Bitter, Rick et al "Introduction to LabVIEW"

*LabVIEW Advanced Programming Techniques*

Boca Raton: CRC Press LLC, 2001
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 http://www.nat-inst.com/custed. 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, while 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 drag-and-drop easy in LabVIEW.
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 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 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, 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 the palette, you will find the Continuous Run, Stop, and Pause buttons. The last three buttons are used for alignment of objects on the panel or diagram. 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. Debugging techniques is a topic covered in Chapter 6.

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, select Save with Options from the File menu. 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 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 beginners, 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 Web sites. 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.

1.2.1 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 1.5 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 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.

1.2.2 Web Sites

Several other sources are also available for help on LabVIEW-related topics. National Instruments’ Web site offers help through online technical support, documents, and free downloads. The following table lists some that may be useful.

The LabVIEW Technical Resource is a quarterly publication generated by LTR Publishing, Inc. The issues contain technical information on LabVIEW as well as
tips on programming style and techniques. LabVIEW books are also reviewed in the publication on a regular basis. A resource disk that contains source code associated with some of the articles accompanies each issue. Readers are encouraged to submit and share LabVIEW programming techniques.

1.3 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’s since Version 1.0. Multitasking is a fundamental ability to

<table>
<thead>
<tr>
<th>TABLE 1.1</th>
<th>Web Sites</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.natinst.com/support">http://www.natinst.com/support</a></td>
<td>Technical support and contact information</td>
</tr>
<tr>
<td><a href="http://www.natinst.com/support/techdocs.htm">http://www.natinst.com/support/techdocs.htm</a></td>
<td>Technical documents, application notes, knowledgebase (searchable database), product manuals</td>
</tr>
<tr>
<td><a href="http://www.natinst.com/dnldgtwy.htm">http://www.natinst.com/dnldgtwy.htm</a></td>
<td>Drivers, updates, example programs, instrument drivers</td>
</tr>
<tr>
<td>ftp://ftp.natinst.com/support</td>
<td>Direct link to ftp site for downloads</td>
</tr>
<tr>
<td><a href="mailto:Info-labview-request@pica.army.mil">Info-labview-request@pica.army.mil</a></td>
<td>Submit request for subscription to LabVIEW user group</td>
</tr>
<tr>
<td><a href="http://www.ltrpub.com">www.ltrpub.com</a></td>
<td>LabVIEW Technical Resource</td>
</tr>
<tr>
<td><a href="http://www.webring.org/cgi-bin/webring?ring=labview;list">www.webring.org/cgi-bin/webring?ring=labview;list</a></td>
<td>List of many LabVIEW-related sites</td>
</tr>
</tbody>
</table>
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.

1.4 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, 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 1.6 shows the pop menu that appears for a digital indicator.

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 1.7 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 Positioning 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 Popup 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 valid also.

### TABLE 1.2

**Shortcuts**

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

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 valid also.

## 1.5 FRONT PANEL CONTROLS

Numerous front panel controls are available in LabVIEW for developing your applications. The Controls palette (shown in Figure 1.8) appears when you make the appropriate selection in the Windows menu. The controls are grouped into subpalette categories. 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 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 programmatically. Controls are basically variables that can be manipulated in the code. The following subsections will briefly describe the various control palettes. Keep in mind that the palettes and controls shown correspond to LabVIEW 5.1 and will vary depending on the version you are using. The ActiveX palette will be described in Chapter 9.

### 1.5.1 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, and long word precision. Bytes are eight bit numbers, words are sixteen bit numbers, and long words are 32 bit numbers. Integers may be used as signed or unsigned quantities.

The controls in the Numeric palette (displayed in Figure 1.9) are self-explanatory. The top row contains digital controls and indicators. These are used for allowing the user to input values or to display output values. The next two rows hold horizontal and vertical slides. A tank and a thermometer are also available and can be useful for displaying output. Finally, there are the knob, dial, gauge, meter, color box, and color ramp in this palette.

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 1.10 displays the window that appears when Format & Precision is selected from the pop-up menu.

The nondigital objects in the numeric palette have an option to associate a digital control or indicator with them through the pop-up menu. Just select Digital Display from the Show submenu after popping-up on the object. Figure 1.11 shows the meter with its associated digital indicator for precise readings. The meter, as most controls,
1.5.2 Boolean

The Boolean palette is illustrated in Figure 1.12. This palette contains various true or false controls and indicators. The first two rows contain square and round buttons, push buttons, LEDs, and radio buttons. The remaining booleans include slide switches, toggle switches, labeled buttons, and checkboxes. Their mechanical action and data range can be modified through the pop-up menu. Some of the controls in this palette are also available in the Dialog palette.

Interesting features 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 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 want to stop. Also note that the mechanical action of subVIs is completely ignored; LabVIEW itself is not considered a user.

In general, it does not seem like there is 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.

1.5.3 String & Table

The String & Table palette is displayed in Figure 1.13. It holds the string control, indicator, and table. The string table is simply a two-dimensional string array. LabVIEW strings are far simpler to use than strings in C. 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.

Display options are very useful for programmers performing communications work. Many strings that are sent to serial instruments and other devices contain nonprintable characters. String displays can be set to show the ASCII 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 whitespace 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.

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.

New to LabVIEW 5.0 is the ability to update the value of the string while the user is typing. Previously, a user would have to stop accessing the string control before the new data was available to the wire diagram. This is undesirable when a
more interactive application is necessary. As an example, a serial console application could not have been written well in LabVIEW 4.0. Console applications typically send each character as they become available from the keyboard buffer. The ability to update strings while typing allows LabVIEW strings to support this type of application.

Attribute nodes allow for additional programming options. It is possible to cause the displayed text to flash with the flash attribute node. String controls can use the position attribute node to set or examine the current position of the cursor in the control.

1.5.4 List & Ring

The List & Ring palette is also displayed in Figure 1.13. 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 1.14 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
The selection listbox with an 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 1.15.

The multiple selection listbox 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.

### 1.5.5 Array & Cluster

The last palette displayed in Figure 1.13 is Array & Cluster. To create an array, you must first drag the array container onto the front panel of a VI. Then 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 were 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 1.16 shows the array and cluster 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. Figure 1.17 shows the array and cluster with objects dropped inside them. A digital control 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 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.
1.5.6 Graphs and Charts

Figure 1.18 displays the Graphs palette with the built-in graph and chart objects. The Waveform Chart and Waveform Graph are located in the top row, while the Intensity Chart and Intensity Graph are in the second row. The XY Graph is also
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 1.19 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. Since 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 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 1.20 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.

1.5.7 PATH & REFNUM

The Path & Refnum palette is displayed in Figure 1.18. The first two objects 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 remaining objects on the palette are refnums, which you may need to employ during programming.

A refnum is a distinct identifier or reference to a specific item. This item can be a file, external device, ActiveX object, network connection, or 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 1.21 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 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.

1.6 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 1.22 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.
1.6.1 Structures

The control structures that are accessible from the Structures palette are shown in Figure 1.23. This palette contains the Sequence, Case, For loop, and While loop structures. You will also find the Formula Node, Global Variable, and Local Variable on this palette.

1.6.1.1 Sequence Structure

Place the 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 1.24 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 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 1.25 will help to define some terms that are related to the Sequence structure. The top window shows Frame 0, while 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.
Sequence structures can be avoided in most applications. The main problem with 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.

Sequence structures do not have equivalents to other programming languages; this is a unique structure to dataflow languages. Text-based languages such as Visual Basic and C perform operations line-by-line; LabVIEW executes things as they become available.

**1.6.1.2 Case Structure**

The Case structure is the second object on the palette and 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 1.26 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.

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. 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. This capability was first added to LabVIEW in Version 5.0. 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.

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 1.27 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 Case structure.

1.6.1.3 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 1.28. 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 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. Since 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 from a single value. Indexing can be enabled in a similar manner if you are using a While loop.

Figure 1.29 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 shorter length of the 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.

A 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 1.30 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 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.

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.

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. Since 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.

### 1.6.1.4 While Loop

The While loop is an iteration construct that executes until a false 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 1.31. The While loop will execute at least once because the condition is evaluated at the end of the current iteration. If 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 1.31** 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 a
false 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.

1.6.1.5 Formula Node

The Formula Node is the first item in the second row of the Structures palette. 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. Popup 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 1.32. 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 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.

1.6.2 Numeric, Boolean, String, and Comparison

The Numeric, Boolean, String, and Comparison palettes are displayed in Figure 1.33. The functions shown in the Numeric palette are straightforward and simple to use. The example in Figure 1.30, 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, Trigonometric, Logarithmic, Complex, 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.

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, 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.
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 1.34 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. Two subpalettes hold additional functions that perform conversion from strings to numbers, byte arrays, and file paths.

1.6.3 Array and Cluster

Both Array and Cluster palettes are displayed in Figure 1.35. 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 1.36 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 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 you should become very familiar with 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 to get into. 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 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.
1.6.4 **TIME & DIALOG**

The Time & Dialog palette, displayed in **Figure 1.35**, contains error handler VIs in addition to functions for retrieving the system time, wait functions for introducing delays, and functions for displaying dialog boxes. Chapter 6 covers the topic on exception handling and describes the error handler VIs in more detail. 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. The functions on this palette are simple to use and are self-explanatory.

A few comments regarding time in LabVIEW should be mentioned. When using the *Tick Count* or *Wait Until Next Multiple* functions, it is possible for the counter to overflow. Windows running on Intel hardware will have a 32-bit counter that is keeping the tick count. This counter will overflow about once every 50 days. For most home users, 50 days is an extremely long time to keep a computer running. In office and lab environments, this is a bit different, machines do not get rebooted every day because you leave them on. Lab machines may be left running for a couple of months at a time, and the clock may roll over at an inopportune time. Computers used in assembly line operations are only rebooted when the line is not assembling products, and for some assembly lines this happens “once every blue moon.” When working with tick counts and multiples, be sure to examine the timestamp returned to verify rollovers do not happen and that you are prepared to compensate for the changing of the number.

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 \( \pi \times 10^7 \) seconds in a year). The concern with system dates and times is 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.

Dialog boxes are great for informing users that something is happening in the system. Dialog boxes need to be avoided in automated applications, however. 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.

1.6.5 **FILE I/O**

**Figure 1.37** 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 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. Support for writing
single-precision and 16-bit numbers is available through the binary VI subpalette. One trick programmers may want to use when storing 32-bit numbers into a binary file is to take each element in the array, split it into a pair of 16-bit numbers, and rebuild the array. The split numbers can easily be written to the binary file. The array length will be twice as long, obviously, but storage space will not change because the storage size of each element is half the size. Your new 16-bit array can be written to the binary file with ease. Reading back the data works basically the same, but instead of the Split Number function, you will use the Join Number function.

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 just 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 do not open or close the files on their own; 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 only be read 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.
Figure 1.38 illustrates a simple block diagram in which a string is written to a file. A file constant, available in the File Constants subpalette, containing the file path information is wired to the *Write Characters to File* function. The error information string is written to an error log, which is simply a text file. The true constant wired to the function causes the information in the string to be appended to the end of the file. If this is changed to false, the previous file can be overwritten with the new string. If you write to a file with this function, remember to format the string in the manner you would like it to appear in the file.

1.6.6 **Instrument I/O, Data Acquisition, and Communication**

The Instrument I/O, Data Acquisition, and Communication palettes all contain various built-in functions to simplify communication with external devices. The three palettes are displayed in Figure 1.39 representing how they appear on a Windows system. The Instrument I/O palette holds VISA, GPIB, Serial, and VXI-related functions. The Communication palette contains functions for ActiveX, DDE, TCP, UDP, Data Socket, and HiQ (Apple Events and PPC replace ActiveX and DDE on the Macintosh). The specific functions in these palettes will not be discussed in this book; however, Chapters 7 and 8 cover ActiveX in detail, which is a relatively new addition to LabVIEW.

When designing an application, there may be a few minor details you should consider for communications. Interapplication communications do not involve cables such as GPIB. For Windows applications, DDE should not be used as a communications protocol. DDE is considered obsolete and only exists for legacy application support. Windows-specific communications can be done with ActiveX/COM functionality. ActiveX is the current Windows standard 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. Macintosh’s PPC and Windows’ DDE are both out-of-date protocols, and neither were particularly easy to use.

GPIB, serial, and VXI communications should be performed with the VISA library. VISA is the future standard for instrument communications in LabVIEW. The IEEE 488 and serial interfaces 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.

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.

1.6.7 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 1.40 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 1.40 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, while 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.

1.6.8 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 1.41 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.

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

Since 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.

1.6.9 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 1.22, 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.

1.6.10 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 1.43, with the Execution Options selected in the drop down box at the top.

The first four checkboxes shown are options for configuring subVIs and are referred to as the “subVI node setup.” These boxes are normally unchecked as default values. In the figure shown, the subVI has been configured to show the front panel when it is called, and to close the panel after it has finished executing. The checkboxes on the right are used to set printing options. The execution options also include allowing reentrant execution, setting VI priority, 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 1.44 displays the VI setup window with Window Options selected in the drop-down menu. These configuration selections allow you to customize the appearance of the VI during execution. In the example shown, Show Scroll Bars, Show Menu Bars, and Show Toolbar have been deselected. These are all enabled by default. Same as VI Name has also been deselected and the Window Title modified. These alterations cause the VI to appear as shown in Figure 1.45 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 1.46 displays the VI Setup Documentation window. LabVIEW provides some built-in documentation support that can be configured through either VI Setup or Preferences. 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. *Use History Defaults* from Preferences has been deselected in the example shown. This informs LabVIEW to use the settings from the VI setup instead of the Preferences. The preference 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 1.47 illustrates the VI history for the VI shown earlier in Figure 1.45. The VI history can be viewed by selecting Show History under the Windows pull-down menu. Chapter 4, Application Structure, discusses the importance of documentation and reveals other documentation methods for LabVIEW applications.

1.6.11 Hierarchical Nature

This section describes how VIs, once developed, can be used as subVIs in larger applications. This creates 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 1.48 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 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.

1.7 SETTING PREFERENCES

This section describes some of LabVIEW’s preferences that can be configured to suit a programmer’s needs. The Preferences selection is available in the Edit pull-down menu. The window that appears is shown in Figure 1.49 along with its default settings. The preferences shown correspond to the Paths selection from the top drop-down menu. Some of the preferences 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.

FIGURE 1.49
1.7.1 Paths

The Paths configurations, shown in Figure 1.49, dictate the directories in which LabVIEW will search when opening or saving libraries, VIs, menus, and other files. The second drop-down menu selector allows you to configure the Library, Temporary, Default, and Menus directories. The last selection in this menu is used to set the VI 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. 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.

---

**TABLE 1.3 Preferences**

<table>
<thead>
<tr>
<th>Preference Selection</th>
<th>Function/Utility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paths</td>
<td>Configure search directories for opening/saving VIs.</td>
</tr>
<tr>
<td>Performance and Disk</td>
<td>Configure to use multithreading and perform check for available disk space prior to launch.</td>
</tr>
<tr>
<td>Front Panel</td>
<td>Settings for front panel editing.</td>
</tr>
<tr>
<td>Block Diagram</td>
<td>Settings for block diagram programming.</td>
</tr>
<tr>
<td>Debugging</td>
<td>Options that are used for debugging VIs, and execution highlighting during execution.</td>
</tr>
<tr>
<td>Colors</td>
<td>Change default colors used by LabVIEW for front panel, block diagram, etc.</td>
</tr>
<tr>
<td>Fonts</td>
<td>Settings for Applications, System, and Dialog Font styles.</td>
</tr>
<tr>
<td>Printing</td>
<td>Configure print settings.</td>
</tr>
<tr>
<td>History</td>
<td>Options for recording revision comments when changes are made to VIs.</td>
</tr>
<tr>
<td>Time and Date</td>
<td>Configure both time and date formats to be used by LabVIEW.</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>Options for tip-strips, native file dialogs, drop-through clicks, hot menus, autoconstant labels, opening VIs in run mode, and skipping navigation dialog at launch.</td>
</tr>
<tr>
<td>VI Server: Configuration</td>
<td>Configure protocols, port numbers, and server resources.</td>
</tr>
<tr>
<td>VI Server: TCP/IP Access</td>
<td>Set access privileges to specific list of clients for VI Server.</td>
</tr>
<tr>
<td>VI Server: Exported VIs</td>
<td>Specify list of VIs that are accessible to clients using VI Server.</td>
</tr>
<tr>
<td>Web Server: Configuration</td>
<td>Enable Web server, configure root directory, set port number and timeout.</td>
</tr>
<tr>
<td>Web Server: Browser Access</td>
<td>Set access privileges to specific list of clients for Web server.</td>
</tr>
<tr>
<td>Web Server: Visible VIs</td>
<td>Specify list of VIs that are accessible to clients from Web server.</td>
</tr>
</tbody>
</table>
1.7.2 Block Diagram

Figure 1.50 displays the Block Diagram preferences window. These options are intended to help you develop code on the block diagram. For the beginning user of LabVIEW, 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.

The last checkbox in this window allows you to configure the maximum number of undo steps per VI. Undo is a LabVIEW enhancement that was introduced in Version 5.0. Undo and Redo are both available in the Edit pull-down menu. When the 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.

1.7.3 History

The History preferences window is displayed in Figure 1.51. Some of these options are duplicated in the VI History settings under VI Setup, as described earlier in Section 1.7.4. If you compare this to Figure 1.46, you will notice that the first four checkboxes are the same. If you have the Use History Defaults box checked in the VI Setup window settings, LabVIEW will use the History preferences.

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 1.51 shows the User Login information. The login name can be modified in this window and is accessed by selecting User Name from the Edit 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 many experiences with troubleshooting 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.

1.7.4 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 7 describes the VI Server in more detail along with the related configurations and some examples.

The Web Server is also an addition to LabVIEW 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.
1.7.5 Palettes

LabVIEW normally displays the default palettes for both Controls (Figure 1.8) and Functions (Figure 1.22). 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 Edit menu. The Select Palette Set submenu allows you to select from the following list: basic, data acquisition (daq_view), default, and test and measurement (t&m_view). This can be further modified to show the standard icons and text, all icons, or all text using the Display Style submenu.

Select Edit Control & Functions Palettes to create and customize a new palette set. A window similar to the one shown in Figure 1.52 will appear that will allow you to perform this. Then select New Setup from the drop-down menu box and enter a name for the new view. A view called “Personalized” was created for the example in Figure 1.52. 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 1.52 to the default palette in Figure 1.22, you will notice the changes that were made. Some palettes were deleted while others were moved to new locations. A VI (Data Logging.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.

**BIBLIOGRAPHY**

*LabVIEW On-line Reference*, National Instruments