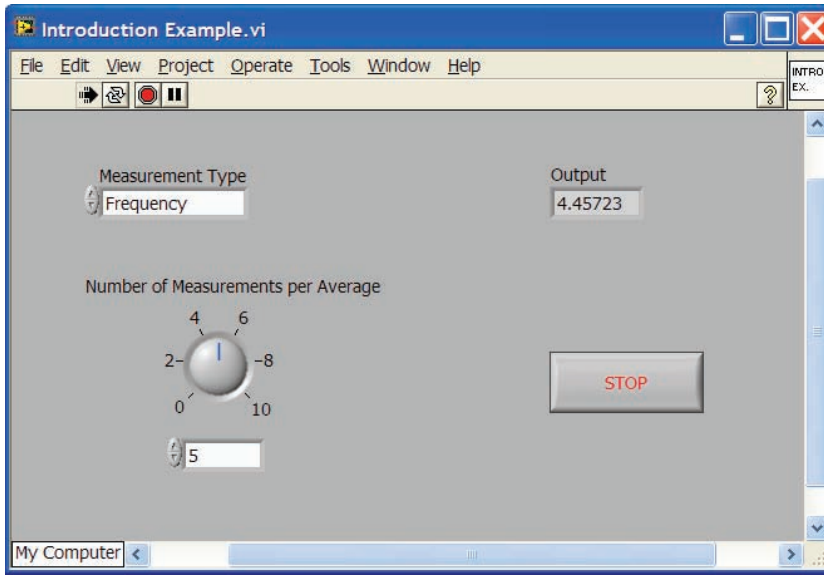


FIGURE 1.3

the palette, you will find the Continuous Run, Stop, and Pause buttons. If you compare Figures 1.1 and 1.3, the last three buttons in Figure 1.1 disappear in Figure 1.3. These buttons are used for alignment of objects on the panel or diagram, and are not available while a program is running. VIs are normally run from the front panel; however, they can also be executed from the block diagram. This allows the programmer to run the program and utilize some of the other tools that are available for debugging purposes.

If the Run button appears as a broken arrow, this indicates that the LabVIEW program or VI cannot compile because of programming errors. When all of the errors are fixed, the broken Run button will be substituted by the regular Run button. LabVIEW has successfully compiled the diagram. While editing or creating a VI, you may notice that the palette displays the broken Run button. If you continue to see this after editing is completed, press the button to determine the cause of the errors. An Error List window will appear displaying all of the errors that must be fixed before the VI can compile. Debugging techniques are discussed further in Chapter 6, which covers exception handling.

The palette contains four additional buttons on the block diagram that are not available from the front panel. These are typically used for debugging an application. The button with the lightbulb is for Execution Highlighting and the three following it are used for stepping through the code. Figure 1.4 shows the code diagram with Execution Highlighting activated. You can see bubbles that represent the data flowing along the wire, from one block to the next. You can step through the code as needed when the Pause button is used in conjunction with Execution Highlighting. As stated earlier, debugging techniques will be covered in Chapter 6.

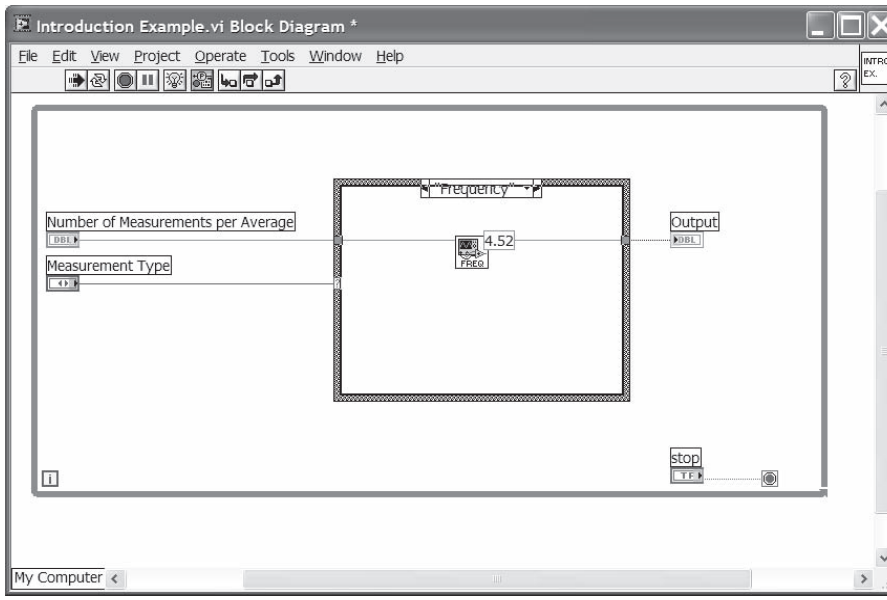


FIGURE 1.4

1.1.4 LabVIEW File Extensions

LabVIEW programs utilize the .vi extension. However, multiple VIs can be saved into library format with the .llb extension. Libraries are useful for grouping related VIs for file management. When loading a particular VI that makes calls to other VIs, the system is able to find them quickly. Using a library has benefits over simply using a directory to group VIs. It saves disk space by compressing VIs, and facilitates the movement of VIs between directories or computers. When saving single VIs, remember to add the .vi extension. If you need to create a library for a VI and its subVIs, you will need to create a source distribution using the LabVIEW Project. If you want to create a new library starting with one VI, you can use Save or Save As. Then select New VI Library from the dialog box. The File Manager can then be used to add or remove VIs from a library.

1.2 LABVIEW PROJECTS

Among other features in LabVIEW 8, the one you should be interacting with daily is the project view. LabVIEW's new project view provides a convenient interface to access everything in a LabVIEW project. Historically, locating all the Vis in an application has required the use of the hierarchy window, but that does not locate some things like LabVIEW libraries and configuration of the application builder. The project explorer provides a tree-driven list of all of these. The set of VI sources and libraries are shown in the first major breakdown: the Source tree. Information related to compilation and installation of an application are kept in

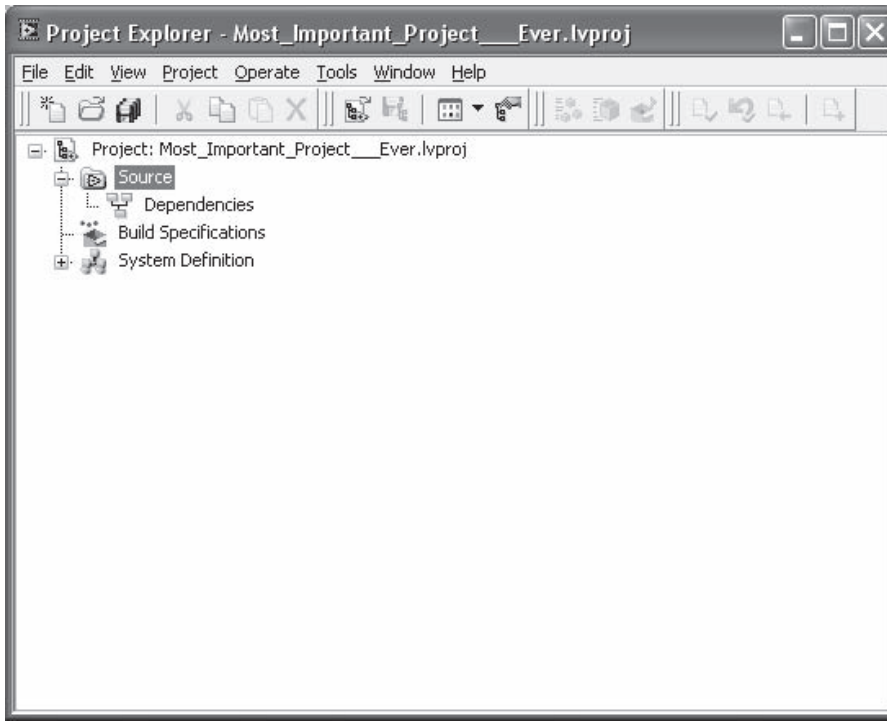


FIGURE 1.5

the second branch of the tree: Build Specifications. Information relating to the target machine environment you are building an application to is located in the last branch: System Definition. Applications that use the same operating system as the development platform will not find the System Definition folder to be of value. If a compile target is something like a Palm Pilot, then this folder is where definitions specific to a Palm based target would be configured. The project window is shown in Figure 1.5.

Among other things worth noting on the project explorer window is the toolbar, which contains buttons to create, save, and save all VIs in the application; compile; the standard cut, copy, and paste buttons; buttons to support compilation of VIs; and buttons to support source code control tools. All of these features will be elaborated on in Chapters 2 and 4.

In general, most work will be done in the Sources branch which provides a listing of all VIs and variables in the project. The Dependencies section is for VIs, DLLs, and project libraries that are called statically by a VI.

1.3 HELP

For beginning users of LabVIEW, there are various sources for assistance to aid in learning the language. Because this book is not a comprehensive guide for begin-

ners, this section was prepared to reveal some of these sources. LabVIEW's built-in help tools will be shown first, followed by outside references and Websites. LabVIEW's online reference is an excellent source of information on the operation of various LabVIEW elements, error code definitions, and programming examples. Few languages can boast of having an online help system that is put together as well as LabVIEW's.

1.3.1 Built-in Help

The first tool that is available to the user is the Simple Help. This is enabled by selecting this item from the Help pull-down menu. When selected, it activates a balloon type of help. If the cursor is placed over the particular button, for example, a small box pops up with its description. This description contains information such as the inputs and outputs the VI accepts in addition to a short text description of what the VI does. Balloon help is available for all wire diagram elements, including primitive elements, National Instruments-written VIs, and user-developed VIs. This tool is beneficial when first working with LabVIEW. It is also helpful when running VIs in single-stepping mode to find out what each of the step buttons will execute.

The Help window will probably be the most utilized help tool available. It is also activated from the Help pull-down menu by selecting Show Help (Ctrl+H). The Help window displays information on most controls, indicators, functions, constants, and subVIs. The type of information displayed varies depending on the object over which the cursor is located. For many of LabVIEW's functions, descriptions are provided along with inputs, outputs, and default values. When the cursor is placed over an icon of a VI that a user has created, that user must input the relevant description to be displayed by the Help window. The same is true for specific controls and indicators used in an application. This is an element of good documentation practices, which is explained further in Chapter 6.

Figure 1.6 shows the Help window as it appears when the cursor is placed over the "In Range?" function. A brief description of the function is provided in the

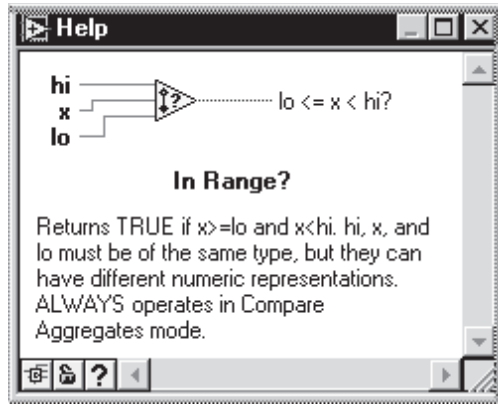


FIGURE 1.6

TABLE 1.1
Websites

http://www.ni.com/support	Technical support and contact information
http://www.ni.com/devzone/idnet/default.htm	Instrument drivers for more than 5000 instruments
http://www.ni.com/support/techdocs.htm	Technical documents, application notes, knowledge base (searchable database), product manuals
mailto://info-labview-on@labview.nhmfl.gov	Submit request for subscription to LabVIEW email user group

window along with the inputs and outputs. The three buttons located in the bottom left corner of the window are used for displaying the simple/detailed diagram, locking help on a specific object, and launching the Online Help for that topic.

The Online Help or Reference can be accessed from the Help menu also. The help files are normally installed with LabVIEW if you choose the typical installation. If you perform a custom installation of LabVIEW, you must ensure that the appropriate box is checked for help files. The Online Reference covers introduction material, overview, information on functions, and advanced topics. It also has a searchable index and word list for specific instances of key words.

1.3.2 Websites

Several other sources are also available for help on LabVIEW-related topics. National Instruments' website offers help through online technical support, documents, free downloads, product demonstrations, the instrument driver network, and the Developer Zone. National Instruments has continuously expanded its online resources, and the result is a full fledged support center. Table 1.1 lists the major websites that will be of value.

1.4 DATA FLOW PROGRAMMING

LabVIEW applications execute based on data flow. LabVIEW applications are broken up into nodes and wires; each element in a diagram that has input or output is considered a node. The connection points between nodes are wires. A node can be a simple operation such as addition, or it can be a very complicated operation like a subVI that contains internal nodes and wires. The collection of nodes and wires comprise the wire diagram. Wire diagrams are derived from the block diagrams and are used by LabVIEW's compiler to execute the diagrams. The wire diagrams are hidden from the programmer; they are an intermediate form used by the compiler to execute code. While you program, the compiler is behind the scenes verifying that diagrams are available to execute. LabVIEW applications that are built using the Application Builder use the execution engine as if LabVIEW were still being used to run the VIs.

A node can be executed when all inputs that are necessary have been applied. For example, it is impossible for an addition operation to happen unless both numbers

to be added are available. One of these numbers may be an input from a control and would be available immediately, where the second number is the output of a VI. When this is the case, the addition operation is suspended until the second number becomes available. It is entirely possible to have multiple nodes receive all inputs at approximately the same time. Data flow programming allows for the tasks to be processed more or less concurrently. This makes multitasking code diagrams extremely easy to design. Parallel loops that do not require inputs will be executed in parallel as each node becomes available to execute. Multitasking has been an ability of LabVIEW since Version 1.0. Multitasking is a fundamental ability to LabVIEW that is not directly available in languages like C, Visual Basic, and C++. When multiple nodes are available to execute, LabVIEW uses a process called arbitrary interleaving to determine which node should be executed first. If you watch a VI in execution highlighting mode and see that nodes execute in the desired order, you may be in for a rude shock if the order of execution is not always the same. For example, if three addition operations were set up in parallel using inputs from user controls, it is possible for eight different orders of execution. Similar to many operating systems' multithreading models, LabVIEW does not make any guarantees about which order parallel operations can occur.

Often it is undesirable for operations to occur in parallel. The technique used to ensure that nodes execute in a programmer-defined order is forcing the order of execution. There are a number of mechanisms available to a LabVIEW programmer to force the order of execution. Using error clusters is the easiest and recommended method to guarantee that nodes operate in a desired order. Error Out from one subVI will be chained to the Error In of the next VI. This is a very sensible way of controlling the order of execution, and it is essentially a given considering that most programmers should be using error clusters to track the status of executing code. Another method of forcing the order of execution is to use sequence diagrams; however, this method is not recommended. Sequence diagrams are basically LabVIEW's equivalent of the GOTO statement. Use sequences only when absolutely necessary, and document what each of the frames is intended to do.

Most VIs have a wire diagram; the exceptions are global variables and VIs with subroutine priority. Global variables are memory storage VIs only and do not execute. Subroutine VIs are special cases of a VI that does not support dataflow. We will discuss both of these types of VIs later. LabVIEW is responsible for tracking wire diagrams for every VI loaded into memory.

Unless options are set, there will be exactly one copy of the wire diagram in memory, regardless of the number of instances you have placed in code diagrams. When two VIs need to use a common subVI, the VIs cannot execute concurrently. The data and wire diagram of a VI can only be used in a serial fashion unless the VI is made reentrant. Reentrant VIs will duplicate their wire diagrams and internal data every time they are called.

1.5 MENUS AND PALETTES

LabVIEW has two different types of menus that are used during programming. The first set is visible in the window of the front panel and diagram. On the Macintosh,

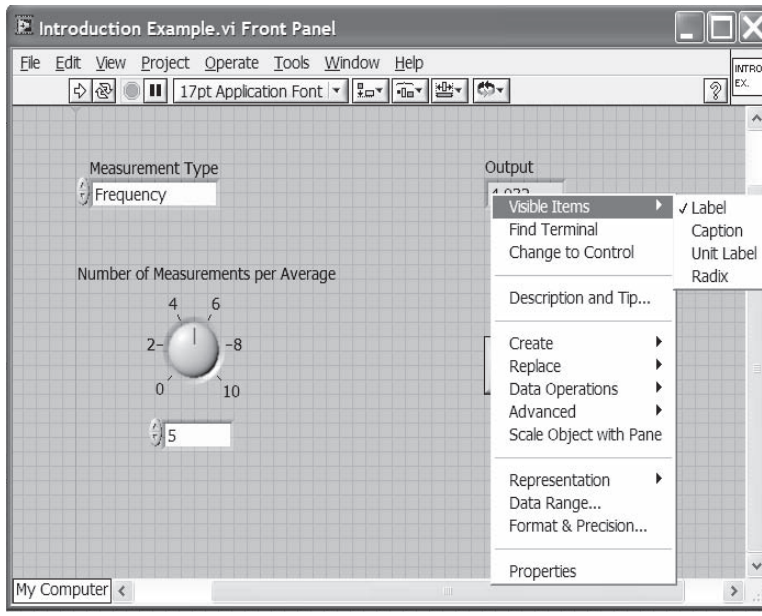


FIGURE 1.7

they are visible along the menu bar when the application is active. These are typical pull-down menus similar to other applications.

The second set of menus are called *pop-up menus* (also referred to as *popping up*). Pop-up menus are made to appear by right clicking and holding down. Macintosh users must hold down the apple key while pressing the mouse button down. The pop-up menu that appears when the cursor is on a blank part of the front panel or block diagram is the Controls palette. Similarly, the Functions palette appears on the block diagram. You can select specific objects on the front panel or block diagram and pop up on them. The menus that appear allow you to customize, modify, or perform other actions on the object. These menus can vary depending on the object that you pop up on. Figure 1.7 shows the pop menu that appears for a digital indicator.

FIGURE 1.8

The Tools palette is made visible by selecting Show Tools Palette from the Windows pull-down menu from either the front panel or block diagram. Figure 1.8 displays the movable Tools palette. The first tool is known as the Operating tool. This is used for editing numbers and text as well as changing values on controls. The arrow represents the Position-

TABLE 1.2
Shortcuts

Shortcut/Key Combination	Description	Menu Item
Tab	Allows you to switch to most common tools without accessing palette.	None
Ctrl, Option, O (Windows, Macintosh, Sun)	Allows duplication of objects. Hold down key, click on object, and drag to new location.	None
Ctrl + E	Lets you toggle between front panel and block diagram.	Show Panel/Show Diagram
Ctrl + H	Displays Help window and closes it.	Show Help
Ctrl + B	Deletes bad wires from code.	Remove Bad Wires
Ctrl + Z	Undo last action.	Undo
Ctrl + R	Begins execution of VI.	Run

ing tool for selecting, positioning, and resizing objects on the front panel or block diagram. Next is the Labeling tool for editing text and creating labels. The Wiring tool is depicted by the spool and is used for wiring data terminals. The Object Pop-up tool is located under the arrow. This is exercised for displaying the pop-up menu as an alternative to clicking the right mouse button. Next to this is the tool for scrolling through the window. The tool for setting and clearing breakpoints is located under the wiring tool. The probe tool is used with this when debugging applications. Debugging tools and techniques are explained further in Chapter 6. Finally, at the bottom is the paintbrush for setting colors, and the tool for getting colors is right above it.

LabVIEW incorporates shortcut key combinations that are equivalent to some of the pull-down menu selections. The shortcuts are displayed next to the items in the menu. The key combinations that are most helpful while you are programming with LabVIEW are listed in Table 1.2. There are also some shortcuts that are not found in the menus. For example, you can use the Tab key to move through the Tools palette. This is a quick way to change to the tool you need. The spacebar lets you toggle between the Positioning tool and the Operating tool. The normal key combinations used in Windows and Macintosh for save, cut, copy, and paste are also valid.

1.6 FRONT PANEL CONTROLS

Numerous front panel controls are available in LabVIEW for developing your applications. The Controls palette (shown in Figure 1.9) appears when you make the appropriate selection in the Windows menu. The controls are grouped into categories in a tree. Categories now include things like the modern control palette, the classic control palette, and specific use selections such as express VIs, and application control references. The subpalettes have a lock in the top left corner to keep the window visible while you are working with the controls. When creating a

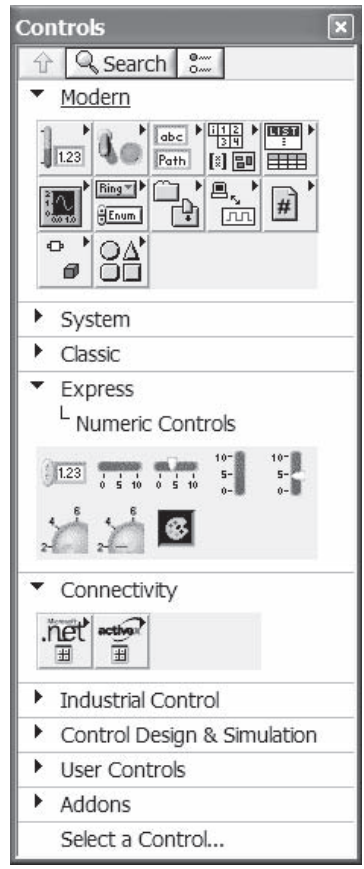


FIGURE 1.9

VI, controls can be simply dragged from the palettes and dropped on the front panel. A terminal, representing the control on the block diagram, then appears for use according to the program. Controls are basically variables that can be manipulated in the code. The following subsections will briefly describe the various control palettes. The Connectivity palette includes .NET and Active X references and will be described in Chapter 8.

1.6.1 User Control Sets

The first three branches in the control tree are modern, system, and classic. These three sections contain all the controls that an application user would interact with such as data entry controls and file path controls. The controls will behave and present data according to the operating system running the application. There is no need for us as programmers to worry about the epic battle between Windows and Linux and the use of a forward or back slash for directory listings.

The system palette contains fewer controls than the classic and modern palettes; in fact, all system controls are strictly user interface controls. Classic and modern palettes contain additional controls that have appeared in previous versions of LabVIEW. The following sections describe the control palettes as they appear in the classic and modern tree sections. The primary difference between them is appearance on the display. A classic numerical control will store data internally as a modern palette control.

1.6.1.1 Numeric

Internally, LabVIEW supports a number of numeric data types. Main types are floating point, integer, and complex numbers. Each type supports three levels of precision. Floating-point numbers are available as single, double, and extended precision. LabVIEW defines the number of digits in the mantissa for single and double precision numbers. Extended precision numbers are defined by the hardware platform LabVIEW is executing on.

Integers are available as byte, word, long word, and quad word precision. Bytes are 8-bit numbers, words are 16-bit numbers, long words are 32-bit numbers, and quad words are 64-bit numbers. Integers may be used as signed or unsigned quantities. LabVIEW supports 64-bit integers on all platforms; 32-bit machines will use code to emulate 64-bit integers. The controls in the Numeric palettes for the classic and modern sets are displayed in Figure 1.10. A full set of controls for allowing a user to enter and view data exists. Simple text representations exist for displaying data in addition to a variety of styled, graphical controls for presentations. User interfaces benefit from a set of controls that are relevant to the application. For example, an application supporting automation of operations at a brewery would benefit from using the tank controls to show the fill level of a primary fermenter. Choice of controls should be used with prudence. An application that contains a

FIGURE 1.10

dizzying array of colors, styles, sizes quickly becomes an eyesore for a user. Use controls to design an appearance that a user will relate to. Palette selection is not mutually exclusive — for example, using a tank in the classic set does not eliminate the ability to use a simple data display from the System set.

Once you have dragged a control or indicator onto the front panel, the pop-up menu can be used to modify its attributes. The type (floating point, integer, unsigned, or complex), data range, format, and representation are typical attributes for a digital control. Representation types that can be displayed for users are decimal, hexadecimal, date/time, and engineering notation. Representation types do not alter the numbers stored in memory; for example, displaying two digits beyond the decimal point does not cause LabVIEW to truncate numbers internally.

Figure 1.11 displays the window that appears when Format & Precision is selected from the pop-up menu. The Numeric Properties pop-up window contains several tabs. The appearance tab contains control configuration properties such as the label and caption visible on the display. The Data Range tab is of importance; it configures the default control value and allows the control to have its valid range of inputs configured. Data validation is critical in any application that is geared towards quality and we strongly encourage all programmers to use this functionality. Data entered outside the minimum and maximum range values can be either coerced to the range or ignored. This functionality does not work if the VI is called as a subVI. In the coercion case, the input data is set to the minimum or maximum range. The control can also be configured to ignore the entry. If the data range functionality is not used, the application should validate ranges in the application itself.

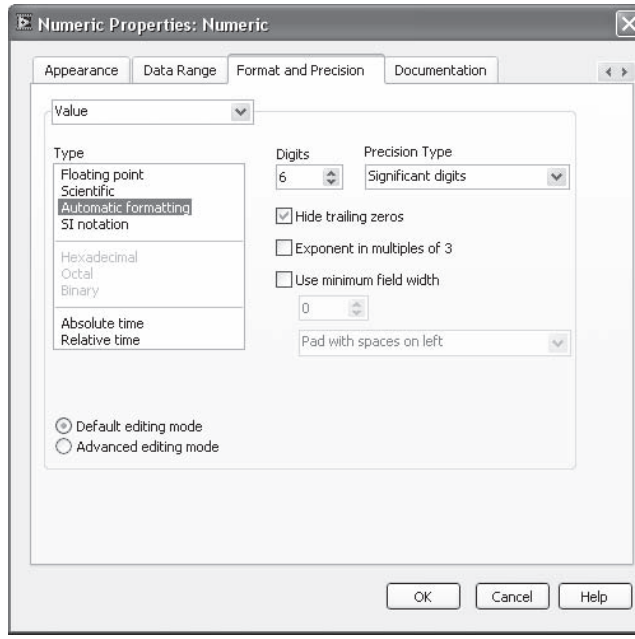


FIGURE 1.11

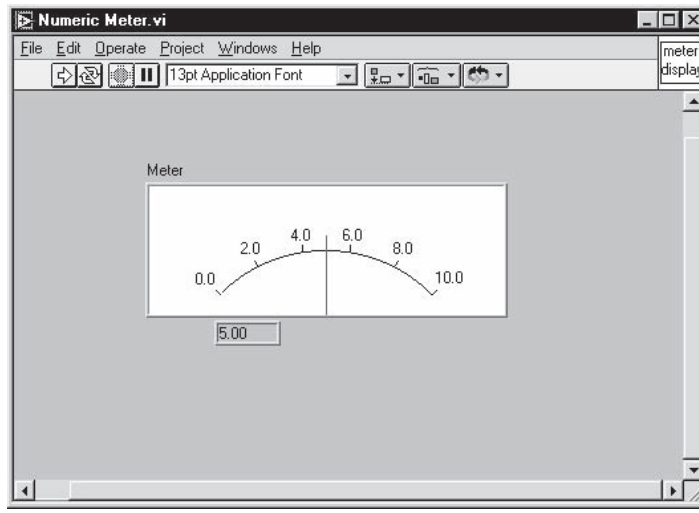


FIGURE 1.12

The format and precision tab affects the display of the data, but not the internal value. The type ranges will determine how data is presented in the control. Internally numerical values are not truncated or rounded when settings in this tab are selected. Floating point data can be shown in various formats, such as truncating the number of digits displayed and integer data can be displayed in decimal, hexadecimal, octal, or binary formats. Nondecimal displays are commonly used and convenient when it comes to data such as fields in communications protocols.

The nondigital objects in the numeric palette have an option to display a digital value with them through the pop-up menu. Just select the Visible Items in the pop up menu and then select Digital Display from the submenu. Figure 1.12 shows the meter with its associated digital indicator for precise readings. The meter, as most controls, can be resized by dragging one of the corners. The scale, markers, and mapping can also be modified on the meter.

1.6.1.2 Boolean

The Boolean palettes for the modern and classic palettes are illustrated in Figure 1.13. These palettes contain various true or false controls and indicators. Buttons to replicate switches, LED indicators, and operating system OK and Cancel buttons are provided. It is unlikely programmers will come up with Boolean indicator requirements that are not captured somewhere in this palette. Some of the controls in this palette are also available in the Dialog palette.

An interesting feature that LabVIEW programmers can use with Boolean controls is the mechanical action of the controls themselves. Configuration options available are switch when pressed, switch when released, switch until released, latch when pressed, latch when released, and latch until released. The major decision is whether the switch should switch or latch. Switching involves a somewhat permanent

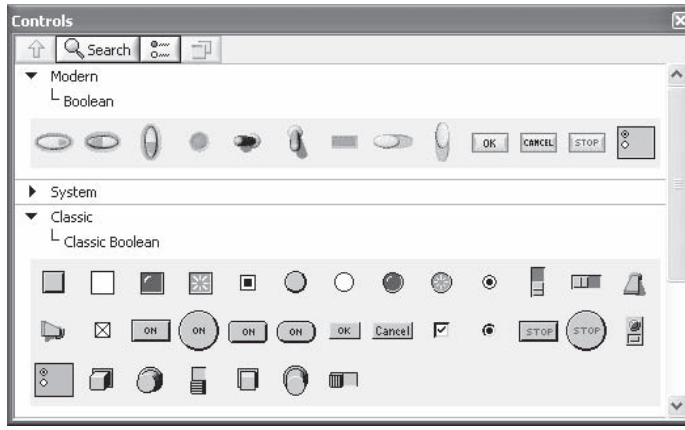


FIGURE 1.13

change. Latching changes the value of the control for a short period of time. The release time is when the user presses the button, and finally lets go. Switch when pressed makes the new value of the Boolean available as soon as the user touches it, and the change stays in place regardless of how long the user holds the button down. Switching when released does not trigger the new value until the user lets go of the control. Switching until released will change the control's value until the user releases the button. When the button is released, it toggles back to its original value.

Latching controls will toggle their value for a short period of time. Unlike switching, latching controls will return to their original value at some point in time. Latch-when-pressed Booleans will make the toggled value available as soon as the user clicks the control. Latch-when-released Booleans are toggled for a short while after the user releases the control. Latch-until-released controls will retain a toggled value while the control is activated by the user, and for a short period of time after the user releases the control.

Boolean controls have a default action of switch when pressed. Latching controls are very helpful in applications that allow users to change the behavior of an application for a short period of time. For example, a test application could have a button titled "e-mail status NOW." This button is not one that should be mechanically switched, where hundreds of e-mails can be sent to your boss when one would have done well. Buttons that switch when released are helpful when users try to time when a VI may have to stop. Also note that the mechanical action of subVIs is completely ignored; LabVIEW itself is not considered a user.

In general, there may not be a lot of material that can be presented on a topic such as programming buttons, but LabVIEW does provide a fair amount of flexibility for programmers as to how users and their programs can interact.

1.6.1.3 String & Path

The String & Path palette for the Modern and Classic control sets is displayed in Figure 1.14. It holds the string control, indicator, and file path control/indicators.

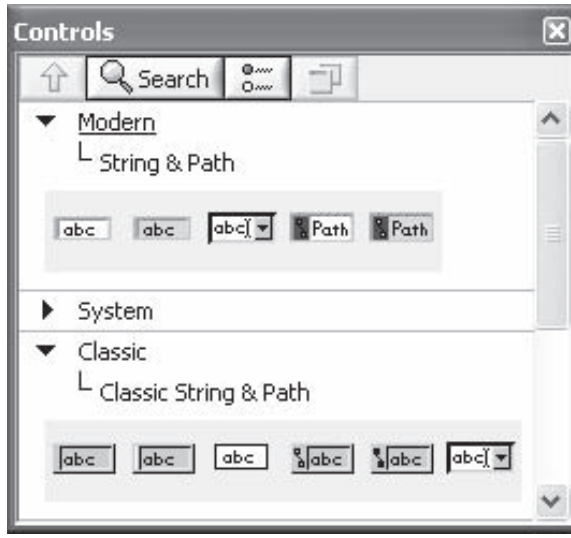


FIGURE 1.14

LabVIEW strings will automatically adjust their size to hold whatever data you place into them. String controls and indicators have a number of options that make them very flexible when programming a user interface. LabVIEW's string display functionality is one of the best in the industry.

Display options are very useful for programmers performing communications work. Many strings that are processed as an implementation of a communications protocol contain nonprintable characters. String displays can be set to show the ASCII or hexadecimal value of the contents. We have used this display option many times when writing drivers and code that use nonprintable arrays of characters. The "slash codes" display option is useful for showing white space used in the string. Spaces would appear as /s in slash code display. Again, this is very useful when writing code that needs to be clearly understood by a user. As an example, when writing code to validate protocol handling and the application needs to generate an Internet Protocol 4 header, it is easier to understand the header information presented in hexadecimal format than it is as a printable string. The first bye is normally 0x45 followed by 0x00. In telecommunications, protocol encapsulation is quite common, such as in IP tunneling. It will not always be practical or necessary to break a message apart field by field. LabVIEW string handling provides tools that make this display trivial where a C# programmer has some work to do.

Information that is sensitive can be protected with the password display option. Similar to standard login screens, password display replaces the characters with asterisks. Few programmers write their own login screens, but there are times when this display is necessary. Later in this book we will demonstrate using an ActiveX control to send e-mail. Before the control can be used to process e-mail, a valid user login must be presented to the mail server. The password would need to be obscured to casual observation.

It is possible to enable scrollbars for lengthy text messages, and also possible to limit values to a single line. If LabVIEW is used to display text files, scrollbars may become a necessary option. Form processing may want to limit the length of data users can insert, and single-line-only mode would accomplish this.

1.6.1.4 Ring & Enum, List & Table

The Ring & Enum and List & Table palettes are displayed in Figure 1.15. You will find the text, dialog, and picture rings along with the enumerated type and selection listbox in the palette. These items allow menu type controls or indicators for the user interface of an application. The text or picture represents a numeric value, which can be used programmatically. The enumerated type has an unsigned number representation and is especially useful for driving case statements. It is a convenient way to associate constants with names. Some of the controls represented in this palette are also available through the Dialog palette.

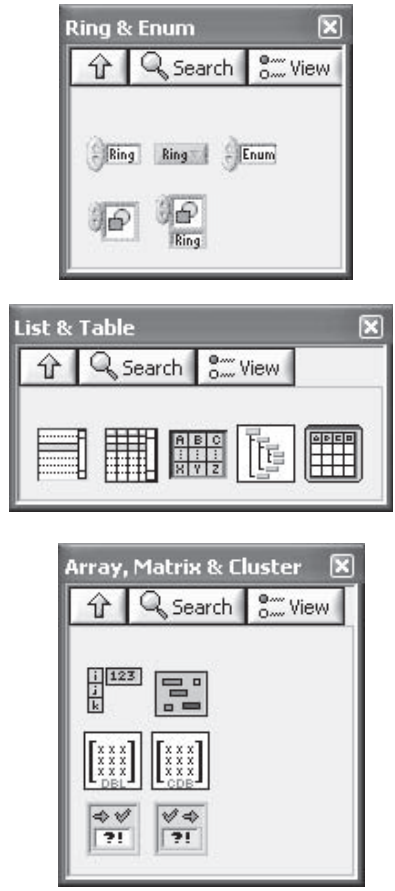


FIGURE 1.15

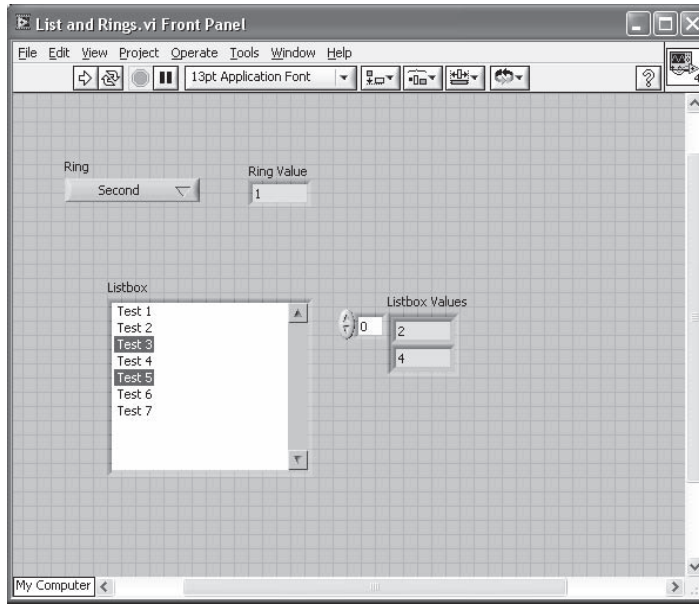


FIGURE 1.16

Figure 1.16 is a simple example that demonstrates how to use the objects in this palette. Shown is the menu ring with a digital indicator next to it, and a multiple selection listbox with a digital indicator array next to it. The menu ring is similar to a pull-down menu that allows the user to select one item among a list. Item one in a menu ring is represented by a numeric value of 0, with the second item being 1, and so on. The second item is selected in this example and its numeric value is shown in the indicator. The menu ring terminal is wired directly to the indicator terminal on the block diagram as shown in Figure 1.17.

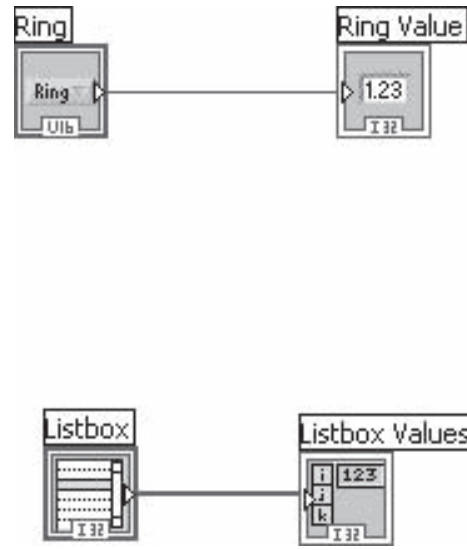


FIGURE 1.17

The multiple selection listbox is not a separate control from a single selection listbox; it's all in the configuration. It's possible to configure listboxes to accept 0 inputs, 1 input, or multiple inputs. It is also possible to allow for the user to modify the text in a listbox. Popping up on a listbox control gives a complete list of the features the control has and how it can

be customized. Symbols can also be shown with text in a list box by enabling symbols in the Visible selection. In this example, a multiple selection listbox was configured and is represented by an array of numbers, with 0 corresponding to the first item on the list. In our example, Test 3 and Test 5 are selected and the corresponding array is next to the list box. The array holds two values, 2 and 4, corresponding to the two tests selected from the listbox. Multiple selections are made from the listbox by holding down the Shift key and clicking on the items needed.

1.6.1.5 Array, Cluster, and Matrix

The last palette displayed in Figure 1.15 is Array, Cluster, and Matrix. To create an array, you must first drag the array container onto the front panel of a VI. This will create an array, but does not define the array type. A control or indicator must be dropped inside the array shell. Arrays of any data type can be created using the objects available in the Controls palette, except for charts or graphs. The array index begins at zero and the index display has a control that allows you to scroll to view the elements. A two-dimensional array can be created by either popping up on the array to add a dimension, or by dragging the corner and extending it.

Unlike C++, LabVIEW arrays are always "safe." It is not possible to overwrite the boundaries of an array in LabVIEW; it will automatically resize the array. Languages like C++ do not perform boundary checking, meaning that it is possible to write to the fifth element of a four-element array. This would compile without complaint from the C++ compiler, and you would end up overwriting a piece of memory and possibly crashing your program. LabVIEW will also allow your application to write outside the boundaries of the array, but it will redimension the array to prevent you from overwriting other data. This is a great feature, but is not one that programmers should rely on. For example, if writing to the fifth element was actually a bug in your code, LabVIEW would not complain and it would also not inform you that it changed the array boundaries!

Array controls and indicators have the ability to add a "dimension gap." The dimension gap is a small amount of space between the rows and columns of the control to make it easier for users to read. Another feature of the array is the ability to hide the array indexes. This is useful when users will see only small portions of the array.

A cluster is a data construction that allows grouping of various data types, similar to a structure in C. The classic example of grouping employee information can be used here. A cluster can be used to group an employee's name, Social Security number, and department number. To create a cluster, the container must first be placed on the front panel. Then, you can drop in any type of control or indicator into the shell. However, you cannot combine controls and indicators. You can only drop in all controls or all indicators. You can place arrays and even other clusters inside a cluster shell.

Figure 1.18 shows the array, cluster, and matrix shells as they appear when you first place them on the front panel. When an object is dropped inside the array shell, the border resizes to fit the object. The cluster shell must be modified to the size needed by dragging a corner. Matrix controls appear in a 3x3 of integers. Figure 1.19 shows the array and cluster with objects dropped inside them. A digital control

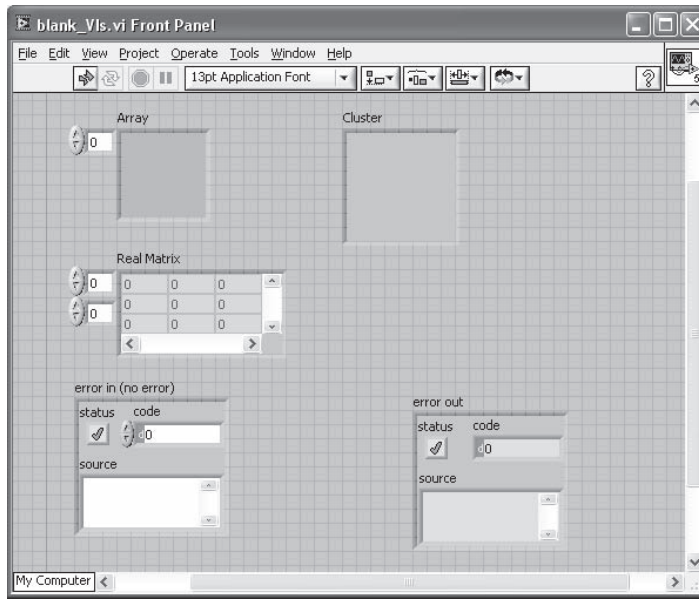


FIGURE 1.18

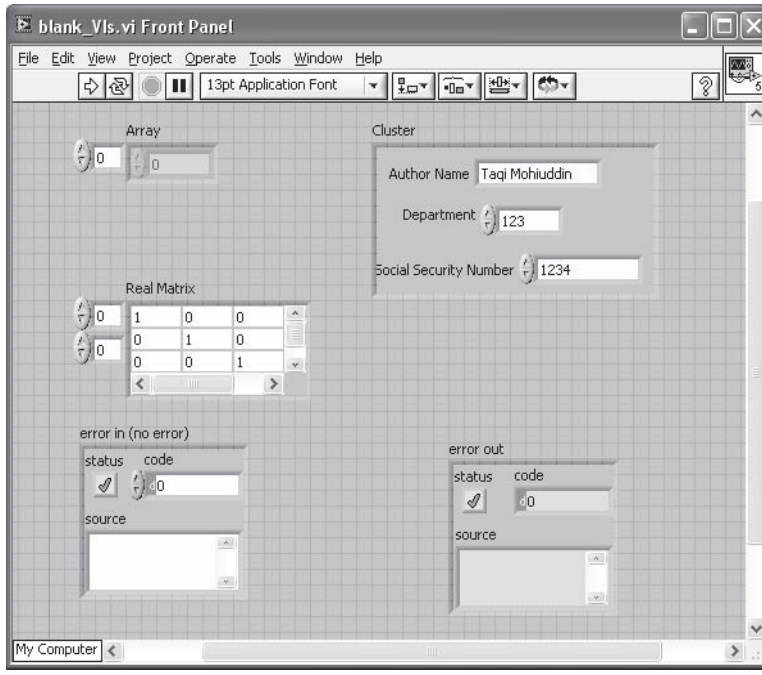


FIGURE 1.19

was dropped in the array shell. The outer display shows the current index number of the array. The cluster now contains a string control for the employee name, a digital control (integer) for the department number, and another string control for the social security number. When only one value from the cluster data is needed when programming, a LabVIEW function allows you to unbundle the cluster to retrieve the piece that is needed. This is explained further in Section 1.6. The matrix control does similar work to an array, but the control's dimensions are kept the same. As the identity matrix was filled out, the x and y dimensions of the array were kept the same. So when a "one" was added to (2,2), the third row and column were filled with zeros. If the cell is part of the matrix it will appear white by default. If the cell is not part of the matrix it will appear gray by default.

The Error In control and Error Out indicator, shown in the two previous figures, are both clusters. These are used for error detection and exception handling in LabVIEW. The clusters hold three objects: a status to indicate the occurrence of an error, a numeric error code, and a string to indicate the source of the error. Many LabVIEW functions utilize the error cluster for error detection. Error handling is discussed in Chapter 6.

1.6.1.6 Graphs and Charts

Figure 1.20 displays the Graphs palette with the built-in graph and chart objects. The Waveform Chart and Waveform Graph are located in the top row, and the

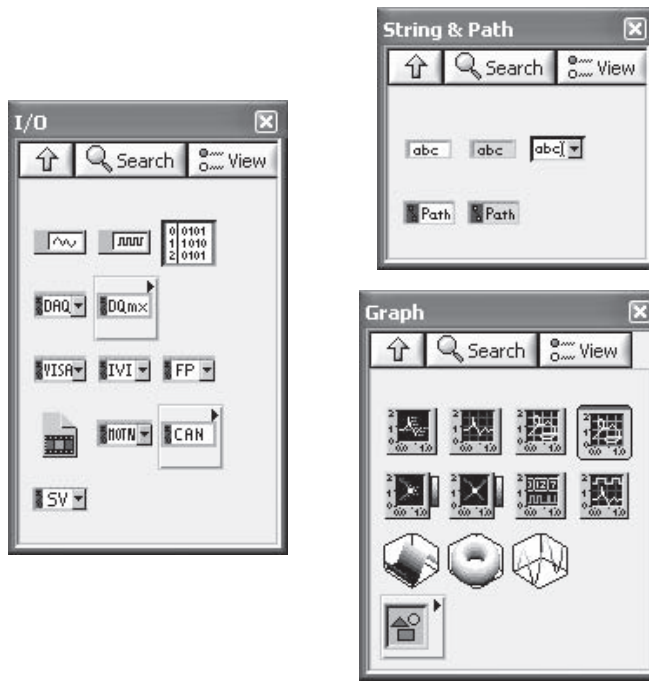


FIGURE 1.20

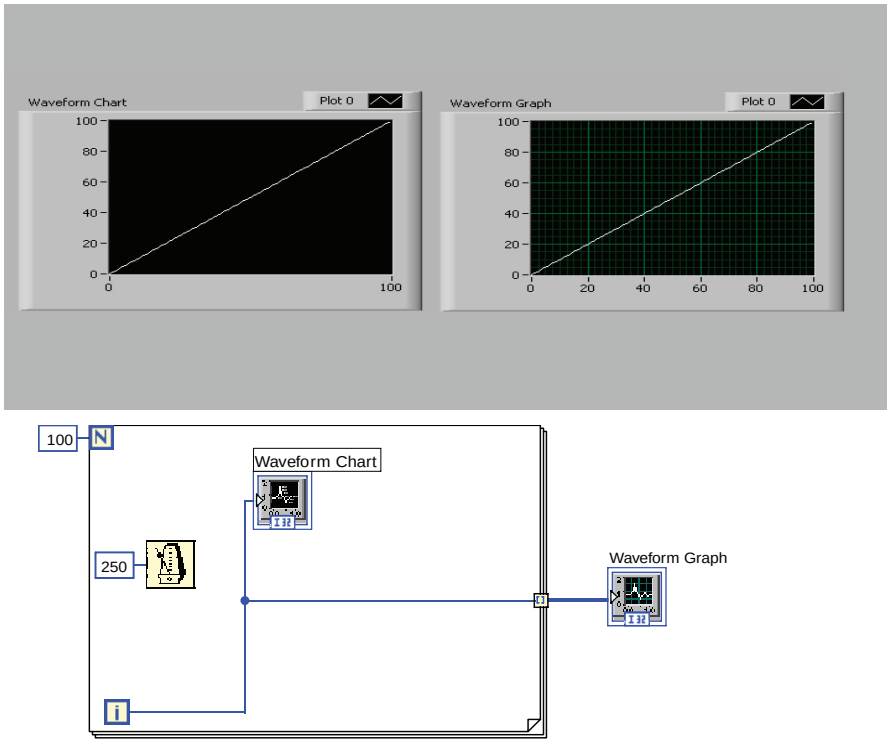


FIGURE 1.21

Intensity Chart and Intensity Graph are in the second row. The XY Graph is also available in the top row of the palette. The graph and chart may look identical at first, but there is a distinction between the two. The graph is used for plotting a set of points at one time by feeding it an array of data values. The chart, on the other hand, is used for plotting one data point or array at a time. A chart also has memory, maintaining a buffer of previous points which are shown in its display.

The example in Figure 1.21 will help to demonstrate the difference between a chart and a graph. A Waveform Chart and Waveform Graph are displayed on the front panel side by side. A For loop is executed 100 times with the index value being passed to the chart. Once the loop is finished executing, the array of index values is passed to the graph. A 250-millisecond delay is placed in the For loop so you can see the chart being updated as the VI executes. Both the chart and graph are used for displaying evenly sampled data.

Graphs and charts have a number of display options enabling programmers to display data in a manner that makes sense. For example, both charts and graphs support a histogram style display. Because histograms plotted with straight lines are awkward to read, interpolation between points and point styles are completely adjustable.

Graph controls and indicators provide a palette for users to adjust the graphs at runtime. The palette allows for auto scaling of both the X and Y axes. Zoom features are available for examining portions of the graph at runtime. Cursors are available

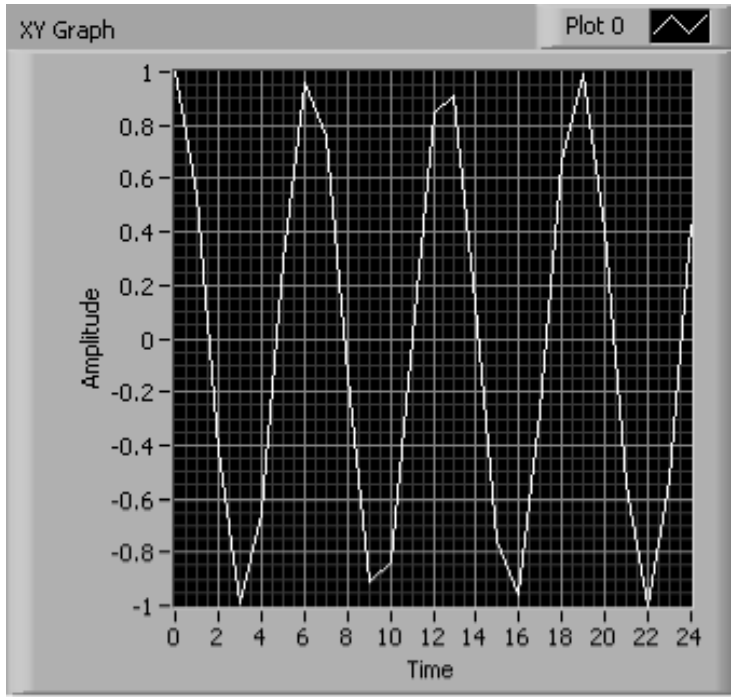


FIGURE 1.22

to measure distances between points. This level of functionality is not very common in graphing packages that come standard with most other languages.

The XY Graph can be used to graph any type of data, similar to a Cartesian graph. Figure 1.22 illustrates the XY Graph with a plot of a cosine wave. Two separate arrays are provided as input to this graph. Several graph and chart attributes can be modified for display purposes. The grid options, mapping (linear or log), scale styles, and marker spacing are some of the items available in the pop-up menu. Their displays can also be resized on the front panel by dragging a corner.

3-D graphs and picture plots are some of the advanced objects available on this palette. The 3-D graphs require three separate arrays of data values for graphing the x, y, and z coordinates. The Polar Plot, Smith Plot, Min-Max Plot, and Distribution Plot are indicators on the Picture subpalette of the Graph palette.

1.6.1.7 String & Path and I/O

The String & Path and I/O palettes are displayed in Figure 1.20. The bottom two objects on the String & Path palette are the File Path Control and File Path Indicator. These are used when performing directory- or file-related operations to enter or display paths.

The I/O palette contains control and indicators for the refnums used in LabVIEW. A refnum is a distinct identifier or reference to a specific communications path. The refnum can point to a file, external device, .NET object, network connection, IVI

device, DAQ Card, or VISA communications channel, or to another VI. This identifier is created when a connection is opened to a specific object. When a connection is first opened, the particulars of the connection need to be defined, such as a file path, instrument address, or an IP address. After the connection is opened, a refnum is returned by the open function. This refnum can then be used throughout an application when operations must be performed on the object. The particulars of the connection need not be defined again.

Figure 1.23 demonstrates the refnum through a simple example. In this illustration, a TCP connection is opened to a host computer. The front panel shows controls for the IP address or host computer name and the remote port number that are needed to define the connection. The Network Connection Refnum is an indicator returned by the function that opens the connection. The block diagram shows TCP Open

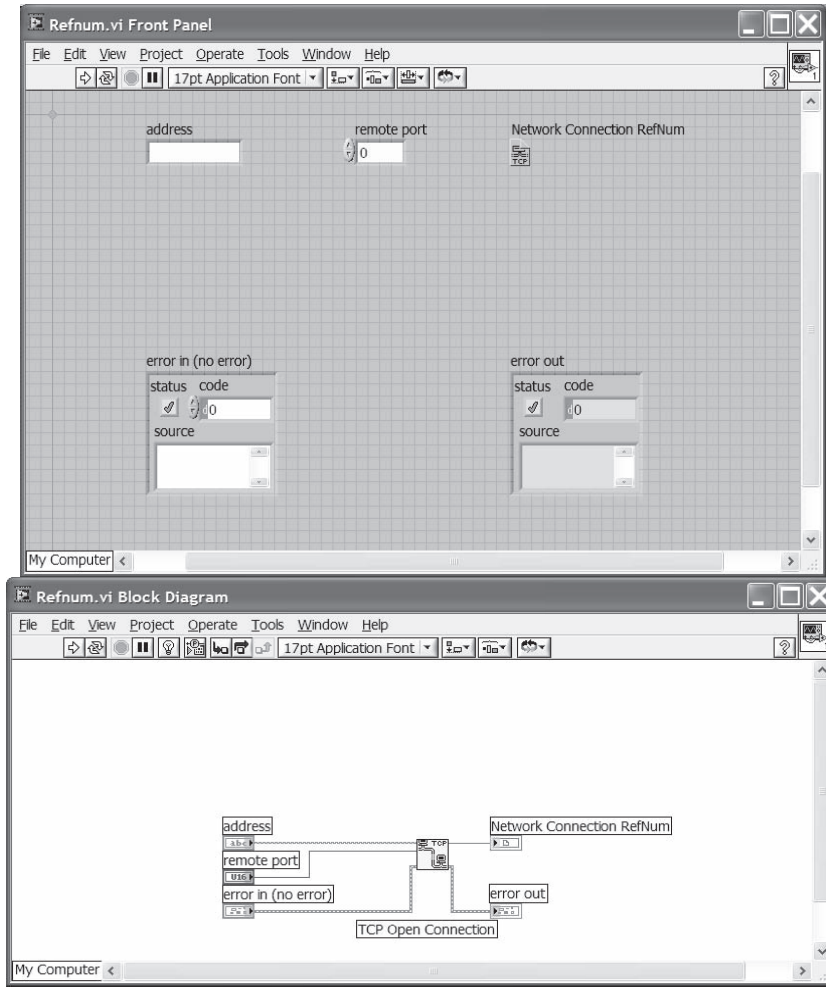


FIGURE 1.23

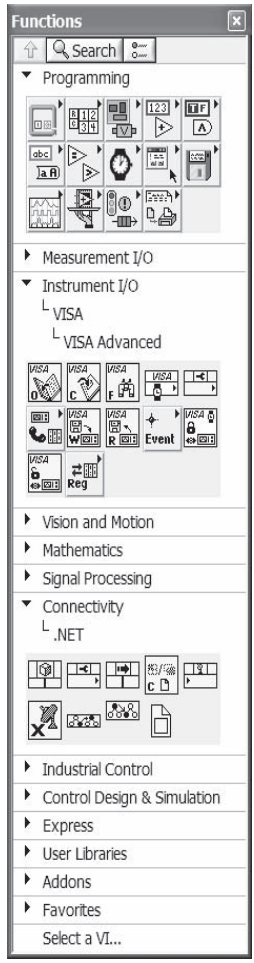


FIGURE 1.24

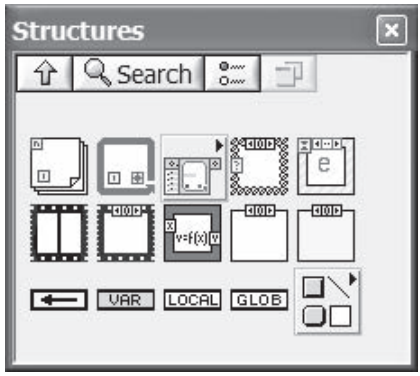


FIGURE 1.25

Connection, a built-in LabVIEW function, with the related data provided. The refnum, or reference, created by this function can then be used to perform other operations. This unique identifier represents the connection, and the specifics do not need to be provided again.

LabVIEW uses refnums to track internally used resources; for example, a file path refnum contains information needed to read or write to a file. This information is using system resources such as memory and must be returned. If the programmer does not close refnums, LabVIEW will leak memory. Over long periods of time, this could degrade the system's performance.

1.7 BLOCK DIAGRAM FUNCTIONS

All coding in LabVIEW is done on the block diagram. Various functions are built in to aid in the development of applications. The Functions palette is displayed in Figure 1.24 and appears when the block diagram window is active. LabVIEW is a programming language and uses the typical programming constructs such as loops, and defines a couple of other structures unique to data flow programming. This section briefly describes some of the tools that are available to LabVIEW programmers.

1.7.1 Structures

The control structures that are accessible from the Structures palette are shown in Figure 1.25. This palette contains several types of structures including the case, For loop and While loop structures. You will also find several types of variables including the Global and Local Variable on this palette.

1.7.1.1 Sequence Structure

There are two types of sequence structure on the structure palette: the Stacked Sequence structure and the Flat Sequence structure. The operation of the two structure types is the same, but the way the sequences are displayed is different. First let us place the stacked sequence structure on the diagram and drag it to the size desired. The structure looks like a frame of film when placed on the diagram. The Sequence structure is used to control the flow or execution order of a VI. In LabVIEW, a node executes when the input data required becomes available to it. Sequence structures can be used to force one node to execute before another, and to ensure that the VI executes in the order intended.

Each frame is basically a subdiagram. The sequence structure will begin executing when the required data becomes available to it, just as any other node. The objects placed inside the first frame (Frame 0) execute first, and the rest of the frames follow sequentially. Within each frame or subdiagram the data flow execution still applies.

The top of Figure 1.26 shows the sequence structure as it appears when first placed on the block diagram. Additional frames are added by popping up anywhere on the border of the structure and selecting Add Frame After (or Before). The second picture depicts the stacked sequence structure after a frame has been added. Only one frame is visible at a time. The display at the top of the frame indicates which frame is currently visible.

The example diagrams in Figure 1.27 will help to define some terms that are related to the Sequence structure. The top window shows Frame 0, and the bottom window shows Frame 1 of the structure. Data can be passed into a Sequence structure by simply wiring it to the border to create a tunnel. The blackened area on the border indicates that a tunnel has been created. Data is passed out of the sequence structure in a similar manner, with the data actually being passed out after all of the frames have been executed. A tunnel is created for each value that needs to be passed in and is available for use in all frames. The same is true for data being passed out of a sequence structure. This point is important because data being passed out of a case structure is handled differently.

Data values can be passed from one frame to the following frames with the use of sequence locals as shown in the top diagram. The sequence local is available in the pop-up menu. The arrow on the local indicates that the data is available for manipulation in the current frame. Note that in Frame 0, the local to the right is not available because the data is passed to it in Frame 1. Frame 2 can use data from both of the sequence locals. The locals can be moved to any location on the inside border of the structure.

Stacked sequence structures can be avoided in most applications. The main problem with stacked sequence structures in LabVIEW programming is readability for other programmers. Controlling the order of execution can be performed with error clusters, or by designing subVIs with dependent inputs. Sequence structures can be a bad habit that is easily developed by some LabVIEW programmers. The authors use sequence diagrams that contain a single frame when working with VIs that do not use a standard error cluster.

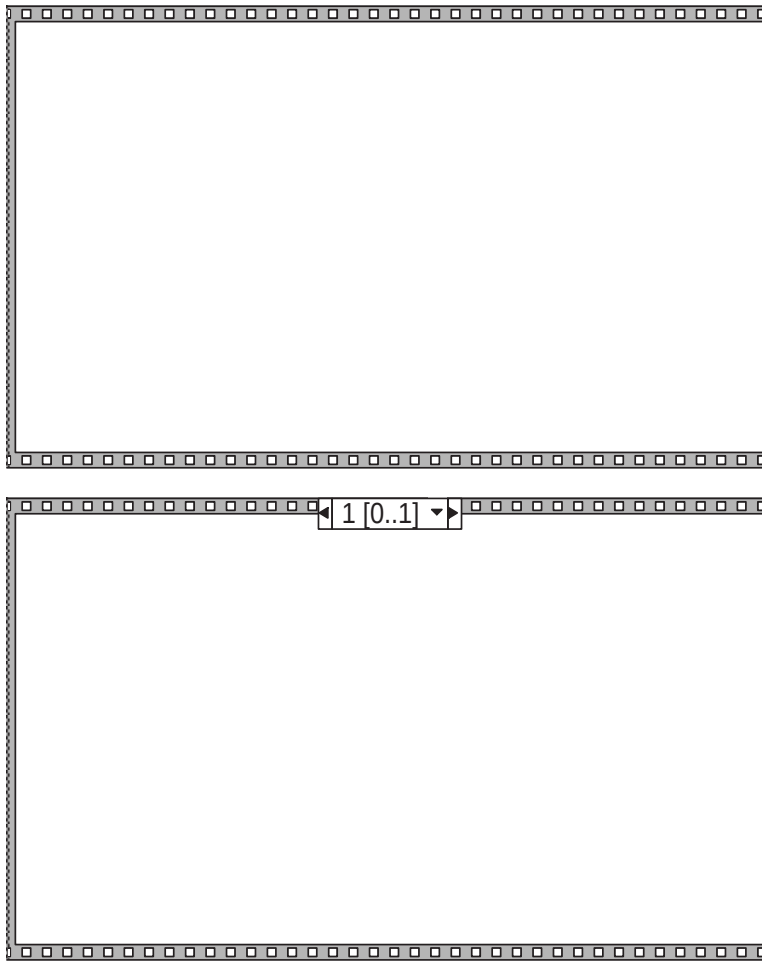


FIGURE 1.26

For years LabVIEW power programmers complained about the use of sequence structures in applications. National Instruments heard the complaints and added a new sequence structure called the flat sequence structure. This structure has the same functionality as its stacked predecessor, but instead of having the sequences stacked on top of one another they are now laid out horizontally. The flat sequence structure truly looks like a section of film now. Figure 1.28 shows a flat sequence structure with two frames. When a flat sequence structure is placed on the code diagram its initial appearance is the same as the stacked. When a second frame is needed the programmer pops up on the structure and selects Add Frame Before (or After) and the new frame will appear adjacent to the existing frame. A definite improvement with respect to readability, but it does result in a loss of valuable diagram real estate.

Sequence structures do not have equivalents to other programming languages; this is a unique structure to dataflow languages. Text-based languages such as Visual

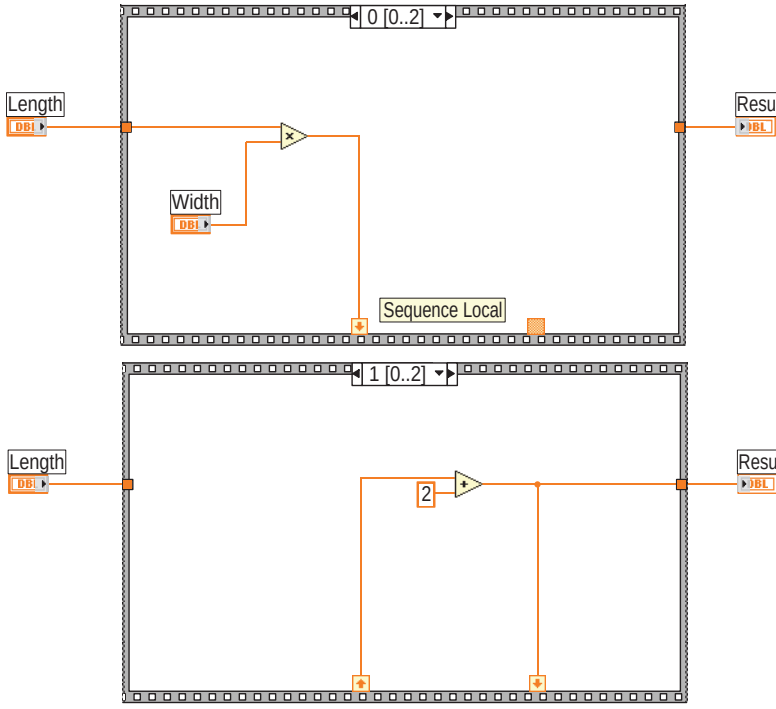


FIGURE 1.27

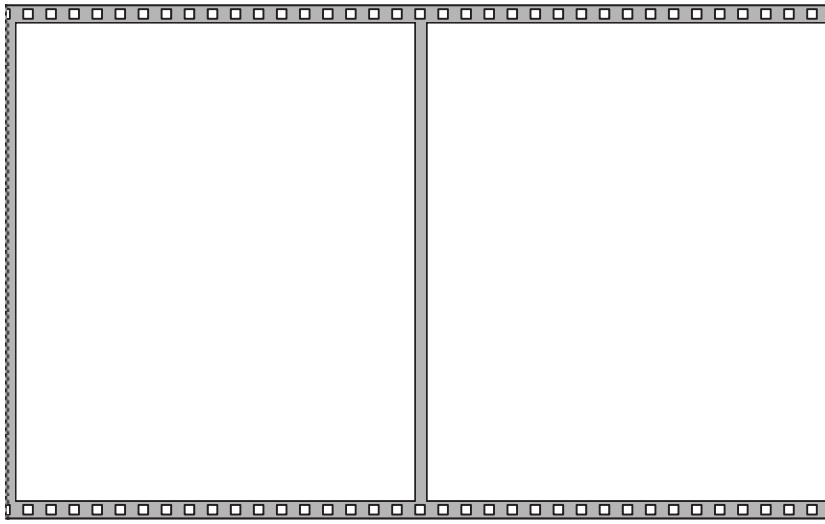


FIGURE 1.28

Basic and C perform operations line-by-line; LabVIEW executes things as they become available.

1.7.1.2 Case Structure

The case structure is placed on the block diagram in the same manner as the sequence structure. The case structure is similar to conditional control flow constructs used in programming languages such as C. The case structure has a bit more responsibility in LabVIEW; in addition to switch statements, it functions as an if-then-else block when used with a Boolean. Figure 1.29 displays case structures and four examples of how they are used.

The first case structure uses a Boolean data type to drive it. A Boolean is wired to the selector terminal represented by the question mark (?). When a Boolean data type is wired to the structure, a true case and a false case are created as shown in the display of the case structure. The false case is displayed in the figure since only one case is visible at a time. As with the sequence structure, the case structure is a subdiagram which allows you to place code inside of it. Depending on the value of the Boolean control, the appropriate case will execute. Of course, the availability of all required data inputs dictates when the case structure will execute.

The flat sequence structure truly looks like a section of film now. Figure 1.28 shows a flat sequence structure with 2 frames. A numerical case structure is shown to the right of the structure driven by the Boolean. When a numeric control is wired to the selection terminal, the case executed corresponds to the value of this control.

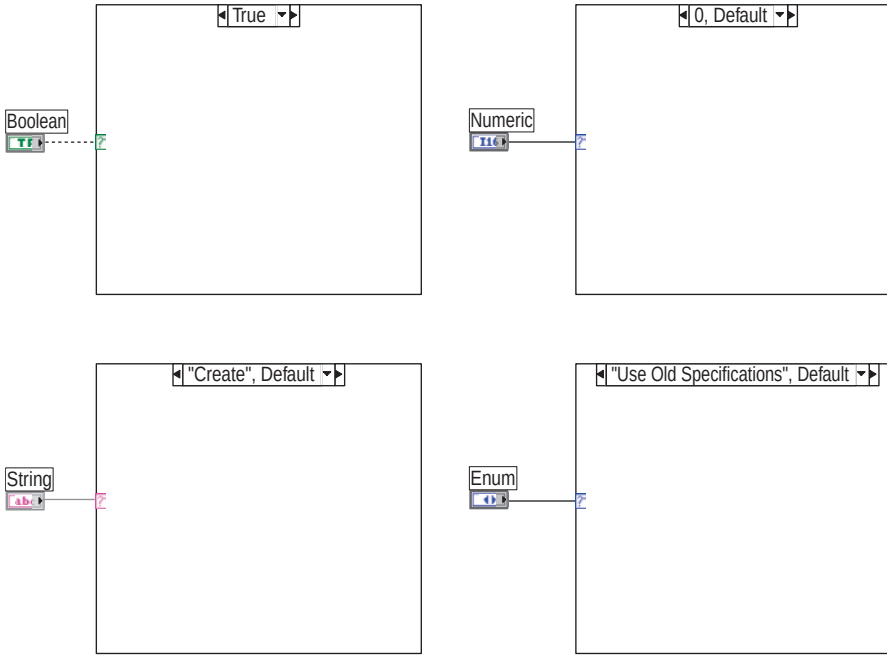


FIGURE 1.29

When the case structure is first placed on the code diagram and the numeric control is wired to the case selector, LabVIEW creates only two cases. You must pop-up on the structure and add as many cases as you need. Normally, Case 0 is the default case, but you can change that to any case you desire. You must specify a default case to account for the different possibilities. If you do not specify a default case, you must create a case for each possibility. You can assign a list or range of numbers to a particular case by editing the display, or case selector label, of the structure with the editing tool. To assign a list to one case, use numbers separated by commas such as 2, 3, 4, 5. To specify a range, separate two numbers by two periods, like 2..5.

You should also be aware that floating point numbers could be wired to the case selection terminal. LabVIEW will round the value to the nearest integer. However, the selector label cannot be edited to a floating point number. The case selector label will display red characters to indicate that it is not valid.

The lower left case structure has a string control wired to the case selector. The case selector display must be edited to the desired string value for each case. The string is displayed in quotes but does not have to be entered that way. The case that matches the string control driving the structure will be executed. LabVIEW allows you to alter the criteria to perform a case-insensitive match to ignore the difference between upper and lower case strings. If there is no match, the default case will execute. Applications with performance requirements should consider using enumerated types to drive the case statements. String parsing and matching can be processor intensive activities.

Finally, an enumerated type is used to drive the case structure in the lower right corner. The text surrounded by the quotes corresponds to the different possible values of the control. When you first wire the enumerated control to the case selector terminal, only two cases are created. You must use the pop-up menu to add the rest of the cases to the structure. Although the enumerated data type is represented by an unsigned integer value, it is more desirable to use than a numeric control. The text associated with the number gives it an advantage. When wired to a case structure, the case selector label displays the text representation of the enumerated control. This allows you to identify the case quickly, and improves readability.

Data is passed in to the case structure by creating a tunnel. Each data value being passed must have a unique tunnel associated with it. This data is made available to all of the cases in the structure. This is similar to the Sequence structure described earlier. However, when data is being passed out of the case, each case must provide output data. Figure 1.30 illustrates this point. The picture shows the code of a VI using an enumerated type to control the execution of the case structure. This VI takes two numeric values as input and performs an operation on them, returning the result as output. Depending on the selection, addition, subtraction, multiplication, or division is performed.

The top window shows the “Subtract” case displayed. Number 2 is subtracted from Number 1 and the result is passed out to Result. Note that the tunnel used to pass the data out is white. This indicates that a data value is not being output by all cases. All of the cases must have a value wired to the tunnel. The bottom window shows the Add case displayed. Now all of the cases have an output wired to the tunnel, making it turn black. This concept holds true for any data type driving the

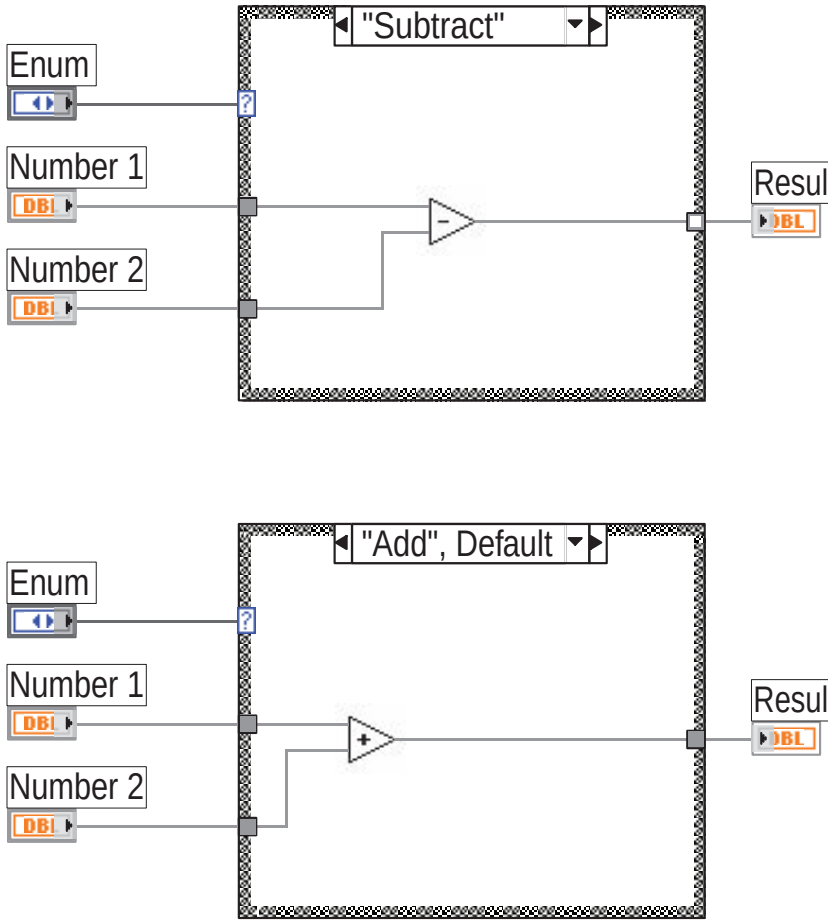


FIGURE 1.30

case structure. An alternative to wiring all the cases to the output in order to remove the error is to right click on the output tunnel and select Use Default if Unwired. This "fills in" the tunnel coloring with just a white dot in the middle indicating that not all the cases are wired, but the structure will output the default value if it is not connected.

1.7.1.3 For Loop

The For loop is used to execute a section of the code, a specified number of iterations. An example of the For loop structure is shown in Figure 1.31. The code that needs to be executed repeatedly is placed inside of the For loop structure. A numeric constant or variable can be wired to the count terminal to specify the number of iterations to perform. If a value of zero is passed to the count terminal, the For loop will not execute. The iteration terminal is an output terminal that holds the number

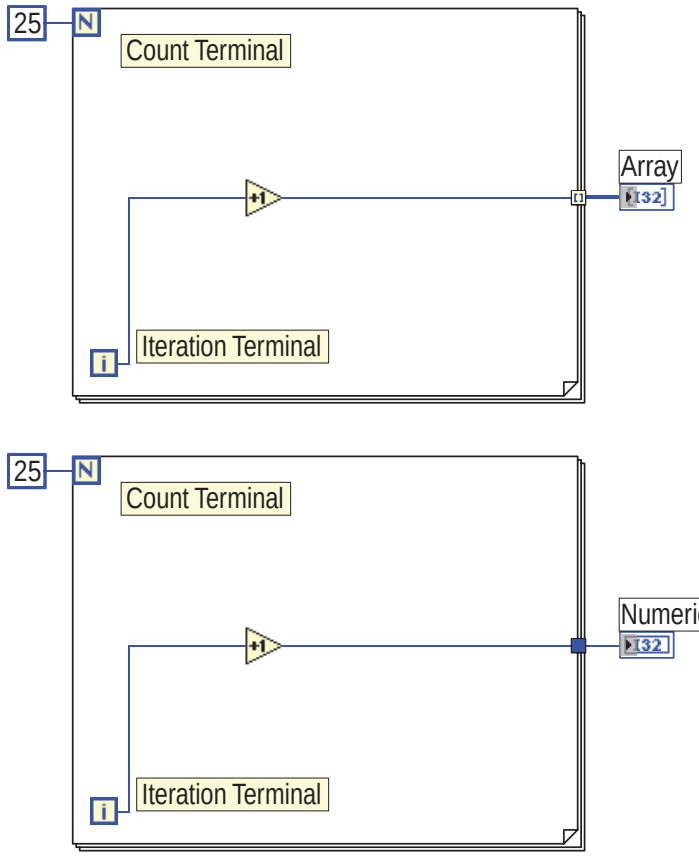


FIGURE 1.31

of iterations the loop has executed. Therefore, the first time the loop executes, the iteration value is 0.

The top block diagram shows a For loop that will execute 25 iterations. A 1 is added to the value of the iteration terminal and passed out to an indicator array via a tunnel. The output of the For loop is actually an array of the 25 values, one for each iteration. Because the loop executed 25 times, LabVIEW passes an array with the 25 elements out of the tunnel. In this case, the array holds values 1 through 25 in indexes 0 through 24, respectively; this is known as auto indexing. Both the For loop and While loop assemble arrays when data is passed out. Auto indexing is the default only for the For loop, however. LabVIEW allows the programmer to disable auto indexing so that only the last value is passed out of the loop. This is shown in the bottom code diagram. Popping up on the tunnel and selecting the appropriate item from the menu disables indexing. The output from the tunnel is wired to a numeric indicator in this diagram. If you observe the wire connecting the indicator and the tunnel, you will notice that the wire is thicker in the top diagram because it is an array. This allows you to quickly distinguish an array

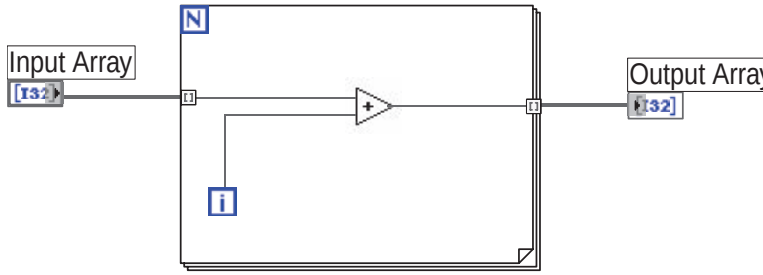


FIGURE 1.32

from a single value. Indexing can be enabled in a similar manner if you are using a While loop.

Figure 1.32 illustrates another example diagram utilizing the For loop. An array is passed into the For loop to perform an operation on the values. In this example, the count terminal is left unwired. LabVIEW uses the number of elements in the array to determine how many iterations to perform. This is useful when the size of the array is variable and not known ahead of time. One element at a time is passed into the For loop structure and the addition is performed. This property of For loops is also a feature of auto indexing and is available by default in For loops. This is the opposite of what the loop does at the output tunnels. Caution needs to be used when working with multiple arrays being fed into a For loop. LabVIEW will perform a number of iterations equal to the array length of the shortest array. Popping up on the terminal and selecting Disable Indexing can disable auto indexing.

What if you do wire a value to the count terminal in this example? If the value passed to the count terminal is greater than the number of elements in the array, LabVIEW uses the number of elements in the array to decide how many iterations to perform. If the value passed to the count terminal is less than the number of elements in the array, LabVIEW will use the count terminal value. This indexing feature on the input side of the For loop can also be disabled by using the pop-up menu. Once indexing is disabled, the whole array is passed in for each iteration of the loop.

The last feature of auto indexing is the ability to handle arrays of multiple dimensions. A two-dimensional array fed into a For loop will iterate the values in one dimension; in other words, a one-dimension array will be fed into the For loop. A nested For loop can be used to iterate through the one-dimension arrays.

Figure 1.33 shows the code diagram of a VI that calculates the factorial of a numerical value. A shift register is utilized to achieve the desired result in this example. The shift register has two terminals, one on the left border and one on the right border of the structure. The shift register is used for passing a data value from the current iteration to the next one. The right terminal holds the data of the current iteration and is retrieved at the left terminal in the next iteration. A shift register pair can be created by popping up on the left or right border of the For loop structure and selecting Add Shift Register. The shift register can hold any LabVIEW data type.

In the example shown, a constant value of 1 is wired to the shift register. This initializes the value of the shift register for the first iteration of the loop. If nothing

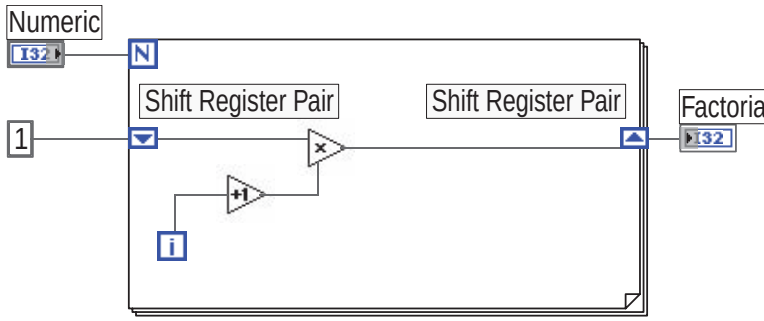


FIGURE 1.33

was wired to the shift register, the first iteration would contain a value of 0. The Numeric control wired to the count terminal contains the value for which the factorial is being calculated. A 1 is added to the iteration terminal and then multiplied to the previous result. This successfully yields the desired factorial result. Shift registers can be configured to remember multiple iterations by popping up and selecting Add Element from either side. A new terminal will appear just below the existing one on the left border of the structure. When you have two terminals, this allows you access to the two previous iteration values. The top terminal always holds the last iteration value.

Care should be used when leaving shift registers uninitialized. When a default value is not wired to a shift register the last stored value will become the initial value. Take the code diagram in Figure 1.34. This VI takes arrays of test name, measured data and status and assembles them into a tab-delimited string that can be written to a text file. Notice that the shift register is not initialized. The first time the VI is run the initial value would be an empty string resulting in the correct assembled string output. If the user were to call this VI a second time the initial value would now be the last assembled string. When the FOR loop executes the second time the old string will get concatenated to the new string. Obviously the

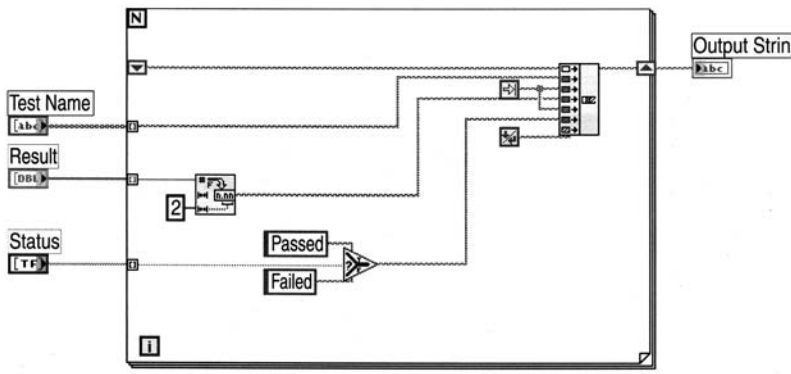


FIGURE 1.34

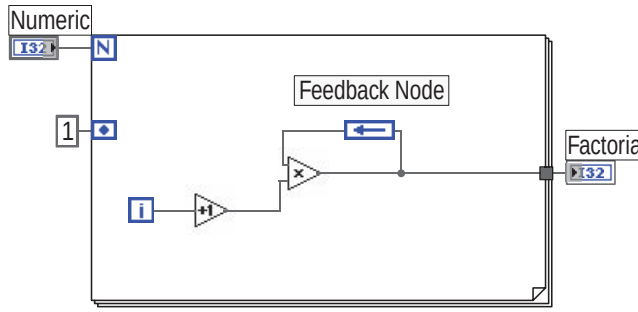


FIGURE 1.35

string could grow very quickly resulting in a memory problem. In some cases the programmer may want this kind of behavior. One application of the unitialized shift register is the Type 2 global, which is discussed in section 2.1.

Shift registers are the only mechanisms available to perform recursive operations in LabVIEW. Recursion is the ability for a function to call itself during execution, and it has frustrated thousands of students learning C and C++. The good news for LabVIEW programmers is that VIs cannot wrap back onto themselves in a wire diagram. There are times when a recursive operation is the best way to solve a problem, and using shift registers simulate recursion. Although not truly recursive, access to the last iterations can be used to perform these ever-popular algorithms in LabVIEW. It is not possible for LabVIEW to overrun a call stack with shift registers, which is very possible with recursive functions in C. One of the problems with recursion is that if exit criteria are not correct, the function will not be able to stop calling itself and will crash the application. Memory usage is also a bit more efficient for shift registers because there is not as much call stack abuse.

A newer option available for passing back values in LabVIEW loops is the feedback node. The feedback node transfers values from one loop to the next in For and While loops. The functionality of the feedback node is similar to the shift register, but is a bit cleaner on the code diagram. Figure 1.35 shows the factorial example discussed above with a feedback node used in place of the shift register. Note in the example that there is an input port on the loop for setting an initial value. The feedback node is on the main level of the Structures palette.

Outputs of a For loop, by default, will be arrays consisting of a collection of outputs for each iteration of the loop. One advantage of the For loop when handling arrays is LabVIEW's efficiency. Because the For loop's iteration count is derived from an iteration count or length of an array, LabVIEW can precompute the number of elements in array outputs. This allows LabVIEW to reserve one contiguous block of memory to write output arrays to. This is important because, as we mentioned earlier, LabVIEW will expand array boundaries, but this involves a performance hit because LabVIEW needs to go to the operating system and reallocate the entire array and perform a duplication of the existing elements. Small arrays will not be a significant performance degradation, but larger arrays can slow things down quite a bit.

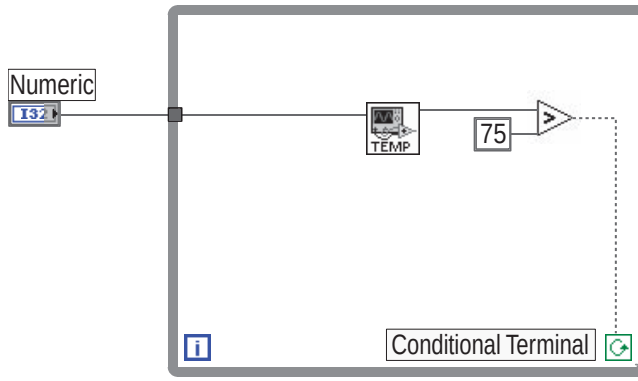


FIGURE 1.36

1.7.1.4 While Loop

The While loop is an iteration construct that executes until a predetermined Boolean value is passed to its conditional terminal. The conditional terminal is located in the lower right corner of the While loop structure, as shown in Figure 1.36. The While loop will execute at least once because the condition is evaluated at the end of the current iteration. When the While loop is placed on the block diagram the conditional terminal is setup to stop if true by default. In this case when a false is passed to the conditional terminal, the loop will execute another iteration before evaluating the value once again. If the terminal is left unwired, the loop will execute infinitely. By right clicking on the conditional terminal the user can change the behavior of the loop to continue if a true is received. If continue if true is selected, and a true value is passed to the conditional terminal, the loop will execute another iteration before evaluating the value once again. If the terminal is left unwired, the loop will execute once before stopping.

Figure 1.36 illustrates the use of the While loop. The output of the subVI is compared to find out if it is greater than 75.0. This evaluation determines whether the loop will execute one more iteration. If the value is greater than 75.0, a true value is passed to the conditional terminal causing it to execute again. If the value is less than or equal to 75.0, a false value causes the loop to terminate.

Automatic indexing is available for the While loop also, but it is not the default. When data is passed in or out of the loop structure, you must use the pop-up menu to enable indexing. Shift registers can be created on the left or right border of the While loop. The shift registers operate in the same manner as described as the For loop.

While loops can be used to perform the functions of a For loop with a little less efficiency. Popping up on the terminals can use auto indexing and array creation. As you will see throughout this book, While loops are used by the authors more often than For loops. This is not a matter of personal preference, but good design decisions. When working with previously collected data, such as reading a file and processing the file contents, For loops will be more efficient and are used in these

types of applications. Points read in the form of arrays can be done far more efficiently with For loops because LabVIEW can precompute memory allocations. The problem with For loops is that there is no mechanism to abort execution of the loop; i.e., there is no break command. While loops stop their execution any time the predetermined Boolean value is fed into the condition terminal.

Stopping execution of a loop is important when performing automation, which is the authors' primary use of LabVIEW. One of the inputs to the condition indicator will be the Boolean value of the error cluster, which we feed through a shift register for every iteration. In an automation application, the ability to break execution is more important than the efficiency of array handling. There is a tradeoff of efficiency against exception handling, but in automation it makes more sense to stop execution of troubled code.

1.7.1.5 Event Structure

An *event structure* is a structure containing one or more subdiagrams corresponding to defined events. When an event occurs, the corresponding subdiagram executes. The event structure will wait until an event occurs or a timeout occurs. By default the timeout value is –1 (never times out). An event could be anything from a control value change to a mouse click. The programmer defines what conditions constitute an event when defining the subdiagram. There is also a dynamic event terminal that allows for events to be registered at runtime.

Perhaps the easiest way to describe an event structure is through an example. Figure 1.37 shows a VI with an enumerated control and indicator and a stop button. The event structure is set up to monitor a value change on the stop button and the enumerated control. If the control value is changed then the event structure executes case 1. The new value is written to the enumerated indicator. If the stop button is pushed the event 0 case is executed. The Boolean value is written to the loop conditional terminal to stop execution. Note that the event structure will wait forever because no value is wired to the timeout input at the top left of the structure.

1.7.1.6 Disable Structure

There are two disable structures available on the structure palette. There is a disable structure and a conditional disable structure. The disable structure is similar to a case structure except there is no conditional input for the structure. The disable can have as many subdiagrams as you care to create, but one and only one must be enabled. This structure allows a user to enter several mutually exclusive subdiagrams in one location. When editing, the programmer can select which subdiagram to execute. The programmer can then change the enabled subdiagram without having to change the code as would be needed for a standard case structure.

The conditional disable structure gives the programmer the ability to select what code to operate based on the target platform. This means there can be different functions called if the code is run on a windows machine vs. a PDA or FPGA. Again, the basic look of the structure is similar to the cases structure without the case selector input.

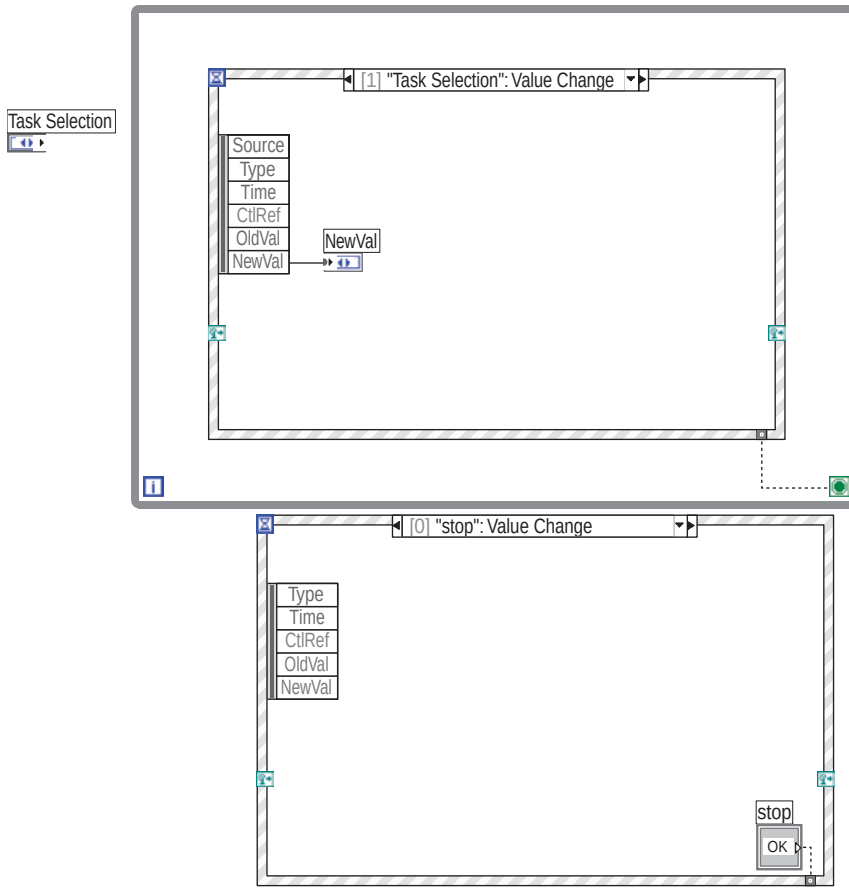


FIGURE 1.37

1.7.1.7 Timed Structure

The timed loop structure is similar to a While loop, but gives the programmer the ability to control the execution rate and run time. The use of the timed structure can allow for exact timing of the loop execution, programmatic change of loop timing, multi-loop timing and loop priorities.

When a timed loop is placed on the code diagram there is an input node on the left of the structure with an input for an error cluster. The input node can be resized to expose additional input parameters. Some of the inputs include the period of the loop execution, the starting offset, a loop timeout and a timing source. When the input node is viewed it starts with an icon representing the input type and the default value next to it. Once a value is wired to the input node the display will change to simply indicate what the parameter is. The programmer can wire values to the input node as they would with a subVI or a timed loop configuration window can be opened by right clicking on the loop structure. By default there are only two inputs

for timing source, priority and period. If additional settings are desired, click on the Configure Advanced Settings checkbox.

There is an output node on the right of the structure. Initially there is only one output available for the error cluster. Again, the node can be expanded to expose additional outputs. The loop can output the expected and actual end iteration values. There is a Boolean output indicating if the loop finished late. Finally, there is an output for the end time. By clicking on the timed loop structure you can make visible a right and left data node. This allows the loop to be able to read loop output values as well as to modify the input parameters.

To illustrate a simple example of timed loops the VI in Figure 1.38 uses timed loops to generate a clock display. The VI has 4 timed loops. The first three are loops for the hours, minutes and seconds. The seconds loop has two inputs. The first is the loop name. This input is needed to be able to abort the loop execution when the stop button is pressed. The second input is the execution time of the loop in milliseconds. Obviously for a second timer the period is 1000ms. The display is the remainder of

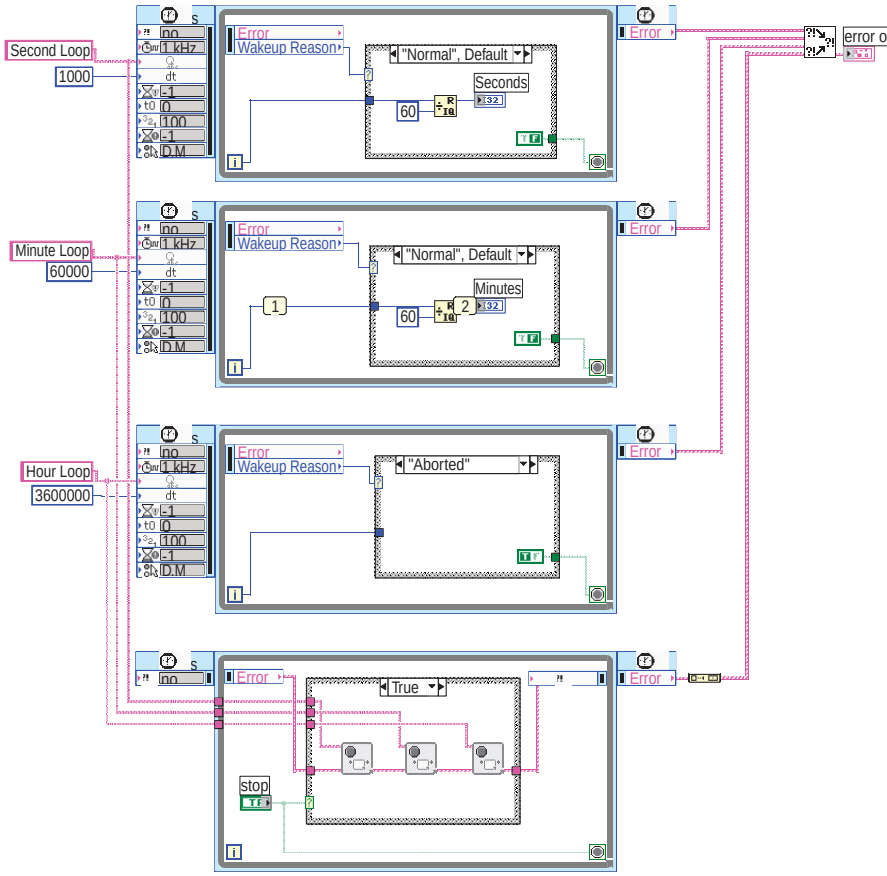


FIGURE 1.38

the loop index divided by 60 in order to reset the value every 60 seconds. The minute and hour loops are exactly the same with a different period setting.

In our example we want to be able to stop execution by clicking on a stop button. In order to perform this action we use the Stop Timed Loop function in the timed loop subpalette. The loop by default executes every 1000ms. If the value of the stop variable is true, the stop timed loop function is called for each of the three named loops. In order to stop a currently executing loop the left data node is needed in each of the three timed loops. The Wakeup Reason input value is Normal by default, but when a stop loop is received the input returns Aborted. This value is used to stop the loops.

1.7.1.8 Formula Node

A formula node is simply a bounded container for math formulas. It allows you to create formula statements, similar to programming in C. Multiple formulas can be enclosed in a single node, and each formula must end with a semicolon.

You can use as many variables as you wish, but you must declare each one as either input or output. Pop-up on the border of the formula node and select either Add Input or Add Output. A terminal is created on the border of the node for which you must enter the name of the variable. An output has a thicker border to help differentiate it from an input terminal. All input terminals must have data wired to them, but output terminals do not have to be used or wired to other terminals. Variables that are not intended for use outside of the Formula Node should be declared as output and left unwired. The input and output terminals can be created on any border of the structure.

The formula node is illustrated in Figure 1.39. The formula node contains a simple formula to demonstrate how it is used. It has one input variable, y, and one output variable, x. The output variable terminal has the thicker border and could have been moved to any location on the structure. The formula Node uses the input variable and calculates the output variable according to the formula created. Consult the Formula Node Syntax topic in Online Help to find out more information on creating formulas and the various operators that are available. You may also find the

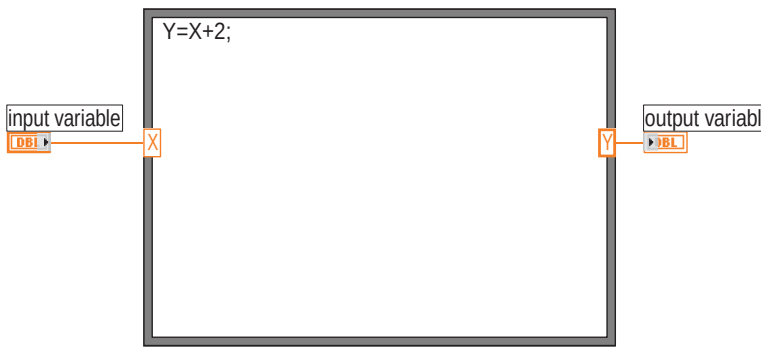


FIGURE 1.39

Formula Node Functions and Operators topic helpful to learn more about the different built-in functions offered.

One advantage of the formula node is that its operation is compiled internally to the node. Long formulas do not take up as much space on your display and can significantly reduce the number of elements in a wire table for the code diagram.

An alternative to using the formula node is the formula express VI. The user can select the formula express VI from the Arithmetic & Comparison palette in the Express palette. This places the express VI on the code diagram. The express VI looks similar to a subVI. Once the VI is placed on the code diagram (depending on your options settings) a configuration screen will display giving you the opportunity to define the formula. This is shown in figure "Express Formula.ai." Express VIs will be discussed further in Section 2.25.

1.7.2 Numeric, Boolean, String, and Comparison

The Numeric, Boolean, String, and Comparison palettes are displayed in Figure 1.40. The functions shown in the Numeric palette are straightforward and simple to use. The example in Figure 1.33, shown previously, utilized the multiply and increment functions. Most of them can be used for any type of number, including arrays and clusters. The multiply function, for example, requires two inputs and yields the product of the two.

The Numeric palette holds the Conversion, Complex, Data Manipulation and Additional Numeric Constants subpalettes. The functions in the Conversion subpalette are primarily used to convert numerical values to different data types. The Additional Numeric Constants subpalette holds such constants as Pi, Infinity, and e. One issue to note about floating point numbers in LabVIEW is that "not a number" quantities are defined. Values for +/- infinity are defined in floating point numbers, and division by zero will not generate an error but will return NaN (Not a Number). When performing calculations, it is up to the programmer (as always) to validate the inputs before performing calculations. The Data Manipulation subpalette contains functions such as flatten to string, split number and logical shift.

Numbers of various types will be converted when they are involved in a math operation. An integer and complex number will sum to be a complex number. The conversion performed is referred to as Coercion. Any numbers that are coerced will be labeled with a gray dot called a "coercion dot." Coercion is rarely a problem, but it needs to be understood that there is a small performance penalty for coercion between types. Numbers will never be converted "backwards," as a complex number being converted to an integer. Performing this type of conversion requires that you use a conversion method.

A rarely used property of floating point numbers is unit support. It is possible to define quantities with a unit attached. Popping up on any floating-point control, indicator, or constant on the diagram will allow you to expand the display menu. One of the display options is Unit. Once the unit is displayed, popping up on the unit shows the menu of units used by LabVIEW. LabVIEW supports sufficient unit types to make sure every chemistry, electronics, mechanical, and assembly lab has little to ask for, if anything. This feature works very well in simulation, measurement,

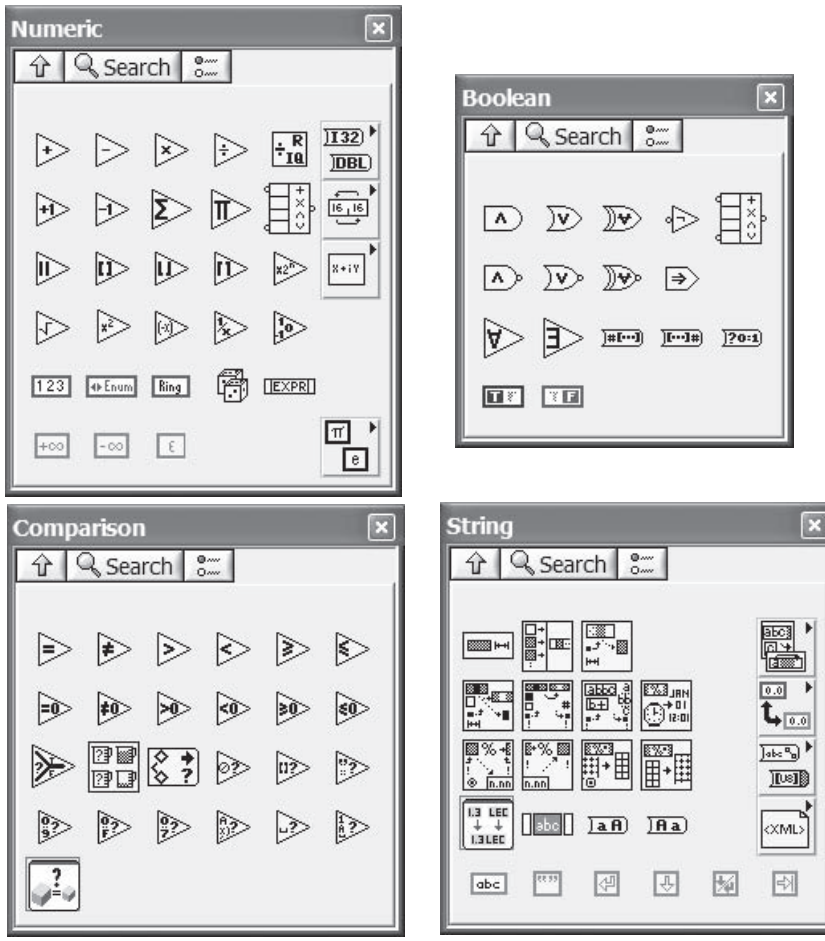


FIGURE 1.40

data display, and educational applications. Unit conversion is also possible, and is done behind the scenes. A floating-point number with a unit of feet can be wired to an indicator with a unit of miles. The display will show in miles; there is no need to perform conversion operations on the results of measurements. In some cases, this represents a possibility for performance enhancement because programmers who perform measurement conversions on their own need to add a few elements to their wire diagrams which will take more time to process. By default, floating-point numbers have no unit dimensions assigned to them.

In LabVIEW 8, there are several changes to the locations of some function palettes as well as some palettes that are available in more than one place. The numeric palette is one example. The Numeric palette is contained in the Programming and the Mathematics palette. Another change to the Numeric palette is the removal of the trigonometric and logarithmic subpalettes. They now are included

in the Elementary & Special Functions subpalette under Mathematics. This may be a little confusing at first to long time LabVIEW programmers, but you do adjust over time.

The Boolean palette holds various functions for performing logical operations. All of the functions require Boolean inputs, except for the conversion functions. A Boolean constant is also provided on this palette. The Comparison functions simply compare data values and return a Boolean as the result. You can compare numeric, Boolean, string, array, cluster, and character values using these functions.

Comparing arrays and clusters is a bit different from comparing primitive types such as integers. By default, LabVIEW comparison functions will return a single value for cluster and array comparison. If every element and the length of the arrays are equal, then a "true" is returned. A "false" is returned if there are any differences. If programmers want to compare an array element-by-element, the Compare Aggregate option can be enabled on the comparison operator. Popping up on the comparison operator will show Compare Aggregates at the bottom of the list of options. An aggregate comparison will return an array with Booleans for the result of a comparison of each and every element in the array or cluster.

Several string functions are provided on the Strings subpalette. Figure 1.41 illustrates the use of Concatenate Strings and String Length functions, the first two items on this palette. When Concatenate Strings is placed on the block diagram, two input terminals are normally available. You must pop up on the function and select Add Input if you wish to concatenate more than two strings at one time. Alternatively, you can drag any corner of the function up or down to add more input terminals. You cannot leave any terminal unwired for this function. The example shown has three inputs being concatenated. A control, a string constant, and a line feed character are concatenated and wired to the String Length function to determine the total length. Four subpalettes hold additional functions that perform conversion from strings to numbers, byte arrays, and file paths.

The Comparison palette is pretty straightforward. There are functions for comparing values. Many of these functions are polymorphic. For example the Equal function can compare two numbers, two strings, two arrays of numbers, an array of numbers and a scalar value, etc… The output is either a scalar Boolean value or an array of Boolean values depending on the inputs to the function. There are also functions for determining if a number is in a range, finding the minimum and maximum value and if the number is an empty path. The Select function outputs either the True or False input based on the Boolean input value. This

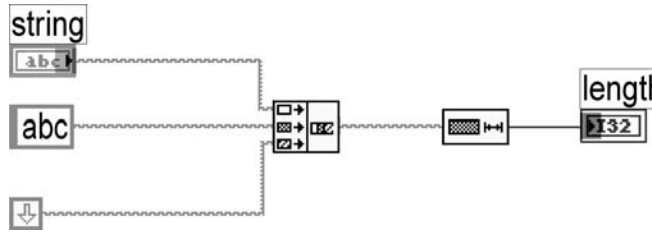


FIGURE 1.41

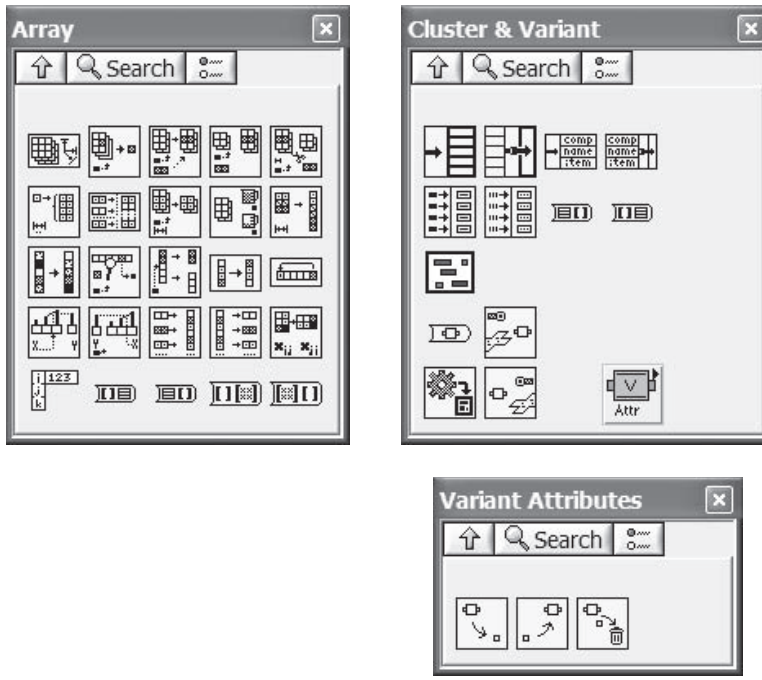


FIGURE 1.42

function is useful when adding conditional execution or exception handling into your application.

1.7.3 Array and Cluster

Array, Cluster & Variant, and Variant Attributes palettes are displayed in Figure 1.42. These palettes contain various functions for performing operations on these data constructs. The array functions provided can be used for multidimensional arrays. You must pop up on the functions and add a dimension if you are working with more than one dimension. Bundle and Unbundle functions are available for manipulation of clusters.

Figure 1.43 displays the front panel and code diagram of an example that uses both array and cluster functions. The front panel shows an array of clusters that contain employee information, similar to the example discussed in Section 1.5.5. This example demonstrates how to change the contents of the cluster for a specific element in the array. The Index Array function returns an element in the array specified by the value of the index wired to it, in this case 0. The cluster at Index 0 is then wired to the Bundle By Name function. This function allows you to modify the contents of the cluster by wiring the new values to the input terminals. Normally, when Bundle By Name is dropped onto the code diagram, only one element of the cluster is created. You can either pop up on the function to add extra items, or drag one of the corners to extend it. The item selection of the cluster can also be changed

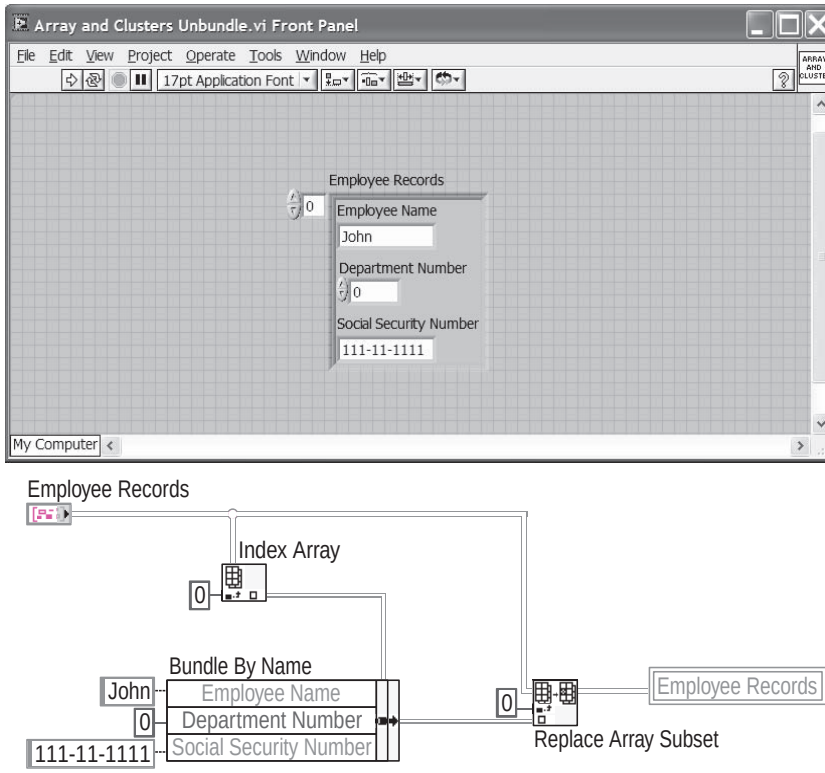


FIGURE 1.43

through the pop-up menu. New values are wired to the function as shown, and are then passed to the Replace Array Element function. This function simply replaces the original data with the values wired to the input terminals at the index specified. The output is then passed to a local variable of the Employee Records control. Local variables can be created by popping up on a control or indicator terminal from the code diagram. Select Local Variable from the Create submenu.

If you work with arrays, one of the array functions with which you should become very familiar is the Dimension array. This function will allow you to set the dimensions on an array. LabVIEW will expand array sizes to prevent users from overwriting the boundaries of an array, but this is bad practice. Each time LabVIEW needs to change the number of elements in a dimension, it must get a memory allocation sufficient to hold the array and copy each and every element into the new array. This is very inefficient, and is a bad programming habit. Pre-dimensioning arrays when you know the length in advance is an efficient habit to develop. The other array function you will become familiar with is the Replace Array element. This function allows you to change the value of an element in an array without duplicating the array.

Other functions in these palettes allow you to perform several other operations on arrays and clusters. The Cluster & Variant palette contains an Unbundle function

for retrieving data from a cluster, a function for building cluster arrays, and functions for converting data between clusters and arrays. The Array palette holds functions for detecting array sizes, searching for a specific value in an array, building arrays, decimating arrays, and several other operations. If you are interested in creating easily-read GUIs, the conversion functions between arrays and clusters is something you will want to look into. On occasion, it will be desirable to use array element access in your application, but arrays on the front panel can be difficult to read. Displaying data on the front panel in the form of a cluster and converting the cluster to an array in the code diagram makes both users and programmers happy.

The variant functions have been combined with the cluster functions on the same palette. A variant is a unique data type that can contain string, numeric, date or user-defined data. The user will need to know how the variable was declared in the original function to be able to properly read and write to a function using the variant data type. The variant functions are the To Variant function, Variant to Data, Variant to Flattened String and Flattened String to Variant functions. These functions are all used to convert LabVIEW data types to or from Variant data types.

On the Cluster & Variant palette there is a Variant Attribute subpalette. This subpalette contains three functions used to manipulate variant data. The attribute functions allow the programmer to set, get and delete variant attributes. The Get Variant Attribute function gives the programmer the ability to either get the names and values of all of the variant attributes or the value of a specified attribute.

1.7.4 Timing

The Timing palette, displayed in Figure 1.44, contains functions for retrieving the system time, wait functions for introducing delays, and functions for formatting time and date strings. The Wait Until Next Multiple function is useful for introducing delays into loop structures. When placed inside a loop, it causes the loop to pause a specified time between iterations of execution. There is a Time Delay Express VI

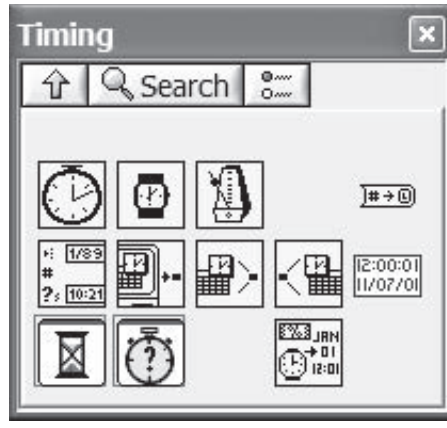


FIGURE 1.44

in this palette that has an error in and out that gives the programmer the ability to control the order of execution. The functions on this palette are simple to use and are self-explanatory.

System dates and times are dependent on the system you run on. Most computers measure the date in the number of seconds that have elapsed since a certain time, for example January 1, 1974, at 12:00am. This number is stored in a 32-bit number and it will be an extremely long time from now before this date rolls over (consider that there are approximately pi * 10^7 seconds in a year). The concern with system dates and times is because of the precision you need. As just mentioned, it is stored in units of seconds. If you need millisecond accuracy, system date and time are not going to be sufficient. Some systems will store hundredths or even tenths of a second, but millisecond accuracy is usually not possible with system times.

1.7.5 Dialog and User Interface

The Dialog & User Interface Palette is shown in Figure 1.45. Dialog boxes are great for informing users that something is happening in the system. Dialog boxes should usually be avoided in automated applications where a user is not monitoring the testing. A dialog box will halt LabVIEW's execution until somebody clicks the "OK" button. If you have an automated system that is expected to run while you are on vacation, it may be a while before you click the button to complete your testing. The dialog functions give the programmer the ability to name the buttons and specify the message displayed.

There are two express dialog functions on this palette. One is an Express Dialog function. This VI acts the same as the standard dialog box, but provides a means for controlling execution flow through the use of the error clusters. The express VI also makes configuring the function easier at the expense of visibility. The second express VI is called Prompt Users for Input. This function displays a dialog box

FIGURE 1.45

with an expandable number of input controls. The programmer can set up each input for numeric, string or checkbox entry. The function outputs the user-provided values.

There are four subpalettes on the Dialog & User Interface palette. The Events subpalette gives the programmer the ability to set up and use user events for program control. The subpalette also contains the Event Structure that is part of the Structures palette. The Menu subpalette contains functions used to add, remove and modify the runtime menus. This can be very useful in an application to give flexibility without creating clutter on the front panel. The Cursor subpalette gives the programmer the ability to control the cursor appearance and disable the mouse on the front panel. The Help subpalette gives control of the help windows and can call up a Web page in the default browser.

The error handler functions are also included in this palette. Chapter 6 covers the topic on exception handling and describes the error handler VIs in more detail. The merge error function merges error I/O clusters from up to four different functions. This function first looks for errors in the four inputs in order and reports the first error found. If the function finds no errors, it will look for warnings and return the first warning found. If the function finds no warnings, it will return no error. This is a helpful function when you have two or more sections of code running in parallel and want to combine the error outputs into a single path.

1.7.6 File I/O

Figure 1.46 shows the File I/O palette in addition to one of its subpalettes, the Advanced File Functions. The basic functions allow you to open, close, create, read from, or write to files. These functions will display a dialog box prompting the user to select a file if the file path is not provided. The advanced functions facilitate accessing file and directory information, modifying access privileges, and moving a file to a different directory, among several others.

LabVIEW's file interfaces give programmers as much or as little control over the file operations as desired. If you want to simply write an array to a tab-delimited file, there is a function to do just that. Supplying the array is about all that is necessary. The interface is very simple; you do not have much control over what the file handler will do. Lack

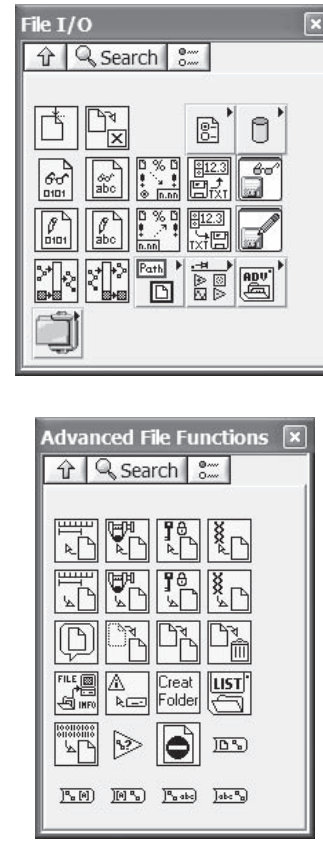


FIGURE 1.46

of control should not be a concern for you if your purpose is to write the tab-delimited string to a file. In fact, the string conversion is done in the function also.

Programmers who are concerned about the amount of space needed by a large set of data can use binary access files. Binary access files will put the bit pattern representing the array directly into the file. The advantages of binary files are the sizes they require. A 32-bit number stored in a binary file takes exactly 32 bits. If the number is stored in a hex format, the number would be 8 digits, requiring 64 bits to store, twice as long. Floating-point numbers have similar storage requirements, and binary files can significantly reduce the amount of disk space required to handle large files.

Binary files also allow programmers to make proprietary file formats. If you do not know the order in which data is stored, it is extremely difficult to read the data back from the file. We are not encouraging developers to make proprietary storage formats — the rest of the engineering community is driving toward open standards — but this is an ability that binary files offer.

Depending on the data being stored in the binary file, the amount of work you need to do is variable. If arrays of numbers are being written to the file, there are binary access VIs to read and convert the numbers automatically. The binary read and write VIs support any data type used in LabVIEW.

If you are trying to write data-like clusters to binary files, there are two options you can use. The first option is to flatten the clusters to a string and write the string to a file. Flattened strings will be binary. File interfaces will be easy to use, but reading back arrays of flattened clusters will be a bit more difficult. You will need to know the length of the flattened string, and be able to parse the file according to the number of bytes each cluster requires. Be sure to provide a robust error handler; the conversion might not work and return all manner of useless data if things go awry. The second option is to use the read and write files directly. Read and write from file is used by all of the higher level file functions, but does not open or close the files directly; you will need to call File Open and Close, in addition to knowing what position in the file to write to.

In general, we do not recommend using binary access files. Binary files can be read only by LabVIEW functions, and a majority of the reasons to use binary files are obsolete. Modern computers rarely have small hard drives to store data; there is ample room to store 1000-element arrays. Other applications, such as spreadsheets, cannot read the data for analysis. Binary files can also be difficult to debug because the contents of the file are not readable by programmers. ASCII files can be opened with standard editors like VI, Notepad, and Simpletext. If parsing or reading file problems show up in your code, it is fairly easy to open up an ASCII file and determine where the problems could be. Binary files will not display correctly in text editors, and you will have to “roll your own” editor to have a chance to see what is happening in the file.

Many programmers use initialization files for use with their applications. LabVIEW supplies a set of interfaces to read and write from these types of files. The “platform independent” configuration file handlers construct, read, and write keys to the file that an application can use at startup. Programmers who do not use Windows, or programmers who need to support multiple operating systems, will

find this set of functions very useful. There is no need to write your own parsing routines. Data that may be desired in a configuration file is the working directory, display preferences, the last log files saved to, and instrument calibration factors. These types of files are not used often enough in programming. Configuration files allow for flexibility in programs that is persistent. Persistent data is data that is written to the hard disk on shutdown and read back on startup.

The Advanced File Function subpalette contains VIs to perform standard directory functions such as change, create, or delete directories. This subpalette has all the major functions needed to perform standard manipulations, and the interface is much easier to use than standard C.

There is a subpalette for interacting with ZIP files. There are functions for creating a new zip file, adding files to an existing zip file and a close zip file function. These functions can be useful when archiving large data files.

1.7.7 Instrument I/O, Connectivity, and Communication

The Instrument I/O and Connectivity palettes contain various built-in functions to simplify communication with external devices. These two palettes along with the Communication subpalette are displayed in Figure 1.47 representing how they appear on a Windows system. The Instrument I/O palette holds VISA, GPIB, Serial, and VXI-related functions. The Connectivity palette contains functions for ActiveX, .NET, Input Devices (keyboard, mouse), Windows Registry Editing, Source Code Control, Communications, Libraries & Executables, and Graphics & Sound. The Communications subpallette contains functions for TCP, UDP, Data

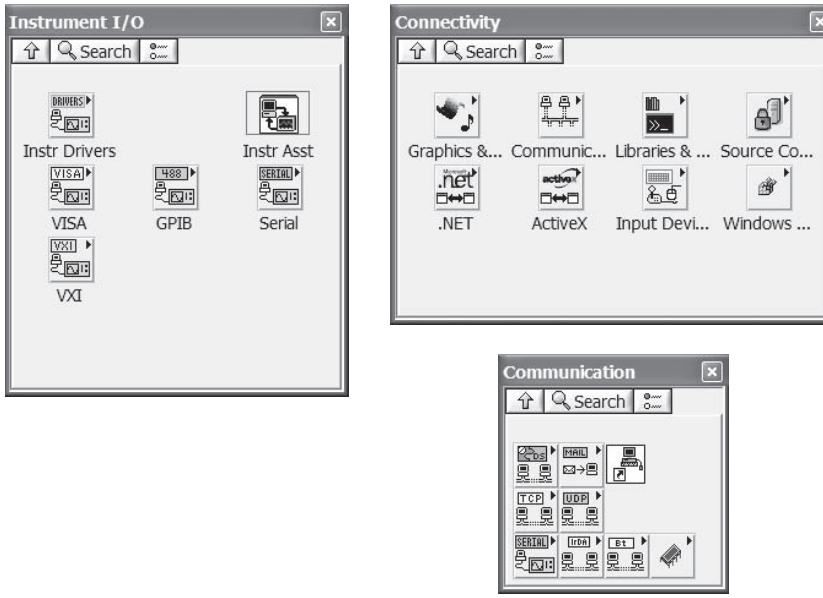


FIGURE 1.47

Socket, Bluetooth, IrDA, SMTP and Port I/O. The specific functions in these palettes will not be discussed in this book; however, Chapter 8 covers ActiveX and .NET in detail.

When designing an application, there may be a few minor details you should consider for communications. Inter-application communications do not involve cables such as GPIB. Windows-specific communications can be done with ActiveX /COM functionality or with .NET. ActiveX and .NET are the current standards for communications in Windows environments.

The only globally available communications protocols are the Unix standards TCP and UDP. Both protocols utilize the Internet protocol (IP). IP-based communications do not need to be between two different computers; applications residing on the same computer can communicate with TCP or UDP connections. TCP or UDP is recommendable because the interfaces are easy to use, standard across all platforms, and will not be obsolete anytime soon.

GPIB, serial, and VXI communications should be performed with the VISA library. VISA is becoming the standard for instrument communications in LabVIEW. The serial support has already been converted to VISA. The serial VIs available in the Instrument I/O palette are built upon VISA subVIs. The IEEE 488 will be supported for some time, but the VISA library is intended to provide a uniform interface for all communications in LabVIEW. Addressing, sending, and receiving from an external device all use the same VISA API, regardless of the communications line. The common API lets programmers focus on talking to the instruments, not on trying to remember how to program serial instruments.

The Instrument I/O palette also contains an express VI called Instrument I/O assistant. This express VI can be used to communicate with instruments and graphically parse the response. This functionality is useful when trying to verify the instrument is connected correctly. This assistant can also be used while verifying the accuracy of a GPIB command before using it in a driver.

LabVIEW VIs are very similar to functions or subroutines in programming languages like C. Once created, VIs can be called inside of other VIs. These subVIs are called simply by placing them on a code diagram, similar to dragging a function from the palettes as discussed in the last section. SubVIs are represented on the block diagram by an icon that you can customize to distinguish it from other subVIs. Once placed on the code diagram, wire the appropriate input terminals to ensure that it will execute correctly. This section explains the activities related in setting up and calling subVIs.

1.7.8 Creating Connectors

VIs can have inputs and outputs, similar to subroutines. A connector must be defined for a subVI if data is to be exchanged with it. It will be necessary for you to define connectors for most VIs that you create. The process consists of designating a terminal for each of the controls and indicators with which data will need to be exchanged. Once the inputs and outputs have been appointed terminals, data can be exchanged with the VI on a block diagram.

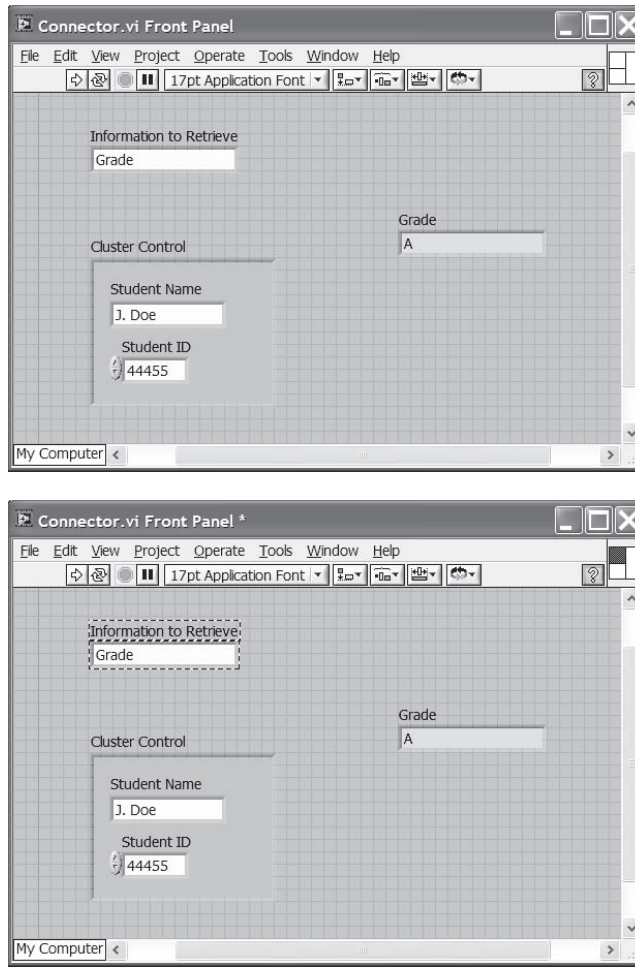


FIGURE 1.48

Figure 1.48 displays the front panel of a VI with the connector pane visible in the top right corner of the window. To display the connector pane on a VI, pop up on the icon that is normally visible and select Show Connector from the menu. Three rectangles or terminals appear in the example, one for each control and indicator. Each control and indicator can be assigned a terminal by using the wiring tool. Click on one of the terminals, then click on a control or indicator to designate the terminal.

The bottom window in Figure 1.48 illustrates how the Information to Retrieve control is assigned the top left terminal on the connector. By default, LabVIEW creates a terminal for each control and indicator on your front panel, but the assignment will be left to the programmer. If the default connector pattern is not appropriate, it can be modified to suit your needs. Once the connector is made visible, use the items in the pop-up menu to select a different pattern, or rotate the current pattern.

Controls and indicators can be assigned to any terminal on the connector. However, controls can only serve as inputs, and indicators can only be used for outputs. You should assign the inputs on the left terminals of the connector and the outputs to the right side, even though you are not required to. All LabVIEW built-in functions follow this convention. This convention also aids the readability of the code. The data flow can be followed easily on a block diagram when subVIs and functions are wired from left to right.

Built-in LabVIEW functions have inputs that are either required, recommended, or optional. If an input is required, a block diagram cannot be executed unless the appropriate data is wired. Correspondingly, LabVIEW allows you to specify whether an input terminal is required. Once you have designated a particular terminal to a control, pop up on that terminal and select This Connection Is from the menu. Then select either Required, Recommended, or Optional. Output indicators have the required option grayed out in the menu. Output data is never required to be wired.

Good programming practice with subVIs is fairly simple. It is a good idea to have a few extra connectors in your VI in case additional inputs or outputs are needed in the future. Default values should be defined for inputs. Defined default values will allow programmers to minimize the number of items on the calling VI's code diagram, making the diagram easier to read. Supplying the same common input to a VI is also tedious; granted, it is not impossible work to do, but it becomes boring. Laziness is a virtue in programming; make yourself and other programmers perform as little work as possible to accomplish tasks.

1.7.9 Editing Icons

Icons are modified using the Icon Editor. Either double-click the default icon in the top right corner of the window or pop up on it and select Edit Icon from the menu. Figure 1.49 is an illustration of the Icon Editor containing a default LabVIEW VI icon with a number. This communicates the number of new VIs opened since initiating the LabVIEW program. Each time you start LabVIEW, the VI contains a "1" in the icon as the default.

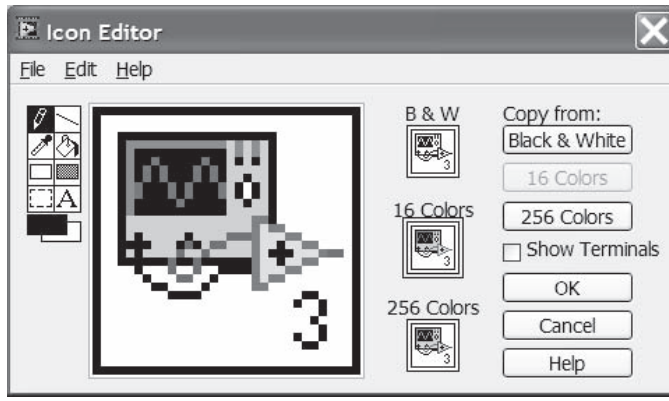


FIGURE 1.49

The Tools palette is located on the left side of the Icon Editor window, and the editing area is in the center. The default foreground color is black, while the background color is white. When you click on the background/foreground color tool, a color palette appears allowing you to select from among 256 colors. You can create different icons for black-and-white, 16-color, and 256-color monitor types. Many people create an icon in color and forget to create in black and white. This is important when you need to print out VI documentation; if you are not using a color printer, the icon will not appear as it should. Try to copy the icon you created from the color area to the black-and-white area.

Figure 1.50 demonstrates the process of customizing an icon. The top window in the figure displays an icon that has been partially customized. First, the contents of the editing area were cleared using the Edit menu. Then, the background color was changed to gray while the foreground was left as black. The Filled Rectangle tool was used to draw a rectangle bordered with a black foreground and filled with a gray background. If you double-click the tool, the rectangle will be drawn for you automatically. The second window displays the finished icon. The Line tool was used to draw two horizontal lines, one near the top of the icon and the other near

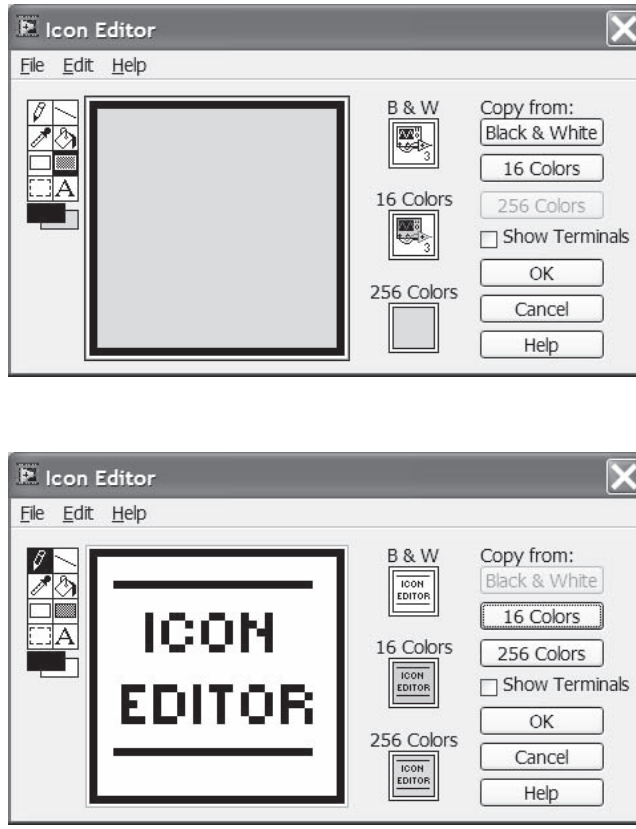


FIGURE 1.50

the bottom. Then, the Text tool was used to write "icon editor" in the editing area. Finally, the same icon was copied into the 16-color and black-and-white icon areas.

Because the icons are graphical representations of the VIs, you can use your imagination and get creative when editing them, if you wish. JPEG- and GIF-formatted picture files can be copied and pasted into the icon editing areas also. Although this can be fun, just remember that the purpose of customizing icons is to allow people to distinguish the VI from other VIs and icons in a program. Try to create icons that are descriptive so that someone looking at the code for the first time can determine its function easily. Using text in the icons often helps achieve this goal. This helps the readability of the code as well as easing its maintenance. Veteran programmers quickly abandon the process of taking an hour to develop an appealing work of art for an icon. We have all had those VIs with the extraordinary icons that were deleted because they became unnecessary in the project.

1.7.10 Using SubVIs

The procedure for using subVIs when building an application is similar to dragging built-in functions from a palette onto the block diagram. The last item on the Functions palette, displayed in Figure 1.24, is used to place subVIs onto block diagrams. When Select a VI is clicked, a dialog box appears prompting you to locate the VI that you want to use. Any VI that has already been saved can be used as a subVI. Place the VI anywhere on the code diagram and treat it as any other function. Once the required inputs have been wired, the VI is ready for execution.

1.7.11 VI Setup

The VI Setup window gives you several options for configuring the execution of VIs. These options can be adjusted separately for each VI in an application. To access this configuration window, pop up on the icon in the top right corner and select VI Setup from the menu. This window is displayed in Figure 1.51, with the Execution Options selected in the drop down box at the top.

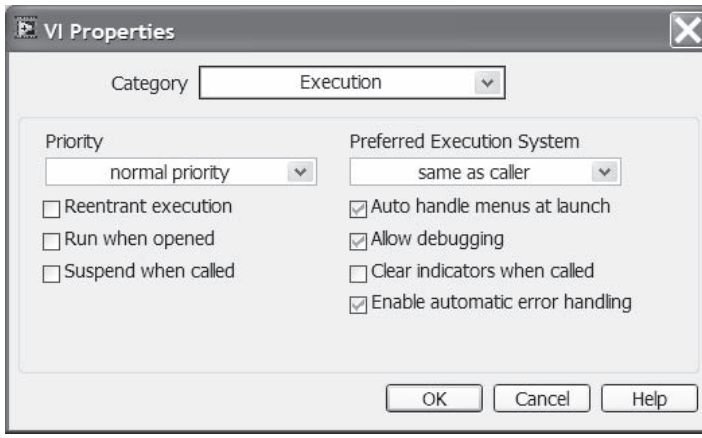


FIGURE 1.51

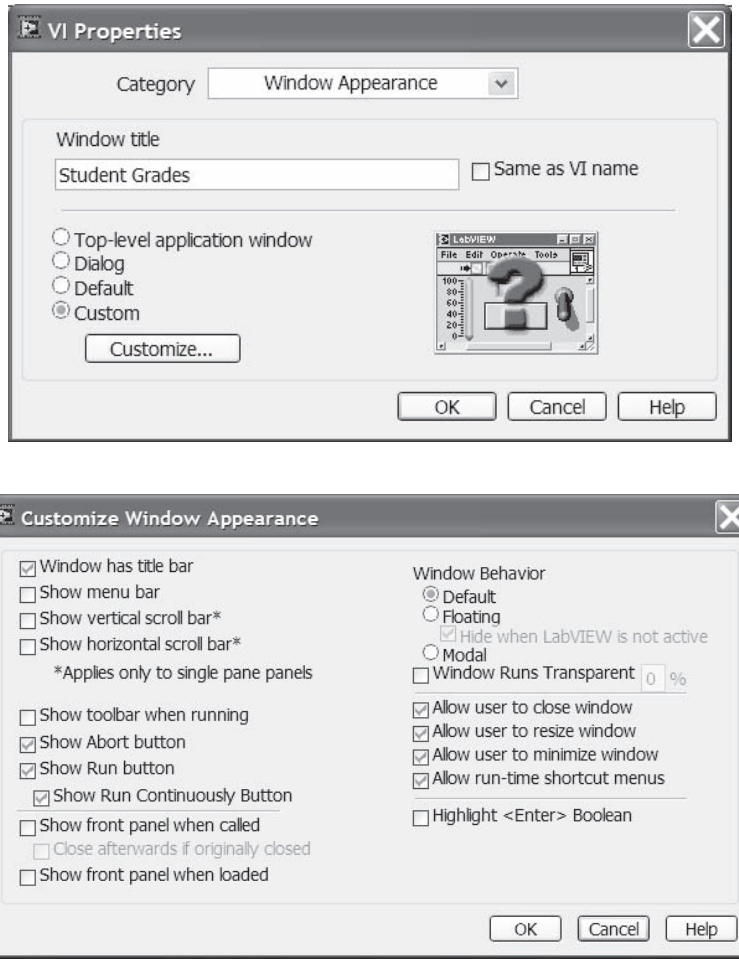


FIGURE 1.52

The items on the execution options panel are used for configuring the VIs execution priority, allowing reentrant execution and setting the preferred execution system. Reentrant execution refers to making multiple calls to the same VI and is covered in the next chapter. VI priority and the execution system selections are used for optimizing the execution of an application. These two topics are discussed further in Chapter 9, which also covers multithreading. We strongly recommend not working with either priority or execution subsystem until you read Chapter 9. This is one of those topics in which not understanding how threads and priorities interact can do more harm than good.

Figure 1.52 displays the VI setup window with Window Appearance Options selected in the drop-down menu. Initially you have the option of Top Level Application Window, Dialog, Default or Custom. These configuration selections allow you to customize the appearance of the VI during execution. The first three options

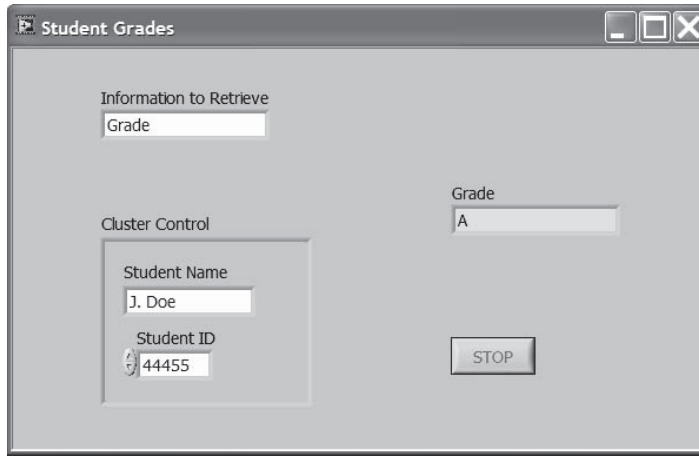


FIGURE 1.53

have predefined settings for appearance and function. The custom option allows the user to setup the VI exactly the way it is needed. In the example shown, Show Scroll Bars, Show Menu Bars, and Show Toolbar have been deselected. These are all enabled by default. There are also checkboxes used to show the front panel when it is called, and to close the panel after it has finished executing. The "Same as VI Name" has also been deselected and the Window Title modified. These alterations cause the VI to appear as shown in Figure 1.53 during execution. When the Stop button is pressed, the front panel returns to its normal appearance. Window options are useful for limiting the actions available to the end user of the program.

Figure 1.54 displays the VI Documentation and Revision History windows. LabVIEW provides some built-in documentation support that can be configured through either VI Setup or Options. A VI history is kept for each VI that is created. This history is used to keep records of changes made to a VI, and serves as documentation for future reference. The "Use the default history settings from the Options dialog box" checkbox has been deselected in the example shown. This informs LabVIEW to use the settings from the VI setup instead of the Options Dialog. The Option settings also allow you to configure the VI history, but this checkbox determines which ones are used.

Also note that two boxes have been checked which configure LabVIEW to add an entry to the VI history every time the VI is saved, and also to prompt the programmer to enter a comment at the same time. The entry LabVIEW adds consists of the time, date, revision number, and the user name. The programmer must enter any comments that will provide information on the nature of the modifications made. Figure 1.55 illustrates the VI history for the VI shown earlier in Figure 1.53. The VI history can be viewed by selecting Show History under the Windows pull-down menu. Chapter 4 discusses the importance of documentation and reveals other documentation methods for LabVIEW applications.

Initially you have the option of Top Level Application Window, Dialog, Default, or Custom. This section describes how VIs, once developed, can be used as subVIs

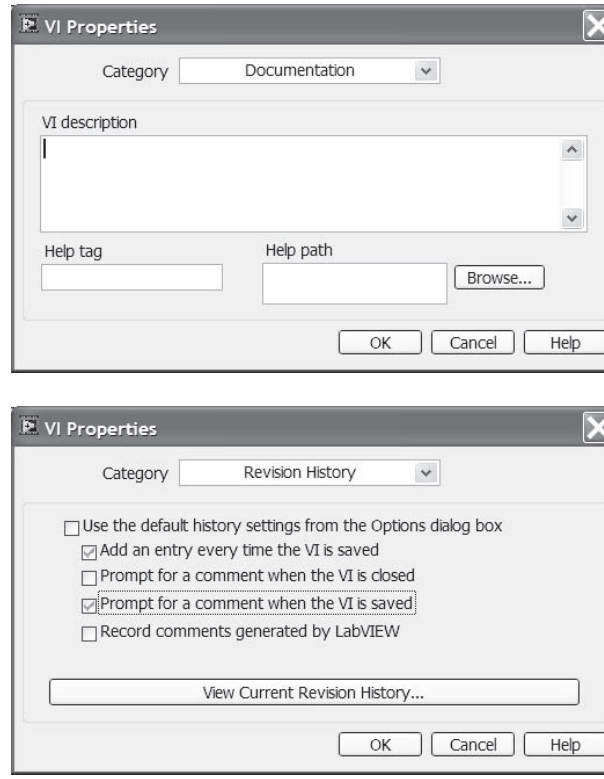


FIGURE 1.54

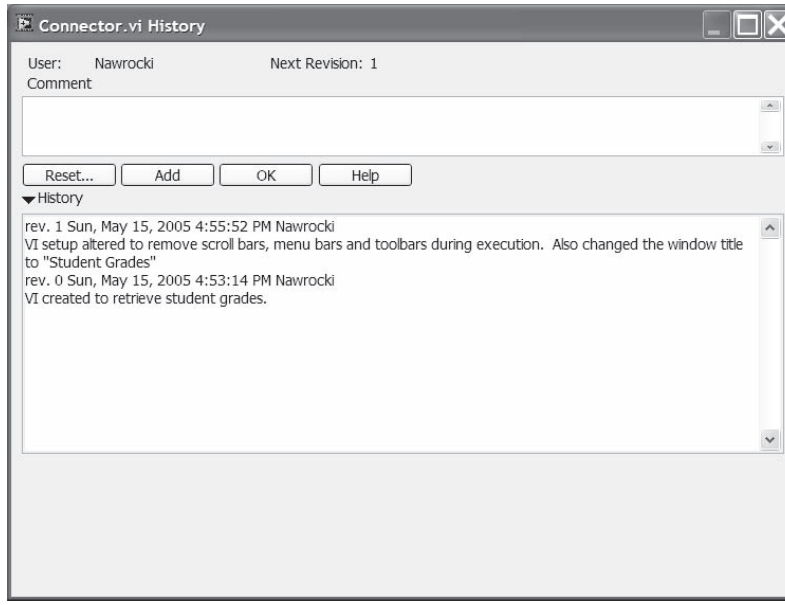


FIGURE 1.55

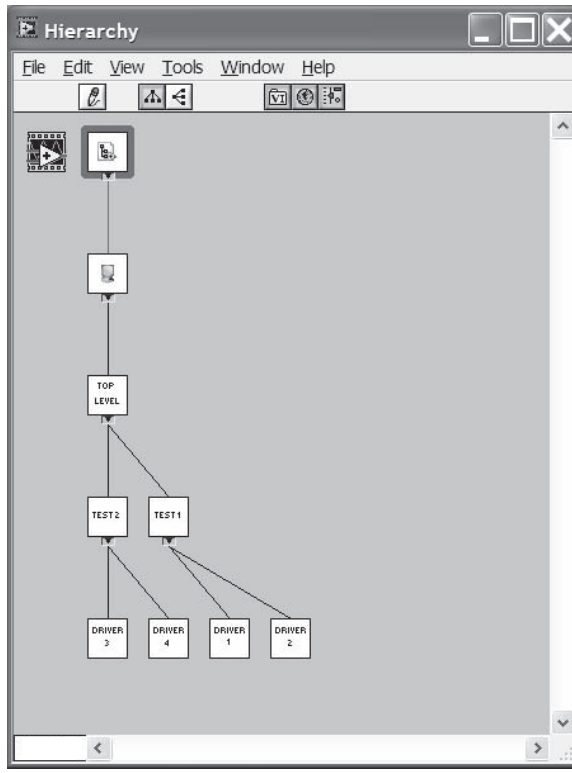


FIGURE 1.56

in larger applications, creating a hierarchy of VIs in an application where layers are created. These layers, or tiers, must be managed during development to increase the readability, maintainability, reuse, and abstraction of code.

Figure 1.56 shows the hierarchy window of a relatively small application. The hierarchy window can be displayed for any VI by selecting Show VI Hierarchy from the Project pull-down menu. This window graphically shows the relationship of a VI to the application. It displays the VI, its callers, and all of the subVI calls that it makes. The hierarchy window shown in the figure corresponds to the main VI at the top. There are two layers of VIs below the main. In this example, the application was developed with three tiers: the main level, the test level, and the driver level.

The inherent structure of LabVIEW allows for reuse of VIs and code. Once a VI is coded, it can be used as a subVI in any application. However, a modular development approach must be used when creating an application in order to take advantage of code reuse. Application architecture and how to proceed with application development are the topics of Chapter 4. This chapter also discusses how to manage and create distinct tiers to amplify the benefits offered by the LabVIEW development environment.

Instrument drivers play a key role in code reuse with LabVIEW. Chapter 5 introduces a formula for the development of drivers to maximize code reuse, based

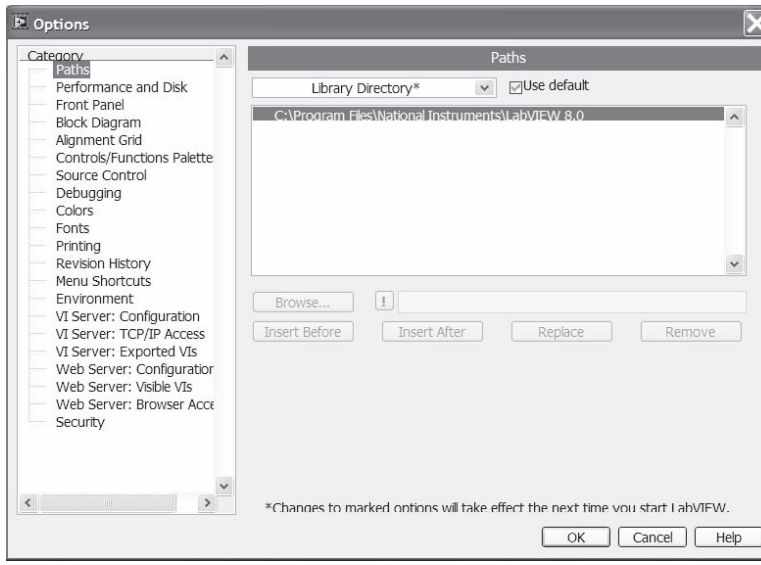


FIGURE 1.57

on National Instruments development method. When this formula is followed, the result is a set of drivers that can be reused in any application while providing abstraction for this lowest tier in the hierarchy.

The intrinsic modularity of LabVIEW can be used to apply an object-oriented methodology to application development. LabVIEW itself is not an object-oriented language; however, it is object-based. The object-oriented approach can be applied to LabVIEW, though in a limited manner. Chapter 10 introduces you to the terminology associated with Object-Oriented Programming, as well as how to apply it in a LabVIEW environment.

1.8 SETTING OPTIONS

This section describes some of LabVIEW's options or preferences that can be configured to suit a programmer's needs. The options selection is available in the Tools pull-down menu. The window that appears is shown in Figure 1.57 along with its default settings. The options shown correspond to the Paths selection from the top drop-down menu. Some of the option selections are self-explanatory and will not be discussed in this section; however, Table 1.3 lists all of the selections and describes the notable settings that can be configured in each.

1.8.1 PATHS

The Paths configurations, shown in Figure 1.57, dictate the directories in which LabVIEW will search when opening or saving libraries, VIs, menus, and other files. The drop-down menu selector allows you to configure the Library, Temporary, Default and Data directories. The last selection in this menu is used to set the VI

TABLE 1.3
Option Headings

Option Selection	Function/Utility
Paths	Configure search directories for opening/saving VIs.
Performance and Disk	Configure to use multithreading and perform check for available disk space prior to launch.
Front Panel	Settings for front panel editing.
Block Diagram	Settings for block diagram programming.
Alignment Grid	Settings for aligning objects on front panel and code diagram to specified grid spacing.
Controls/Functions Palettes	Settings for palettes.
Source Control	Settings for source code control of LabVIEW files.
Debugging	Options that are used for debugging VIs, and execution highlighting during execution.
Colors	Change default colors used by LabVIEW for front panel, block diagram, etc.
Fonts	Settings for Applications, System, and Dialog Font styles.
Printing	Configure print settings.
Revision History	Options for recording revision comments when changes are made to VIs.
Menu Shortcuts	Allows setting key combination shortcuts to menu functions.
Environment	Options for tip-strips, native file dialogs, drop-through clicks, hot menus, auto-constant labels, opening VIs in run mode, and skipping navigation dialog at launch.
VI Server: Configuration	Configure protocols, port numbers, and server resources.
VI Server: TCP/IP Access	Set access privileges to specific list of clients for VI Server.
VI Server: Exported VIs	Specify list of VIs that are accessible to clients using VI Server.
Web Server: Configuration	Enable Web server, configure root directory, set port number and timeout.
Web Server: Browser Access	Set access privileges to specific list of clients for Web server.
Web Server: Visible VIs	Specify list of VIs that are accessible to clients from Web server.
Security	Allows setting of password for LabVIEW login.

Search Path. This informs LabVIEW of the order in which to search directories when opening VIs. When you open a VI containing subVIs that are not part of a library, this search order will be followed to find them. You can configure this to minimize the time it takes to search and find subVIs.

If your group uses a number of common VIs, such as instrument drivers, the directories to the drivers should be added to the VI search path. Current projects should not be added to the search path. The VI search path was intended to allow programmers to easily insert common VIs. The VIs that are written as part of a project and not intended to be part of a reusable library would end up cluttering up the search path, lengthening the time LabVIEW takes to locate VIs.

1.8.2 Block Diagram

Figure 1.58 displays the Block Diagram options window. These options are intended to help you develop code on the block diagram. For the beginning user of LabVIEW,

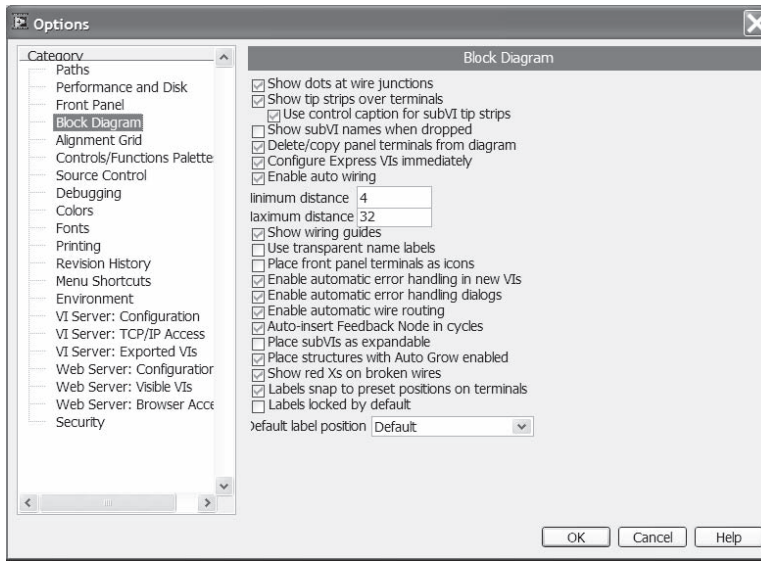


FIGURE 1.58

some of these settings can help you get familiar with the programming environment. Tip-strips, wiring guides, and junction dots are very useful when wiring data to functions and subVIs. Displaying subVI names is also handy because the icons are not always descriptive enough to determine their roles in an application.

1.8.3 Environment

The Environment option window offers the ability to set preferences for the LabVIEW environment. Options for the environment include auto tool select, Just In Time (JIT) advice, undo and abridged menus. Undo and Redo are both available in the Edit pull-down menu. When the option box is unchecked, you can change the default number from 8 to another suitable number. Keep in mind that a higher number will affect the memory usage for your VIs during editing. Since actions are recorded for each VI separately, the number of VIs that you are editing at any one time also affects memory usage. Note that once a VI is saved, the previous actions are removed from memory and cannot be undone. The Environment Option Window is shown in Figure 1.59.

1.8.4 Revision History

The Revision History options window is displayed in Figure 1.60. Some of these options are duplicated in the VI History settings under VI Properties. If you compare this to Figure 1.54, you will notice that the first four checkboxes are the same. If you have the Use History Defaults box checked in the VI Property window settings, LabVIEW will use the Revision History options.

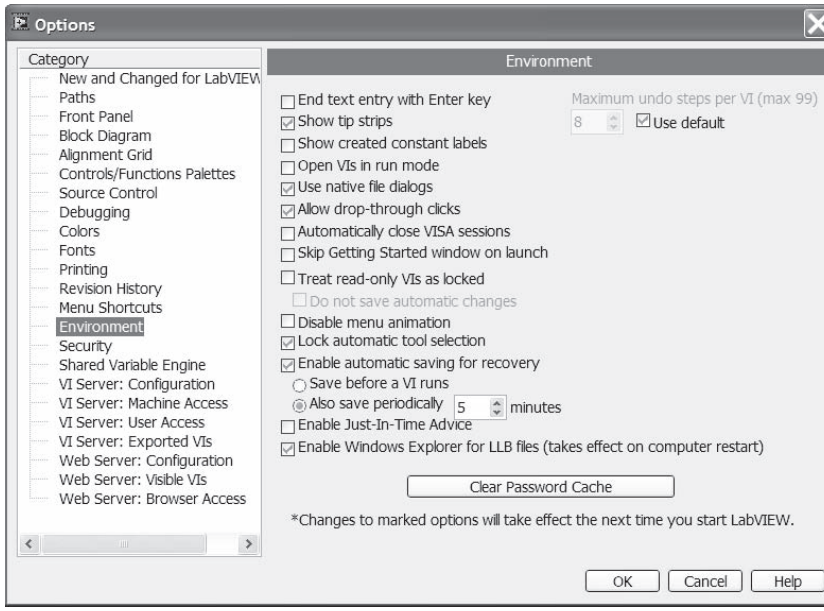


FIGURE 1.59

The radio buttons let you configure the login settings for LabVIEW. These settings will be used to determine the name entered by LabVIEW in the VI History box that records the comments when an entry is made. The second window in Figure 1.60 shows the User Login information. The login name can be modified in this window and is accessed by selecting User Name from the Tools menu.

Using the VI history is simply good programming practice. Listing the change history of a VI allows other programmers to understand what modifications a VI has which can be used to help debug applications. It does not take too long with troubleshooting before you see why an application stopped working because "someone else" made a modification to code and did not communicate or document the modification. Using history alone is not quite enough. When making comments in the history, note the changes that were made, and, equally important, note why the changes were made. It is fairly common practice to comment code as you write it, but to not keep the comments up to date when modifications are made. Giving other programmers a hint as to why a change was made allows them to see the thought process behind the change.

1.8.5 VI Server and Web Server

The VI Server functionality is a feature that was added to LabVIEW in Version 5.0. It allows you to make calls to LabVIEW and VIs from a remote computer. You can then control them through code that you develop. This also permits you to load and run VIs dynamically. Chapter 8 describes the VI Server in more detail along with the related configurations and some examples.

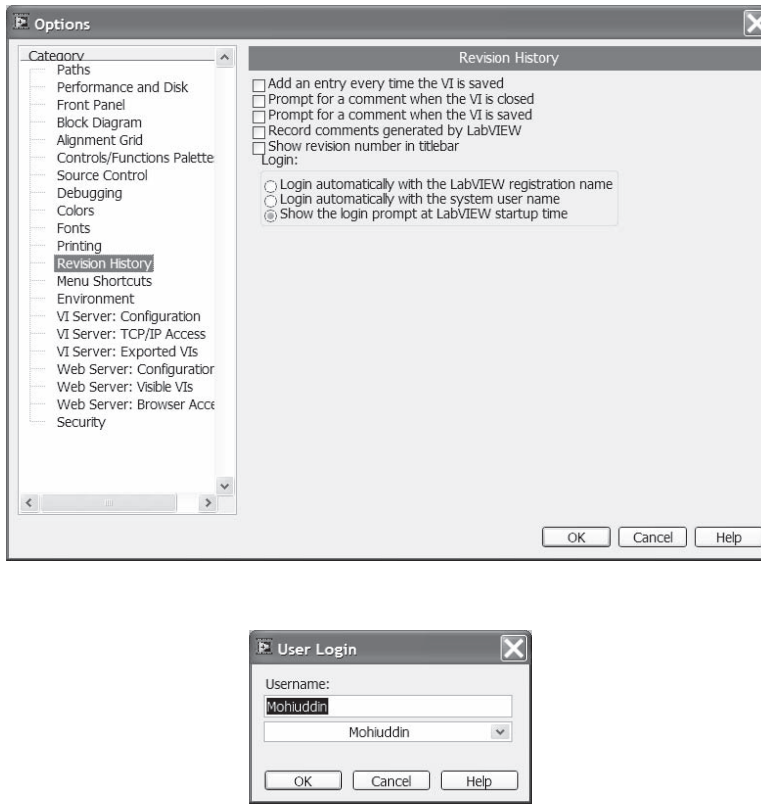


FIGURE 1.60

The Web Server is also an addition to LabVIEW starting in Version 5.0. The built-in Web server must be enabled through the preference settings. The Web server will allow you to view any VIs that are loaded on the same machine using a browser. You can then view the front panel of a VI that may be running from any remote machine. The Web Server and its configurations are discussed further in Chapter 2.

1.8.6 Controls/Functions Palettes

LabVIEW normally displays the default palettes for both Controls (Figure 1.9) and Functions (Figure 1.24). You can change the palette view to match your programming needs by either selecting a new palette set or creating your own palette. The view can be changed easily through the Options menu. The programmer has the option to show the standard icons and text, all icons, all text or tree view by using the Format pull down menu. The Control/Function Palette is shown in Figure 1.61.

Select Tools, Advanced, and then Edit Palette Views to create and customize a new palette set. A window similar to the one shown in Figure 1.62 will appear that will allow you to perform this action. Then select New Setup from the drop-down menu box and enter a name for the new view. A view called “Personalized” was

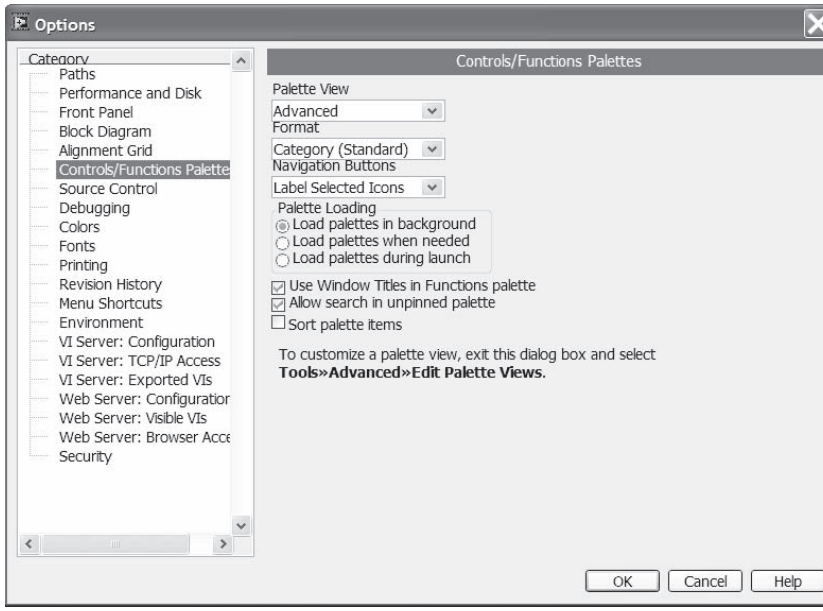


FIGURE 1.61

created for the example in Figure 1.62. The customized Functions palette is also shown, along with the modified User Libraries subpalette. A new setup must be created because LabVIEW does not directly allow you to modify the default palette set. It serves as protection in case the changes a user makes are irreversible.

Once you have created the new setup, the Functions and Controls palettes contain the default subpalettes and icons. The user is allowed to move, delete, and rename items in the palettes as desired. All of the available editing options are accessible through the pop-up menu. Simply pop up on the palette icon or the specific function within a subpalette to perform the desired action. If you compare the Functions palette in Figure 1.62 to the default palette in Figure 1.24, you will notice the changes that were made. Some palettes were deleted while others were moved to new locations. A VI (Data Log.vi) was added to the Users Library displayed in the bottom window. VIs that you have created and may use regularly can be added to a palette in this manner. After a new setup has been created, it will be available to you as an item under the Select Palette Set submenu.

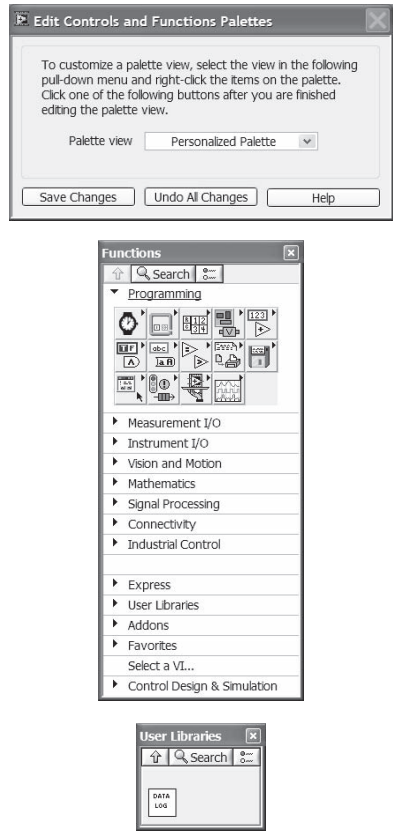

Edit Controls and Functions Palettes
To customize a palette view, select the view in the following pull-down menu and right-click the items on the palette. Click one of the following buttons after you are finished editing the palette view.
Palette view
Personalized Palette
Save Changes
Undo All Changes
Help
Functions
Search
Programming
Measurement I/O
Instrument I/O
Vision and Motion
Mathematics
Signal Processing
Connectivity
Industrial Control
Express
User Libraries
Addons
Favorites
Select a VI...
Control Design & Simulation
User Libraries
Search
DATA LOG

FIGURE 1.62

2 LabVIEW Features

The previous chapter covered many of LabVIEW's basic functions. The functions give a programmer the ability to produce a wide range of applications in a relatively short time. Although the previously discussed functions provide enough of a basis to build an application, there are a number of LabVIEW features that can make an application more flexible and easier to use, and can give your application a professional appearance. Some of these features will be discussed in this chapter.

2.1 GLOBAL AND LOCAL VARIABLES

Global variables are used when a data value needs to be manipulated in several VIs. The advantage of using a global variable is that you only have to define that data type once. It can then be read from or written to multiple VIs. The use of global variables is considered poor programming practice; they hide the data flow of your application and create more overhead. National Instruments suggests that you structure your application to transfer data using a different approach when possible. However, there are instances when global variables are necessary and are the best approach for an application. One example would be updating a display from data being generated in a subVI. The application could have two While loops running in parallel. Data could be generated in a subVI in the top loop while the bottom loop reads the data from the global and writes the information to the user interface. There is no other method for obtaining data from a subVI while it is still running (pending discussion of the Shared Variable).

The global variable must be created and its data types defined before it can be used. To create a global, first drag the icon from the Structures palette and drop it onto a block diagram. Figure 2.1 shows the global as it appears on the diagram. The question mark and black border indicate that it cannot be used programmatically. The global has a front panel to which you can add controls, identical to a VI. A Global does not have a block diagram associated with it. To open the front panel of the global variable, simply double-click on the icon. The front panel of the global is shown in the bottom window of Figure 2.1.

Two controls have been created on the global front panel. A global variable can contain multiple controls on the front panel. Try to logically group related controls and tie them to a single global variable. Once the data types have been defined, save the global as a regular VI. The global can then be accessed in any VI by using the same method you normally follow to place a subVI on the code diagram. If you have more than one control associated with the global variable, pop-up on the icon

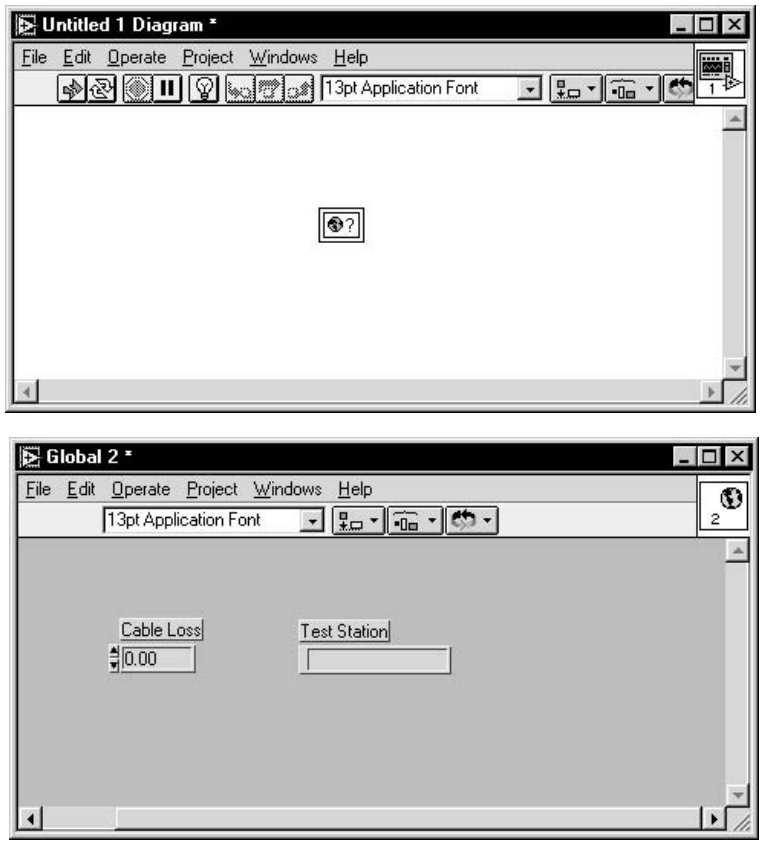


FIGURE 2.1

once you have dropped it onto a block diagram and use the Select Item submenu to select the appropriate one.

A value can be either written to or read from a global. Use a "read" global to read data from and a "write" global to write data to a global variable. The first selection in the pop-up menu allows you to change to either a read or write variable. Figure 2.2 demonstrates how a global and local variable can be used on the block diagram. The global created in Figure 2.1 is used in this VI to retrieve the Cable

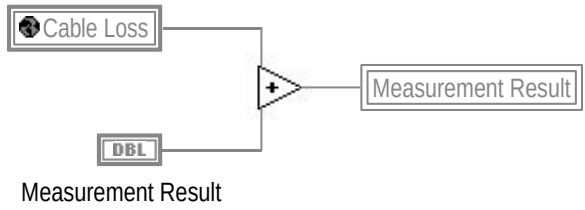


FIGURE 2.2

Loss parameter. Global variables are easy to distinguish on the block diagram because of the unique icon that contains the name of the variable. The thicker border indicates that it is a read variable.

Measurement Result is a control used in this VI. The result of the addition is being passed to the local variable of Measurement Result. Local variables can be created by popping up on a control or indicator terminal and selecting Local Variable from the Create submenu. Alternatively, drag and drop the local variable from the Structures palette. Then, pop up on it and use the Select Item submenu to choose the name of a control or indicator. A local variable can be created for any control or indicator terminal. As with the global, the local can be used as a read or write variable and toggled using the pop-up menu. In the example shown, the name of the local, Measurement Result, appears in the icon. The icon does not have a thick border, indicating that it is a write variable.

The main difference between local and global variables is access. The local variable is only available on the code diagram it was created on. The global variable can be used in any VI or subVI on the source machine. Due to the fact that the global variable is loaded from a file, any VI has access to this data. While this flexibility seems like a benefit, the result is a loss of data control. If a specified global variable is placed in a number of VIs, one of the VIs could be used by another application. This could result in errant data being written to the global in your main program. With local variables, you know the only place the data can be modified is from within that VI. Data problems become easier to trace.

One alternative to a standard global variable is the use of a *functional global*. A functional global is a VI that contains a While loop with an uninitialized shift register. The VI can have two inputs and one output. The first input would be an Action input. The actions for a simple global would be read and write. The second input would be the data item to store. The output would be the indicator for the item to read back. The case structure would have two states. In the Read state, the program would want to read the global data. The code diagram wires the data from the shift register to the output indicator. The Write case would wire the input control to the output of the shift register. The code diagram is shown in Figure 2.3. The benefit of using this type of global is the prevention of race conditions; an application cannot

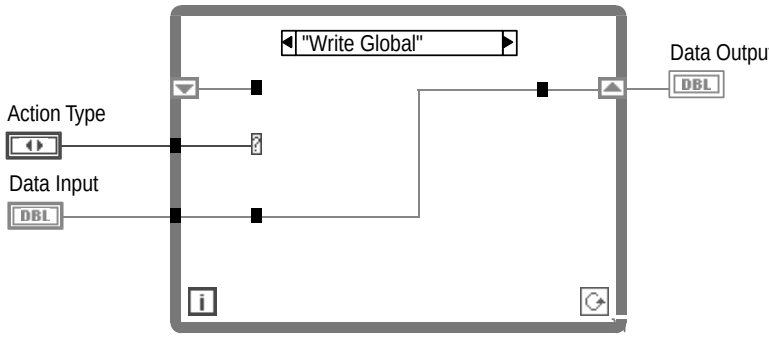


FIGURE 2.3

attempt to write to and read from the global at the same time. Only one action will be performed at a time.

2.2 SHARED VARIABLES

Shared variables are a new type of variable added to LabVIEW 8. The shared variable is similar to a global variable in that you obtain and pass data to other parts of your program or other VIs without having to connect the data by wires. The shared variable has several distinct differences. First, data can be shared not only between one VI and another on your computer, but it can be shared over the network. Another difference is that shared variables can be bound to a source. For example the data could be read in from another shared variable, a front panel control or a LabVIEW RT target. A shared variable can also use data buffering and restrict inputs to one writer at a time. A shared variable node on the code diagram can provide a time stamp to be able to determine the "freshness" of the data. Finally, there is an error in and out for the shared variable, which will help with both exception handling and forcing the order of execution.

To create a shared variable you must first open a LabVIEW project. The shared variable can only be created within a LabVIEW project and must be contained in a project library. If a library does not exist, one will be created for the new variable. Once you have the project open you need to right click in a library and select New, and then Variable. The Shared Variable Properties window will appear. This dialog box is shown in Figure 2.4. Here you will enter the variable type. A shared variable

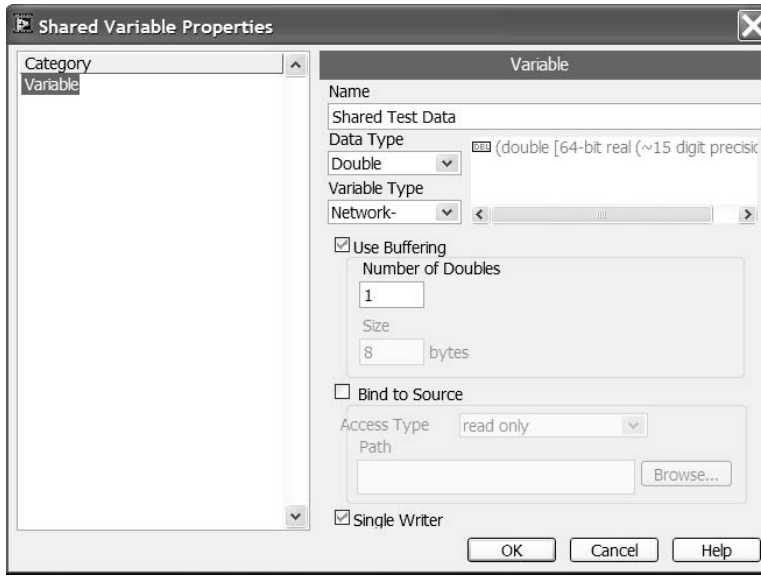


FIGURE 2.4

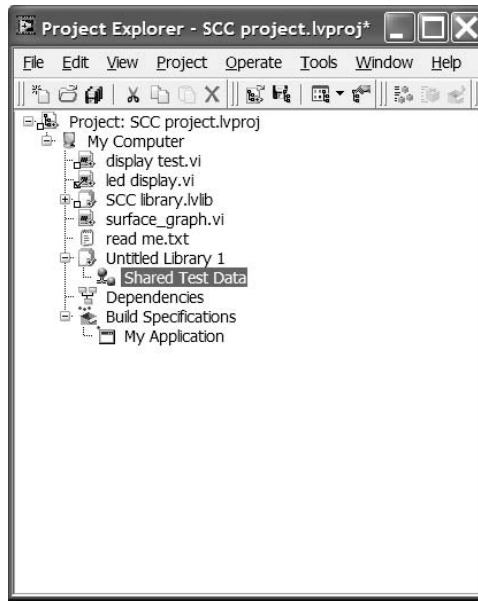


FIGURE 2.5

can be almost any type of variable including numbers, strings, Booleans, waveforms and arrays of the aforementioned datatypes. If those options do not meet your needs you can import a custom data control type.

Once the datatype is selected you need to choose whether the variable will be a network variable. There is a choice of whether to use buffering. If you choose to use buffering make sure you make your buffer large enough. If there is a buffer overflow data will be lost, and there will be no error generated. In the properties window you can select a source to bind the variable to. Finally you have the option to set the variable to only accept changes from one writer at a time. Once you have completed changes to the variable properties you will see the variable has been added to the project. Figure 2.5 shows the new shared variable was added to an empty library since one did not previously exist in this project. Also notice that based on the icon you can see that the variable was setup as a network variable.

Now that the variable is set up you can use the variable in your code. There are two ways to insert the variable in the code. The first is to do a drag and drop from the Project Manager to the VI. The second method is to insert a shared variable from the Structures palette. Once the shared variable node is on the code diagram you can double click on it to select the shared variable to link to. This will only work if the VI you are working with is a part of the project. Figure 2.6 shows a loop that reads a shared variable every 5 sec. The data and time stamp are reported to the front panel. Through this you will be able to see what the current value of the variable is and when it was last changed.

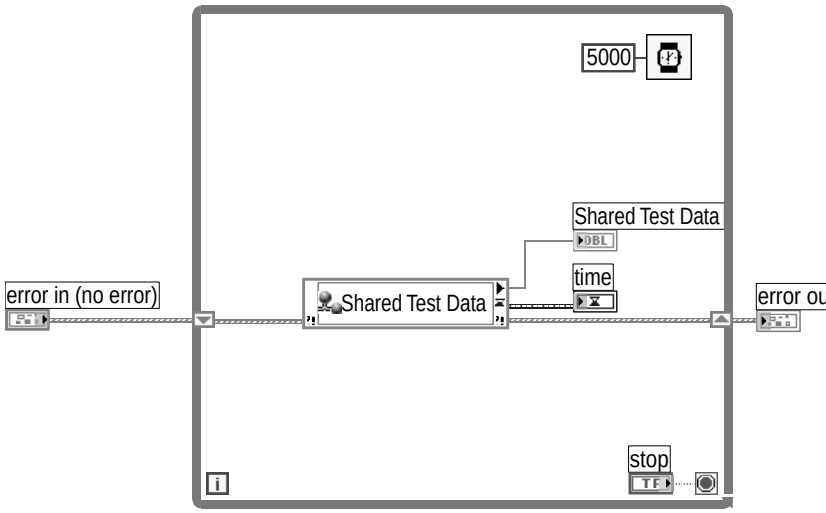


FIGURE 2.6

2.3 CUSTOMIZING CONTROLS

Controls and indicators can be customized, saved, and reused in different VIs. This allows you to modify the built-in controls and indicators to accommodate your applications. This section describes the procedure for creating custom controls and type definitions. A "Type Definition" is a master copy of a control or indicator. When you need to use the same control in several VIs, you can create a type definition and save it. Then, when changes are made to the type definition, they will automatically be applied to all of the VIs that use that control.

2.3.1 Custom Controls

To customize any control on your front panel, select the control and choose Edit Control from the Edit pull-down menu. A new window will appear with the control shown on the panel. This panel has no diagram associated with it and cannot be executed. Figure 2.7 shows this window with a text ring control on the panel. Also note that Control is the current selection in the drop-down menu on the toolbar. The control can be modified in either the Edit mode or the Customize mode; the default is Edit mode when the window first appears. The Edit mode is similar to the Edit mode of the front panel of any VI where alterations can be made. It allows you to make some of the basic changes to a control, such as size and color. The Customize mode lets you make additional changes to specific elements of the control. The top window in Figure 2.7 shows the Edit mode, and the bottom window shows the Customize mode. The first button in the window toolbar allows you to toggle between the two modes of operation.

Each control is a composite of smaller parts. When you modify a control in the Customize mode, you are then able to modify the parts of the control. If you open

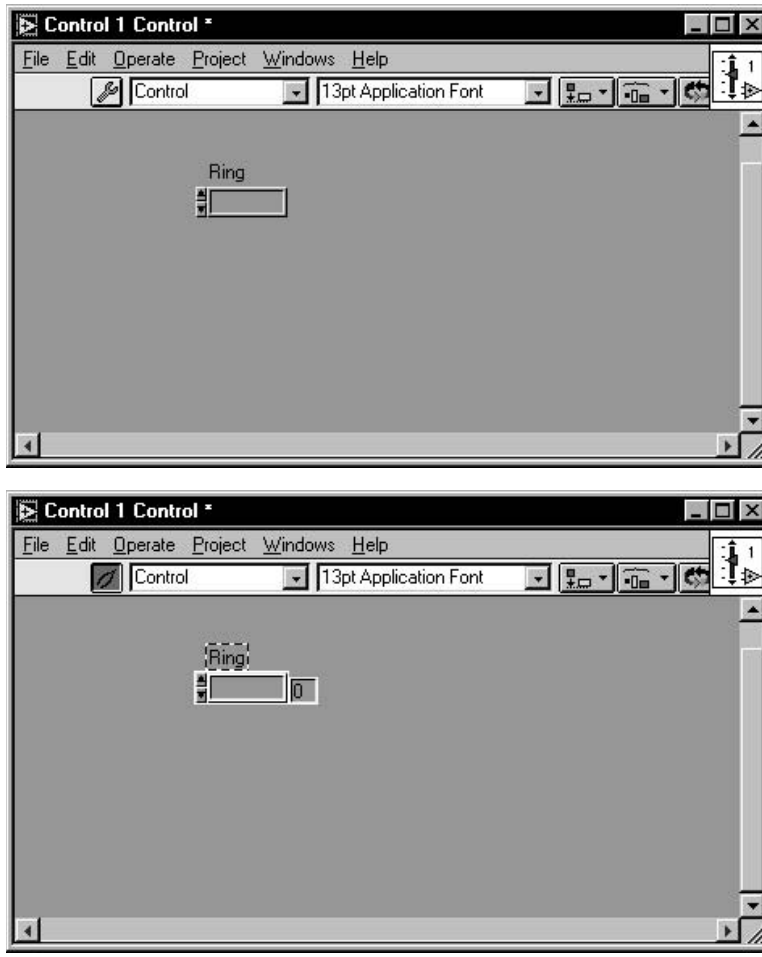


FIGURE 2.7

the Parts Window from the Windows menu, you will be able to see the labels for each part, as well as the position and size of each part. You can scroll through each part via the increment arrow. One of the benefits of this capability is the ability to create custom controls. Text or pictures can be copied and pasted into the control editor. The pictures can become part of the control. This capability makes the creation of filling tanks, pipes, and other user-friendly controls possible.

Figure 2.8 shows the modified ring control on the front panel. The up and down scroll arrows were altered for the ring control. Once the desired modifications are made to a control, you can replace the original control with the modified one without saving the control. Select Apply Changes from the Control Editor window before closing it to use the modified control. Alternatively, you can save the control for use in other VIs. Simply give it a name and save it with a .ctl extension or use Save As

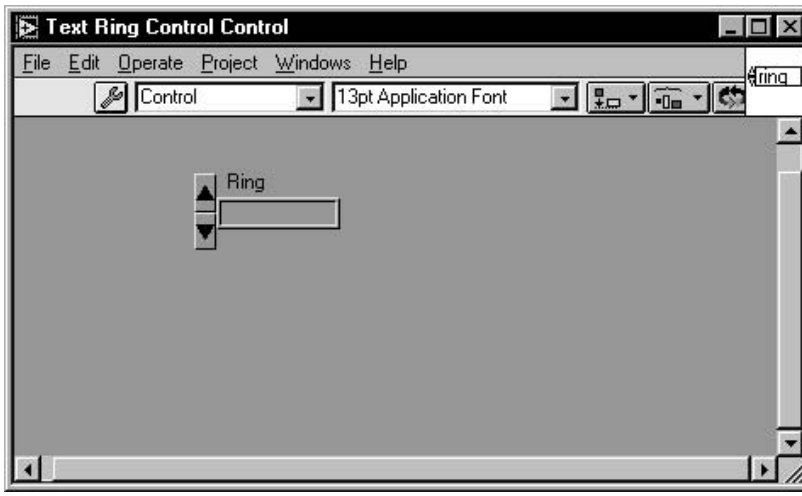


FIGURE 2.8

from the File menu. To use it on another VI front panel, choose Select a Control from the controls palette and use the file dialog box to locate the control.

2.3.2 Type Definitions

A type definition allows you to set the data type of a control and save it for use in other VIs. This may be useful if you change the data type and want that change reflected in several VIs. A type definition allows you to define and control the data type from one location. It can prove to be very practical when using clusters and enumerated types. When items need to be added to these controls, you only have to do it once. Default values cannot be updated from a type definition.

You create a type definition following a similar procedure as a custom control. Select the control that you wish to create a type definition from and choose Edit Control from the Edit pull-down menu. Figure 2.9 displays the window that appears for an enumerated control. To create a type definition instead of a custom control, select Type Def. from the drop-down menu in the toolbar. The enumerated control has been defined as an unsigned word and three items have been entered into the display. This type definition was saved using the Save As selection from the File menu. The window title tells you the name of the control and that it is a type definition.

The type definition can be used in multiple VIs, once it has been saved, by using Select a Control from the Controls palette. When you need to modify the type definition, you can open the control using Open on the File menu. You could also select the control from a front panel that uses it and choose Edit Control from the Edit menu, which then opens the window of the saved type definition. A final way to open the control is by double-clicking on it (if this option is selected in your preferences).

Any changes made to the type definition will be reflected automatically in its instances if they have been set to auto-update. The instances include controls, local

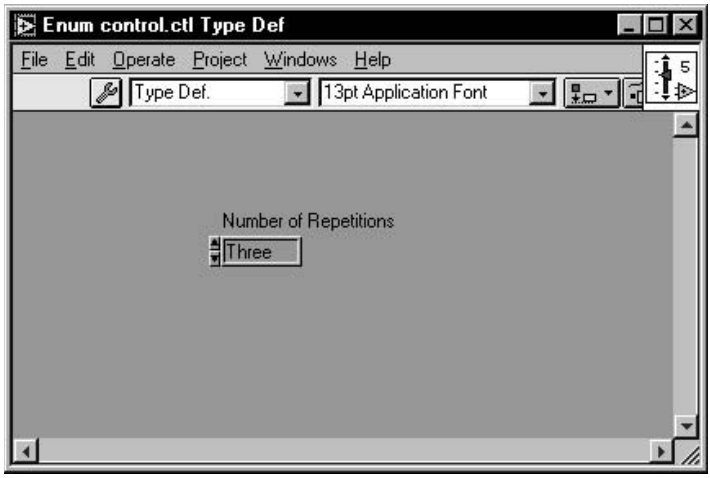


FIGURE 2.9

variables, and constants created from the control. All VIs that use instances of the type definition are set to auto-update by default. When you pop up on an instance of the type definition, you will see a menu similar to the one shown in Figure 2.10. You can then choose to auto-update the control or disconnect it from the type definition. Items can be added to the enumerated control type definition shown in Figure 2.9, and all VIs that use the type definition will be automatically updated. Items cannot be added to the instances unless the auto update feature is disabled.

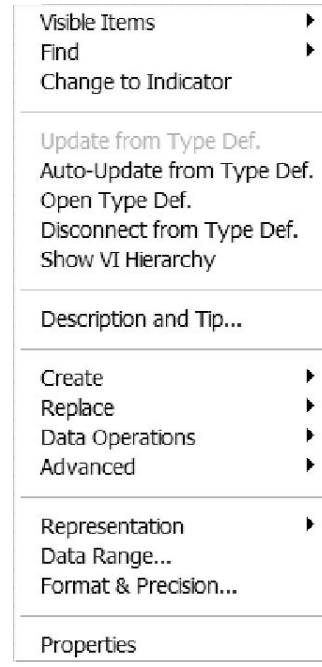


FIGURE 2.10

2.3.3 Strict Type Definitions

Type definitions cause only the data type of a control to be fixed in its instances. Other attributes of the type definition can be modified within the instances that are used. Size, color, default value, data range, format, precision, description, and name are attributes that can be adjusted. Strict type definitions can be used to fix more of the attributes of a control to create uniformity across its instances. Only the name, description, and default value of a strict type definition can be altered in the instances. This allows you to maintain more control of the type definition. In addition, auto updating cannot be

disabled when strict type definitions are used. This forces all changes to be applied to all of the occurrences.

Strict type definitions are created in the same manner as type definitions. The only difference is the drop-down menu in the toolbar should be set to Strict Type Def. After the strict type definition has been saved, changes can be made only to the master copy. The changes made to the master copy are reflected in the VI only when it is open. If a VI is not in memory, the changes are not updated. This could be an issue if a VI is copied to a new location, such as a different PC, without opening the VI between the time the control was modified and the copy was performed. In the absence of the Strict Type Def., the VI would first ask you to find the control. If the control is unavailable, the control will appear grayed out. If you right-click on the control you have the option of disconnecting from the Strict Type Def. If you disconnect, the VI would use the last saved version of the control. In this case, the modifications would not be reflected in the VI on the new PC.

2.4 PROPERTY NODES

Property nodes are a means for getting and setting the properties of a control or indicator during program execution. The properties available will vary depending on the particular control or indicator being used on the front panel of your application. Pop up on either the control from the front panel or the terminal from the code diagram and Select Property Node from the Create submenu. By providing the ability to change the appearance, location, and other properties programmatically, property nodes provide you with a tremendous amount of flexibility while designing and coding your application.

Figure 2.11 illustrates some of the characteristics of property nodes. An enumerated type control, Number of Repetitions, will be used to describe property nodes. A property node was created for the Number of Repetitions control and is placed just below the terminal. In this example, the Visible attribute is being set to "false." When the VI executes, the enumerated control will no longer be visible. All of the properties associated with Number of Repetitions are shown in the property node to the right on the block diagram. Multiple properties can be modified at the same time. Once a property node has been created, simply drag any corner to extend the node or use the pop-up menu and select Add Element.

You can read the current setting or set the value of a control through the property node from the block diagram. Use the pop-up menu to toggle the elements in the node between read and write. An arrow at the beginning of the element denotes that you can set the property, while an arrow after the property name denotes that you can read the property. In the figure shown, the first ten elements have the arrow at the beginning, indicating that they are write elements. When there are multiple properties selected on a property node, they can be selected as either read or write. If both operations need to be performed, separate property nodes need to be created.

The following example demonstrates how to use property nodes in a user interface VI. The front panel of the VI is displayed in Figure 2.12. Depending on the selection made for the Test Sequence Selection, the appropriate cluster will be

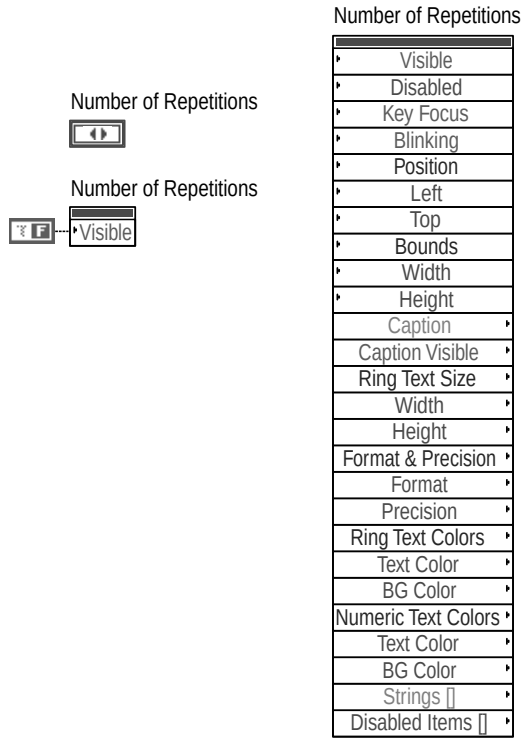


FIGURE 2.11

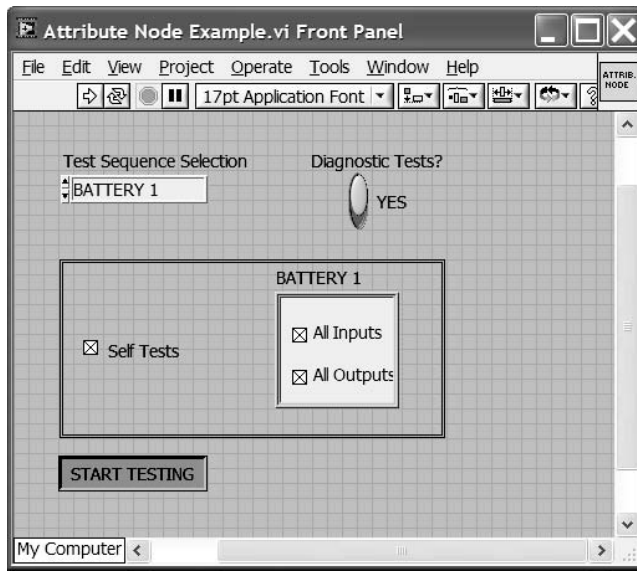


FIGURE 2.12

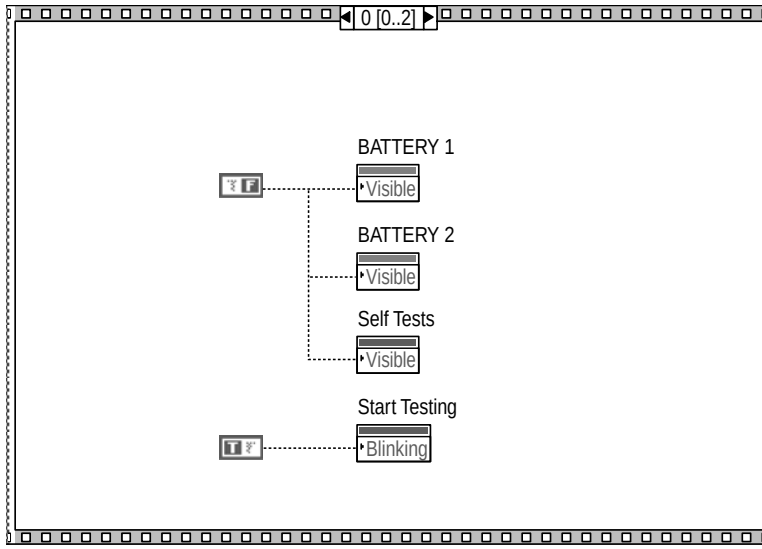


FIGURE 2.13

displayed in the framed border below it. The cluster shown for Battery 1 has two checkboxes that can be manipulated when Battery 1 is selected in the enumerated control. If Diagnostic Tests? is true, the Self Tests checkbox will be visible. When the user is finished making the selections, the Start Testing button is pressed.

Figure 2.13 shows Frame 0 of the sequence structure on the block diagram associated with this VI. The first action taken is to set the visible properties of Battery 1, Battery 2, and Self Tests to false. Also, the Start Testing button's blinking property is set to true. This ensures that the VI begins in a known state in case it was run before. Frame 1 of the sequence is shown in Figure 2.14. The purpose of this frame is to continually monitor the actions the user takes from the front panel. The While loop repeats every 50 milliseconds until Start Testing is pressed. The 50-millisecond delay was inserted so that all of the system resources are not used exclusively for monitoring the front panel. Depending on the selection the user makes, the corresponding controls will be displayed on the front panel for configuring. This example shows that property nodes are practical for various applications.

In the following example, shown in Figure 2.15, more of the properties are used to illustrate the benefits of using the property node. The VI has a number of front panel controls. There is a numeric control for frequency input, a set of Boolean controls to allow the user to select program options, and a stop Boolean to exit the application. To allow the user to begin typing in the frequency without having to select the control with the mouse, the property for key focus was set to "true" in the code diagram. This makes the numeric control active when the VI starts execution. Any numbers typed are put into the control. The second property in the property node for the digital control allows you to set the controls caption. This caption can be changed during execution, allowing you to have different captions based on program results.

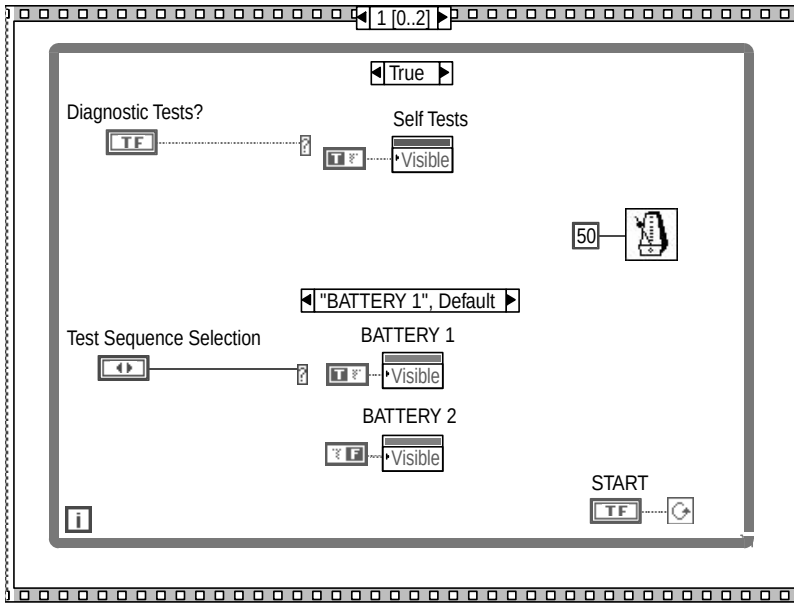


FIGURE 2.14

The second set of controls is the Booleans for setting program options. One of the requirements for this application is to make two additional options available if the second option is set to "true." Setting the Disabled attribute to 2 (which is disabled and grayed out) performs this action when the program starts. If the second option is set to "true," the Disabled attribute is set to 0. This enables the control, allowing the user to change the control value. The other option disables the control, but the control remains visible. The final property node in this example sets the Blinking property of the Stop Boolean to "true." When the VI is run, the Stop button will flash until the program is stopped. The front panel and code diagram are shown in Figure 2.15.

2.5 REENTRANT VIs

Reentrant VIs are VIs configured to allow multiple calls to be made to them at the same time. By default, subVIs are configured to be nonreentrant. When a subVI is nonreentrant, the calls are executed serially. The first subVI call must finish execution before the next one can begin. Calls to the subVI also share the same data space, which can create problems if the subVI does not initialize its variables or start from a known state. Shift registers that are not initialized, for example, can cause a subVI to yield incorrect results when called more than one time.

When a subVI is configured to be reentrant, it can speed up the execution of an application as well as prevent problems caused by sharing the same data space in the system. Each call to the subVI will execute in its own data space. A separate instance is created for each call so that multiple calls can be executed in parallel.

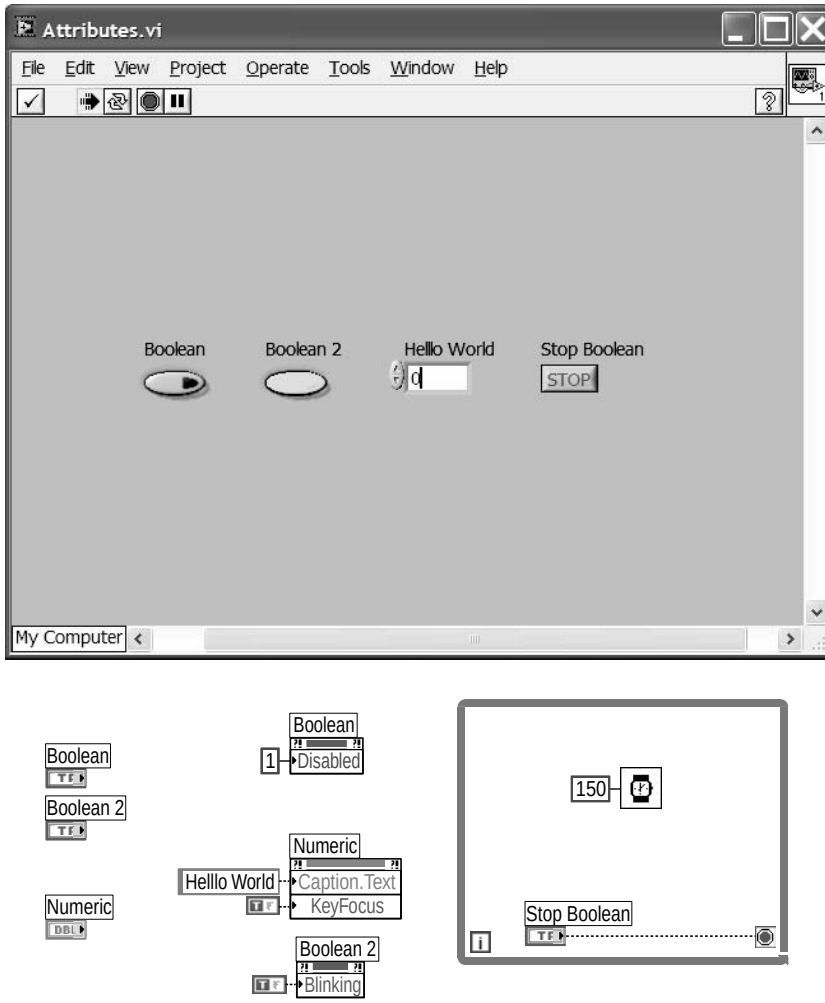


FIGURE 2.15

The calls will, in effect, execute independently of each other. The use of a separate data space for each call will result in more memory usage. Depending on the data types used, and the number of times the VI is executed, this can become an important issue. When creating an application, you should be aware that subVIs of a reentrant VI cannot be reentrant.

A VI can be enabled for reentrant execution through the Execution Options in the VI Properties dialog box. Pop up on the icon in the top right corner of the VI window and select VI Properties from the menu. The Execution options of the VI Properties window are displayed in Figure 2.16. In previous versions of LabVIEW several checkboxes would become disabled when Reentrant Execution is enabled: Show Front Panel When Loaded, Show Front Panel When Called, Run When Opened, Suspend When Opened, and all of the printing options. In addition, Exe-

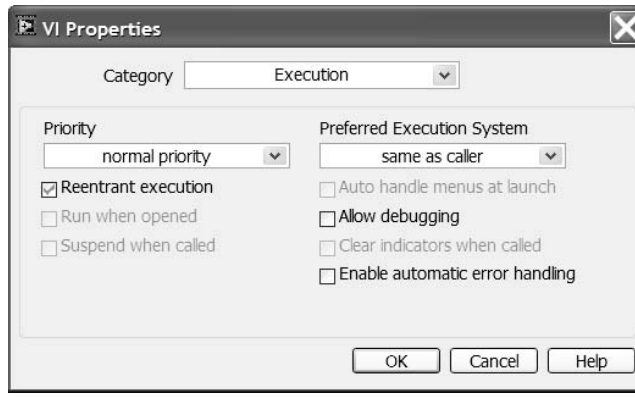


FIGURE 2.16

cution Highlighting, Single-Stepping, and Pausing were no longer available. In LabVIEW 8 reentrant VIs can now be debugged.

When you put a reentrant VI on a code diagram, a copy of the reentrant VI is placed on the code diagram. If you put 2 copies of the reentrant VI on a code diagram or on more than one open code diagram two individual copies of the subVI are created. When you open up a copy of the reentrant subVI, named Reentrant Function SubVI for this example, the front panel opens up with the title reentrant function subVI.vi:1 (clone). When subsequent copies are opened the number after the colon is incremented. Each subVI is a mutually exclusive copy. From the clone copy you cannot modify the original VI. If you want to make changes to the reentrant subVI you must open the original VI from file through the File menu. Changes made to the original VI will be reflected in the cloned copies. You would debug a reentrant VI the same way you would debug a normal VI. In the example in Figure 2.17, the subVI is a function used to perform a select action on 2 numbers. The main level VI has two copies of the subVI. One copy is used to perform addition on two arrays of numbers and one copy of the subVI is used to perform multiplication on the same two arrays of numbers. When you single step through the multiplication version of the subVI you can see that the correct value is seen at the input. You are now able to evaluate the actual numbers going through a specific instance of the reentrant subVI.

2.6 LIBRARIES (.LLB)

The VI library was briefly mentioned in Chapter 1 when saving a VI was discussed. This section will describe the procedure for saving and editing a VI library. There are both advantages and disadvantages to saving a VI inside of a library. National Instruments suggests that VIs be saved as separate files (.vi extension) rather than as libraries unless there is a specific need that must be satisfied. Table 2.1 lists benefits of saving as libraries and separate files; the table is also available through the on-line help VI Libraries topic. These issues need to be considered when saving files.

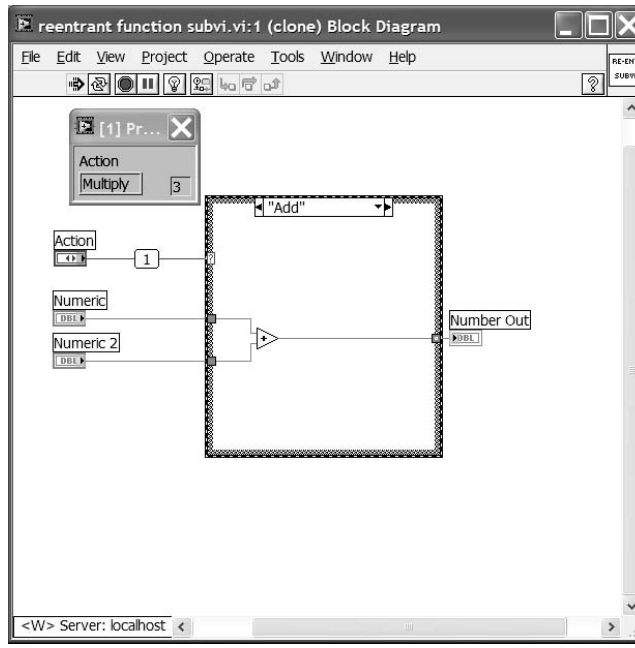


FIGURE 2.17

TABLE 2.1
Library Benefits and Drawbacks

Benefits of Saving as .VI	Benefits of Saving as .LLB
1. You can use your own file system to manage and store VIs.	1. 255 characters can be used for naming files (may be useful for Macintosh, where filenames are limited to 31 characters).
2. Libraries are not hierarchical. Libraries cannot contain subdirectories or sublibraries.	2. Easier for transporting VIs to different platform or to a different computer
3. Loading and saving VIs is faster and requires less disk space for temporary files.	3. Libraries are compressed and require less disk space.
4. More robust than storing entire project in the same file.	4. Can now be viewed in Windows Explorer (starting in LabVIEW 7).
5. There is the possibility of a library becoming corrupt.	5. Can set one or several VIs to start when the library is opened (top level VIs).
6. Source Control cannot operate on individual VIs in a LLB, only the entire LLB.	

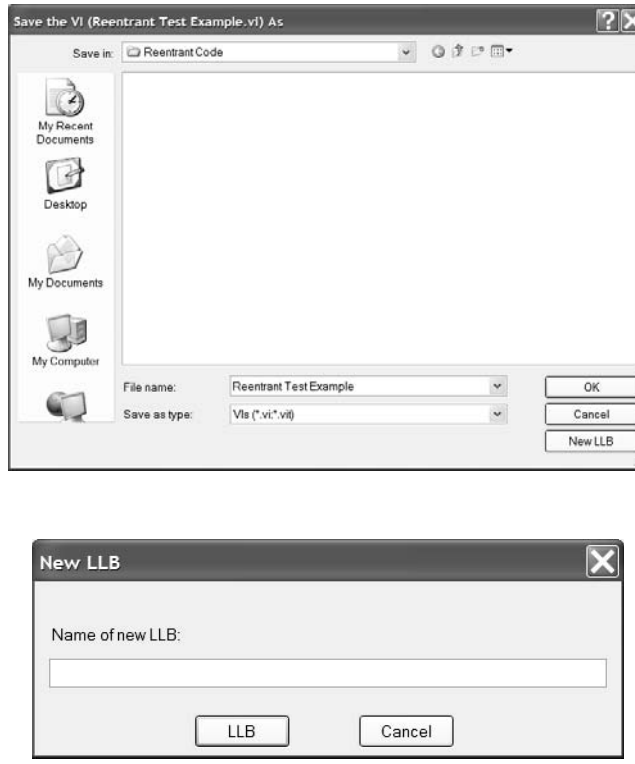


FIGURE 2.18

To save a file as a VI library, you can use Save As from the File pull-down menu. Figure 2.18 shows the file dialog box that appears when you select Save As and then select to make a copy of the VI. One of the buttons in the dialog box window lets you create a new VI library. When you press New VI Library, the window shown below the dialog box in Figure 2.18 appears. Simply enter a name for the library and press VI Library; the extension will be added for you. If you want to add a VI to an existing library, you can perform a Save As option and find the library you wish to save it in. The library is treated as a folder or directory and you can save VIs inside it. The second option available (on Windows platforms) is to open the library in Windows Explorer. To Windows Explorer, the library looks just like another folder so that you can add and delete files from the library.

If you do not want to use the Windows Explorer or if you are running on another OS, you can edit a library by using the LLB Manager. The LLB Manager dialog box is shown in Figure 2.19. In the LLB manager you have the ability to add, remove and copy VIs in the LLB. You can also convert the LLB to a folder or a folder to a LLB. While editing a LLB you can designate a VI in the LLB as a top level VI. A top level VI is opened when the LLB is opened automatically. This allows the programmer to launch a user interface VI that obscures how the underlying code is arranged or used.

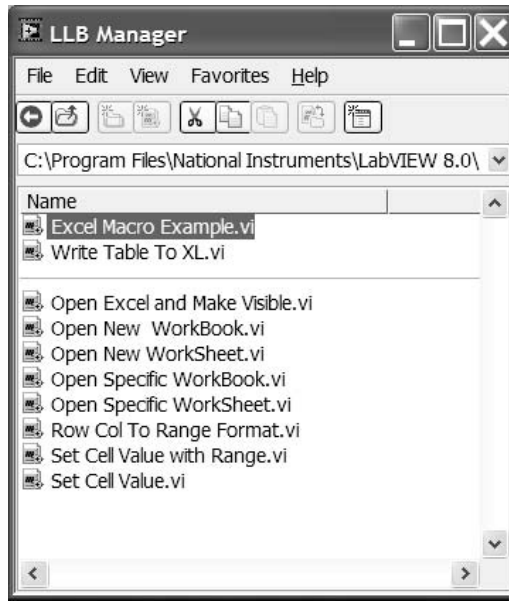


FIGURE 2.19

LLB files should not be confused with Project Libraries. In LabVIEW 8, a project library is defined as a collection of VIs, shared variables, type definitions, and other files. When you create a project library a file containing the properties and references for the project library is generated. The project library file will have a .lvlib extension. There are some similarities and differences between a LLB and a project library. Both functions give you the ability to group files for an application. You cannot make a top level VI in a project library though. When using a project library, VIs from the library can be dragged and dropped on the code you are editing.

2.7 WEB SERVER

National Instruments incorporates a Web server with versions of LabVIEW 5.1 or later. Once enabled and configured, the Web server allows users to view the front panels of applications (VIs that have been loaded) from a remote machine using a browser. Both static and dynamic images of a front panel VI can be viewed remotely. Not only will the Web server allow you to view the front panel of an application, but you have the ability to control the VI as well. You can control the application or front panel remotely using a browser. You have the ability to interact with the application through the Web server, but you cannot modify the code. This section will go through the steps for configuring the Web server, and will show an example of how the front panel will appear on an Internet browser. This will be followed by an explanation of controlling the application using remote front panels.

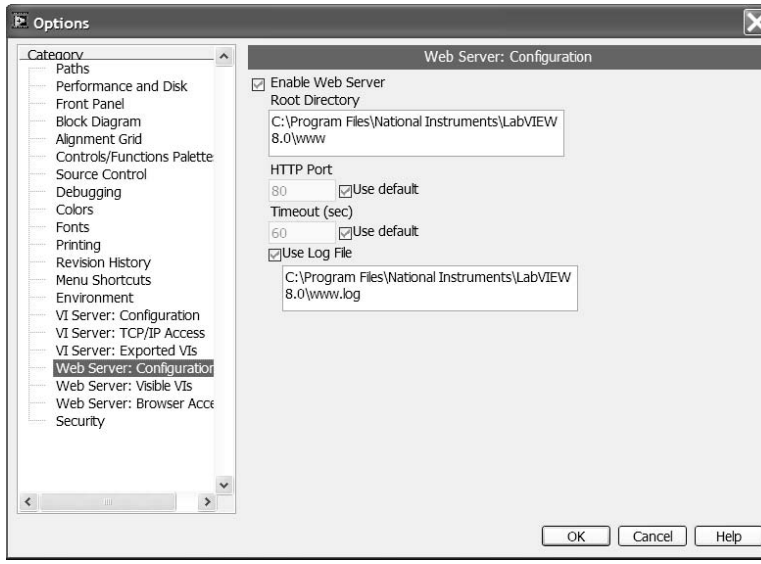


FIGURE 2.20

The Web server is set up through the Options selection on the Tools pull-down menu. This window is shown in Figure 2.20 with Web Server: Configuration selected. The Web server is not running by default, so you must first check Enable Web Server. The default Root Directory is the folder that holds the HTML files published to the Web. The HTTP Port is the TCP port the Web server uses to publish the pages. This port number may have to be changed depending on whether another service is utilizing this default on your machine. Finally, the log file saves information on the Internet connections to the Web server. Once this is done, front panels of loaded VIs can be viewed from a remote browser. However, you may want to perform additional configurations to set access privileges, as well as determine which VIs can be viewed through the server. Figure 2.21 displays the Option window for the Web Server: Browser Access selection. By default, all computers have access to the published Web pages. This window allows you to allow or deny specified computers access to the Web pages. You can enter IP addresses or domain names of computers into the browser access list. The browser list allows you to use the wildcard character (*) so that you do not have to list every computer name or IP address individually. The "X" in front of an entry indicates that a computer is denied access while a $\surd$ indicates that it is allowed access. If a diamond appears in front, then this is a signal that the item is not valid.

Figure 2.22 illustrates the window when Web Server: Visible VIs is selected in the left pane. This window is similar to the browser access settings but applies to VIs. You can add names of VIs into the listbox and select whether you want to allow or deny access to them via a browser. Again, the wildcard character is valid when making entries. The X and $\surd$ indicate whether the VI can be accessed, and the

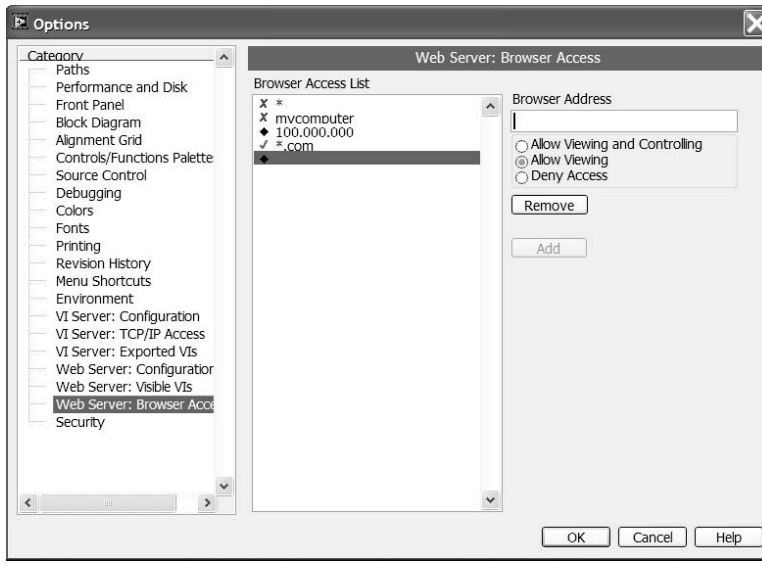


FIGURE 2.21

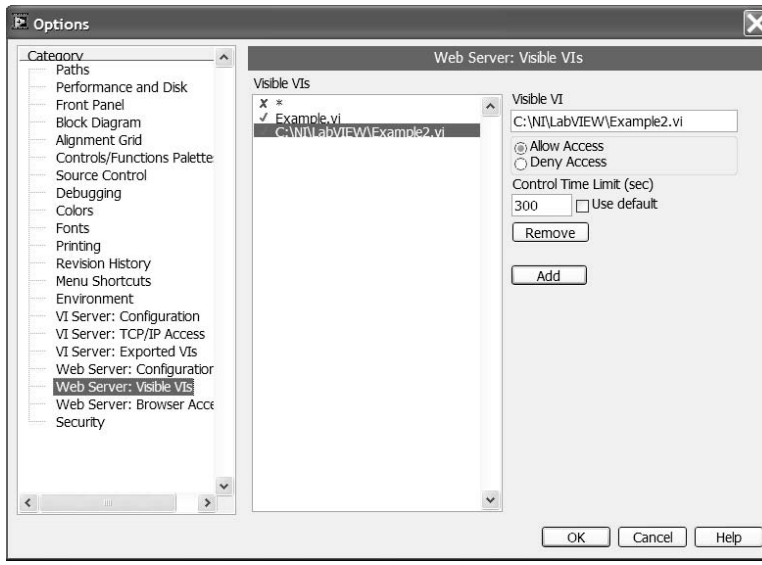


FIGURE 2.22

diamond signifies that the syntax may be incorrect. You can enter the name of a VI or designate the specific path if you have multiple VIs with the same name.

Viewing the front panel of a VI with a Web browser is a straightforward procedure. Before you can view the front panel, make sure the Web server has been enabled and the VI has been opened. The VI must be loaded in memory before anyone can look at it from the Web. Then, from any machine that is allowed access to the Web server, type the URL of the machine running the Web server into the browser. To the end of the URL add the VI name with the html extension. Insert a plus sign or % 20 in the URL to represent the spaces in your VI name. For example, to monitor a VI named "test program.vi" you would enter in the following URL into your browser: http://111.111.111.11/test program.html. This assumes the IP address of the Web server is 111.111.111.11. If you are trying to view a VI on the local machine you could use "localhost" instead of the IP address. Depending on the company you are working for, and your network specifics, the system administrator may need to set permissions or perform other system administrator intervention.

As mentioned earlier you can control a LabVIEW Application or Front Panel Remotely Using a Browser. The first step is to set up the Web server. Once the Web server is set up for allowing viewing and control access you must load or run the application or VI. Once in memory you would navigate to the proper html page in your browser. Now you should see what you saw above. To control the VI you would right click on the VI panel in the browser and select Request Control of VI. If there is no other client controlling the VI a message appears indicating that you have control of the front panel. If another client is already controlling the VI, the Web server queues the request until the other client releases control. To disconnect you can right click on the panel and select Release Control of VI from the Remote Panel Client window or you can close the browser window.

From the server you have the ability to regain control of a VI that is being controlled remotely. If you right click on the VI panel a Remote Panel Client subwindow comes up. You can select Switch Controller, which gives the local VI control again. The browser that was in control will see a message stating that the server has regained control. The person connecting through the browser can request control again. In order to prevent the remote control, the person at the server can select Lock Control from the Remote Panel Client subwindow. Now when the remote browser tries to take control a message will state that either the server has locked control or another client has control. When the server is unlocked the control will transfer to the remote host and the remote host will be notified that control has been granted.

2.8 WEB PUBLISHING TOOL

The Web Publishing Tool can be used to help you customize the way your Web page appears in a browser. Normally, only the front panels of the VIs that you have loaded are displayed in the browser when accessed remotely. This tool lets you add a title with additional text, which is then saved as an HTML file. When you want to view the front panel of a VI, you use the URL of the HTML page you have saved. The

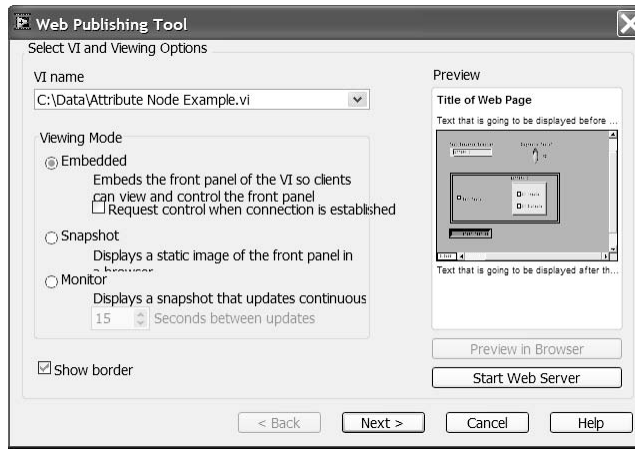


FIGURE 2.23

page then loads the picture of the front panel for you to view along with the title and additional text.

To access this feature, select Web Document Tool from the Project pull-down menu. The Web Document Tool window is displayed in Figure 2.23. Select a VI using Browse from the VI Name drop-down menu, or type in a VI name for which you want to create a page. A dialog box will ask you if you wish to open the VI if it is not already opened. Then you can enter in a title that you want to appear on the page, text before the image, and text after the image. The buttons available allow you to save the changes to an HTML file, start the Web server if it has not already been enabled, and preview the Web page. When you select Save To Disk, a dialog box will tell you the URL that is to be used to access the page from a browser. Remember that if the VI has not been loaded, the Web page will appear without the image of the front panel. Figure 2.24 shows an example of a page created using the Web Publishing Tool in a browser window.

2.9 INSTRUMENT DRIVER TOOLS

There are tools in LabVIEW 8 that help you manage both the external devices connected to your computer and the instrument drivers that are available from National Instruments. An instrument driver is a collection of VIs used to control instruments. LabVIEW drivers abstract the low-level commands that programmable instruments respond to. Drivers allow you to control instruments without having to learn the programming syntax for each instrument. Instrument drivers are discussed in more detail in Chapter 5, including the recommended style to follow when developing them. LabVIEW ships with a second CD-ROM that contains instrument drivers for numerous instruments from various manufacturers.

Under the tools menu there is an Instrumentation submenu. This submenu contains two tools for working with instrument drivers. The first tool is the Find Instrument Drivers function. When this function is selected, a dialog box comes up

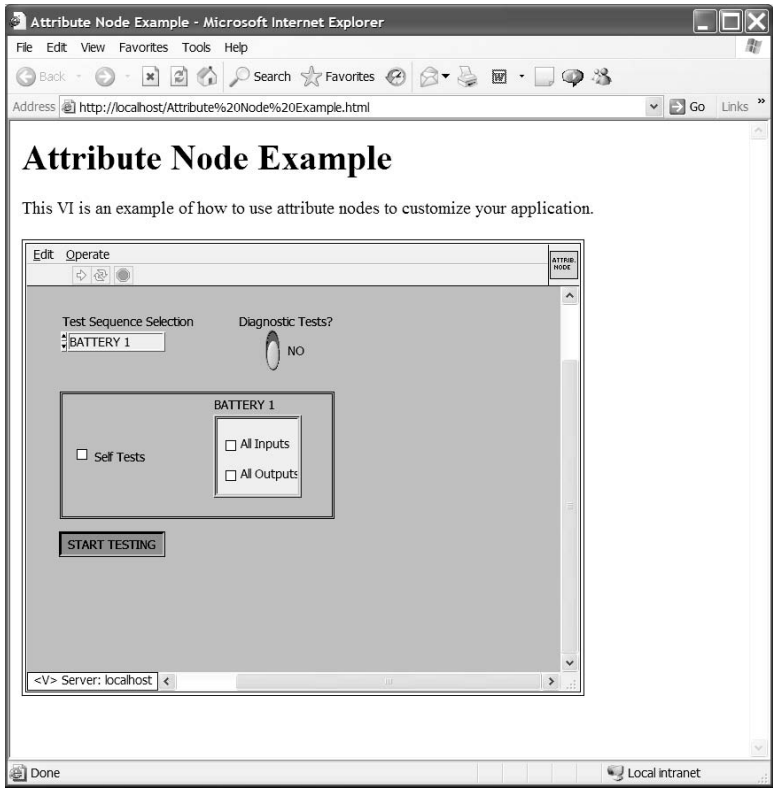


FIGURE 2.24

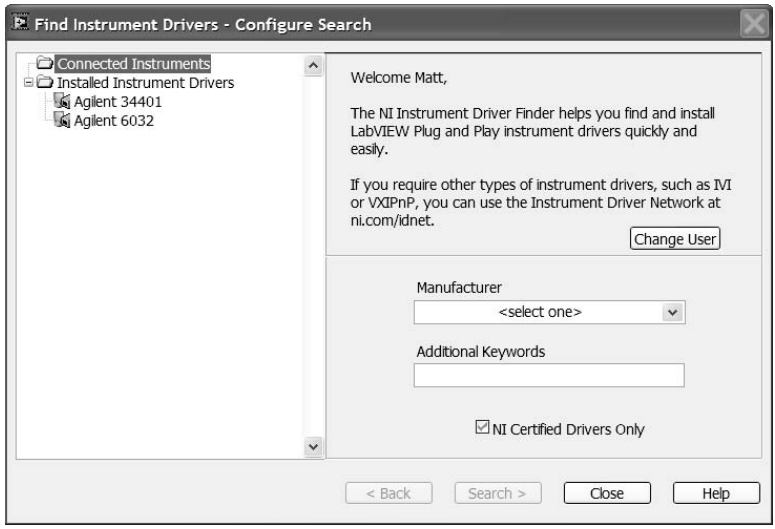


FIGURE 2.25

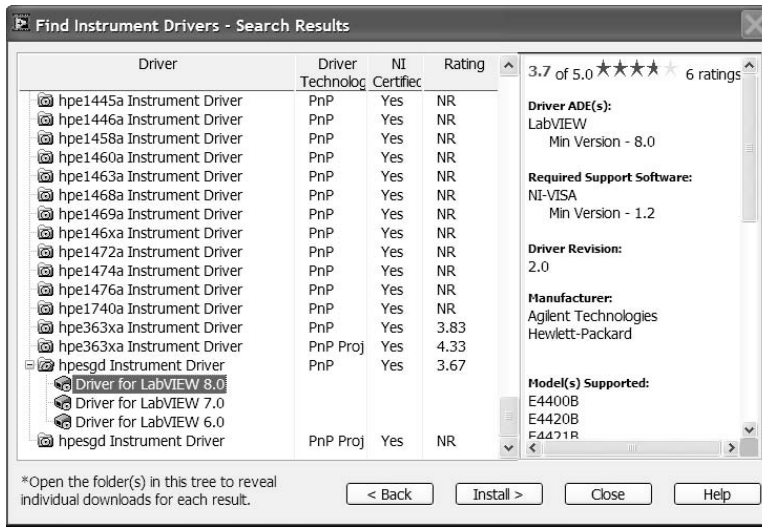


FIGURE 2.26

giving the programmer a list of currently installed drivers as well as a list of any connected instruments. The dialog box is shown in Figure 2.25. If the driver you are looking for is not installed you can search the National Instruments Website for the driver you need. At the top right of the dialog box you can log into the National Instruments Website. Once logged in you can select what manufacturer you are searching for as well as any key words to narrow the search. The result of the search is shown in Figure 2.26.

Now that you have a list of drivers you can search the list for the one that meets your needs. Platform support, LabVIEW version, required support software and ratings are just a few pieces of information available to you. Some drivers also have the option of installing as a standard driver or as a driver project. Once you find the driver you need you can select Install. The driver will install on your computer and will be available to use through the functions palette in the Instrument I/O subpalette.

The second tool available is Create Instrument Driver Project. This tool is a starting point for creating your own driver for an instrument. When the tool is launched it will open the dialog box shown in Figure 2.27. Here you have the choice of creating a new driver form template or creating a new driver from an existing driver. If you select to copy an existing driver, a list of installed drivers will appear in the second pull-down window. If you want to start from a template the second pull-down will have a list of the available driver templates. Once you select the closest matching instrument type the tool will generate a driver project. The project manager view of the generated driver project is shown in Figure 2.28. As you can see, the standard array of VIs are generated such as the VI tree, initialize, configure instrument, reset and close. From here you would modify each of the VIs to work with your instrument. In this example, when you open the Initialize VI, you see that it was made general enough to work in many cases. The initialize VI window is shown in Figure 2.29.

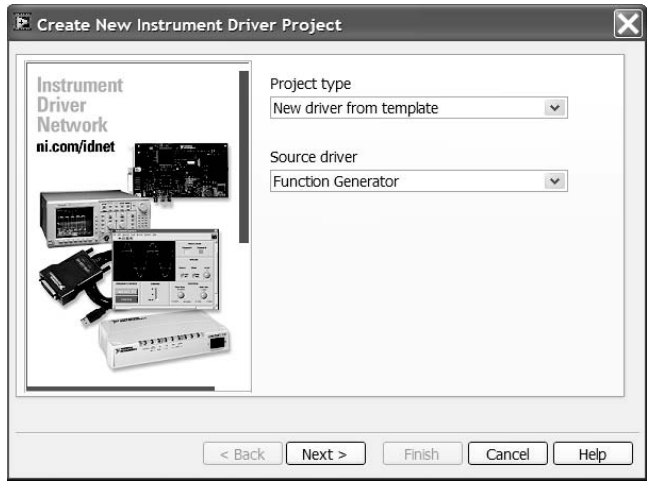


FIGURE 2.27

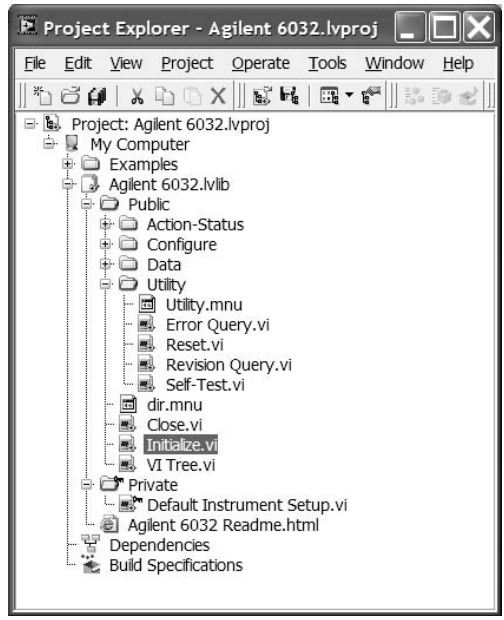


FIGURE 2.28

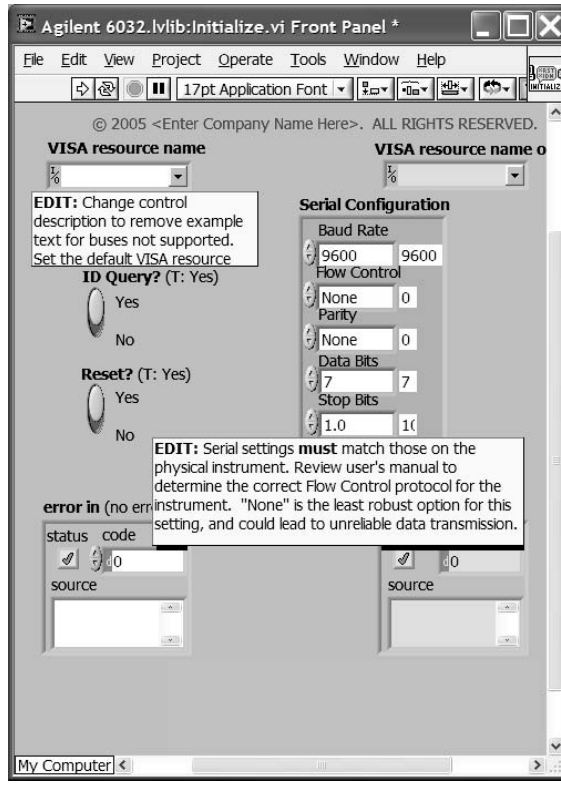


FIGURE 2.29

There is also a selection in the Instrumentation submenu for going directly to the Instrument Driver Network on National Instruments Website. In the Instrument Driver Network you have numerous search options available for finding the driver you need.

2.10 PROFILE FUNCTIONS

In order to make more efficient applications it is often helpful to be able to take a step back and look at the application from a statistical standpoint. This can give the user the ability to see where the weak link in the chain is so as to be able to optimize the code based on the needs of the application. The user can look for VIs that require the most time to execute, use memory inefficiently or where code may be too complex for the application. There are three profile functions in LabVIEW 8 that will help with these issues.

2.10.1 VI PROFILER

LabVIEW has a built-in tool, the VI Profiler, to help you optimize the execution of your applications. It can reveal beneficial information such as how long VIs in the

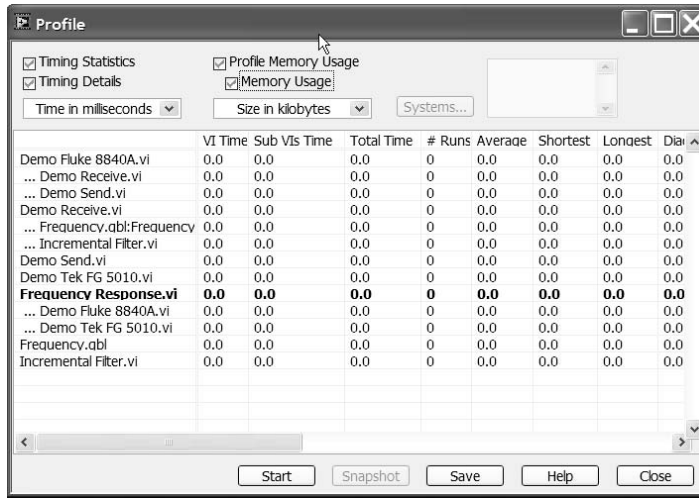


FIGURE 2.30

application take to execute, and how much memory each VI is using. These statistics can then be used to identify critical areas in your program that can be improved to decrease the overall execution time of your application. I/O operations, user interface display updates, and large arrays being passed back and forth are some of the more common sources that can slow down your application. This tool is part of the Full and Professional versions of LabVIEW.

Selecting Performance and Memory from the profile submenu located in the Tools pull-down menu allows you to access the VI Profiler. The profile window is shown in Figure 2.30 as it appears before it is started for capturing the data. The name of the VIs and subVIs, are displayed in the first column, with various pieces of data exhibited in the columns that follow.

Before you start the profiler, you can configure it to capture the information you wish to see by using the checkboxes. VI Time, SubVIs Time, and Total Time are the basic data provided if the other boxes are not selected. The memory usage data is optional because this acquisition can impact the execution time of your VIs. This can result in less accurate data on the timing of your application. Once you start the profiler, the memory options cannot be changed midstream.

Table 2.2 describes the information provided in the various columns in the profile window. It includes items that appear when all of the checkboxes are selected in the profiler window. If memory usage is selected, statistics will be provided for bytes of memory, as well as blocks of memory used. The memory size used is calculated at the end of the execution of a VI and may not be a precise representation of the actual usage during execution.

Frequency Response.vi will be used to demonstrate some actual data taken by the profile window. The Instrument I/O demonstration is an example program that is part of the LabVIEW installation when you choose the recommended install. Figure 2.31 displays the profile window for Frequency Response.vi with the option

TABLE 2.2
Profile Window Data

Statistic	Description
VI time	Total time taken to execute VI. Includes time spent displaying data and user interaction with front panel
SubVIs time	Total time taken to execute all subVIs of this VI. Includes all VIs under its hierarchy
Total time	VI time + subVIs time = total time
Number of runs	Number of times the VI executed
Average	Average time VI took to execute calculated by VI time number of runs
Shortest	Run that took least amount of time to execute
Longest	Run that took longest amount of time to execute
Diagram	Time elapsed to execute code diagram
Display	Amount of time spent updating front panel values from code
Draw	Time taken to draw the front panel
Tracking	Time taken to follow mouse movements and actions taken by the user
Locals	Time taken to pass data to or from local variables on the block diagram
Average bytes	Average bytes used by this VI's data space per run
Minimum. bytes	Minimum bytes used by this VI's data space in a run
Maximum bytes	Maximum bytes used by this VI's data space in a run
Average blocks	Average blocks used by this VI's data space per run
Minimum blocks	Minimum blocks used by this VI's data space in a run
Maximum blocks	Maximum blocks used by this VI's data space in a run

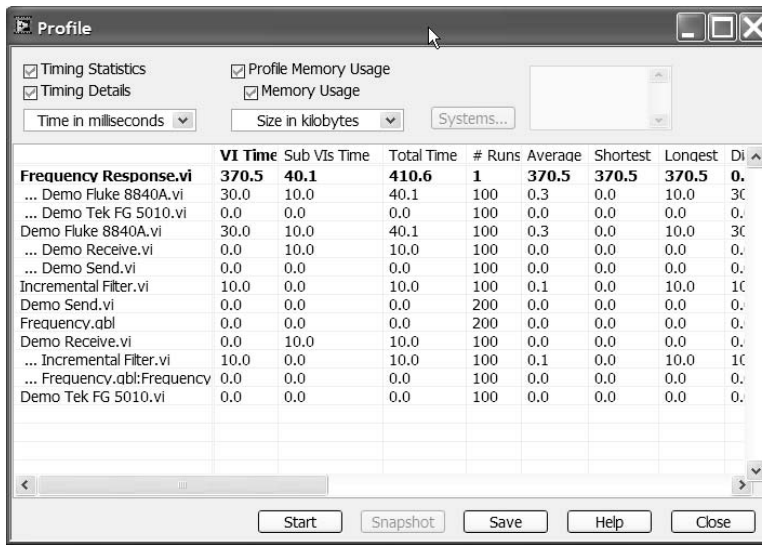


FIGURE 2.31

for Timing Statistics selected. The data shown in the profile window represents a single run, after which the profiler was stopped. This data will vary depending on the configuration of the system from which this VI is executed.

Information on subVIs is normally hidden in the profile window. When you double-click on a cell containing a VI name, its subVIs will appear below. You can show or hide subVIs by double-clicking. Clicking on any of the column headings causes the table to sort the data by descending order. The time information can be displayed in microseconds, milliseconds, or seconds by making the appropriate selection in the drop-down menu. Memory information can be displayed in bytes or kilobytes.

2.10.2 Buffer Allocations

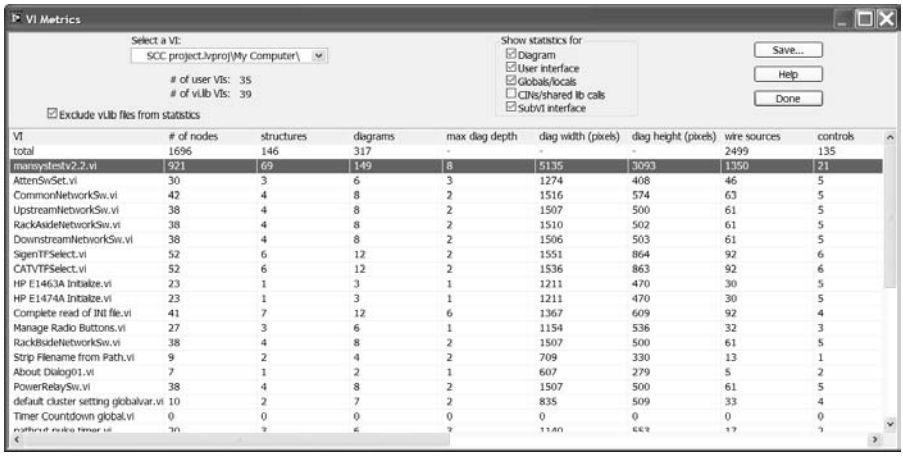
There are two additional options under the profile submenu. The first is to show Buffer Allocations. This function shows the user where buffers are created (memory is allocated) for data operations on the code diagram. This can be a useful tool for optimizing your code for efficient use of memory as well as speed. There may be a case where you are doing manipulation on arrays that could cause an exponential increase in memory needed. If you catch this before pumping a lot of data through your VI you could change the way you are handling the arrays such as predefining the array and then inserting the values into the defined array instead of doing an append which makes a new copy each time. Sometimes problems like this can be missed when the code is debugged with a small data set, but when it is on a live system with large amounts of data the system can come crashing down.

When you select Show Buffer Allocations from the profile submenu, a dialog box will come up giving you the option of what data types to analyze. You can select any combination of data types that you want to look at. Once you have selected the appropriate types you select Refresh. This will put a small black box everywhere a buffer is created.

2.10.3 VI Metrics

The final Profile option is VI Metrics. This function gives the user a means for providing a view into the actual code used in a VI or Project. The dialog that comes up gives the user options for what portion of the VI or VIs you want to analyze such as Code Diagram, User Interface and Globals/Locals. Once all the inputs are set you generate a window containing the statistics. Depending on your selections you will have statistics on parameters such as number of nodes, structures, wire sources and maximum diagram depth. From the window you can select to save the information to a tab-delimited text file. An example of the metrics window is shown in Figure 2.32.

The VI metrics can be a useful tool for code optimization and for project planning. Sections like the maximum diagram depth can alert the developer that there may be too many nested structures leading to code that will be hard to understand and maintain. The information for number of structures, nodes, subVIs,



VI	# of nodes	structures	diagrams	max diag depth	diag width (pixels)	diag height (pixels)	wire sources	controls
total	1696	146	317	-	-	-	2499	135
mansystestv2.2.vi	921	69	149	8	5135	3093	1350	21
AttenSwSet.vi	30	3	6	3	1274	408	46	5
CommonNetworkSw.vi	42	4	8	2	1516	574	63	5
UpstreamNetworkSw.vi	38	4	8	2	1507	500	61	5
RackAsideNetworkSw.vi	38	4	8	2	1510	502	61	5
DownstreamNetworkSw.vi	38	4	8	2	1506	503	61	5
SigenTFSelect.vi	52	6	12	2	1551	864	92	6
CATVTFSelect.vi	52	6	12	2	1536	863	92	6
HP E1463A Initialize.vi	23	1	3	1	1211	470	30	5
HP E1474A Initialize.vi	23	1	3	1	1211	470	30	5
Complete read of INI file.vi	41	7	12	6	1367	609	92	4
Manage Radio Buttons.vi	27	3	6	1	1154	536	32	3
RackBsideNetworkSw.vi	38	4	8	2	1507	500	61	5
Strip Filename from Path.vi	9	2	4	2	709	330	13	1
About Dialog01.vi	7	1	2	1	607	279	5	2
PowerRelaySw.vi	38	4	8	2	1507	500	61	5
default cluster setting globalvar.vi	10	2	7	2	835	509	33	4
Timer Countdown global.vi	0	0	0	0	0	0	0	0
without pulse timer.vi	20	3	6	3	1140	553	17	2

FIGURE 2.32

and wire sources can help give some view into the complexity of the code. This can later be used for planning future projects and for measuring overall efficiency.

2.11 AUTO SUBVI CREATION

Creating and calling subVIs in an application was discussed in the introductory chapter in Section 1.6.9. Connector terminals, icons, VI setup, and LabVIEW's hierarchical nature were some of the topics that were presented to give you enough information to create and use subVIs. There is another way to create a subVI from a section of code on a VI diagram. Use the selection tool to highlight a segment of the code diagram that you would like to place in a subVI and choose Create SubVI from the Edit pull-down menu. LabVIEW then takes the selected code and places it in a subVI that is automatically wired to your current code diagram.

Figure 2.33 will be used to illustrate how Create SubVI is used. The code diagram shown is used to open a Microsoft Excel file, write the column headers, and then write data to the cells. A section of the code has been selected for placing inside of a subVI using the menu selection. The result of this operation is shown in Figure 2.34, where the selected segment of the code has been replaced with a subVI. The subVI created by LabVIEW is untitled and unsaved, an activity left for the programmer. When the subVI has been saved, however, the action cannot be undone from the Edit pull-down menu. Be cautious when using this feature so as to prevent additional time spent reworking your code. Also, a default icon is used to represent the subVI, and the programmer is again left to customize it.

Notice that although the Test Data control was part of the code selected for creating the subVI, it is left on the code diagram. These terminals are never removed from the code diagram, but are wired into the subVI as input. Then, the appropriate controls and indicators are created in the subVI, as shown in Figure 2.35. The

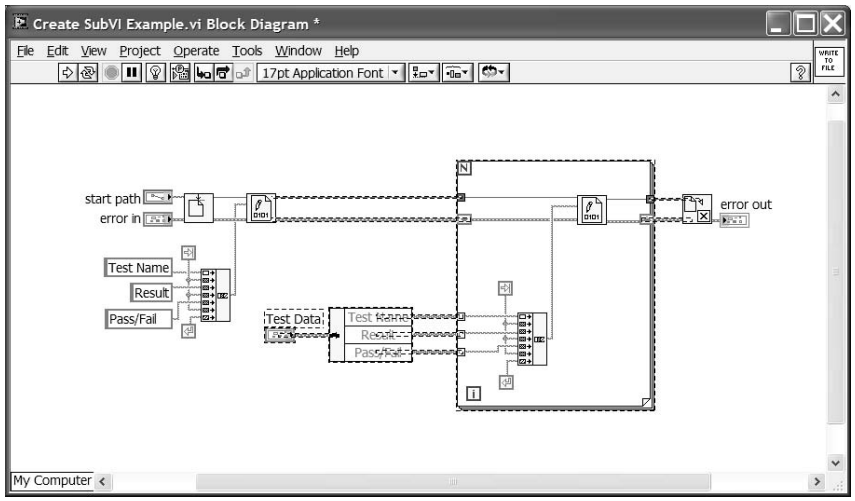


FIGURE 2.33

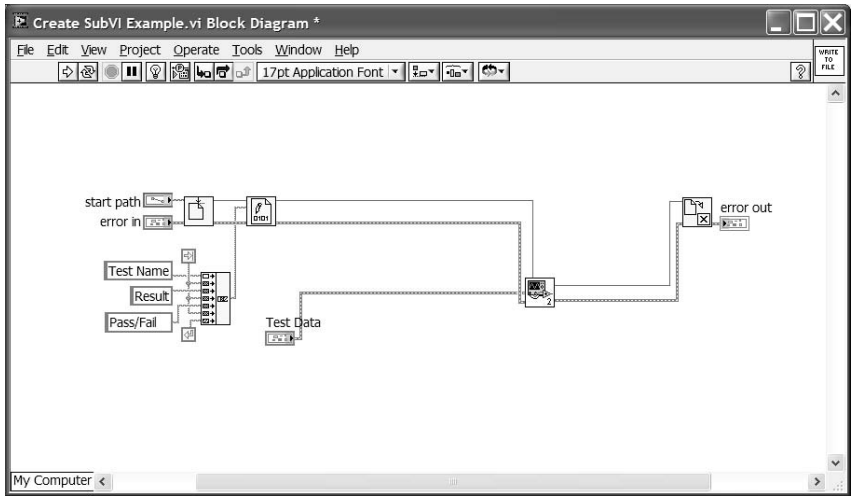


FIGURE 2.34

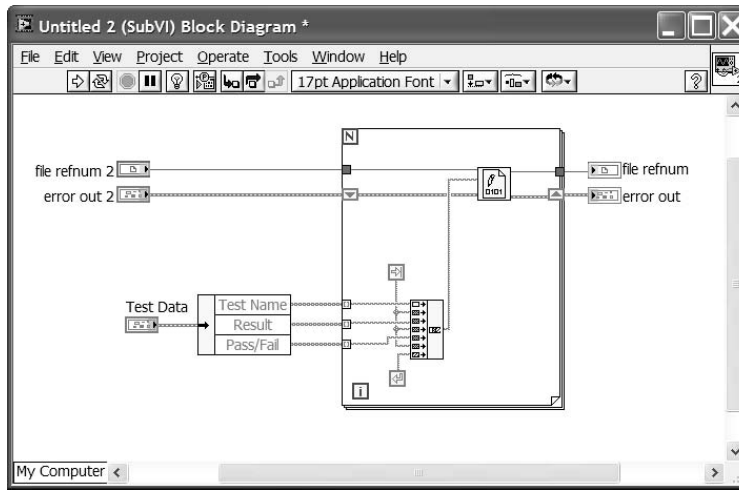


FIGURE 2.35

refnums, error clusters, and the Test Data cluster terminals appear on the code diagram, with their counterparts on the front panel.

There are instances when creating a subVI using the menu option is illegal. When there are real or potential problems for creating a subVI from the selected code, LabVIEW will not perform the action automatically. For potential problems, a dialog box will notify you of the issue and ask you whether you want to go ahead with the procedure. If there is a real problem, the action will not be performed, and a dialog box will notify you of the reason. Some of the problems associated with creating subVIs are outlined in the on-line help under the "Cycles" topic. A cycle is data initiating from a subVI output and being fed back to its input. Attribute nodes within loops, illogical selections, local variables inside loops, front panel terminals within loops, and case structures containing attribute nodes, local variables, or front panel terminals are causes of the cycles described in the help.

This tool should not be used to create all of your subVIs. It is a feature intended to save time when modifications to VIs are needed. National Instruments suggests that you use this tool with caution. Follow the rules and recommendations provided with on-line help when using this feature. Planning is needed before you embark on the writing of an application. There are several things that should be considered to get maximum benefit from code that you develop. Chapter 4, Application Structure, discusses the various tasks involved in a software project, including software design.

2.12 GRAPHICAL COMPARISON TOOLS

Keeping track of different versions of VIs is not always an easy task. Documenting changes and utilizing version control for files can help if the practices are adhered to strictly. When the size of an application grows, or if there is a team involved in the development, it becomes more difficult to follow the guidelines. In LabVIEW 5.0, graphical comparison tools were introduced to help manage the different ver-

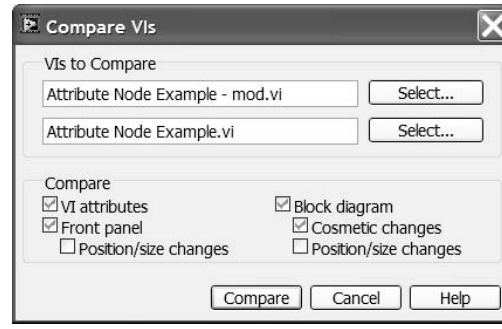


FIGURE 2.36

sions of VIs. These tools are available only with the Professional Development System and Professional Developers Toolkit. You can compare VIs, compare VI hierarchies, or compare files from the source code control tool.

2.12.1 Compare VIs

To compare two VIs, you must select Compare VIs from the Compare submenu in the Tools pull-down menu. This tool graphically compares two VIs and compiles a list of the differences between them. You then have the option of selecting one of the items to have it highlighted for viewing. Figure 2.36 displays the Compare VIs window as it appears.

Use the Select buttons to choose the two VIs for comparison. Only VIs that have already been opened or loaded into memory can be selected as shown in Figure 2.37. The listbox displays VIs, Globals, and Type Definitions that are currently in

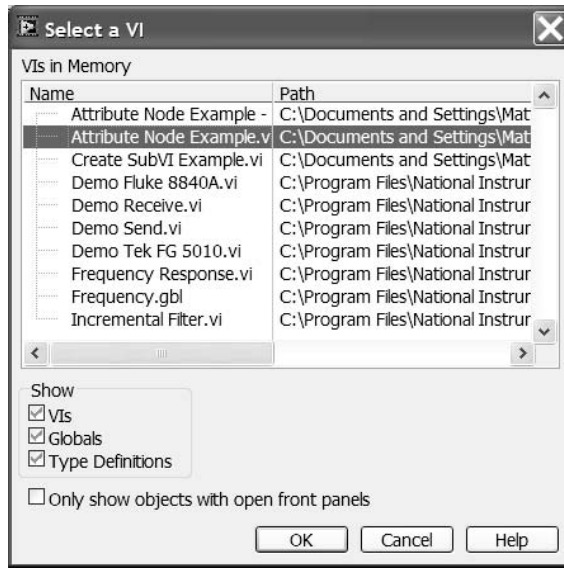


FIGURE 2.37

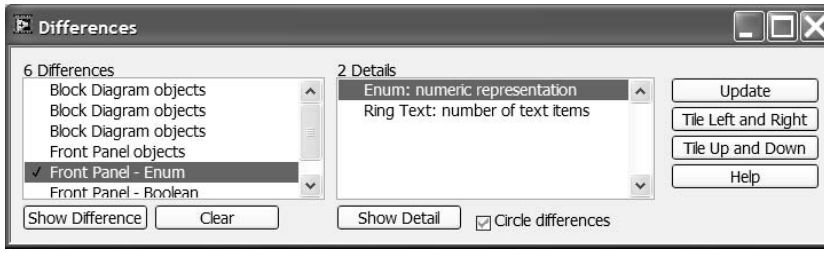


FIGURE 2.38

memory. You should keep in mind that LabVIEW does not allow you to open two VIs with identical names at the same time. If you want to compare two VIs that have identical names, you must rename one of them so they both can be loaded at the same time. Once renamed, both will appear in the listbox.

Once both VIs have been selected, simply press Compare to allow LabVIEW to begin compiling the list of differences. Figure 2.38 shows the window that lists the differences found during the comparison. The first box lists the difference and the second lists the details of the difference. There are two details associated with the difference selected in the figure shown. You can view the differences or the details by clicking the appropriate button. The comparison tool can tile the two VIs' windows and circle the differences graphically to make them convenient for viewing. A checkmark is placed next to the items that have already been viewed in both listboxes. The selection of differences in the list can sometimes be very long. Small differences in objects' locations, as well as cosmetic differences, will be listed even though they may not effect the execution of your VI. To graphically show the changes select Show Detail from the list of differences. Figure 2.39 shows one of the changes displayed.

2.12.2 Compare VI Hierarchies

You can compare two VI hierarchies by selecting Compare VI Hierarchies from the Compare submenu in the Tools pull-down menu. This is to be used for differentiating two versions of a top-level or main-level VI. Figure 2.40 displays the Compare VI Hierarchies window. Use the buttons next to file paths to select any VI for comparison through the file dialog box. With this tool, two VIs with the same name can be selected for comparison of hierarchies, unlike the tool for comparing VI differences. LabVIEW takes the second VI selected, renames it, and places it in a temporary directory for comparison purposes. This saves you the trouble of having to rename the VI yourself when you want to use the tool to find differences.

When Compare Hierarchies is clicked, descriptions of the differences are displayed along with a list of all of the VIs. All of the descriptions provided are relative to the first VI selected. For example, if a VI is present in the first hierarchy and not in the second, the description will tell you that a VI has been added to the first hierarchy. If a VI is present in the second hierarchy and not in the first, the description will tell you that a VI has been deleted from the first hierarchy. Shared VIs are

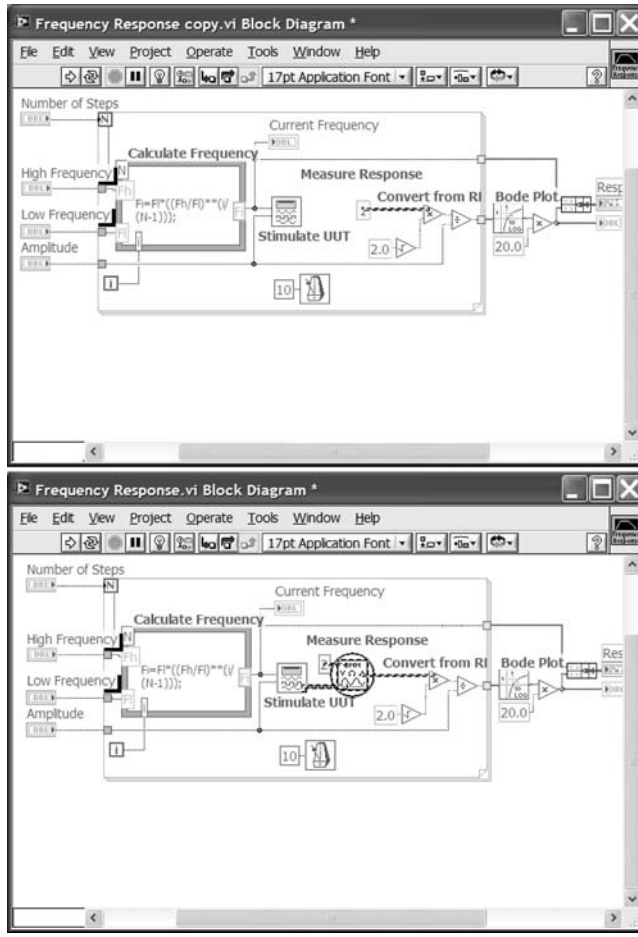


FIGURE 2.39

present in both hierarchies. A symbol guide is provided in the window to assist you while reviewing the list. An option is also available to allow you to view these differences graphically by having the two window displays tiled for convenience. This is similar to the highlighted differences shown when comparing two VIs. The variance is circled in red on both VI hierarchies for quick identification.

2.12.3 SCC Compare Files

Comparing files through the Source Code Control (SCC) Tool is similar to the procedures for comparing VIs and VI hierarchies. The use of the third party Source Code Control applications is described further in Section 2.18. Select Show Differences from the Source Control submenu from the Tools pull-down menu or by right clicking on the VI in the Project Explorer and selecting Show Differences.

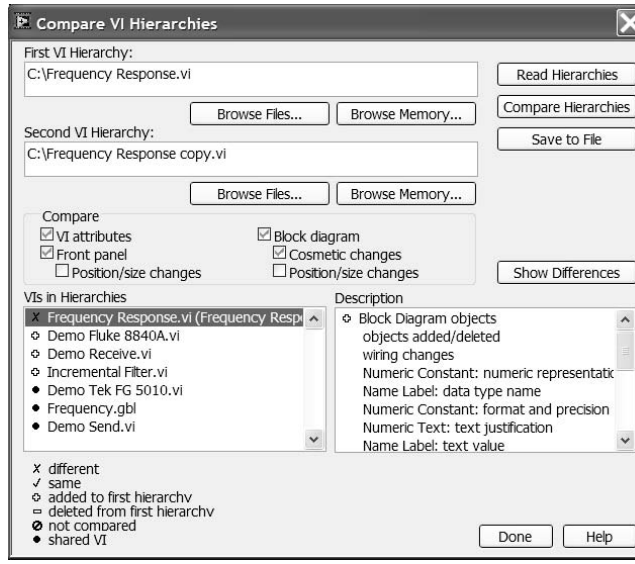


FIGURE 2.40

The SCC Compare Files tool allows you to select a project that you have already created from the pull-down menu. You also have the option of comparing files and having the differences shown by clicking on the respective buttons. Unlike the previous compare VIs function, as the VIs are in SCC, the VIs still have the same names. The SCC utilities keep track of which VI is which, making the function easier to use.

2.13 REPORT GENERATION PALETTE

The Report Generation VIs allow users to programmatically send text reports to a system printer. The Report Generation palette and the Report Layout subpalette are shown in Figure 2.41. These VIs are only available for Windows 2000 and XP because they only work on Win32 systems. The VIs use an ActiveX server, NI-Reports Version 1.1, to perform all of the functions provided in the palette. The NI-Reports ActiveX object is installed on your system at the same time you install LabVIEW.

Chapters 7 and 8 explain ActiveX, the related terminology, and show several examples to help you get started using this powerful programming tool. You can view the source code of the Report Generation VIs by opening them as you would any other VIs. These VIs use the built-in functions from LabVIEW's ActiveX subpalette. National Instruments developed this ActiveX object, as well as the VIs that use the server, to offer a simplified method for generating reports. Once you become familiar with using the ActiveX functions, you will be able to create your own report-generation VIs using the NI-Reports object, and utilize other ActiveX servers that are available.

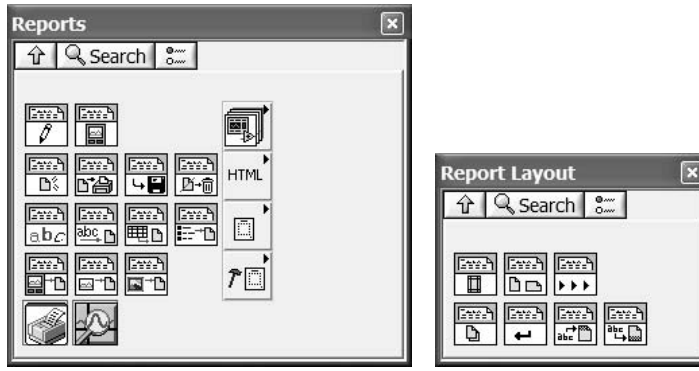


FIGURE 2.41

The first VI available on the palette is Easy Text Report.vi. This is a high-level VI that performs all of the formatting as well as sending the report to the designated printer. Use this VI if you just want to send something to the printer without concern for detailed control or formatting of the report. Easy Text Report performs all of the actions you would normally have to execute if it were not available. It calls other VIs on this palette to perform these actions. All you have to provide is the relevant information and it is ready to go. Simply wire the printer name and the text you want printed. You can optionally set margins, orientation, font, and header and footer information with this VI.

Figure 2.42 illustrates a simple example on the use of Easy Text Report.vi. The VI shown is the same one shown in Figure 2.33, where test data is written to a file for storage. The VI has been modified slightly to send the same cluster information to the printer using the report generation feature. The printer name, desired text, and header and footer information is wired to Easy Text Report.

The other VIs on the palette let you dictate how the report is printed in more detail. In order to print a report, you must first use New Report.vi to create a new report. This VI opens an Automation Refnum to NI-Reports server object whose methods will be used to prepare and print the report. Once the report is created, you can use Set Report Font.vi or any of the other VIs on the Report Layout subpalette for formatting. Append Numeric Table to Report.vi lets you add numeric data,

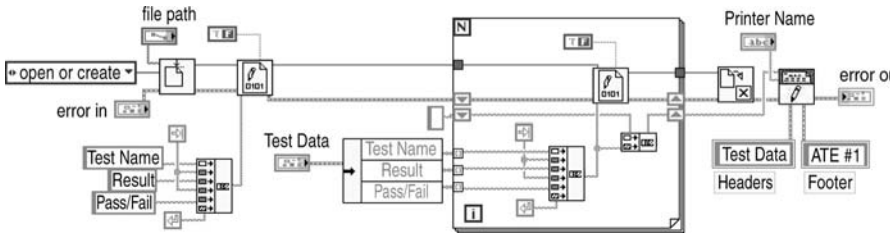


FIGURE 2.42

through a two-dimensional array, to your printout. You can then print your information with the Print Report VI. Finally, remember to release the resources being used for printing by executing Dispose Report.vi. You can open and view the source code for Easy Text Report.vi to get a better understanding of how to use the Report Generation VIs. This VI performs the sequence of actions described in using the NI-Reports ActiveX object.

2.14 APPLICATION BUILDER

The Application Builder allows you to create standalone applications, shared libraries and source distributions. The Application Builder is an add-on package that must be purchased separately, normally shipping only with the Professional Development System of LabVIEW. The tool is accessible through the project explorer. The application builder enables you to generate an executable version of your LabVIEW code, create an installer for your application and install any support files using one dialog box.

The first step to create a standalone application is to create a build specification. By selecting a file in your project explorer and selecting Build Application from the tools menu you will get the dialog box shown in Figure 2.43. In this window you have several sections that can be configured including Application Information, Destinations and Source Files. The Application Information category gives you the ability to set the build name, target name, destination directory, version information and miscellaneous other parameters. The Source Files category gives you the ability to select the VIs that will launch with the application as well as to include files that

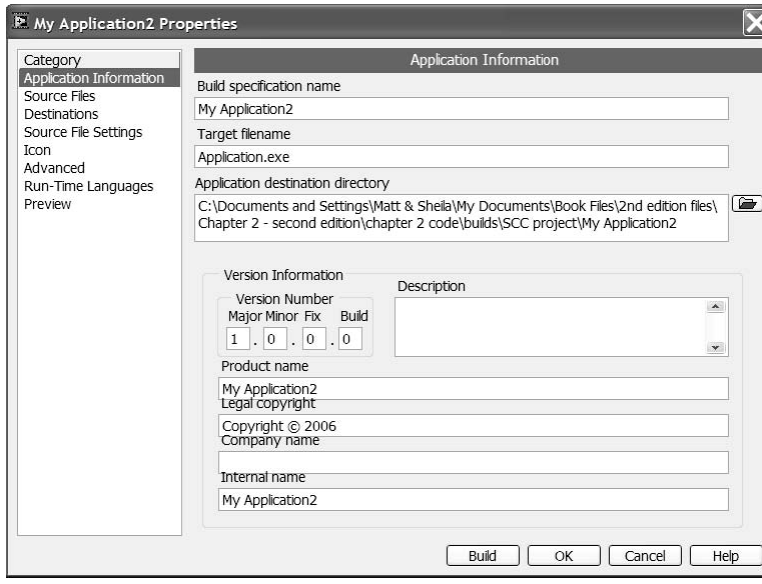


FIGURE 2.43

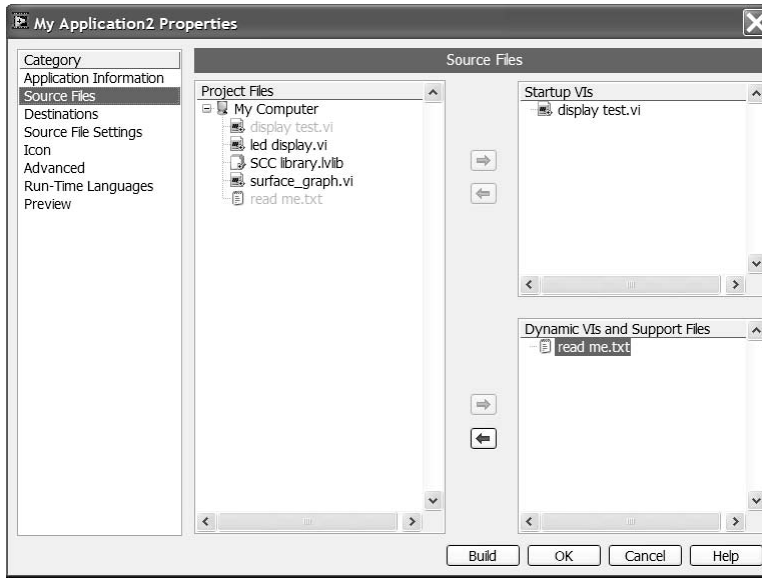


FIGURE 2.44

are not part of the application such as VIs that are dynamically loaded or documentation files. The builder will include all of the subVIs in the top level VI's hierarchy so that you do not have to create a library. An example of this is shown in Figure 2.44.

The Source File Settings category gives you the ability to select display and function options for each VI in the build. You can also set whether a VI is a startup VI, dynamic VI or Include only if referenced. In the Advanced category there are some additional options like the ability to enable debugging.

Once you are done updating all the settings in the dialog box you will see the new build specification listed in the project explorer. This can bee seen in Figure 2.45. To generate the executable you simply right-click on the build specification and select Build. The option is also available under the Project menu in the project explorer.

After you have created the executable, you can place it on any machine for use as a distributed program. When you choose to create an installer, LabVIEW's runtime library gets installed automatically on the target computer. This allows you to run executable LabVIEW programs on computers that do not have LabVIEW installed on them. LabVIEW's runtime library consists of the execution engine and support functions that are combined into a dynamic link library (DLL).

2.15 SOUND VIs

Sound VIs are found on the Sound subpalette of the Graphics and Sound palette. These built-in VIs permit you to easily incorporate sound and the manipulation of sound files into your applications. This section is only a brief overview of the sound VIs, and detailed information on their use will not be provided. The Sound palette and its associated subpalettes are displayed in Figure 2.46.

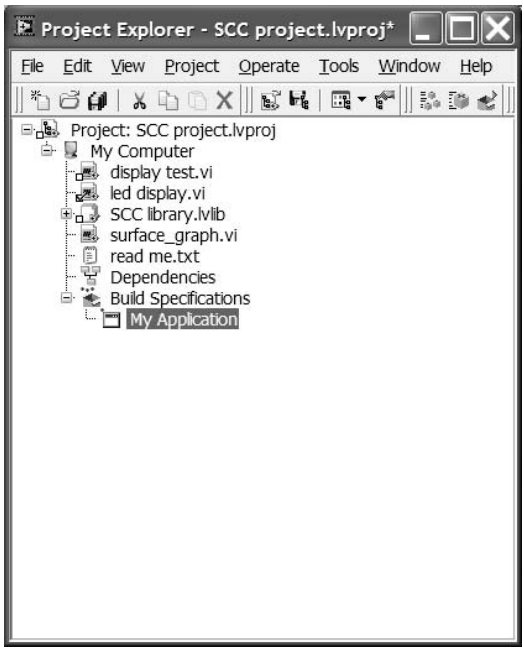


FIGURE 2.45

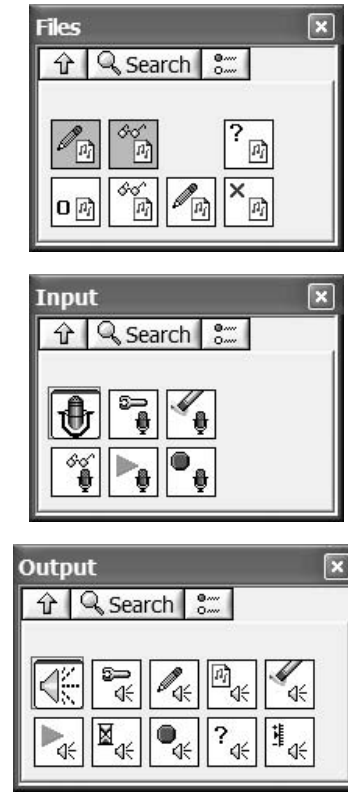


FIGURE 2.46

You can read and write waveform files to an output device using Snd Read or Write Waveform VIs. These are both general VIs that simplify the use of the sound VIs. You should ensure that your multimedia playback and recording audio devices and drivers are installed properly on your system before using the sound VIs.

For more control, use the VIs located on the Input, Output and Files subpalettes. The Sound Input VIs allow you to configure and capture data from an input device. The Sound Output VIs let you configure, send data, and control volume for your output device. The File VIs give you the ability to read and write sound files. Use Sound Input Configure or Sound Output Configure functions first when sending or retrieving data from the devices, and pass the Task ID when performing other functions. The Task ID serves as a reference to your device that is then used by the other VIs.

Two VIs that you may find useful in your applications are Beep.vi and Play Sound File.vi. The first VI simply makes a beep sound on Windows, Macintosh, and Unix. It can be used to send an audio signal to the user when an action has occurred. For example, you may want to have your application beep to let the user know that it needs attention when a test has completed its run or when an error has occurred. Play Sound File can be used in a similar manner. It is capable of playing

back any waveform audio file (*.wav). Just provide the file path of the audio file you wish to play.

2.16 APPLICATION CONTROL

There are several LabVIEW functions that give the developer added control of the way a LabVIEW program executes. In older versions of LabVIEW these functions were grouped in one Palette (Application Control), but they are now located in several palettes. These application-control functions will be discussed below.

2.16.1 VI Server VIs

The VI server functions are part of the Application Control Palette. The Application Control Palette is shown in Figure 2.47. The VI server functions allow the LabVIEW programmer control over a VI's properties, subVI calls, and applications. Through the use of the VI server functions, you can set the properties of the VI user interface through the code diagram. You can set whether to show the run button, show scrollbars, set the size of the panel, and even the title that appears at the top of the window. VI Server functions also allow you to dynamically load VIs into memory, which can speed up your application by not having VIs in memory unless they are being used. This is especially useful when you have seldom used utility VIs that can be loaded when needed and then released from memory to free up system resources.

Remote access is another capability made available through the VI server. A VI can be opened and controlled on a remote computer with relative ease. Once the connection is made to the VI, the interaction is the same as with a local VI using the server VIs. A brief description of the VI server VIs is provided below, followed by some examples.

Open VI Reference allows the user to obtain a reference to the VI specified in the VI Path input. You also have the option of opening this VI on the local machine

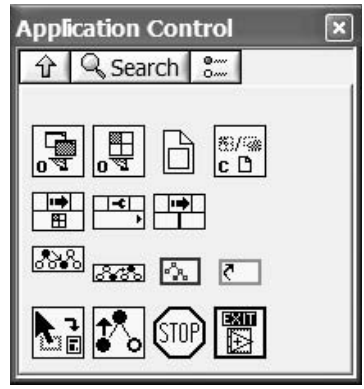


FIGURE 2.47

or a remote machine. If you want to open the VI on a remote computer, the Open Application Reference VI needs to be used. The input to this VI is the machine name. The machine name is the TCP/IP address of the desired computer. The address can be in IP address notation or domain name notation. If the input is left blank, the VI will assume you want to make calls to LabVIEW on the current machine. The output of this VI is an application reference. This reference can then be wired to the Open VI Reference VI. Similarly, the application reference input to the Open VI Reference VI can be left unwired, forcing LabVIEW to assume you want the LabVIEW application on the current computer.

Once the application reference has been wired to the Open VI Reference, there are a few inputs that still need to be wired. First, a VI path needs to be wired to the VI. This is the path of the VI to which you want to open a session. This session is similar to an instrument session. This session can then be passed to the remainder of the application when performing other operations. If only the VI name is wired to this input, the VI will use relative addressing (the VI will look in the same directory as the calling VI). The next input will depend on whether you want to use the Call by Reference Node VI, which allows the programmer to pass data to and from a subVI. You can wire inputs and outputs of the specified VI through actual terminal inputs. The Call by Reference Node VI will show the connector pane of the chosen VI in its diagram. To be able to use this function, the Type Specifier VI Refnum needs to be wired in the Open VI Reference VI. To create the type specifier, you can right-click on the terminal and select Create Control. A type specifier refnum control appears on the front panel. If you go to the front panel, right-click on the control, and choose Select VI Server Class, there will be a number of options available. If you select Browse, you can select the VI for which you want to open a type specifier refnum. The Browse selection brings up a typical open VI window. When the desired VI has been selected, the connector pane of the VI appears in the refnum control as well as in the Call By Reference Node VI. There is also a Close Application or VI Reference VI. This VI is used to close any refnums created by the above VIs.

An example of these functions would be helpful to see how they fit together. We created a VI that has two digital control inputs and a digital indicator output. The code diagram simply adds the two controls and wires the result to the indicator. The VI in this example is appropriately named “Add.vi.” In our VI reference example, we first open an application reference. Since we are using the local LabVIEW application, we do not need to wire a machine name. Actually, we do not need to use this VI, but if we want to be able to run this application on other computers in the future, this VI would already be capable of performing that task. The values of the two inputs are wired to the connector block of the Call by Reference Node VI. The Output terminal is wired to an Indicator. The code diagram of this VI is shown in Figure 2.48.

There is a second way to perform the same function as the above example. The example could be executed using the Invoke node instead of the Call by Reference Node VI. The Invoke node and the Property node are both in the Application Control Palette. These nodes can be used for complete control over a subVI. The Property node allows the developer to programmatically set the VI’s settings such as the

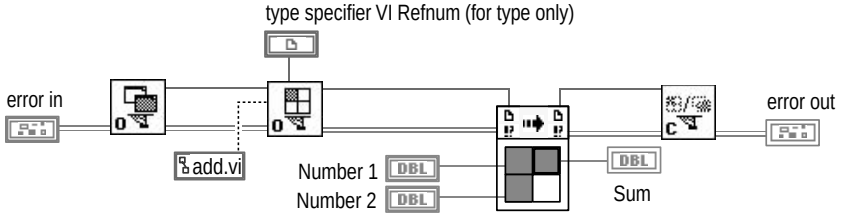


FIGURE 2.48

execution options, history options, front panel window options, and toolbar options. The settings can be read from the VI or written to the VI. The Invoke node allows the developer to perform actions like set control values, read control values, run the VI, and print the VI. The previous example of calling the Add VI to add the two numbers was rewritten to use the Invoke node instead of the Call by Reference Node VI. Figure 2.49 shows the code diagram of this VI.

As can be seen from the example, more code is necessary to perform the task of executing a subVI. The true power in the use of the Invoke and Property nodes is the control the programmer has over the VI. By using these nodes, you can configure the VI in the required manner for the application. For example, you may want to use the same VI for a number of applications but need the front panel to be displayed in only one of the applications. The application that needs the front panel to be displayed can use the Property node to set that value to "true." This allows the code to be reused while still being flexible.

We will now provide an example of using the Property node to control a VI's property programmatically. This example VI will call a subVI front panel to allow the user to input data. The data will then be retrieved by the example and displayed. The desired subVI front panel that needs to be displayed during execution is shown in Figure 2.50. This VI will first open a connection to a subVI through the Open VI Reference function previously described. Once the reference to the VI is opened, we will set the subVI front panel properties through the Property node. By placing the Property node on the code diagram, and left-clicking on the property selection with the operator tool, a list of available properties appears. From this menu we will select our first property. First, we will select Title from the Front Panel Window section. This will allow us to change the title that appears on the title bar when the VI is visible.

The next step is to change additional properties. By using the position tool, you can increase the number of property selections available in the Property node. By dragging the bottom corner, you can resize this node to as many inputs or outputs as are needed. The properties will execute from top to bottom. For our user interface, we will set the visibility property of the ScrollBar, MenuBar, and ToolBar to "false." The final property to add is the front panel visibility property. We will set this property to "true" in order to display the front panel of the subVI.

Next, we will use the Invoke node to select the Run VI method. This will perform the same action as clicking on the Run button. The Invoke node only allows one method to be invoked per node. A second Invoke node is placed on the code diagram

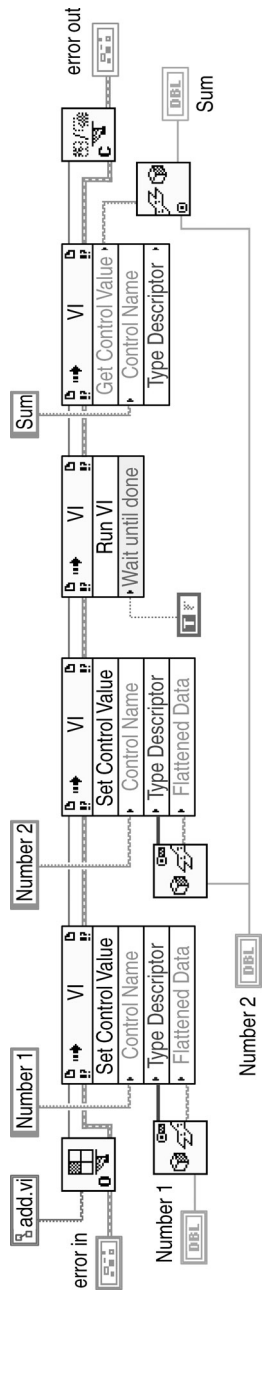

error out
Sum
VI
Get Control Value
Control Name
Type Descriptor
Sum
VI
Run VI
Wait until done
Number 2
VI
Set Control Value
Control Name
Type Descriptor
Flattened Data
Number 1
VI
Set Control Value
Control Name
Type Descriptor
Flattened Data
add.vi
error in
Number 1
Number 2
DBL

FIGURE 2.49

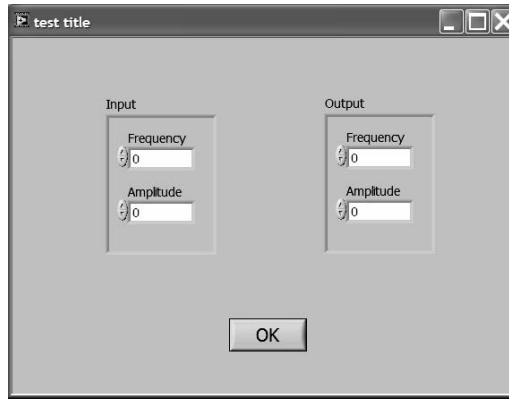


FIGURE 2.50

in order to obtain the subVI output. The name of the output control is wired into the Invoke node. The output of the node is wired to an Unflatten from String function, with the resulting data wired to the front panel. A Property node is then placed on the code diagram to set the subVI front panel visibility attribute to "false." The final step is to wire the VI reference to the Close function. The code diagram is shown in Figure 2.51.

2.16.2 Menu VIs

The Menu subpalette can be found in the Dialog & User Interface palette. The Menu VIs allow a programmer to create a menu structure for the application user to interact with. The menus are similar to the typical window menus at the top of the toolbar. The first VI is the current VI's menu. This function provides the current VI's menu refnum to be use by the other functions. The next VI is the Get Menu Selection VI. This VI returns a string that contains the menu item tag of the last selected menu item. The function has two primary inputs. The programmer can set the timeout value for the VI. The VI will read the menu structure for an input until the timeout value has been exceeded. There is also an option to block the menu after an item is read. This can be used to prevent the user from entering any input until the program has performed the action necessary to handle the previous entry. The Enable Menu Tracking VI can be used to re-enable menu tracking after the desired menu has been read.

There are two VIs in the palette for specifying the menu contents. The Insert Menu Items VI allows the programmer to add items to the menu using the item names or item tags inputs. There are a number of other options available and are described in the on-line help. The Delete Menu Items VI allows the programmer to remove items from the menu. There are also functions available for setting or getting menu item information. These VIs manipulate the menu item attributes. These attributes include item name, enabled, checked, and shortcut value. Finally, there is a VI that will retrieve the item tag and path from the shortcut value.

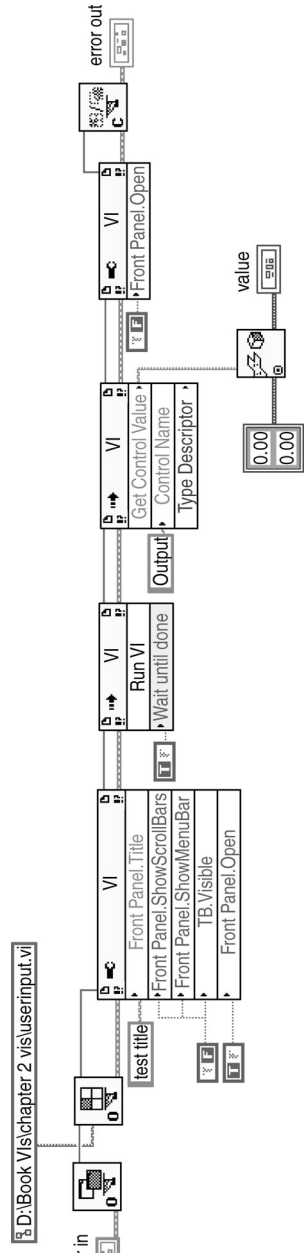

error out
value
D:\Book VIs\chapter 2 vis\userinput.vi
error in
test title
VI
Front Panel.Title
Front Panel.ShowScrollBars
Front Panel.ShowMenuBar
TB.Visible
Front Panel.Open
Run VI
Wait until done
Output
Get Control Value
Control Name
Type Descriptor
Front Panel.Open
0.00
0.00

FIGURE 2.51

Before continuing with an example of using the menu VIs, we will discuss the Menu Editor. If you open the Edit menu from either the front panel or code diagram of a VI, there will be a selection for Run-Time Menu. You will need to make this selection to launch the menu editor. The first choice is a menu ring control that allows you to select the default menu structure, the minimal menu structure, or a custom menu structure. If you look at the remainder of the panel while the default menu structure is selected, you will see that there is a preview section showing the menu names, as they would appear on the front panel during execution. If you click on the menu, the submenus will be displayed. When you are creating your own menu, this preview section will allow you to see how the menu will appear during runtime. Below the preview area is a display area. This area is used to create the new menu. To the right of this display is a section to set the item properties.

To create your own menu, you must select Custom. This will clear the list of menus from the display. If you want to use some of the items from the default menu structure, you can select Item Type from the Property section. This will allow you to copy one item, the entire default structure, or something in between. The items that appear in the window can be deleted, added, moved, or indented. In addition, menu separators can be added to make the menu easier to read. Finally, in the Properties section there is an option to set up shortcut keys for your menu selection. These will work the same way as Ctl-C works for Copy in most menu structures.

We will now add a simple menu to a VI through the menu editor. This VI will contain two menus: a File menu and an Options menu. Under each menu heading there will be two submenus. They will be named “First” and “Second” for convenience. Once we open the menu editor and select Custom, we must select either Insert User Item from the menu or click the Plus button on the editor window. This will add a line in the display with question marks. You will now be able to add information to the Properties section. In the Item Name section we will type in “File.” The underscore is to highlight a specific letter in the label for use with the ALT key. If the user presses ALT-F, the file menu will be selected. Notice that the name placed in the Item Name section is also written into the Item Tag section. These can be made different, if you choose.

The next step is to add another entry. After pressing the Add button, another group of question marks is placed below the File entry. You can make this entry a submenu by clicking on the right arrow button on the editor. Using the arrow keys will allow you to change an item’s depth in the menu structure, as well as move items up and down the list. The menu editor display is shown in Figure 2.52. When you have completed the desired modifications, you must save the settings in an .mnu file. This file will be loaded with the VI to specify the menu settings.

The same menu structure can be created at run time by using the Insert Menu Items function. The first step is to create the menu types. This is done by wiring the insert menu items function with the menu names wired to the item tags input, and nothing wired to the menu tag input. These can be done one at a time, or all at once through an array. You can then call the insert menu items VI for each menu, inserting the appropriate submenu items as necessary. The different options can be modified through the Set Menu Item Info function. Figure 2.53 shows the code used to create the same menu as described above.

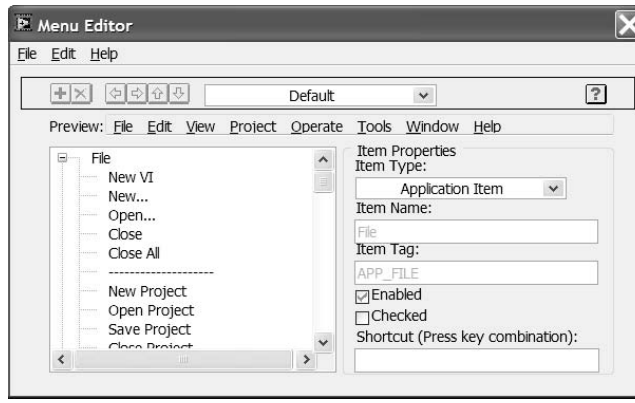


FIGURE 2.52

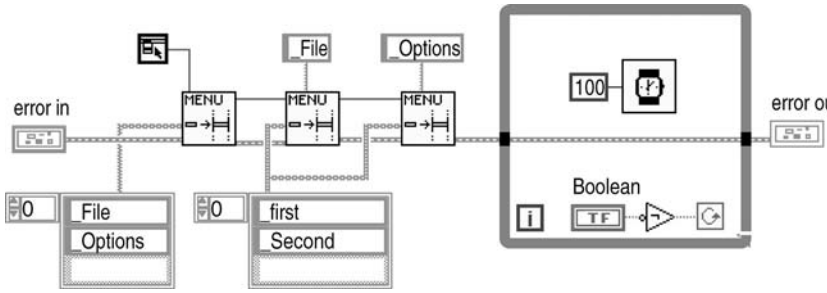


FIGURE 2.53

The following example will illustrate how to use the menu in an application. The VI menus created in the previous example will be modified to have one of the File items display Close. The example will require the use of a state machine. State Machines are covered in Chapter 3. The Get Menu Selection VI is inside the While loop. The output of the VI is wired to the selector of the case structure. The timeout value of the VI is set for 250ms. This is to prevent the VI from using all of the processor time. The Block Menu input is set to "true" to allow the state machine to process the inputs without additional inputs being selected. If the user does not select a menu, an empty string is returned from the VI. A case for the empty string will need to be created to do nothing. This will allow the state machine to continually poll the menus for input. A case matching the close input must be created (_File:_Close). This case will exit the state machine by wiring a "false" to the conditional terminal of the While loop. Cases for the options will need to be created to account for those inputs. The code that you want to run when the menu item is selected should be placed in cases corresponding to the menu items. After the case structure executes, the Enable Menu Tracking VI will need to be wired since menu tracking was disabled when the Get Menu Selection VI was executed. Finally, one of the cases will need to be made the default. A default case is always required when

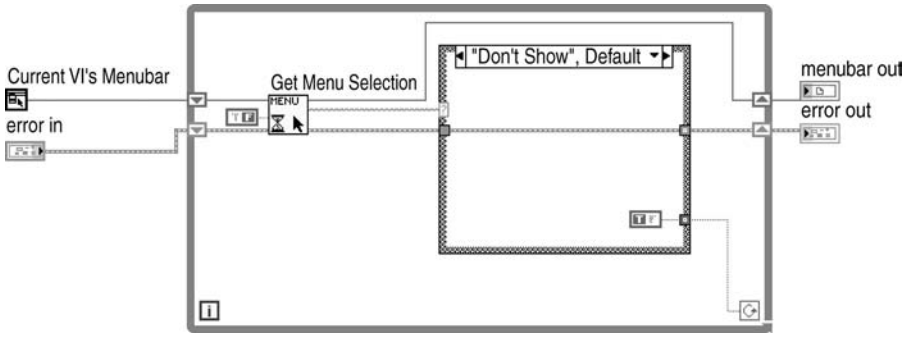


FIGURE 2.54

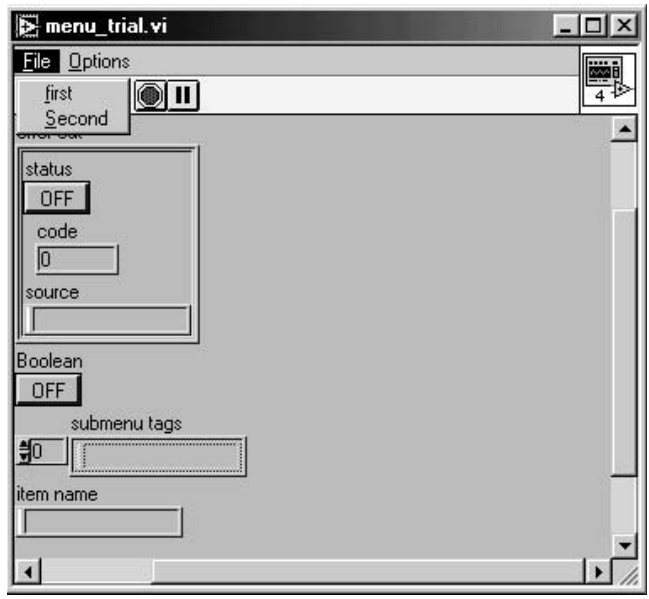


FIGURE 2.55

a case structure is driven by strings. The code diagram for this VI is shown in Figure 2.54. The front panel is shown in Figure 2.55.

2.16.3 Help VIs

The Help VIs are located in the Dialog and User Interface palette. There are four VIs in the Help subpalette. The first VI, Control Help Window, allows the programmer to select the visibility of the help window as well as its location. The second VI, Get Help Window Status, reads the status of the help window (visibility and position). The third VI, Control Online Help, allows the programmer to display the online help window. The inputs allow the programmer to display the contents, the

index, or jump to a specific section of the online help. The VI can also close the online help window. The final VI, Open URL in Default Browser, can be used to display Web-based help in the browser window. These VIs are especially useful when creating custom menus. The menu selections can include items to query the online help or display the help window.

2.16.4 Other Application Control VIs

There a couple of additional functions in the application control palette that should be discussed. The first is the Quit LabVIEW VI. The VI will stop all executing VIs and close the LabVIEW application. It will not execute if a "false" is wired to the input. Next is the Stop VI. This VI stops execution of the current application. It performs the same function as the Stop button on the toolbar. Finally, there is the Call Chain VI. This VI returns an array of VI names, from the VI in which the function is executing to the top level VI in its hierarchy. This function is especially useful when performing error handling. It will allow you to locate a user-defined error quickly and perform the necessary actions to deal with it.

2.17 ADVANCED FUNCTIONS

There are several functions in LabVIEW that are intended for the advanced LabVIEW user. They give the user the ability to call external code modules, manipulate data and synchronize code execution. The functions were part of the Advanced palette in older versions of LabVIEW, but currently have several locations. These VIs and their uses will be discussed in the following sections.

2.17.1 Data Manipulation

There are some functions that give the programmer the ability to manipulate data. These VIs are grouped together in a subpalette in the Numeric palette. The first function in the Data Manipulation subpalette is the Type Cast function. This function converts the data input to the data type specified. For instance, the Type Cast function can convert an unsigned word input to an enumerated type. The resulting output would be the enumerated value at the index of the unsigned word. Another function on this palette allows you to split a number into its mantissa and exponent.

There are a number of low-level data functions. There is both a Left and Right Rotate with Carry function, which rotate the bits of the input. There is a Logical Shift function to shift the specified number of bits in one direction depending on the sign of the number of bits input. A Rotate function is available to wrap the bits to the other end during the data shift. This function also derives direction from the sign of the number of bits input. Moving on to a slightly higher level, there are functions to split a 16- or 32-bit number and to join an 8- or 16-bit number. Finally, there are functions to swap bytes or words of the designated inputs.

The final functions in this palette are the Flatten to String and Unflatten from String functions. The Flatten to String function converts the input to a binary string. The input can be any data type. In addition to the binary string, there is a type string

output to help reconstruct the original data at a later time. The Unflatten from String function converts the binary string to the data type specified at the input. If the conversion is not successful, there is a Boolean output that will indicate the error. Appendix A of the G Programming Reference Manual discusses the data storage formats. This information can also be found in your on-line help.

2.17.2 Calling External Code

LabVIEW has the ability to execute code written in C, as well as execute functions saved in a DLL. There are two methods for calling outside code. The programmer can call code written in a text-based language like C using a Code Interface Node (CIN). The programmer also has the ability to call a function in a DLL or shared library through the use of the Call Library function. The CIN and the Call Library functions reside in the Libraries and Executables subpalette. This subpalette is located in the Connectivity palette. Further descriptions of the CIN and Call Library functions are in Chapter 5.

2.17.3 Synchronization

The Synchronization palette contains five categories of VIs: semaphores, occurrences, notifications, rendezvous, and queues. The semaphore is also known as a "mutex." The purpose of a semaphore is to control access to shared resources. One example would be a section of code that was responsible for adding or removing items from an array of data. If this section of code is in the program in multiple locations, there needs to be a way to ensure that one section of the code is not adding data in front of data that is being removed. The use of semaphores would prevent this from being an issue. The semaphore is initially created. During the creation, the number of tasks is set. The default number of simultaneous accesses is one. The next step is to place the Acquire Semaphore VI before the code that you want to protect. The Boolean output of the VI can be used to drive a case structure. A "true" output will indicate that the semaphore was not acquired. This should result in the specified code not being executed. If the semaphore is acquired, the size of the semaphore is reduced, preventing another semaphore from executing until the Release Semaphore VI is executed. When the release is executed, the next available semaphore is cleared to execute.

The Occurrence function provides a means for making a section of code halt execution until a Set Occurrence function is executed. The Generate Occurrence function is used to create an occurrence reference. This reference is wired to the Wait on Occurrence function. The Wait function will pass a "false" Boolean out when the Occurrence has been generated. If a timeout value has been set for the Wait function, a "true" Boolean will be wired out at the end of the specified time if no occurrence has been generated. This is a nice feature when you need to perform polling. For example, you may want to display a screen update until a section of code completes execution. A While loop running in parallel with the test code can be created. Inside the While loop is the code required for updating the display. The Wait on Occurrence function is wired to the conditional terminal. The timeout is set

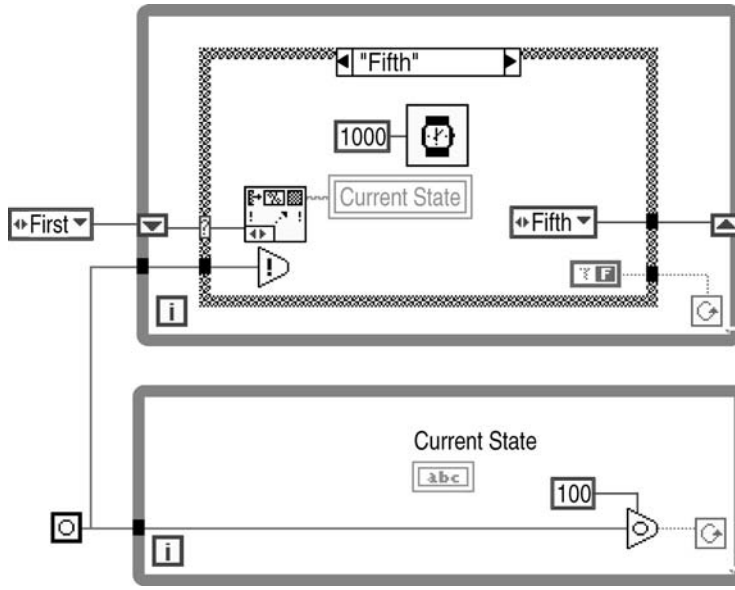


FIGURE 2.56

for the amount of time desired between While loop iterations. If the occurrence is not generated, a "true" is passed to the conditional terminal, and the While loop executes again. If the occurrence is generated, a "false" is sent to the conditional terminal, completing execution of the While loop. An example code diagram using occurrences is shown in Figure 2.56.

The Notification VI is similar to occurrences. Both functions wait for a message to be generated in order to execute. There are two main differences between occurrences and notifications. The Send Notification VI sends a text message as well as instructing the Wait on Notification VI to continue. This is true of one or multiple Wait on Notification VIs. The second difference is the ability to cancel a notification. In addition to these differences you also have the ability to obtain the status from the Get Notifier Status VI.

The Rendezvous VIs are used to synchronize multiple parallel tasks. When the rendezvous is created, the number of items or instances is entered. In order for code at the Wait at Rendezvous VI to execute, the specified number of Wait at Rendezvous VIs must be waiting. The programmer has the option of setting a timeout on the wait VI. There is also a function that can increase or decrease the number of rendezvous required for the code to continue. When all of the required Wait on Rendezvous VIs execute, the tasks all continue at the same time.

The final set of functions is the Queue functions. The queue is similar to a stack. Items are placed on the stack and removed from the stack. The first step in using the queue functions is to create a queue. The create queue function allows the programmer to set the size of the queue. If the maximum size is reached, no additional items will be able to be put into the queue until an item is removed. The Insert Queue Element and Remove Queue Element VIs perform as advertised. There

is one option for both of these VIs that can be of use: the Insert Queue Element has the option of inserting the element in the front (default) or back. The Remove Queue Element allows the data to be taken from the end (default) or beginning. The Queue functions are beneficial when creating a list of states to execute in a state machine. More information and examples of the queue used with state machines is included in the state machines chapter.

There is a lone function in the Synchronization Palette that does not belong in one of the subpalettes. This function is the First Call function. The function has no inputs and one output. The output is a Boolean value indicating if this is the first time that this section of code or subVI has executed. This can be a useful function when there are tasks that need to be performed the first time through a loop or the first time a call has been made to a subVI. Instead of having a shift register running through a loop to track if you have been to a certain case or not can be simplified with this function.

2.18 SOURCE CODE CONTROL

Source Code Control (SCC) is a means for managing projects. The use of SCC encourages file sharing, provide a means for centralized file storage, prevents multiple people from changing the same VI at the same time, and provides a way to track changes in software. The Professional Developers version of LabVIEW and the Full Development version of LabVIEW with the Professional G Developers Toolkit provide means for source code control.

In earlier versions of LabVIEW there was support for Microsoft Visual SourceSafe, Rational ClearCase for Unix and a built-in source code control application. In LabVIEW 8 you have numerous options for source code control implementation including PVCS (Serena) Version Manager, MKS Source Integrity, and CVS with the Push OK Windows client software. If you are using Linux or Mac, the only source code control software that will work in the LabVIEW environment is Perforce. The built-in source code control application is no longer available and will not be discussed here. Some general setup and usage examples will be discussed here, but will not be discussed in depth. Additional information for SourceSafe and Perforce should be obtained from the Professional G Developers Tools Reference Guide and the specific vendor user manuals.

2.18.1 CONFIGURATION

To start using source code control you first need to install the application software. Once installed you can configure LabVIEW to work with the third party application by going to the Tools menu and selecting Options and navigating to the Source Control section, or by selecting Source Control and selecting Configure Source Control. The dialog box that displays is shown in Figure 2.57. The Source Control Provider Name pull down menu initially comes up as < None >. If you have installed your SCC application properly you should see the name in the pull down menu when you click on it. You then have the ability to select where the database (Source Control Project) is located. The database can be installed either on a server or

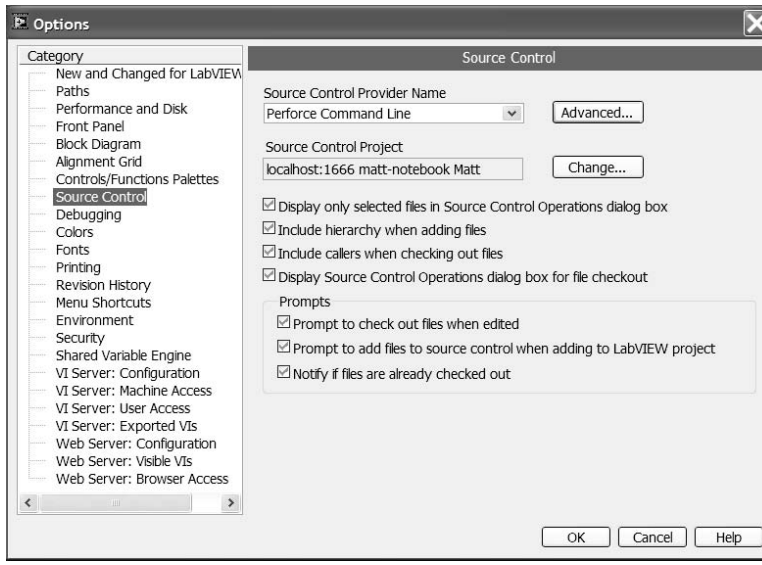


FIGURE 2.57

locally. Depending on which SCC application is installed, the available options may be different.

2.18.2 Adding and Modifying Files

The first step to adding files to SCC is to create a VI or project. Once created the time can be added to source code control by either right clicking on the item in the Project Manager window and selecting Add to Source Control or by Selecting Source Control from the Tools Menu and choosing Add to Source Control. If you add a VI to a project and it is not in SCC you will get a dialog box asking if you want to add it to source control.

You can remove files from SCC by going to the Tools menu in the Project Manager and selecting Remove from Source Control in the Source Control submenu. You will get a warning message that the file will be removed from the SCC database and may be deleted from disk depending on the SCC setup.

To check out a file you can right click on the name in the Project manager window and selecting Check Out. To check the file back in or to undo the checkout you can right click on the file name in the Project Manager and select the appropriate item from the list. Figure 2.58 shows the Project Manager with files stored in source control. When you check in a file you have the ability to add in a comment for the history. Adding comments cannot be encouraged enough. This is the best way to make sure you can find out to what version you would need to rollback in the event major problems are found after several modifications are made. The history is a valuable tool that should be used every time.

Most of the time, when the files are stored on a separate server, you will want to get the latest version of your files in order to have the most up-to-date code before

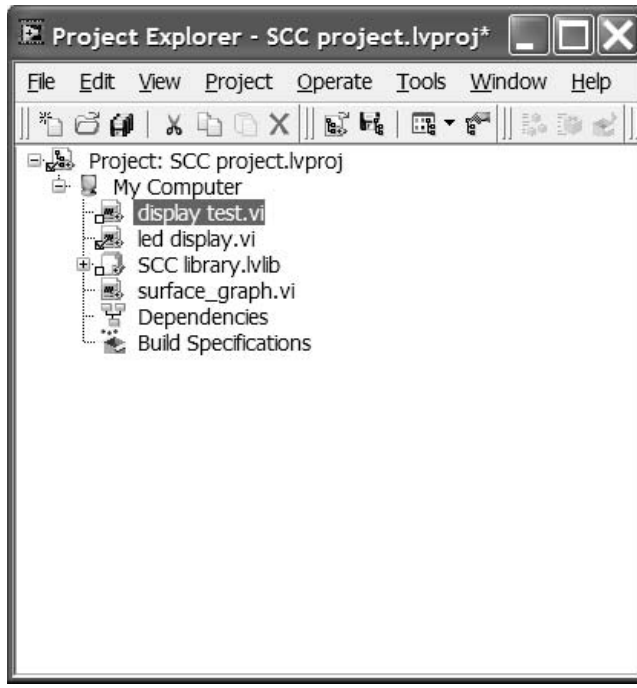


FIGURE 2.58

making modifications. This might also be useful if you have been changing a VI and then decide you want to start over. You will be replacing the modified VI with the one stored in the SCC application. To get the latest version you would go to the Source Code selection from the Tools menu and click on Get Latest Version. This will get the version of the code that the SCC application is presenting as the latest version. Depending on the settings in the SCC application one can set the latest version to a specific version label or VI version based on what code is desired. This is a topic for how to use the SCC application and will not be discussed further here.

2.18.3 Advanced Features

There are a number of advanced features available including Show History and Show Differences. Often when modifying a VI or debugging your code you may notice something had been modified from what you remember. This is especially true in a multiple developer environment. By selecting Show History from the Source Control submenu you can get a list of changes for the VI or project. The list should include who saved the VI, when it was saved and any comments that were entered. This is where the history really pays off. You will be able to see what changes were made each time the VI was saved in order to see what had been modified and hopefully why. The LabVIEW function calls up the native file history in the SCC application. If you are running Perforce through LabVIEW the Show History selection will look like Figure 2.59.

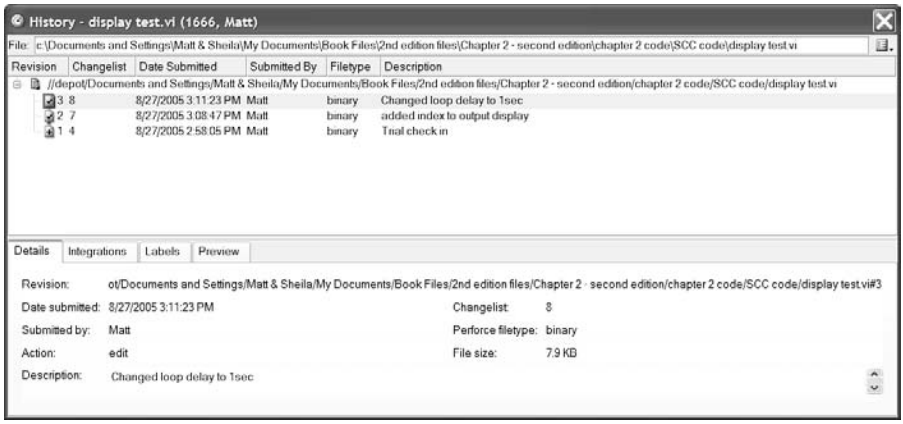


FIGURE 2.59

There are times when the history does not give you enough information on changes that have been made (or maybe even no comments at all). This is where the LabVIEW differencing tool comes in handy. In the past you would have to save the old version of the VI under a new name and then call the LabVIEW differencing tool. Now the process is integrated in LabVIEW so that you can just select Show Differences in the Source Control submenu or by right clicking on the VI in the project manager and the software will compare the versions without having to change names.

2.19 GRAPHS

There are a number of graphing controls available in the Functions palette on the front panel. In addition to these functions on the front panel, there are a number of VIs available in the Graphics and Sounds subpalette on the code diagram. The three main types of graphs will be discussed below.

2.19.1 Standard Graphs

The standard graphs include the waveform graph and chart, the intensity graph and chart, and the XY graph. These graphs have been standard LabVIEW controls for a long time. The waveform graphs and charts are simply displays of 2-D data. A graph is created by collecting the data into an array and then wiring the resulting data to the graph. Each time the graph is updated, the graph is refreshed. The chart is a continual display of data. Each data item is written to the chart, with the previous data remaining. The XY graph is capable of displaying any arrangement of points. Each point is plotted individually, without any relation to time or distribution.

The intensity chart and graph are slightly different than the waveform counterparts. The intensity chart was created to display 3-D data in a 2-D graph. This is achieved by using color over the standard display. The values of the X and Y parameters correspond to a Z parameter. This Z parameter is a number that represents the color to be displayed.

2.19.2 3-D Graphs

Three graphs were added to the Graph palette with LabVIEW 5.1. These graphs are the 3-D surface graph, the 3-D parametric graph, and the 3-D curve graph. These graphs utilize ActiveX technology, and therefore are only available on the Windows versions of the full development and professional development systems. The 3-D surface graph displays the surface plot of the Z axis data. The 1-D X and Y data are optional inputs that displace the surface plot with respect to the X and Y planes. The 3-D parametric graph is a surface graph with 2-D surface inputs for the X, Y, and Z planes. The 3-D curve graph is a line drawn from 1-D X, Y, and Z arrays.

When the 3-D graphs are placed on the front panel, the corresponding VI is placed on the code diagram with the plot refnum wired to the connector. Since these graphs are designed using ActiveX controls, the properties and methods can be modified through the code diagram using the Property and Invoke nodes. These nodes can be found in the ActiveX subpalette in the Communication palette. The nodes can also be found in the Application Control palette. Properties and Methods will be discussed in depth in the Active X chapters (7 and 8). Using the Property node, the programmer is able to set values like the background color, the lighting, and the projection style.

A VI that generates data for the 3-D Surface graph is presented as an example. The VI modifies a number of the attributes through the Property and Invoke nodes. The resulting front panel and code diagram are shown in Figure 2.60.

In addition to the graph VIs, there are additional 3-D graph utility VIs in the Graphics & Sound palette. There are VIs for setting the Basic, Action, Grid, and

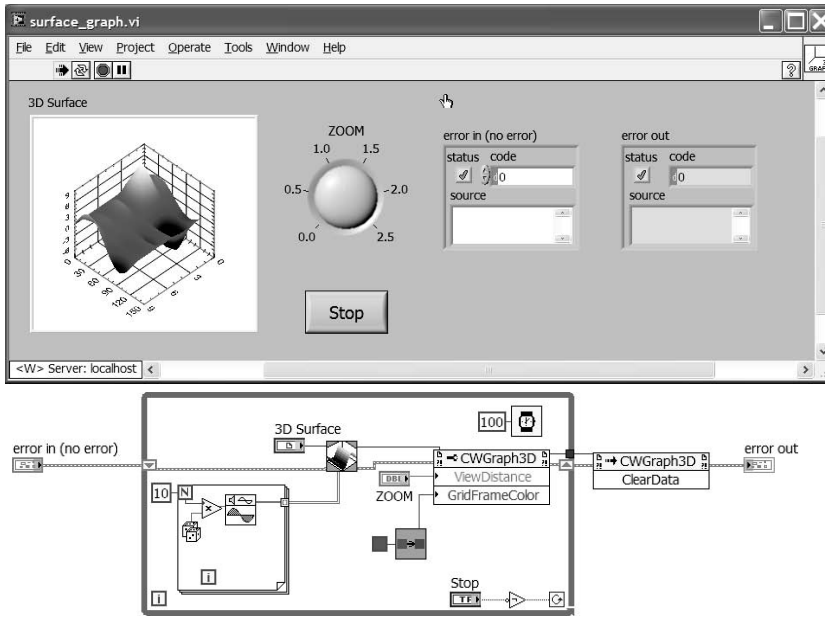


FIGURE 2.60

Projection properties. These VIs simply call the property node internally, allowing the user to simply input the desired properties without having to do as much wiring. These VIs are essentially drivers for the graph properties. There are two additional VIs in the palette. There is a VI to set the number of plots available on a given 3-D graph. And finally, the Convert OLE Colors VI is used to convert LabVIEW colors to OLE colors.

2.19.3 Digital and Mixed Signal Graphs

In LabVIEW 7, National Instruments added support for Digital graphs. Someone taking data from a data acquisition card can now graph the outputs on a digital graph. With the digital graph the user can now combine timing and digital signal information in a waveform graph. There is also a palette of functions for building digital waveforms, generating a digital waveform and for manipulating the information in the waveform. In the digital waveform palette there is a subpalette containing functions for converting Boolean arrays and spreadsheet strings to or from digital waveforms. The digital waveform is simply a cluster of information relevant to a digital signal.

In LabVIEW 8, National Instruments introduced a Mixed Signal Graph. The mixed signal graph gives the user the ability to plot analog and digital waveforms on the same graph. Waveforms of any type and number can be bundled together and wired to a mixed signal graph. The graph will group the analog and digital waveforms and plot them together on the graph. All graphs on the mixed signal graph will share the same x-axis. Figure 2.61 shows an example of a mixed signal graph. The Generate Digital Waveform function was used to generate 3 waveforms. A function was also used to generate a sine wave. These waveforms are then bundled together to produce the resulting mixed signal graph.

2.19.4 Picture Graphs

In the Graph palette, there is a subpalette that contains additional graphing functions. The Picture subpalette contains a Polar plot, Smith plot, Min-Max plot, Distribution plot, and a Radar plot. This subpalette also contains a picture control and a subpalette of graphing related data types.

The Functions palette contains additional picture graph VIs in the Graphics & Sounds subpalette in the programming palette. There is a Picture Plot subpalette that contains the VIs corresponding to the above mentioned plots, as well as functions for building different waveforms and parameter inputs. There is a Picture Functions subpalette that provides a variety of functions such as draw rectangle, draw line, and draw text in rectangle. Finally, there is a subpalette that reads and writes data in the different graphics formats.

2.20 DATA LOGGING

One feature available for application debugging and data storage is front panel data logging. This feature stores the values of all front panel controls with a

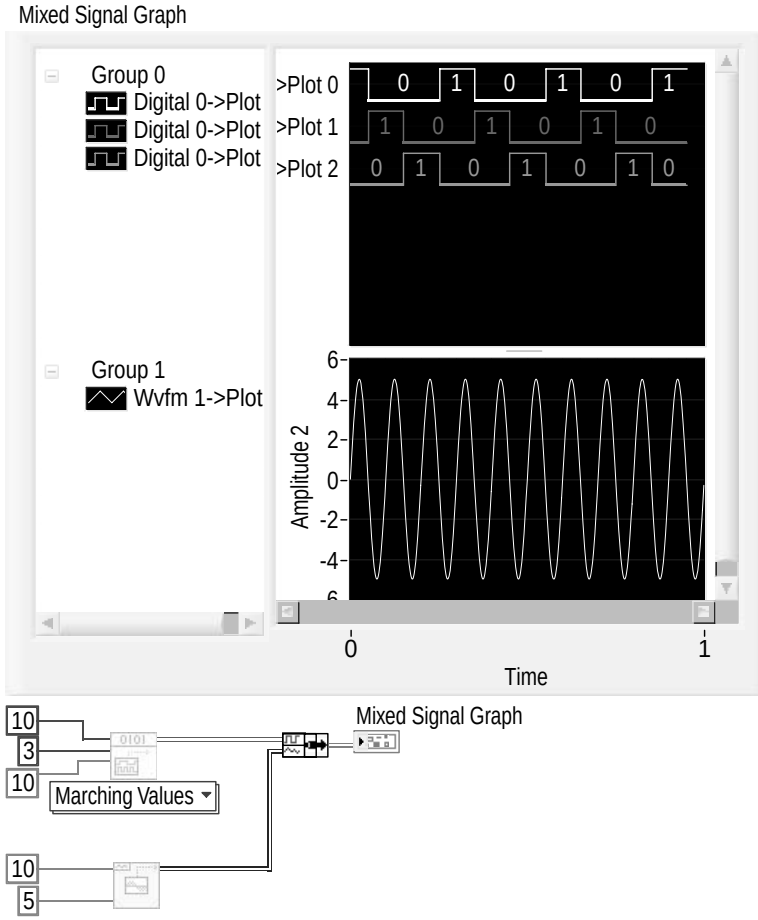


FIGURE 2.61

corresponding time and date stamp to a file. The data from this file can be retrieved and displayed. The VI from which the data is stored can retrieve and display the data through an interactive display. The data can also be retrieved interactively using the file I/O functions. This function is described in more detail in the Exception Handling chapter.

2.21 FIND AND REPLACE

There are times when developing an application that you want to find instances of an object. For example, when wiring a VI with Local variables, you want to make sure that there is no opportunity for a race condition to occur. In order to prevent any possible race conditions you will have to find all occurrences of the Local variable. By finding each instance of the Local, you can ensure there is no possibility

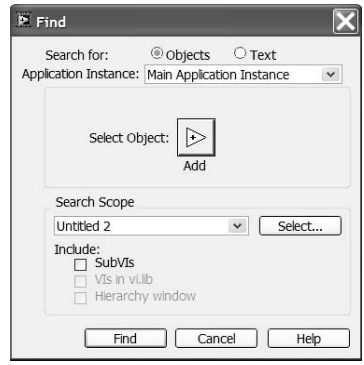


FIGURE 2.62

of another section of the code changing the value of the Local before the specific section has made its modifications.

There is a function in LabVIEW that gives the programmer the ability to find the location of a selected object. The Find and Replace function can be found in the Edit menu. By selecting Find and Replace, a dialog box appears on the screen. The dialog box is shown in Figure 2.62.

The dialog box gives you the opportunity to select what kind of object you want to search for. There are a number of different objects and options available. To select a specific type of object, you need to click on the Add button. A pull-down menu will appear giving you a list of objects to search for. A list of the object categories, and a brief description, is shown in Table 2.3.

TABLE 2.3
Find and Replace Function Objects

Menu Items	Description
Functions	This selection brings up the Functions palette. Any object available in the Functions palette can be selected for the search.
VIs	This option lets you select from any VIs in memory except VIs contained in the vi.lib.
Type Defs	This choice allows you to select any Type Definitions that are in memory and not contained in the vi.lib.
Globals	This selection allows you to choose any Global variables that are in memory and not contained in the vi.lib.
Objects in vi.lib	Choosing this option displays all VIs, Global variables, and Type Definitions located in vi.lib.
Others	This option allows you to select Attribute Nodes, Break Points, and Front Panel terminals. This search does not let you look for specific items, only for all instances of the given object.
VIs by name	This selection provides the user with a dialog box to input the desired VI to search for. This display contains a list of all VIs in memory in alphabetical order. You have the option of selecting whether to show any or all of the VIs, Globals, and Type Definitions.

After selecting the object you want to search for, you will need to specify the search parameters. There is a pull-down menu that allows you to select all VIs in memory, selected VIs, or the active VI. If you choose selected VIs, you can select any number of VIs currently in memory by clicking the Select button. You can also choose to search VIs in vi.lib and the hierarchy window.

At the top of the Find dialog box, there is a choice of what to search for. The default selection is Objects, but you can also search for text. If you select Text, the dialog box will show choices for what to search for, and the search scope. To choose what to search for, the Find What section allows you to type in the text string you are attempting to find. You can select whether to find an exact match including case (Match Case), match the whole word, or to match regular expression. By clicking on the More Options button, you are able to limit your search. You will have the option to select or deselect items including the diagram, history, labels, and descriptions.

Some objects also have the ability to launch their own search by right-clicking on the object and selecting Find. For example, if you select a Local variable and select Find from the pop-up menu, you will have the option of finding the indicator/control, the terminal, or the Local variables. If you select Local variables, a list of all instances of the Local, including the one you selected, will be displayed. You have the option to clear the display, go to the selected Local in the list, or launch the Find function. A checkmark will be displayed next to any Local variables you have already selected.

2.22 PRINT DOCUMENTATION

Many LabVIEW programmers print out VIs and documentation only when they need to create a manual-type document for their application, or need to view some code while away from their computers. There are a number of options when printing that can save time and provide you with documentation that you can use to create your own on-line references. By using the RTF functions, you have the ability to create help-file source code.

The first step to creating your own documentation is selecting Print Documentation from the File menu. A Print Documentation dialog box will pop up, prompting you to enter a format for your documentation. You have options from printing everything, just the panel, or custom prints that allow you to configure the options you want to select. Once you select what you want in your documentation, you then have the option of what style to print.

By selecting the Destination, you have the choice of printing to a printer, creating an HTML file, creating an RTF file, or creating a text file. If you select HTML file, you will be prompted for additional selections. These selections include image format and depth. Your image format options are PNG (lossless), JPEG (lossy), or GIF (uncompressed). In the Depth field you have a choice of black and white, 16 colors, 256 colors, or true color (24 bit). The RTF file options include the depth field and a checkbox for Help compiler source. The option for printing to a text file is the number of characters per line.

When you select Print, you need to be careful. If you are printing HTML documents, the print window will ask you for a name and location for the file. Each

of the images will be saved in separate picture files. If you save a large VI in this manner, you could get a lot of files spread around your drive. You need to make sure to create a folder for your storage space to ensure that your files are in one place and are not difficult to find. The same holds true for RTF file generation.

2.23 VI HISTORY

The history function provides a means of documenting code revisions. This function is useful if no form of source code control is being used. This history will allow you to record what modifications have been done to a VI, and why the changes were made. Along with the text comment you enter, the date, time, user, and revision number are stored. Selecting VI Revision History from the Edit menu can access the VI History window. The window that appears is shown in Figure 2.63.

If you want to use the History function, you will first need to set up the VI's preferences. There are three methods for modifying a VI's history preferences. You can select History from the Preferences found in the Edit pull-down menu. The history preferences accessed here allow you to define when the history is saved. You can choose to have an entry created when the VI is saved, prompt for comments when the VI is closed, prompt for comments when the VI is saved, or allow LabVIEW to record its generated comments. Other options involve different login selections.

You can also select preferences from the VI Setup. In the documentation section of the VI Setup, there is an option to use the default settings from the history preferences. If this option is not selected, a number of options for creating the history become enabled. The final method for accessing history preferences is through the VI server. In the Property node there is a selection for History. A number of the

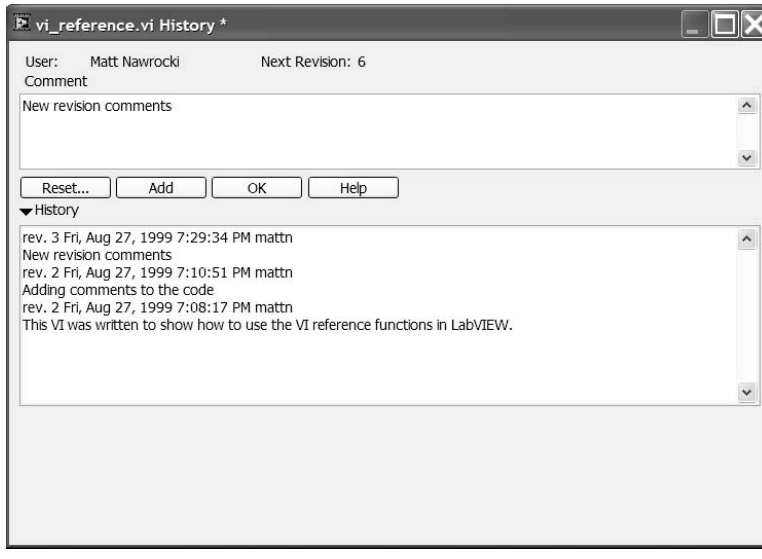


FIGURE 2.63

options available in the preferences are available. The use of the VI server allows you to set the preferences for any VI that you can open with the VI server. If you have a large number of VIs in which to change preferences, you could create a VI to set the preferences programmatically through the VI server.

If you want to add a history entry, you can enter your comments into the comment section of the history window. When all of your comments have been made, you can select Add in the history window. This will add the comments to the VIs history. Clicking on the Reset button will delete the history. The Reset option will also give you the option of resetting the revision number. You have the ability to print the history information when you select Print Complete Documentation or History from the Custom Print settings.

2.24 KEY NAVIGATION

The Key Navigation function provides a means to associate keyboard selections with front panel controls. An example of key navigation is when you have to enter information into a form; pressing the tab key cycles through the valid inputs. Another example of Key Navigation is the ability to press a function key to select a desired control. If you want the user to be able to stop execution by pressing the Escape key, the Stop Boolean can be associated with the Escape key. If the Escape key is pressed while the program is running, the Boolean control is selected, causing the value to change.

To open the Key Navigation dialog box, you need to right click on the control you want to configure, select Advanced and select Key Navigation. The dialog box that is opened is shown in Figure 2.64. The input section on the top left allows you

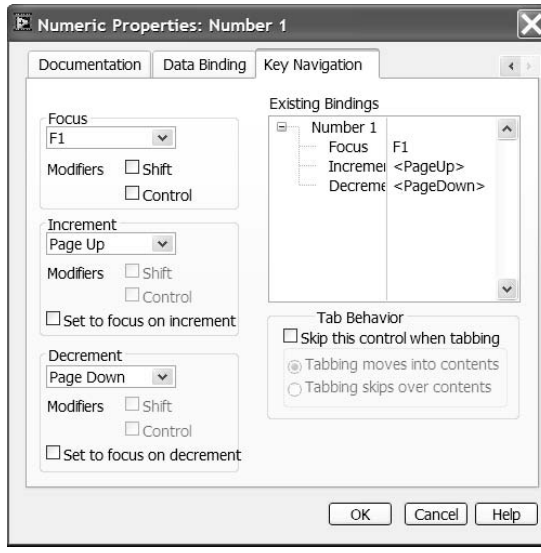


FIGURE 2.64

to define the key assignment for the control. You can combine specific keys like the Escape key, or a function key with the Shift or Control key to expand the possible setting combinations. The listbox on the right displays the currently assigned key combinations. Since you cannot assign the same key combinations to multiple controls, this window can be used to avoid previously defined assignments. If you do select a previously defined combination, the previous setting will be removed.

Below the windows is a selection for skipping the control while tabbing. If this checkbox is selected, the user will not be able to tab to this control. The changes will take effect when you click on the OK button. There are a few issues to keep in mind when using Key Navigation. If you assign the Return (Enter) key to a control, you will not be able to use the Return key to enter the data in any controls. The exception to this rule is if you are tabbing to the controls. If you tab to a control and press Return, the value of the control will be switched. A final consideration is the ability to use your application on different computers or platforms. Not all keyboards have all of the keys available for assignment. This will prevent the user from being able to use keyboard navigation. In addition, there are a few different key assignments among Windows, Sun, and Macintosh computers. If an application will be used on multiple platforms, try to use common keys to make the program cross-platform compatible.

2.25 EXPRESS VIs

An Express VI is similar to a subVI or function. The main difference is with express VIs you have the ability to configure the operation through a dialog box instead of through wiring. This can minimize the wiring needed on your code diagram, but will obscure settings making the code more difficult to debug. The inputs and outputs that are displayed on an Express VI node will change depending on how the Express VI is configured. The express VI will look like a box with input and output connections (National Instruments calls this format expandable nodes). The Express VI will always have a blue border as is shown in Figure 2.65. A subVI with expandable nodes will have a yellow border.

If you wanted to save a configured Express VI as a subVI for use in other applications you would right click on the Express VI and select Open Front Panel.

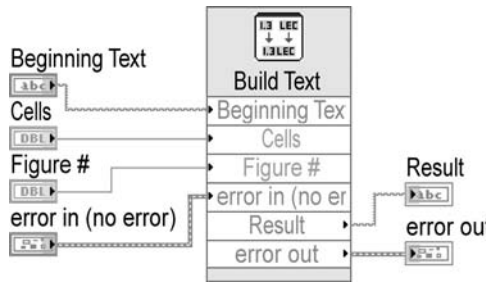


FIGURE 2.65

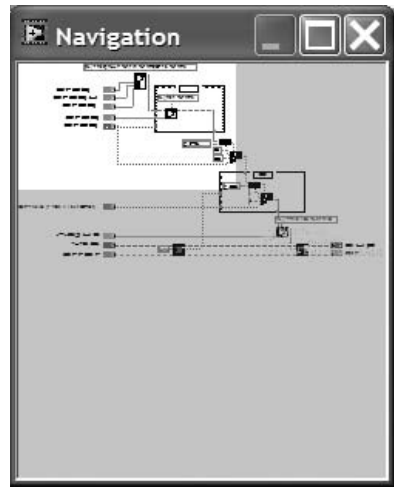


FIGURE 2.66

You will receive a warning that you will not be able to configure this VI as an Express VI anymore. Once you dismiss the message, the Express VI will be converted to a subVI. From here you can save the subVI and edit as needed.

2.26 NAVIGATION WINDOW

All of us at one time or another have had to work with code diagrams that would not fit on three monitors. Now for the sake of argument I will assume that we are talking about editing code written by someone else. It can be a laborious task to go from one end of the diagram to the other trying to find pieces of the code you are modifying. One LabVIEW tool that will make the work a little easier is the Navigation Window. This tool is located in the View pull down menu. When this tool is launched a window appears that allows the user to see a zoomed-out view of the entire code diagram. There is a white box that shows the portion of the code diagram that is visible in the current window. By dragging this white box around you can move the displayed portion of your window to the section of the code you want to view. The navigation window is shown in Figure 2.66.

2.27 SPLITTER BAR

Splitter Bars are a tool used for partitioning the front panel into sections. This tool is similar to splitting a worksheet in Microsoft Excel. The splitter bar can be used to make the diagram look more professional, make navigation easier, freeze portions of the panel from moving or even for creating a tool bar.

To insert a splitter bar on your front panel you would select a vertical or horizontal splitter bar from the Containers palette. Once the splitter bar has been added you can right click on the bar and select from several settings. You can select

whether there is a scrollbar (frozen window or unfrozen). You can select whether the items in the window scale with changes to the panel size and if the items in the panel are locked to a location in the panel. There are also a couple different styles of splitter bars to choose from.

BIBLIOGRAPHY

G Programming Reference, National Instruments, Austin, TX, 1999.
LabVIEW On-line Reference, National Instruments.
Professional G Developers Tools Reference Manual, National Instruments.
LabVIEW 5.1 Addendum, National Instruments.

3 State Machines

State machines, as a programming concept to provide an intelligent control structure, have gained a large degree of acceptance from the LabVIEW programming community. LabVIEW itself supports state machines in the form of template code. LabVIEW has also changed the types of inputs to a case structure to further support State Machines. This chapter will discuss state machine fundamentals and provide examples of how to implement different versions of state machine.

3.1 INTRODUCTION

State machines revolve around three concepts: the state, the event, and the action. No state machine operates effectively without all three components. This section will define all three terms and help you identify meaningful states, events, and actions in your programming. Major mistakes programmers make when working with state machines is not defining appropriate states. We will begin with the concept of state.

"State" is an abstract term, and programmers often misuse it. When naming a state, the word "waiting" should be applied to the name of the state. For example, a state may be waiting for acknowledgment. This name defines that the state machine is pending a response from an external object. States describe the status of a piece of programming and are subject to change over time. Choosing states wisely will make the development of the state machine easier, and the robustness of the resulting code much stronger. Relevant states allow for additional flexibility in the state machine because more states allow for additional actions to be taken when events occur.

Events are occurrences in time that have significant meaning to the piece of code controlled by the state machine. An event that is of significance for our previous example is the event "Acknowledgment Received." This external occurrence will inform the state machine that the correct event has occurred and a transition from states is now appropriate. Events can be generated internally by code controlled by the state machine.

Actions are responses to events, which may or may not impact external code to the state machine. The state machine determines which actions need to be taken when a given event occurs. This decision of what action needs to be taken is derived from two pieces of information: the current state and the event that has occurred. This pair of data is used to reference a matrix. The elements of this matrix contain the action to perform and the next state the machine should use. It is possible, and often desirable, for the next state to be equal to the current state. Examples in this section will demonstrate that it is desirable to have state changes occur only when

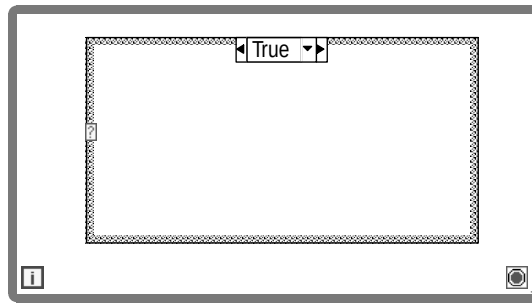


FIGURE 3.1

specific actions occur. This type of behavior is fairly typical in communications control. Unless a specific sequence of characters arrives, the state should not change, or perhaps the state machine should generate, an error.

The state machine itself always makes state changes. The current state is not normally given to code external to the state machine. Under no circumstances should external code be allowed to change the current state. The only information external code should give to the state machine is an event that has occurred. Changing state and dictating actions to perform is the responsibility of the state machine.

3.1.1 State Machines in LabVIEW

A state machine, in simple terms, is a case structure inside a While loop, as shown in Figure 3.1. The While loop provides the ability to continuously execute until the conditional operator is set "false." The case statement allows for variations in the code to be run. The case that is selected to run can be, and usually is, determined in the previous iteration of the While loop. This allows for a relatively simple block of code to make decisions and perform elegant tasks. In its simplest form, a state machine can be a replacement for a sequence structure. In more complex forms of the state machine, the resulting structure could be used to perform the operations of a test executive. The topic of state machines is covered in a number of places, including the National Instruments LabVIEW training courses; however, there is not much depth to the discussions. This chapter will describe the types of state machines, the uses, the pitfalls, and numerous examples.

When used properly, the state machine can be one of the best tools available to a LabVIEW programmer. The decision to use a state machine, as well as which type to use, should be made at an early stage of application development. During the design or architecting phase, you can determine whether the use of a state machine is appropriate for the situation. Chapter 4 discusses how to approach application development including the various phases of a development life cycle.

3.1.2 When to Use a State Machine

There are a number of instances where a state machine can be used in LabVIEW programming. The ability to make a program respond intelligently to a stimulus is

the most powerful aspect of using state machines. The program no longer needs to be linear. The program can begin execution in a specified order, then choose the next section of code to execute based on the inputs or results of the current execution. This can allow the program to perform error handling, user-selected testing, conditional-based execution, and many other variations. If the programmer does not always want the code to be executed in the same order for the same number of iterations, a state machine should be considered.

An example of when a state machine is useful to a programmer is in describing the response to a communications line. An automated test application that uses UDP to accept commands from another application should be controlled with a state machine. Likely states are "waiting for command," "processing command," and "generating report." The "waiting for command" state indicates that the application is idle until the remote control issues a command to take action. "Processing command" indicates that the application is actively working on a command that was issued by the remote application. "Generating report" notes that the state for command has completed processing and output is pending.

Events that may occur when the code is executing are "command received," "abort received," "error occurred," and "status requested." All of these possibilities have a different action that corresponds to their occurrence. Each time an event occurs, the state machine will take a corresponding action. State machines are predictable; the matrix of events, actions, and states is not subject to change.

The purpose of the state machine is to provide defined responses to all events that can occur. This mechanism for control is easily implemented, is scalable for additional events, and always provides the same response mechanism to events. Implementing code to respond to multiple events without the state machine control leads to piles of "spaghetti code" that typically neglect a few events. Events that are not covered tend to lead to software defects.

3.1.3 Types of State Machines

There are a number of different styles of state machines. To this point there is no defined convention for naming the style of a state machine. In an effort to standardize the use of state machines for ease of discussion, we propose our own names for what we feel are the four most common forms of state machines in use. The four styles are the Sequence, the Test Executive, the Classical, and the Queued style state machines. Discussions of the four types of state machines, as well as examples of each type, follow in Sections 3.3 to 3.6.

3.2 ENUMERATED TYPES AND TYPE DEFINITIONS

In the introductory chapters on LabVIEW programming, we stated that an "enumerated type control" is similar to a text ring. The enumerated type control is basically a list of text values associated with a numeric value. The main difference for the enumerated control is that the string is considered part of the data type. The enumerated control can be found in the "List & Ring" section of the Control palette. The default data representation of the enumerated control is an unsigned word.

However, its representation can be changed to unsigned byte or unsigned long by popping up on the control and selecting Representation. When an enumerated type control is copied to the back panel, the result is an enumerated constant. Using the "numeric" section of the Function palette can also create an enumerated constant.

Enumerated constants can make state machines easier to navigate and control. When an enumerated type is connected to a case structure, the case indicator becomes the enumerated text instead of a numeric index. When a programmer is using a case structure with a number of cases, navigation of the case structure becomes difficult if the input is a numeric constant. When the user clicks on the case structure selector, a list of case numbers is shown. It is difficult for the user to determine which case does what, requiring the user to go through a number of cases to find the one that is desired. A benefit of using an enumerated constant with the case structure is readability. When someone clicks on the selector of a case structure controlled by an enumerated-type input, the lists of cases by name are shown. If the enumerated constant values are used to describe the action of the case, the user can easily find the desired case without searching through multiple cases. Similarly, you can use strings to drive state machine structures to enhance readability. However, string matching functions can be time consuming for performance critical applications. When enumerated or string constants are used with a state machine, there are a few added advantages. When the state machine passes the next state to execute to the case structure, the state to be executed becomes obvious. When a state branches to subsequent states, the user can see by the constant which state will execute next. If numerics are used, the person going through the code will have to go through the states to see what is next.

A second advantage involves the maintenance of existing code. When numeric inputs are used to control the state machine, a numeric constant will point to whichever state corresponds to the defined index. A better way to aid in modifications is the use of enumerated types. When the order of states is changed, the enumerated constants will still be pointing to the state with the matching name. This is important when you change the order of, add, or remove states. The enumerated constants will still point to the correct state. It should be noted that in the event a state is added, the state needs to be added to the enumerated constant in order to make the program executable; however, the order of the enumerated constant does not have to match the order of the case structure. This problem does not exist when using string constants to drive the case structure. This leads into the next topic, type definitions used with state machines.

3.2.1 Type Definitions Used with State Machines

A "type definition" is a special type of control. The control is loaded from a separate file. This separate file is the master copy of the control. The default values of the control are taken from this separate file. By using the type definition, the user can use the same control in multiple VIs. The type definition allows the user to modify the same control in multiple VIs from one location.

The benefit of using type definitions with state machines is the flexibility allowed in terms of modifications. When the user adds a case in the state machine, each

enumerated type constant will need to have the name of the new case added to it. In a large state machine, this could be a very large and tedious task — not to mention fairly error prone if one enumerated type is not updated. If the enumerated constant is created from a type definition, the only place the enumerated type needs to be modified is in the control editor. Once the type definition is updated, the remaining enumerated constants are automatically updated. No matter how hard we all try, modifications are sometimes necessary; the use of type definitions can make the process easier.

3.2.2 Creating Enumerated Constants and Type Definitions

Selecting Enumerated Type from the List & Ring section of the Tools pallete creates the enumerated control on the front panel. The user can add items using the Edit Text tool. Additional items can be added by either selecting Add Item Before or After from the popup menu, or pressing Shift + Enter while editing the previous item. The enumerated constant on the code diagram can be created by copying the control to the code diagram via "copy and paste," or by dragging the control to the code diagram. Alternative methods of creating enumerated constants include choosing Create Constant while the control is selected, and selecting the enumerated constant from the Function pallete.

To create an enumerated-type definition, the user must first create the enumerated-type control on the front panel. The user can then edit the control by either double-clicking on the control or selecting Edit Control from the Edit menu. When the items have been added to the enumerated control, the controls should be saved as either a Type Definition or a Strict Type Definition. The strict type definition forces all attributes of the control, including size and color, to be identical. Once the type definition is created, any enumerated constants created from this control are automatically updated when the control is modified, unless the user elects not to auto-update the control.

3.2.3 Converting between Enumerated Types and Strings

If you need to convert a string to an enumerated type, the Scan from String function can accomplish this for you. The string to convert is wired to the string input. The enumerated type constant or control is wired to the default input. The output becomes the enumerated type constant. The use of this method requires the string to match the enumerated type constant exactly, except for case. The function is not case-sensitive. If the match is not found, the enumerated constant wired to the default input is the output of the function. There is no automatic way to check to see if a match was found. Converting enumerated types into strings and vice versa is helpful when application settings are being stored in initialization files. Also, application logging for defect tracking can be made easier when you are converting enumerated types into strings for output to your log files.

There are two methods that can be used to ensure only the desired enumerated type is recovered from the string input. One option is to convert the enumerated output back to a string and perform a compare with the original string. The method

needed to convert an enumerated type to a string is discussed next. A second option is to use the Search 1-D Array function to match the string to an array of strings. Then you would use the index of the matched string to typecast the number to the enumerated type. This assumes that the array of strings exactly matches the order of the items in the enumerated type. The benefit of this method is that the Search 1-D Array function returns a –1 if no match is found.

If the programmer wants to convert the enumerated-type to a string value, there is a method to accomplish this task. The programmer can wire the enumerated type to the input of the format into string function in the Function palette. This will give the string value of the selected enumerated type.

3.2.4 Drawbacks to Using Type Definitions and Enumerated Controls

The first problem was mentioned in Section 3.2.2. If the user does not use type definitions with the enumerated type constant, and the constant needs to be modified, each instance of the constant must be modified when used with a state machine. In a large state machine, there can be a large number of enumerated constants that will need to be modified. The result would be one of two situations when changes to the code have to be made: either the programmer will have to spend time modifying or replacing each enumerated control, or the programmer will abandon the changes. The programmer may decide the benefit of the changes or additions does not outweigh the effort and time necessary to make the modifications. This drawback limits the effectiveness of the state machine because one of the greatest benefits is ease of modification and flexibility.

The programmer needs to be careful when trying to typecast a number to the enumerated type. The data types need to match. One example is when an enumerated type is used with a sequence-style state machine. If the programmer is typecasting an index from a While or For loop to an enumerated type constant, either the index needs to be converted to an unsigned word integer, or the enumerated type to a long integer data type. The enumerated data type can be changed in two ways. The programmer can either select the representation by right-clicking on the enumerated constant and selecting the representation, or by selecting the proper conversion function from the Function palette.

If the programmer needs to increment an enumerated data type on the code diagram, special attention needs to be paid to the upper and lower bounds of the enumerated type. The enumerated values can wrap when reaching the boundaries. When using the increment function with an enumerated constant, if the current value is the last item, the result is the first value in the enumerated type. The reverse is also true; the decrement of the first value becomes the last value.

3.3 SEQUENCE-STYLE STATE MACHINE

The first style of state machine is the sequence style. This version of the state machine is, in essence, a sequence structure. This version of the state machine executes the states (cases) in order until a false value is wired to the conditional terminal. There

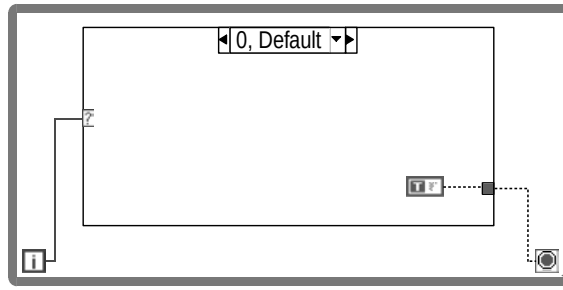


FIGURE 3.2

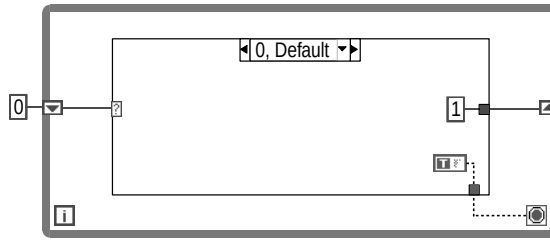


FIGURE 3.3

are a couple ways in which to implement this style of state machine. The first and simplest way is to wire the index of the While loop to the case statement selector. Inside each case, a Boolean constant of "true" is wired to the While loop conditional terminal. The final case passes a false Boolean to the conditional terminal to end execution of the loop. Figure 3.2 shows a sequence machine using the index of a While loop. This type of state machine is more or less "brain dead" because, regardless of event, the machine will simply go to the next state in the sequence.

A second way to implement the sequence-style state machine is to use a shift register to control the case structure. The shift register is initialized to the first case, and inside each case the number wired to the shift register is incremented by one. Again, the state machine will execute the states in order until a false Boolean is wired to the conditional terminal of the While loop. This implementation of a state machine is modified for all of the other versions of state machines described in this chapter. Figure 3.3 shows a simple sequence machine using a shift register.

3.3.1 When to Use a Sequence-Style State Machine

This style of state machine should be used when the order of execution of the tasks to be performed is predefined, and it will always execute from beginning to end in order. This state machine is little more than a replacement for a sequence structure. We generally prefer to use this type of structure instead of sequences because it is far easier to read code that uses shift registers than sequence locals. The single biggest problem with the sequence structure is the readability problems caused by sequence locals. Most programmers can relate to this readability issue.