

Learning LabVIEW for the First Time

1

1.1 INTRODUCTION

This chapter is written for version 6i of LabVIEW. If you do not have LabVIEW, and if the evaluation version of LabVIEW is not still with this book, you can get the free evaluation version (good for 30 days; programs cannot run for more than five minutes) or the real thing from National Instruments (www.ni.com or 1-800-IEEE-488).

If you know LabVIEW already and want to dive into DAQ now, skip this chapter except for Section 1.6, and go directly to Chapter 2 (for an overview of non-LabVIEW-related DAQ fundamentals) or Chapter 3, where working through Section 3.1.5 will hopefully get you a LabVIEW-DAQ connection.

I've intentionally made this chapter the most densely compacted, interactive introduction of LabVIEW possible—it will be like taking a drink from a fire hose. So unless you're already proficient in LabVIEW (and a speed-reader), don't expect to make it through this chapter in one day! Although compact, this chapter will introduce every aspect of LabVIEW needed to write the majority of useful DAQ programs.

If you have never programmed in any language before, you will not only need to learn LabVIEW, but you will need to learn basic programming concepts as well. First, become familiar with the information in Appendix A—preferably with an experienced software nerd around to explain any difficult concepts. (Note: Do not refer to your nerd as such when asking for help—try “professional.”) If you have some programming experience but are new to LabVIEW, carry on with this chapter. Even if you are quite familiar with LabVIEW, at least skim this chapter to make sure you understand the concepts herein. But if this chapter seems too difficult, try the well-written *LabVIEW for Everyone*, by Lisa K. Wells and Jeffrey Travis, which serves a slower-paced, more thorough LabVIEW tutorial “for everyone.” Whether Lisa bribed me to say this will never be publicly known.

During reviews, we considered omitting the next paragraph; however, I intend to tell things the way they are, and that means pointing out the negative along with the positive. Based upon the title, you may expect me to imply throughout this book that LabVIEW is *always* better than other languages for DAQ.



I have built DAQ applications in every style I can think of (three graphical languages, numerous text-based languages, assembly language on multiple computer platforms, microcontrollers in text *and* assembly, hardware-only, and more)—and many times over in each style. *Yet LabVIEW is usually my first choice for most DAQ jobs.* That’s why I feel it worthwhile to write this book. But read on for alternatives. The truth is this: If you are already very good with C, or if your application is going to be extremely complex, you may be better off using NI’s LabWindows/CVI for DAQ applications instead of LabVIEW. If you are good with Microsoft Visual Basic or Microsoft Visual C++, NI’s Measurement Studio has excellent DAQ tools for you to use instead of LabVIEW. Or you may be able to get away without buying any NI software if you have a very simple application, by interfacing directly to NI-DAQ from another programming environment. See Chapter 11 for details on these options. Also, unless you follow the guidelines in Chapter 6, Section 6.1.1, LabVIEW can be tough to handle when your program gets very large or complex. Although DAQ is the focus of this book as a whole, it is not the focus of this chapter. This chapter is for LabVIEW in general, yet is subtly optimized to give you the necessary prerequisites for most DAQ applications.

I highly recommend that you work through the examples in this book with LabVIEW and a real DAQ device. If you don’t have a real DAQ device, do what you can in LabVIEW. Like a math book, it is important that you read

everything *sequentially* in this chapter and that you understand this chapter before proceeding to the rest of the book. Unlike a math book, this book is designed to make sense to most people. Check that—most people with any engineering or programming experience. In order to keep this chapter as short as possible, yet thorough enough to get you going with DAQ, each important topic is covered only once as the tutorial progresses, and any given topic may rely upon previous work; hence the need for sequential reading.

You should already know how to operate a computer on a very fundamental level. You should be familiar with

1. files and folders (folders are also called directories);
2. how to move, copy, and delete files and folders;
3. floppy disks and hard drives;
4. how to operate a text editor (like Microsoft Word, WordPad, or Notepad);
5. memory (also called RAM); and
6. the term “*operating system*.”

Throughout this book, you will occasionally be instructed to *pop up* on an object. To pop up means to move your cursor so that it is over that object, and right-click (click with the right mouse button, not the left). Only one point on every cursor, called the *hot point*, actually “works” when you click; you may need to experiment to find exactly where the hot point is. An object on which you can pop up is said to have a *pop-up* menu. Finally, you will often need the Positioning tool  or the Operating tool  in order to get the proper pop-up menu—these tools will be introduced in Section 1.2.2.

As you’re working through this book, you may close all VIs after each section, or your screen will become quite cluttered.

1.2 VI BASICS

“What is a VI?,” you might ask. VI (pronounced vee-eye) means Virtual Instrument, and that is what LabVIEW was originally geared for—making software versions of instruments that you might find in a laboratory. In a more general context, a VI is any program module written in LabVIEW.

1.2.1 The Front Panel, the Block Diagram, and Saving Your VIs

LabVIEW is a graphical programming language in which the programs are written using pictures, not words. Instead of having text, like

```
sumOfSquares = a * a + b * b;
```

LabVIEW instead has a graphical representation of its operations, as in the VI shown in Figure 1-1, saved as `Sum of Squares.vi`.

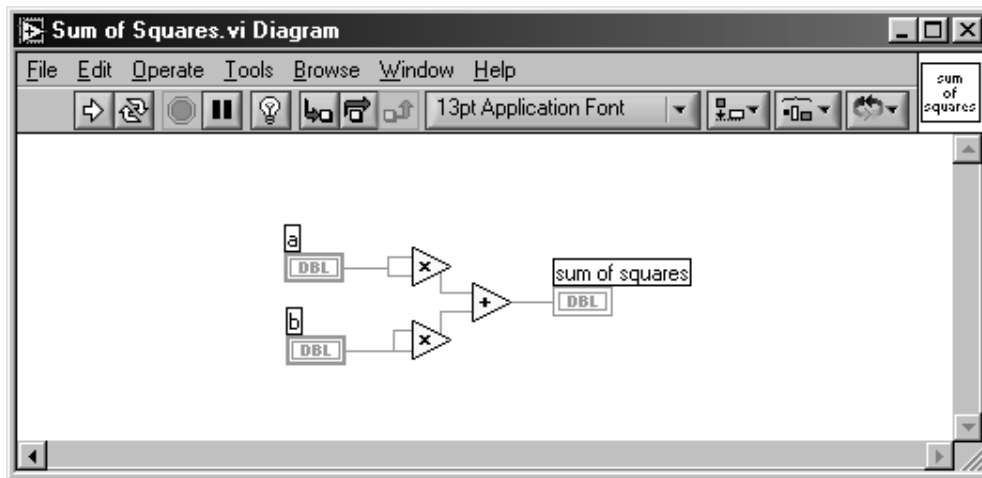


Figure 1-1

LabVIEW code (a block diagram) does not look like the usual text code found in most programming languages.

A VI consists of three major parts: (1) a front panel, (2) a block diagram, and (3) an icon/connector (consisting of an icon and a connector pane). We will discuss the first two of these three parts in this section.

The window in Figure 1-1 is called a block diagram, and it is always associated with a front panel, which is another window. Every front panel has at most one block diagram, and every block diagram has exactly one front panel. The front panel is what the user sees (sometimes called a GUI, or graphical user interface), and the block diagram is the *code*, or the heart of the program. A reasonable front panel for the block diagram in Figure 1-1 would be the example shown in Figure 1-2.

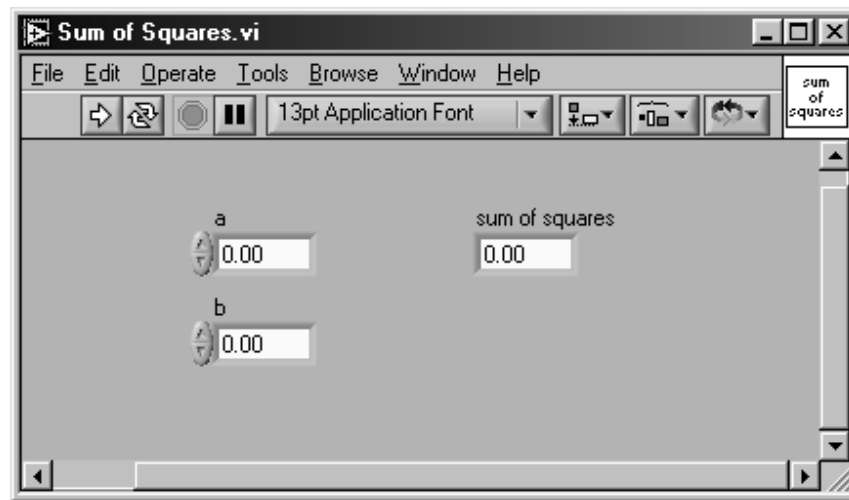



Figure 1-2
Example of a LabVIEW front panel.



The front panel is gray by default, and the block diagram is white by default. You may find it useful to keep them colored this way so you can instantly distinguish front panels from block diagrams during development. If you must change their colors, at least keep the block diagrams white and the front panels whatever subtle color you like; this helps both you and your users.

Initially, one of the most confusing parts of LabVIEW is the relationship between the front panel and the block diagram. Referring to Figures 1-1 and 1-2, you can see how the three rectangles on the block diagram have a correlation to those on the front panel, since their labels are the same. On the front panel, a user can type any number into the boxes labeled **a** or **b**. The sum of the squares of these numbers is calculated and shown on the front panel in the **sum of squares** box whenever the **Run** button  is pushed (the **Run** button can be found near the upper left of the front panel).

We will now use LabVIEW to create an empty front panel and block diagram (if you don't have LabVIEW, the evaluation version included with this book, or available free from NI, can get you started). Later in this lengthy chapter, we will build the VI shown in Figure 1-1.

First, create a folder on your hard drive where you should store all VIs produced in this book. Since the upcoming chapters will use VIs from previ-

ous chapters, this will make more sense than creating a separate folder per chapter. In this book, this folder will be C:\Bruce\Projects\LV DAQ Book\VIS.

In Windows, find the **Start** button in one of the corners of your screen, typically the lower left corner. Select **Start»Programs»National Instruments LabVIEW 6i** from the Windows task bar (this menu path may vary on your computer). If LabVIEW asks you to log in and you don't want to log in every time LabVIEW launches, see Appendix E, item 15. If LabVIEW was not running, you might see an introductory LabVIEW window asking if you want to create a new VI or open an existing one; if you see this window, create a new VI; otherwise, one will have been created for you. You will see, as shown in Figure 1-3, a new front panel come up, called **Untitled 1** (or, if LabVIEW was already running, the untitled window will be numbered according to the number of times an untitled window has been created—Untitled 2, Untitled 3, and so on).

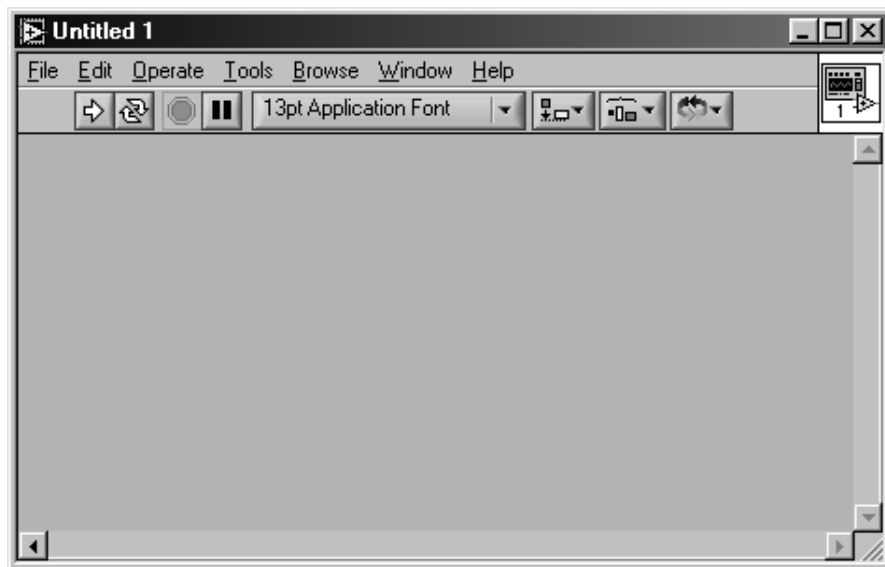


Figure 1-3
A VI's front panel, newly created.

This is a blank front panel upon which you will later add the boxes from *Sum of Squares.vi* seen in Figure 1-2. Instructions for adding these boxes will be in Section 1.2.4, but first, a few more LabVIEW basics will be

introduced. To see the block diagram, which is currently hidden, select the **Window»Show Diagram** menu item. Alternatively, <Ctrl-E> is a keyboard shortcut for switching between front panel and block diagram views.

At this point, save the VI: Select the **File»Save** menu item, and save it as `Sum of Squares.vi` in the folder you created earlier. Quit LabVIEW now with the **File»Exit** menu item. Launch LabVIEW again, and this time, use the **File»Open...** menu item to open this same VI. You now know how to save a VI and open it again later. You could have also double-clicked the VI from your graphical operating system, because the `.vi` extension on the file name tells your operating system to use LabVIEW to open the VI. Another useful LabVIEW feature is the **File»Recently Opened Files** menu item.


Suppose you want to save all of your LabVIEW work into a single file, which might help you transfer your work from computer to computer. An LLB is a LabVIEW LiBrary, which is a special LabVIEW file (ending with `.llb`) in which you can save your VIs into a single file as if it were a folder from LabVIEW's point of view. I prefer to not use LLBs in general, but to simply place all my VIs in a folder. When I want to save my VIs later as a single file, I use a compression utility like WinZip (available at www.winzip.com).

1.2.2 Controls, Indicators, and Tools

In Figure 1-2, the boxes on the front panel are called *controls* or *indicators*, depending on whether the data moves into or out of them. The boxes labeled `a` and `b` are controls, and the box labeled `sum of squares` is an indicator.



From the point of view of the block diagram, controls are sources of data, and indicators are destinations of data. If this is not clear, go back to your `Sum of Squares.vi` block diagram, and see that `a` and `b` are sources of data, meaning that data comes out of them and goes elsewhere in the block diagram. Similarly, indicators on the block diagram, such as `sum of squares`, are destinations of data, meaning that data goes into them. From the point of view of the front panel, this logic is reversed: Controls are destinations of data (the source is the user or possibly another VI, as we'll see later) and indicators are sources of data (the destination is the user or another VI).

When the **Run** button  is pushed, *dataflow* begins. Dataflow occurs on the block diagram, not the front panel. In dataflow terms, a *node* is a “stopping point” for data on a dataflow diagram. In LabVIEW, wires connect nodes on a block diagram. A dataflow diagram consists of nodes, wires, and

data packets. In a block diagram (such as the one in Figure 1–5), the nodes might look like those in Figure 1–4.



Figure 1–4
Examples of nodes on the block diagram.

The wires are the lines connecting these nodes, and the data packets are the two little circles on the wires.

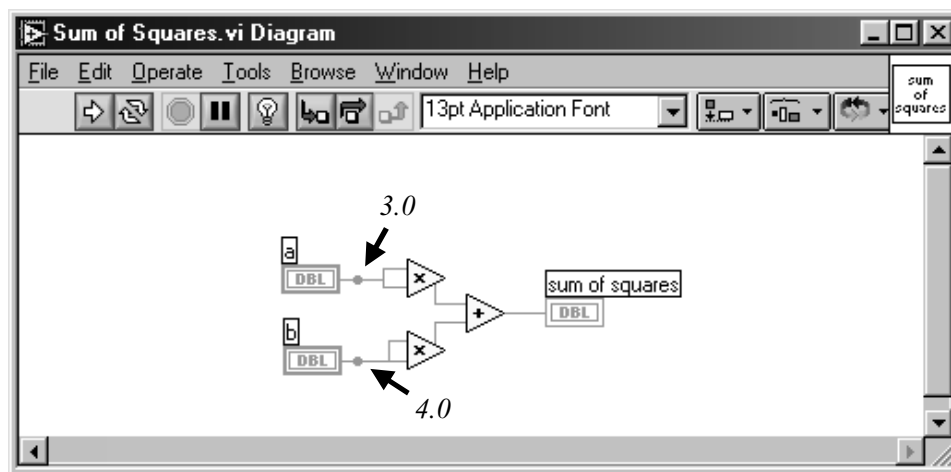


Figure 1–5
Data packets, drawn as little circles on a block diagram.



I used a drawing program to draw the data packets on the wires and the numbers with their lines on the block diagram in Figure 1–5; none of these are really a visible part of LabVIEW.

Data packets are usually invisible, but when writing your LabVIEW program, it is helpful to imagine them flowing from controls to indicators, which in Figure 1–5 would be from left to right. If they were *really* visible whenever a LabVIEW program ran, it would slow the programs down to a

crawl. Later, we'll see a debugging tool called *execution highlighting* that allows you to see data packets similar to the ones shown above. The rules of dataflow are follows:

1. Data packets can flow in only one direction on any given wire.
2. Data must flow from one node to another.
3. A node can release data to its output wires only after it has received data on all of its input wires.

In LabVIEW, a *terminal* is an area to which a wire may be connected. The nodes shown in Figure 1–6 each have one or three terminals. In this case, all the nodes' terminals are wired, but some terminals on some nodes may remain unwired. Later, we will see nodes with many more terminals. Since all front panel objects have nodes with only one terminal, their block diagram connection points are usually called terminals rather than nodes, though either name is technically correct.

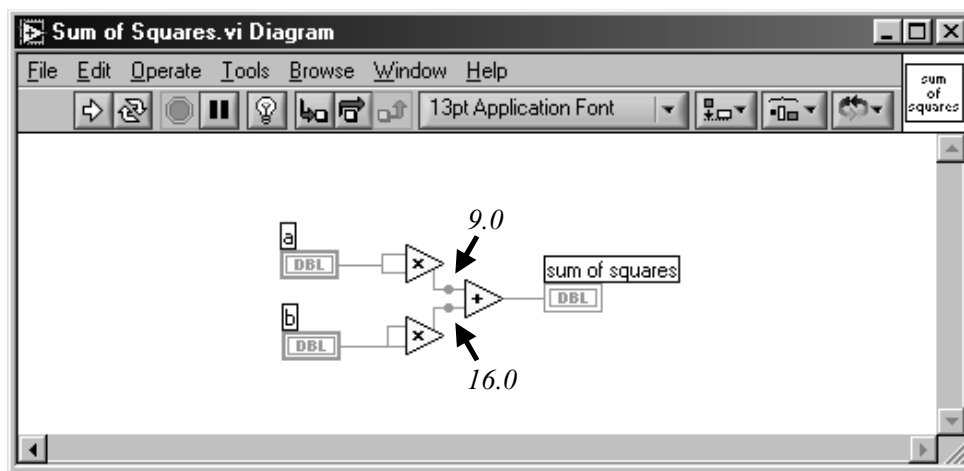




Figure 1–6

The same data packets as in Figure 1–5, a moment later.

Imagine the data packets (the little circles I've drawn) shown in Figure 1–5 coming out of the a and b terminals and moving to the right along the orange lines (orange on your screen, not in your book), called *wires*. These circles move towards the two Multiply functions . Just before the circles







reach the Multiply functions, they split in two where the wires fork, so that the top Multiply function produces a $3 \times 3 = 9$ on the orange wire to its right. Similarly, the bottom Multiply function produces a $4 \times 4 = 16$ on the orange wire to its right, as shown in Figure 1-6.

Finally, when all data packets have hit the Add function , it sums the 9 and 16 to produce 25, which is then sent to the sum of squares node, thus showing up on the front panel. As users running the final program, we usually never see the block diagram in action; we see only front panels.

In this example, our data type is strictly numbers. Later, we will see other data types.

On the front panel and block diagram, the mouse cursor can serve several different functions. These basic tools are summarized in Table 1.1. To toggle between these different modes, use the <Tab> key.

Table 1.1 Basic VI Editing Tools

Icon	Tool	Purpose
	Operating	Changes the data value of a control on the front panel or of a constant on the block diagram.
	Positioning	Changes the position, size, or shape of an object.
	Labeling	Allows you to edit any editable text or to create a "free label."
	Coloring	Sets the color of an object to the "current color."
	Color Copying	Reads the current color of an object as the "current color."
	Wiring	Front panel: Used only in the connector pane (to be described later). Block diagram: Wires nodes together via their terminals.

1.2.3 Data Types

Make sure you're familiar with the concepts in Appendix A, if you're not already.

In the last section, we saw little data packets flowing along wires. As shown there, they had the data type of floating-point (numeric) data, which can have a fractional part, such as 2.5, 0.0000323, or 3.14159. For now, let's discuss the three most basic data types in LabVIEW: (1) numeric, (2) Boolean, and (3) string.

Numeric data types represent numbers, as the name suggests, and they can come in many flavors. First, we will describe all numeric subcategories and their limitations, then we will show you how to create and use them in LabVIEW.

Starting with the three major numeric categories, Table 1.2 lists the numeric data types. Table 1.3 lists the different types of integers.

How is this relevant to your LabVIEW/DAQ programming? Suppose you have a DAQ board and are acquiring analog data. At the hardware level, your data comes in as an integer, then is usually transparently converted to a floating-point number for you. Depending on the precision of your DAQ device, you might need a certain range of integers to be able to distinguish all the different values that could come from the board. This will be discussed in more detail in the "Resolution" section of Chapter 2, Section 2.2.1, where 12-bit and 16-bit integers are discussed. Appendix A offers even more fundamental information about numeric types.

Be careful when changing from one data type to the other on the block diagram, and realize that numbers could be changed without your knowledge, unless you notice the gray circle. Generally, a little gray splotch will appear, warning you of such a change in data type, as seen on the left side of the i16 indicator in Figure 1-7.

Table 1.2 Numeric Data Types

Major Numeric Categories	Description
Integer	These numbers cannot have a fractional part; examples of integers are -2, 0, or 123.
Floating-Point	These numbers may have a fractional part, like 0.5, -12.345, or 0.0000001, although a fractional part is not required.
Complex	These numbers have both a real and an imaginary part, and are composed of two floating-point numbers. For most DAQ work, it is not necessary to use these or even to understand them. Only under certain circumstances involving arrays of numbers, usually for frequency analysis, are complex numbers useful, so they will not be discussed further in this book.

Table 1.3 Types of Integers

Integer Type	LabVIEW Abbreviation	Range
Signed 32-bit	I32	-2,147,483,648 to 2,147,483,647
Signed 16-bit	I16	-32,768 to 32,767
Signed 8-bit	I8	-128 to 127
Unsigned 32-bit	U32	0 to 4,294,967,295
Unsigned 16-bit	U16	0 to 65,535
Unsigned 8-bit	U8	0 to 255

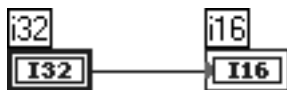


Figure 1-7

This dangerous conversion will give you an incorrect number if *i32* is not in the range of -32,768 to 32,767.

As a rule, use wires having data types large enough to accommodate any size number that could possibly travel along them.

Floating-point numbers also have precision issues, but unlike integers, we often need not concern ourselves with their precision in the field of DAQ when dealing with data. Here's one big exception—when dealing with times and dates in LabVIEW's standard Time & Date format, we will need to use double precision floating-point numbers (DBL, eight bytes) rather than single precision floating-point numbers (SGL, four bytes). In this case, the SGL data type does not have enough significant figures, or digits of precision, to accurately represent time values. The SGL data type is usually appropriate for analog DAQ data. The other type of floating-point data is called extended precision floating-point (EXT, platform-dependent number of bytes). The EXT data type is generally not used in DAQ unless you're doing some sort of math that requires extreme floating-point precision, such as certain types of iterative numeric analysis. See the end of Appendix A for more floating-point details.

In LabVIEW, these different numeric data types are called the number's *representation*. You can pop up (by right-clicking) on a numeric control, indicator, or constant and change its representation with its **Representation** menu item.

The other two basic data types in LabVIEW are Boolean and string. A Boolean data type can have a value of either True or False (just one bit—not even a byte). A string data type contains an arbitrary number of bytes, which can be considered 8-bit integers. Often, strings contain human-readable text, wherein each byte corresponds to a letter, number, symbol character, or “formatting character” like a space, tab, or return character.

Two more data types should be mentioned at this point—the array and the cluster. These are both ways of grouping other LabVIEW data types (including themselves!) so that many pieces of data can conceptually flow along a wire in a single data packet. For example, you could have a group of 100 numbers in an array and visualize it moving along a wire as one data packet. Or you could have two numbers, a Boolean, and three strings grouped as a cluster, also moving on a wire as one data packet. Almost any combination of data can be grouped by using the array and cluster data types to flow along a LabVIEW wire. These will be discussed in more detail later.



Wire colors change depending on your data type. I'm afraid this black & white book won't show you many colors, unless you spill something on it, but observe the various wire colors as you build your block diagrams.

Table 1.4 provides a quick summary of what the colors usually mean on the block diagram for wires and terminals.

Table 1.4 *Data Type Colors on a Block Diagram*

Color	Data Type
Blue	Integer (Numeric)
Orange	Floating-point (Numeric)
Magenta (hot pink)	Cluster containing any non-numeric data type (Booleans, other clusters, arrays, strings, etc.)
Green	Boolean
Brown	Cluster containing only numerics

As of version 6i, LabVIEW introduced a *new data type* called the Waveform (see Figure 1–8), which I capitalize in this book to distinguish it from a generic waveform, which is a generic sequence of analog data points. LabVIEW's Waveform is like a cluster containing three parts, described in Table 1.5.

Table 1.5 *LabVIEW's Waveform*

Waveform component	Description
t0	This is the time corresponding to the initial point in the Waveform's data, Y. Its data type is DBL and is defined as a time, just like the output of LabVIEW's Get Date/Time In Seconds function.
dt	If the Waveform's data, Y, has more than one data point, these points are equally spaced in time by an interval of dt seconds.
Y	This is the Waveform's data, which is also described by the previous two parameters. Y may be an array or a single number.

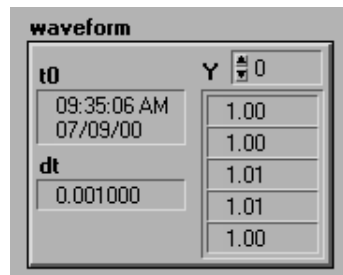


Figure 1-8
A front panel Waveform object with sample data.

Although cluster-like, the Waveform data type cannot be manipulated by normal Cluster functions—it has its own special set of functions.

For those DAQ VIs designed to handle waveforms of just one point, there's another version of the Waveform data type called the *single-point* Waveform, which is just like the normal Waveform, but certain components are hidden (see Figure 1-9).

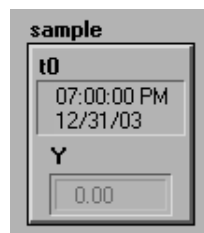






Figure 1-9
The single-point Waveform.




1.2.4 Creating Your First Functional VI

We will now build our first functional VI. Use LabVIEW to open `Sum of Squares.vi`, which you should have created in an earlier section. If you see one or two smaller windows opening up with your front panel containing tools and controls, close them. This book will not use these palettes as ever-present windows (many users find them annoying in a cluttering sort of way), but if you want to, they can be turned on and off in the **Windows**

menu. Rather than using the **Tools** palette, *close it*, then select your tools by hitting the <Tab> key on your keyboard.



Position your cursor (with the mouse) anywhere over the front panel and verify that you scroll through the following tools when you hit the <Tab> key: , , , .



If you are able to continuously <Tab> through these four tools, you are in edit mode and ready to edit. If you ever find yourself tabbing through these tools , ,  on the block diagram, this means you are in run mode. To get out of run mode, first make sure your VI is unlocked by hitting <Ctrl-L> and adjusting the **Security** item under the **Category** menu's window to "unlocked"—otherwise you will be stuck in run mode.

Secondly, make sure you are in edit mode, as opposed to run mode, by checking whether an item in the **Operate** menu says **Change to Edit Mode** or **Change to Run Mode**. <Ctrl-M> toggles between these two modes, unless your VI is "locked."

Suppose you try to run the VI, and the block diagram stops and looks something like Figure 1-10 with a red border surrounding the block diagram.

This means you are in run mode, and when you had this particular tool , you clicked the block diagram. Unless you're familiar with this debugging tool and you did this on purpose, get this tool again (which now may look like this ) by getting to run mode (if not there already), clicking the block diagram so the red rectangle disappears, getting back to edit mode, then proceeding as normal. Yes, this can be confusing.

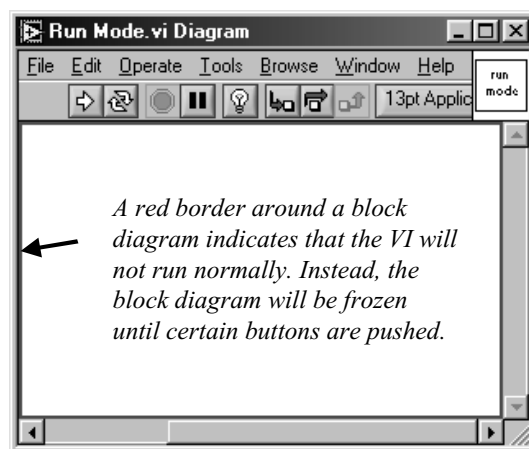




Figure 1-10

A red border around the diagram indicates run mode.

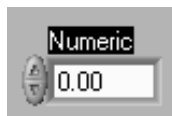
Be careful to click only when told to in the next paragraph. If you make a mistake, you can undo it with the **Edit»Undo...** menu item or <Ctrl-Z>. The Undo feature is very well implemented in LabVIEW and should be useful quite frequently, unless you never make mistakes, like me. See Appendix E for more information on Undo.

Refer to the `Sum of Squares.vi` front panel shown in Figure 1-2 while building this front panel. To create your first control, select any tool other than the Coloring tool  (hit the <Tab> key until you see this other tool) and pop up on (right-click) the front panel, thus bringing up a temporary **Controls** palette. This is the way we will always access the **Controls** palette in this book, so if you already had the “permanent” **Controls** palette window (a little window with the word **Controls** in its title bar) in the background, close it now. If you tried to pop up with the Coloring tool , you would not have gotten the **Controls** palette. With the **Controls** palette still showing from your recent right-click, move your cursor in the **Controls** palette and over the upper left icon (**Numeric**). The **Numeric** subpalette will pop up. Then, move the cursor over the upper left icon in the new subpalette (**Digital Control**). See Figure 1-11.



*In this book, I am keeping the **Controls** palette window closed so that you must pop up on the front panel to see it. This is a personal preference, as I like uncluttered computer screens. To open it once it's closed, use the **Window»Show Controls Palette** menu item. If you want to skip ahead and get a quick description of the many items in the **Controls** palette, see Figure 1-29 and Table 1.6 (some of this won't make sense until you get there). The **Controls** palette will not open and close per VI—it stays open or closed for LabVIEW in general.*

When you click this, your next click on the front panel will create a new Digital Control (a box with a number you can edit) wherever you click. This sort of action will be hereafter succinctly described as “dropping the Digital Control from the **Controls»Numeric** palette” (when you see **Controls»** preceding the name of a palette or other object, this means to start with the **Controls** palette, found by popping up on the front panel). Click slightly towards the upper left of your front panel, thus creating your first control:



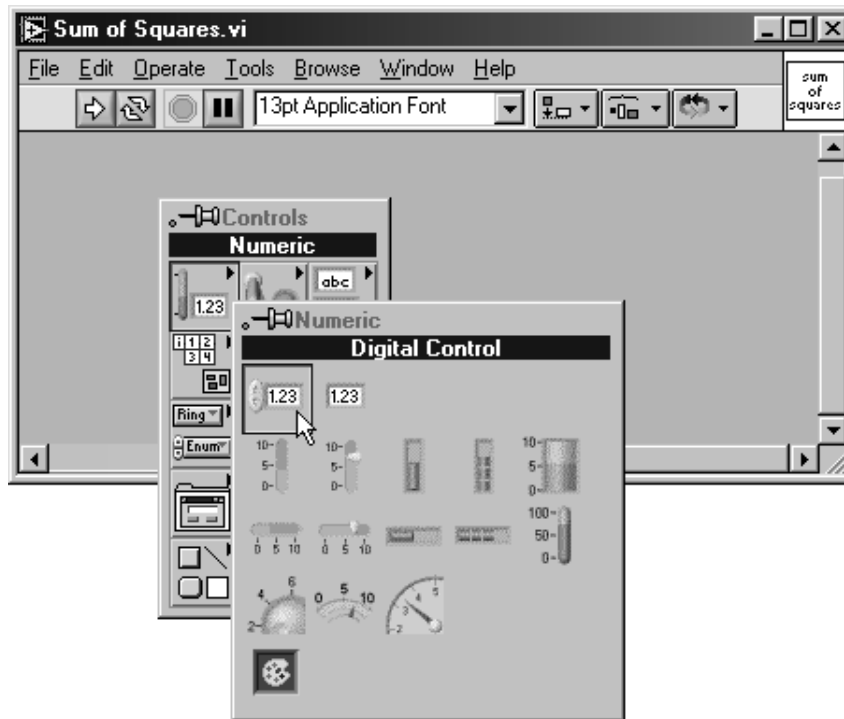
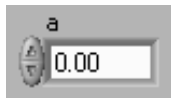



Figure 1-11


The Sum of Squares.vi front panel with the Numeric and Digital Control subpalettes—you will not create the icon (the white box saying “sum of squares”) until later.

Before clicking again, while the label is still highlighted, type a lowercase a, and your new control has a label:



If you were to click elsewhere before you began typing the label, the label would lose its highlighting, so you would need to use the Labeling tool  to edit the label.





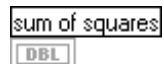


If any control or other front panel object doesn't show up quite where you want, you can use the Positioning tool  to move it around.


Remember how to use this **Controls** palette, as it's the most common way to drop new objects on the front panel. It will be discussed in detail in a later section.


Create a similar control just below this one, and label it b.

Finally, create another digital control to the right, and label it `sum of squares`. Pop up on it and change it to an indicator. A shortcut here would have been to select the Digital Indicator rather than the Digital Control in the **Controls»Numeric** palette.

Notice the difference between a numeric control and indicator on the front panel. Go to the block diagram, and you should see three orange rectangles, which are called terminals, each of which corresponds to a control or indicator on the front panel.

First, notice that the controls  and  have thicker borders than indicators like ; this is common with many node types. Pop up on any terminal (terminals are on the block diagram) and select its **Find Control** or **Find Indicator** menu item. Notice how this helps you find terminals' front panel objects! The opposite sort of mechanism works from these front panel objects with their **Find Terminal** menu items. From the block diagram, you can also double-click terminals (with the Positioning tool  or Operating tool ) to do the same thing.



Next, use the Positioning tool  on the block diagram to arrange the terminals, with a and b on the left and `sum of squares` on the right, then add the Multiply and Add functions as shown previously in the block diagram of Figure 1-1. To do this, pop up anywhere in the block diagram to see the **Functions** palette, select the **Numeric** subpalette therein, then find the appropriate functions, placing them like you did the objects on the front panel. This is the way we will always access the **Functions** palette in this book, so if you had a "permanent" **Functions** palette window (a little window with the word **Functions** in its title bar) in the foreground, you might want to close it now.





Hit the <Tab> key until you find the Wiring tool . Wiring is where you must be the most careful while building VIs. You must also have a steady hand—limit that caffeine consumption until later in the book, when instructed. First, be aware that block diagram objects will often try to wire themselves together by default if you drag them close to one another; I recommend you disable this feature until you become more acquainted with LabVIEW. To toggle this initially cumbersome automatic wiring, hit the space bar while you are dragging an object to be wired on the block diagram.



As with the **Controls** palette window, I am keeping the **Functions** palette window closed so that you must pop up on the front panel to see it. This is a personal preference, as I like uncluttered computer screens. To open it once it's closed, use the **Window»Show Functions Palette** menu item. The **Functions** palette is only seen with block diagrams, while the **Controls** palette is only seen with front panels. If you want to skip ahead and get a quick description of the many items in the **Functions** palette, see Figure 1-30 and Table 1.7.

Earlier, I recommended that you keep the **Functions** and **Controls** palettes, along with the Tools palettes, closed. There is one special case where it's especially useful to have the **Functions** palette or **Controls** palette open. When open, you may click on the little magnifying glass icon  that will take you to a handy browser that allows you to type key words to search for objects. This browser is called the Functions Browser for the **Functions** palette, and the Controls Browser for the **Controls** palette.

If you botch the wiring in this section, undo your steps <Ctrl-Z> carefully until enough wires are gone. If you have too much trouble wiring, skip ahead and read Section 1.2.8. Later, we will use more efficient wiring tricks. But for now, be aware that the math functions mentioned here (the yellow triangles) have three underlying, invisible terminals arranged like this: . Whenever you move the Wiring tool  over the terminal to which it is ready to connect, that terminal will flash. Make sure you don't accidentally wire two wires to the same terminal! If so, or if you make certain other wiring mistakes, you will see the wires turn into dashed lines. This means the wires are broken because they're illegally wired for some reason, and your VI won't run—use the Undo function <Ctrl-Z> and try again.

Once you have everything wired up as shown in the block diagram of Figure 1-1, you should be able to hit the **Run** button  on the front panel and watch your VI work. If the **Run** button looks broken , click it and use the resulting dialog box to track down what's wrong. If you cannot find the error, you may need to recreate the VI from scratch. Once the VI is working, enter numbers other than zero in the front panel controls a and b (must have  or  just prior to typing) so this VI is somewhat less boring!

Yes, I know this seems like a trivial program. And it is. But building it is nontrivial for a LabVIEW newcomer. Gradually, as you realize the power of LabVIEW, you will build much more complex and useful VIs.

1.2.5 Coloring










Let's experiment with coloring. Coloring is usually used on the front panel only, not on the block diagram. First, expect to accidentally color something wrong, so remember <Ctrl-Z> undoes your last action, including coloring mistakes. Okay, now let's go—create a new VI, then drop two new Digital Controls on its front panel. To color one of these controls, go to the front panel, select the Coloring tool , then *right-click* the object you want colored. Up pops a palette from which you can select a color, as shown in Figure 1-12.



Figure 1-12
The Coloring tool palette.

I prefer to use the “More Colors”  button to select a color, so I can select *exactly* the same colors for the many VIs I create. Using that  button, select your favorite shade of green, if any—if not, select your least despised shade of green. On the same control, make it white again. Now try this on another control, and select something ghastly, like burnt orange.

We will now learn about the Color Copying tool . Make sure you have the Coloring tool , hold the <Ctrl> key down, and notice that your Coloring tool changes to the Color Copying tool . To use this tool, move it over an object of the color you want, and click that object with the <Ctrl> key held down. Do this to on the control that you just colored white to “suck up” the color white into the eye-dropper-looking thing, release the <Ctrl> key (so you get the Coloring tool  again), then color the burnt orange section white.

To color the text itself, instead of its background, select the text area you want colored with the Labeling tool , then click the **Text Settings** menu, which looks something like this , and find the “color” section in the menu that pops up.




Undo is particularly helpful when you make a coloring mistake!

Coloring works with most front panel objects and a few block diagram objects. Like the background and foreground of text, many objects have different parts that can be colored.

1.2.6 Selecting Objects for Manipulation

In order to manipulate objects, you must first be familiar with the concepts of selection. If an object is *selected*, you are then able to perform a number of operations on it such as moving it, removing it, copying it, and so on. Multiple objects can be selected, as well.

The best way to learn about selection is to actually do it. Open your Sum of Squares.vi created earlier. *You must have the Positioning tool*  *to select objects.* Once you do, click the a control so you see the “marching ants” (technically, *selection box*, which is the moving pattern of dotted lines around an object to show it’s selected), as shown in Figure 1–13.

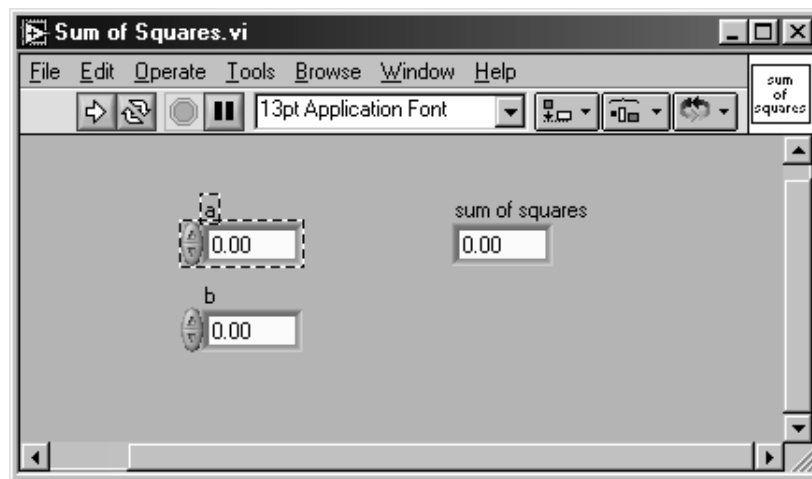



Figure 1–13

The selection box consists of moving dashes, which I call the “marching ants.” This shows you what’s selected.

At this point, the object is selected, and you can do things with it, like moving it and copying it, to be overviewed in the next section.



If you click an object to select it, and you hold the left mouse button down too long while moving the mouse even a tiny bit, you will accidentally drag the object. <Ctrl-Z> quickly undoes this.

To select multiple objects that are physically located near one another, you can drag a rectangle around them with the Positioning tool . See Figure 1-14.

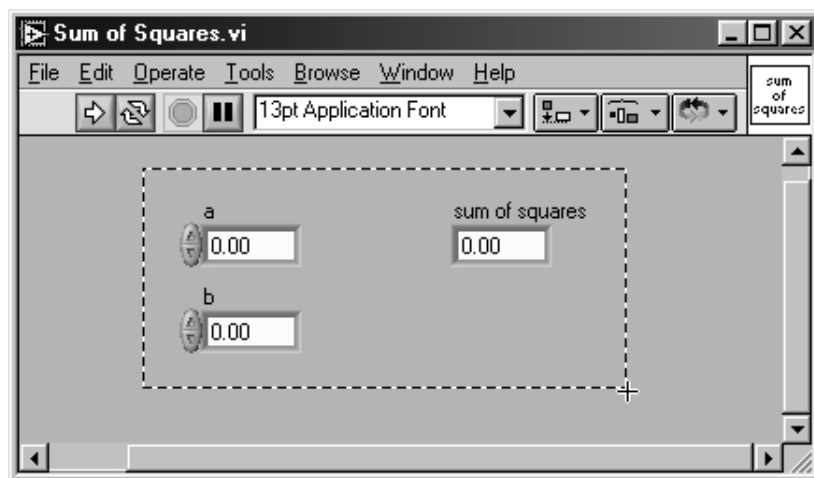




Figure 1-14

Selecting multiple objects by dragging a selection rectangle.

To select multiple objects one at a time, hold down the <Shift> key while clicking each object with the Positioning tool . This <Shift> key trick also allows you to deselect objects one at a time.

1.2.7 Manipulating Objects

This section of the book may save you the most time, especially once you become a good LabVIEW programmer, so read it carefully! The Positioning

tool  is used for manipulating objects, in terms of moving them around and sizing them. Several functions will be discussed in this section concerning object manipulation:

1. Cut, Copy, and Paste
2. Clear
3. Clone
4. Move
5. Resize

Everything listed above, except resizing, can be performed upon multiple objects by selecting the desired objects first, then performing the action.


The **Cut**, **Copy**, and **Paste** functions utilize LabVIEW's *clipboard*, and can be found in the **Edit** menu. LabVIEW's clipboard is a normally invisible storage space where LabVIEW can remember one object or a group of objects from a front panel and/or block diagram. For example, any or all of the three controls on the front panel of your previous VI could have been copied to the clipboard. Once there, you can view the clipboard through the **Windows** menu. When you *copy* an object, you place a copy of that object onto the clipboard. When you *cut* an object, a copy of that object goes to the clipboard as with the *copy*, but you also remove it from the front panel or block diagram. When you *paste* an object, the object is copied from the clipboard to the front panel or the block diagram. Go to the **Edit** menu now, and memorize the keystrokes for these three functions, as you will often use them (cut = <Ctrl-X>, copy = <Ctrl-C>, and paste = <Ctrl-V>).

To demonstrate these three functions in LabVIEW, first create a new VI (via the **File»New VI** menu item). Next, create any new small object on the front panel, such as a Digital Control. To cut or copy an object, you must first select it by clicking on it with the Positioning tool. Now, perform a cut (the **Edit»Cut** menu item or <Ctrl-X>), and watch the selected object disappear. Once cut, the object is residing in the clipboard. To use this object elsewhere, you can now click where you want it, and paste it (with the menu item or <Ctrl-V>). Try the same thing while copying instead of cutting (<Ctrl-C>).

When using LabVIEW, there are really two clipboards involved. One belongs to LabVIEW, and the other belongs to your operating system. LabVIEW's clipboard can remember an image or text copied from another application. This will come from your operating system's clipboard, which can only pass simple images and simple text to LabVIEW (simple in terms of format, not size). Similarly, your operating system's clipboard can only accept


such simple things from LabVIEW. Unless otherwise noted, “clipboard” will hereafter refer to LabVIEW’s clipboard, not your operating system’s.


The **Clear** function, also found in the **Edit** menu, simply removes an object. It is similar to the **Cut** function, except clear does not involve the clipboard. The <Delete> key works similarly.

The **Cloning** function copies an object, but cannot be found in any menu. It is an action that can only be performed with the mouse *and* the keyboard! It is similar to a copy immediately followed by a paste, but the object never sees the clipboard. To clone an object, simply drag it (remember, dragging requires the Positioning tool ) with the <Ctrl> key held down.



When you copy or cut front panel objects’ terminals from the block diagram, the associated front panel objects are placed in the clipboard, but you cannot clone them from the block diagram. When you cut a front panel objects’ terminal from the block diagram, it is not deleted. To delete that front panel object, you must do it from the front panel.

Moving an object can be done in two general ways. The first way is to *drag* the object (click it with the Positioning tool, move the mouse with the left mouse button held down, then release this button when finished moving). The second way to move it is to select it, so that the marching ants are seen, then use the <Arrow> keys (for one-pixel moves) or <Shift-Arrow> keys (for eight-pixel moves). Objects can be precisely arranged by using the **Align Objects** and **Distribute Objects** menus in the tool bar . First, select multiple objects, then select your alignment or distribution method, using those menus in the tool bar. Holding the <Shift> key down while dragging an object limits the cursor movement to horizontal or vertical, depending on the initial direction of movement.

Resizing objects is often difficult even for a pro, because you must first move your Positioning tool very precisely over certain corners of only certain objects. For example, take your *Sum of Squares.vi*, select the Positioning tool, and move the mouse over the lower right-hand corner of any front panel object until your cursor changes into two little angled black lines () , which look like Figure 1–15 over a Digital Control.

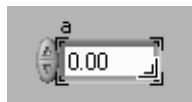


Figure 1–15
Cursor appearance when resizing objects.

When you have this tool, you will be able to drag the corner of a Digital Control or Digital Indicator in a horizontal direction, thus changing its size. Many other objects allow similar resizing in a vertical direction as well as horizontal. Holding the <Shift> key down while resizing limits the cursor movement to horizontal, vertical, or diagonal, depending on the initial direction of movement; however, not all objects support all of these directions.

1.2.8 Wiring in Detail: Dragon Alert

For you LabVIEW newcomers Here There Be Dragons. Prepare yourself. Wiring in LabVIEW is *very* tricky at first. Most objects on the block diagram, other than labels, have terminals to which you can attach wires. This is the basis of LabVIEW programming, so you must be skilled in wiring.

Figure 1-16 shows a few examples of common block diagram objects.



Figure 1-16
Common block diagram objects.

For some objects, or nodes (such as these), you can pop up on them and select **Show»Terminals** so that you see the terminals (where you can attach the wires, no more than one wire per terminal), shown in Figure 1-17.



Figure 1-17
Inherent terminal patterns (hidden) from Figure 1-16.

All front panel objects with block diagram terminals have nodes with just one terminal, such as the three objects in *Sum of Squares.vi*, shown earlier. The data flowing to or from such terminals may be a single number, or it may be an array, or it may be quite a complex data structure—you will be able to determine this data type before the end of this chapter.

Wiring difficulty in LabVIEW stems from the inherent difficulty of attaching the wire to the right place; the (sneaky, invisible) terminal. Usually, you can't see the terminal explicitly, so it's easy to attach a wire to the wrong terminal. If ever your LabVIEW program doesn't work as expected, miswiring could be the reason.

For example, suppose you were wiring to the Add function in `Sum of Squares.vi`. Also suppose you attempt to wire the two input terminals on the left of the Add function, and suddenly you have broken wires and a program that won't run, as in Figure 1-18.

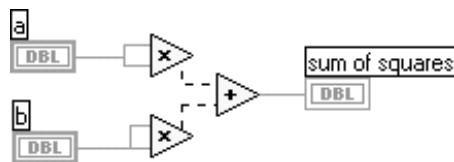



Figure 1-18

It is not clear why these wires are broken.

Notice that the **Run** button is broken ; this means your VI won't run. What's going on here? Even a pro might think there should be no broken wires here, but in this case, you have two wires wired to the *same* terminal, either the top left or the bottom left terminal of the Add function. To verify this, you could triple-click either broken wire and see that they're actually the same wire, as the marching ants will indicate in Figure 1-19.

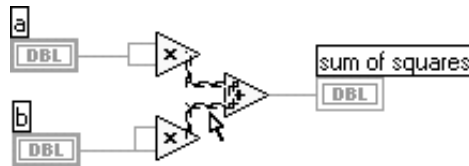



Figure 1-19

After triple-clicking the broken wires, it becomes clear why they are broken.

As a rule, once you have broken wires, hit <Ctrl-Z> to undo your last wire, then try again. If you continue to fail, hit <Ctrl-B> to delete all broken wires (some little ones might be hidden), and if that doesn't fix things, then start from scratch. Depending on the situation, once you get more wiring experi-

ence, you may find faster ways to remedy your wiring problems, such as by deleting certain wire segments and rewiring carefully.



If you click your broken Run button , and all your errors are wiring-related, a single <Ctrl-B> (Remove Broken Wires) often fixes everything.

A *wire tree* is simply all wires connected together. For example, Figure 1–20 shows two wire trees (terminals a and b belong to Digital Controls on the front panel and have a representation of I32, a 32-bit integer).

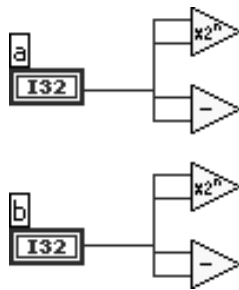


Figure 1–20
Two valid wire trees.

In order for your VI to run, a fundamental requirement is that any wire tree must have exactly one source and one or more destinations. If you were to wire the above two trees together, you would have one large tree, shown in Figure 1–21, but it would not be legal because it would have two sources (a and b).

If you now try to run the VI by clicking on the broken **Run** button, you'll see the message box shown in Figure 1–22, which is designed to help you eliminate your errors.

It may be a little tough to decipher these messages without experience, but you should pay attention to this line (wording subject to change with future versions of LabVIEW): "This wire connects more than one data source." Click on that line, and in the lower box, you will see the message shown in Figure 1–23.

1.2 VI Basics

29

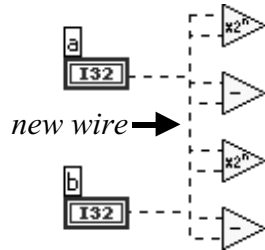


Figure 1–21

One invalid wire tree, which should be two trees. (I drew the “new wire” text and the arrow.)

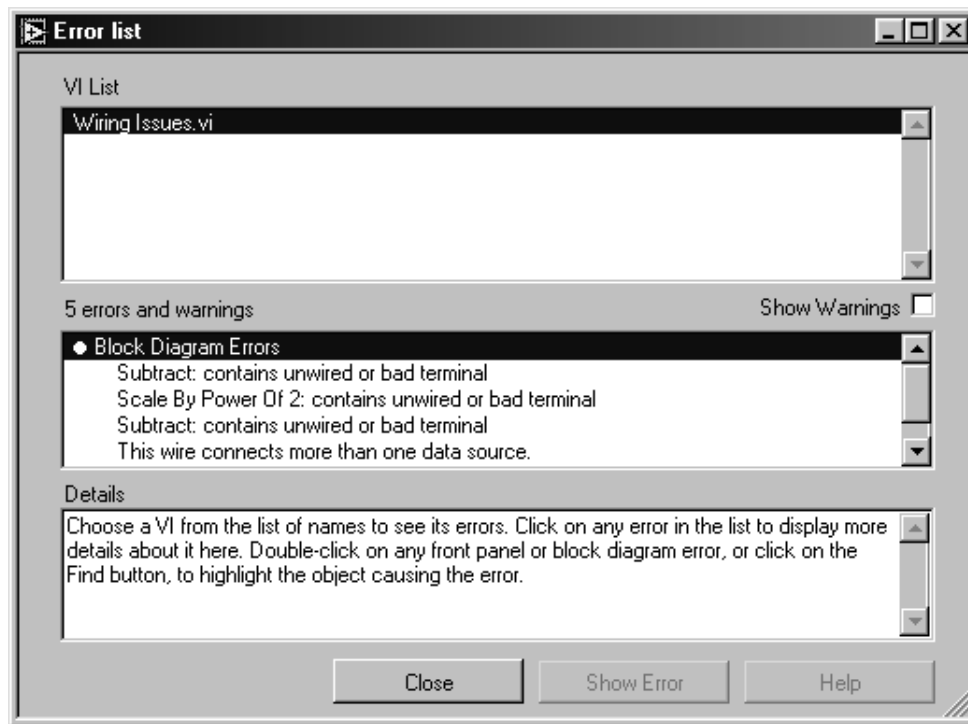


Figure 1–22

LabVIEW's Error list box, from clicking a broken Run button.

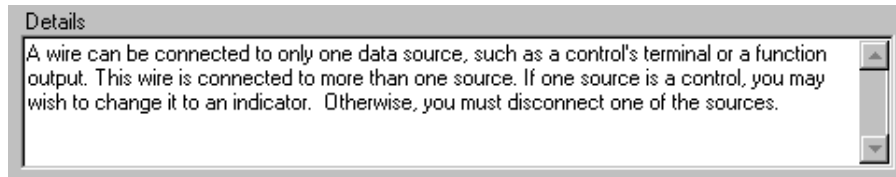


Figure 1-23

Result of clicking the “This wire connects more than one data source” line from Figure 1-22.

That means that your wire tree has more than one source, as in our wire tree in Figure 1-21. In this case, these sources are a and b, both of which are trying to set the value of the wire tree—the tree has no way of deciding which value to take. If you’re an electrical engineer, this would be like connecting two different voltage sources to the same wire. Luckily, this wiring mistake won’t physically fry your computer.

These error messages may be reworded from version to version of LabVIEW, but the underlying problems are the same. The rest of the messages are complaints from $x2^n$ and $-$, both whining that they don’t have valid wires connected to their inputs.

On the other hand, suppose we start with the two valid wire trees, as in Figure 1-20, then change the upper control to an indicator, so we get broken wires (and, of course, a broken **Run** button). See Figure 1-24.

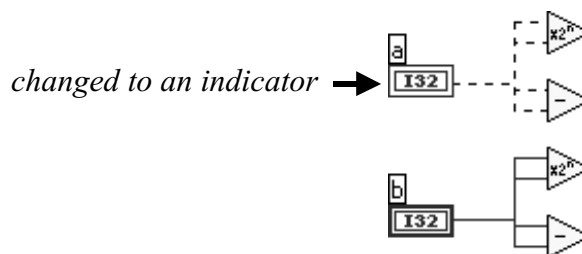


Figure 1-24

Another way to break a wire tree. (I drew the “changed to an indicator” text and the arrow.)

This time, the error box looks like Figure 1-25 (after I clicked on the appropriate line).

Read the message under **Details**, which describes the problem very well. The source of this data is the a indicator. You can fix the block diagram by changing it back to a control.

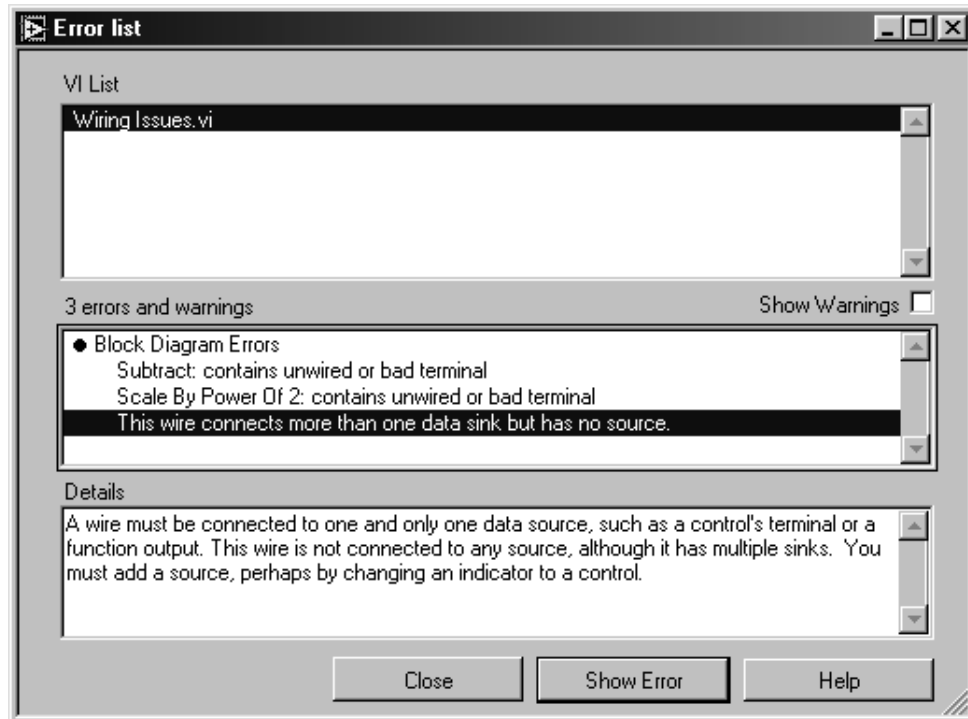


Figure 1-25
Error message resulting from Figure 1-24.

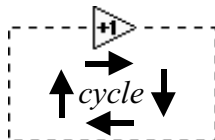



Figure 1-26
A cycle (wiring error).

The last major class of wiring error you may (rarely) run into is a *cycle*. Figure 1-26 shows the simplest example of a cycle.


If LabVIEW allowed this type of construct, which it doesn't, your program would never end, as it would get caught in an *infinite loop*. A number would come into the Increment function from the left, it would be increased by one, then it would go back into the Increment function, and so on forever.

Whenever you see the word "cycle" in your error list box relating to wiring, you have a situation like this, which is not legal. Usually, cycles happen in much more complex wiring situations and are correspondingly more difficult to spot. As soon as you run into a cycle, check your wiring logic. If you can't find your mistake, undo your way out of this situation then rewire very carefully.

Here is a list of wiring tips, which should greatly increase your productivity:

1. Wiring diagrams, like circuit diagrams, should be neat and tidy, with a minimal number of crossing wires.
2. Hit <Ctrl-B> to remove broken wires, especially when your wiring is complex. If you try wiring again, and it still doesn't work, remove more valid wires around where you're working, then rewire.
3. With the Positioning tool , single-click a wire to select one linear segment, double-click to get one branch of a wire tree, and triple-click to get the entire wire tree.
4. To determine exactly which terminal (or terminals) of a block diagram node your wire is connected to, triple-click the wire to let the marching ants lead you to its hidden connections.
5. To fine tune the positioning of your wires, single-click a wire segment, then move it with the arrow keys in a direction perpendicular to that segment's direction.
6. Let's hope NI doesn't start charging us for wire by the inch, which is all the more reason to *minimize your total amount of wire*. For example, suppose you're happily wiring along, then encounter this message: "You have run out of wire again. Please visit our Web site with your credit card ready to buy more wire."

1.2.9 Free Labels

If you left-click with the Labeling tool  somewhere other than on editable text, you can place a *free label* anywhere on the front panel or block diagram. On the front panel, these are one of the few objects without block diagram terminals, like the objects in the **Controls»Decorations** palette, thus they have no block diagram data associated with them. They are simply used to clarify and describe things on the front panel or the block diagram. On the block diagram, they are useful for helping newcomers, including *you* many months later, to figure out your code; there they are often called *comments*, keeping consistent with general programming lingo. On the block diagram, I prefer to color my comments' backgrounds yellow with black text, as this prints out clearly on most color printers.



I find it convenient to not color anything else but comments on the block diagram, in the spirit of consistent programming.


As real estate is at a premium on block diagrams, keep your comments few and at a high level (don't get too detailed). I won't be making many such comments in this book, in order to save space.

1.3 LABVIEW HELP

There are many components being introduced in this chapter, as I'm sure you've noticed. When you're building the block diagrams, you need not bother to pop up on each component to read the LabVIEW's help information, unless it is key to your understanding. Unless you have a serious brain capacity, you would be swamped!

Within LabVIEW, there are several convenient help utilities. Outside of LabVIEW's own help resources, you can get help through various news-groups, Internet resources, and from NI—these external options are detailed in the preface. For this chapter, let's discuss your help options *inside* LabVIEW, then when you need further help, refer to the preface for your external options.

1.3.1 The Help Window

Let's look at the Help window contents of a simple function, the Subtract function. Open up a new VI, drop the Subtract function  on your block diagram, then, while your cursor is over the function, show the Help window (<Ctrl-H> is a nice shortcut for showing and hiding the Help window). Figure 1-27 shows what it looks like.

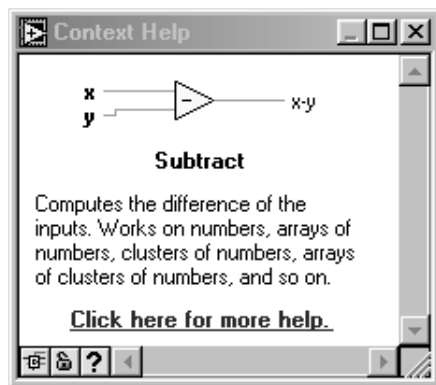






Figure 1-27

An example of LabVIEW's Help window.

In general, the Help window shows you a description of a function's inputs and outputs, the name of the function, and an overall description. The contents of the Help window will change as you move your cursor over certain objects on the block diagram. If these contents are sluggish when changing the Help window, wiggle your cursor a bit over the relevant object, or click the object.

See the three boxes in the lower left area of the Help window? The first box  will show (or hide) normally hidden input and output wires (some functions with many inputs and outputs will hide those of lesser importance). The second box  will lock the Help window onto its current contents, despite where you move your cursor on the block diagram. The third box  will bring up the information in LabVIEW's *help files* (described soon) concerning the topic of the Help window, as will the *Click here for more help* link.

1.3.2 LabVIEW's Help

In addition to LabVIEW's Help window, LabVIEW has its own help in the format of standard Windows Help and PDF files. This is an even larger store of knowledge than the Help window. For example, if you had clicked on the  box or the [Click here for more help](#) link in the Subtract function's Help window, a window would appear with a different description of the Subtract function. Clicking on this window's Index button, you would see a standard help interface, as shown in Figure 1–28.

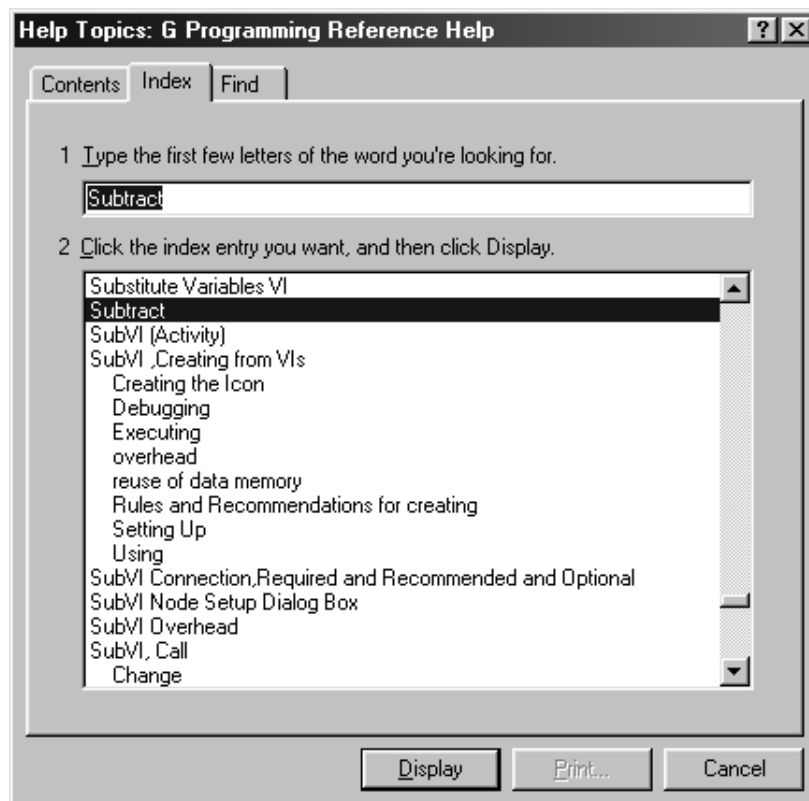


Figure 1–28
Another LabVIEW help window.

This help window can also be accessed via the **Help»Contents And Index** menu item. There are also some very helpful and well-written PDF files to be

found from the **Help»View Printed Manuals...** menu item. You may want to read these to supplement this book.

1.3.3 Help Per VI

Finally, VIs can contain descriptive information in their **File»VI Properties...»Documentation** box, easily accessed with <Ctrl-I>, and in many of their objects' **Description and Tip...** boxes. If a VI is used as a *subVI* within another VI's block diagram (subVIs will be described later), these pieces of information can be available in the Help window of the subVI.

1.4 The Controls Palette (Front Panel Only)

One of my *few* complaints with LabVIEW is that it's sometimes too complicated for newcomers. But in my experience with newcomers, it ties for the "least complicated" award with Microsoft Visual Basic! As a veteran, I love the complexity because it serves to enhance the power of LabVIEW, but it takes a long time to sift through all the information when you're starting out.

A perfect example of complexity is the **Controls** palette (see Figure 1-29), described briefly in Section 1.2.4. Table 1.6 provides a quick description of its subpalettes.

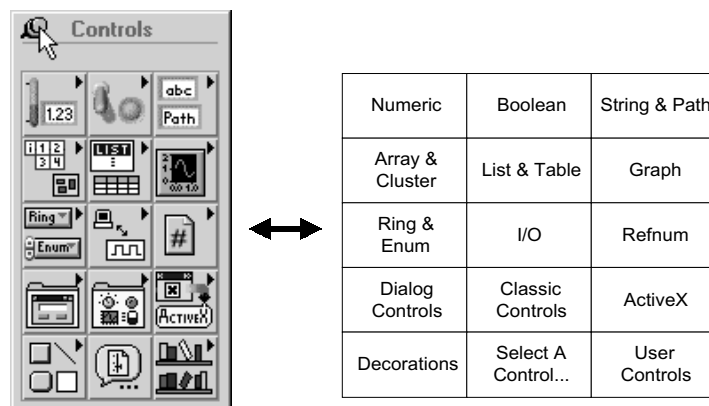


Figure 1-29
The Controls palette.

1.4 The Controls Palette (Front Panel Only)

37



For every item in the **Controls** palette (listed in Table 1.6) that has associated data (most of them, but not the **Controls»Decorations** objects), a corresponding terminal will be created on the block diagram once you create the front panel object. Also, when you see the word *control* in Table 1.6, an indicator will sometimes drop, depending on the particular object you select within the subpalette. You can change any control to an indicator, or vice versa, after it's been dropped.

Table 1.6 Controls Palette Summary

Subpalette	Description	Data Type
Numeric	A wide variety of numeric controls.	Numeric
Boolean	A wide variety of Boolean controls.	Boolean
String & Path	A string can display text or hold generic data bytes, and a path points to a filename, like C:\Folder\My File.txt; either type can hold an arbitrary number of bytes/characters.	String and Path
Array & Cluster	Objects that allow you to group other controls.	Array and Cluster
List & Table	Lists allow you to display multiple text items, all in one control, and tables are grids of strings (like a spreadsheet).	Numeric and 2D Array of Strings
Graph	Rectangular objects that allow you to show your data graphically (pictures).	Various
Ring & Enum	Like a list, but acts like a pop-up menu as only one item is displayed at a time.	Numeric
I/O	Many controls related to input/output, including DAQ Channel Name and Waveform.	I/O (DAQ Channel Name), Waveform, etc.
Refnum	Reference numbers for open connections of various types, such as the Byte Stream File Refnum for open files.	Different types of Refnums like the Byte Stream File Refnum

Table 1.6 Controls Palette Summary (continued)

Subpalette	Description	Data Type
Dialog Controls	A variety of controls appropriate for dialogs (pop-up windows that do not allow you to click other windows).	Various
Classic Controls	A variety of controls used in previous versions of LabVIEW; I prefer these when screen space is an issue, as they're more space-efficient!	Various
ActiveX	Controls that allow you to integrate ActiveX components (Windows only).	Various
Decorations	Front panel objects that are just for looks; they control no data, thus they have no block diagram nodes.	None
Select a Control...	Lets you select a custom control that you or somebody else may have built.	Various
User Controls	User-defined controls, which will be discussed later in this book as <i>custom controls</i> ; see help files documentation on how to get them into this palette.	Whatever you want

Open a new VI, and drop controls from each of the these subpalettes just to see what they look like. With the Positioning tool , you can move them around and resize most of them. With the Coloring tool , you can color different parts of them. For a few of the controls, pop up on them and select the **Find Terminal** menu item to see what the terminals look like. Different colors of the terminals represent different data types, as described in Section 1.2.3.

Once you've seen at least one control from each of the subpalettes in Table 1.6, close your cluttered, useless VI <Ctrl-W> without saving it.

1.5 THE FUNCTIONS PALETTE (BLOCK DIAGRAM ONLY)

Figure 1–30 shows what the **Functions** palette looks like (used only on the block diagram), and Table 1.7 summarizes its basic subpalettes.

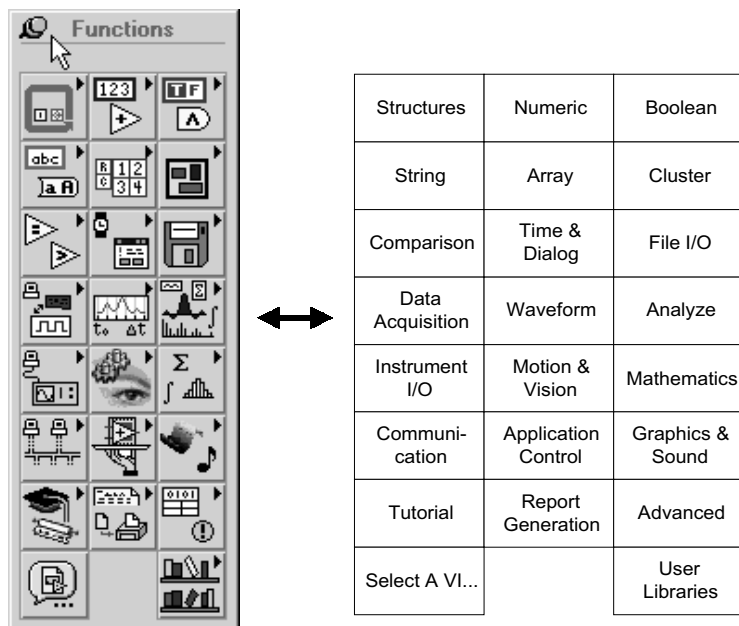


Figure 1–30
The Functions palette.

Unlike **Controls** palette items, which usually place a corresponding node on the block diagram, the reverse is not true. These **Functions** palette items never place an object on the front panel—they only place an object on the block diagram.

If you have the **Functions** palette permanently open, you can click the **Options** button in the **Functions** palette toolbar to access the Function Browser. This lets you customize the **Functions** palette.

Like the previous section, this section is only intended to give you an overview, and the important details will be explained later in the book.

Table 1.7 Functions Palette Summary

Subpalette	Description	DAQ relevance
Structures	High-level program control.	Nearly always
Numeric	Operations on numeric data types.	Always
Boolean	Operations on Boolean data types.	Nearly always
String	Operations on string data types.	Nearly always
Array	Operations on array data types.	Nearly always
Cluster	Operations on cluster data types.	Nearly always
Comparison	Compares many types of data, but usually numeric data, such as determining whether one number is greater than the another.	Nearly always
Time & Dialog	Timing and pop-up windows (dialog boxes).	Nearly always
File I/O	Reads and writes data to files (typically hard disks, floppy disks, etc.).	Often relevant
Data Acquisition	Provides functionality for all of NI's data acquisition (DAQ) products.	No comment
Waveform	Creates and manipulates the Waveform data type.	Sometimes
Analyze	Mathematics for Waveforms and arrays.	Frequency, uh, I mean frequently
Instrument I/O	Provides communication with a wide range of instruments having certain types of interfaces.	Relevant if you consider these instruments "DAQ"
Motion & Vision	VIs for NI's motion and vision hardware (controlling motors and video cameras).	Not officially considered DAQ, but related
Mathematics	Analysis for arrays of numeric data, such as averaging, standard deviation, frequency response, etc.	Often relevant
Communication	Provides communication with other computers, other applications on your computer, etc.	Sometimes relevant
Application Control	More powerful control of your application.	Sometimes relevant
Graphics & Sound	Very fancy graphics controls, plus controls for a standard PC sound card.	Often relevant



1.5 The Functions Palette (Block Diagram Only)

41

Table 1.7 Functions Palette Summary (continued)

Subpalette	Description	DAQ relevance
Tutorial	VIs used in LabVIEW's built-in tutorial, which is the purpose of this chapter as well; try it!	Only if you need a more detailed tutorial than this
Report Generation	Helps generate printed reports from LabVIEW.	Often relevant
Advanced	Advanced data manipulation.	Often relevant for advanced projects
Select A VI...	Required for subVIs.	Nearly always
User Libraries	For your own LabVIEW subVIs; see help files documentation on how to get them into this palette.	Could be relevant, if you want

Go ahead and drop some of these objects on a block diagram, but do not save anything if asked. Some of these functions (generally the non-yellow ones) are actually subVIs; they often come from LabVIEW's folders, such as `vi.lib`. If you change these subVIs and save your changes, you could easily foul up your copy of LabVIEW, so be careful—see Appendix C for details.

Notice the Compound Arithmetic function in the **Functions»Numeric** palette (when you see the name of a palette or other object starting with **Functions»**, this means to start with the **Functions** palette, found by popping up on the block diagram). If you wanted to add more than two elements, you could use this function. You can operate on as many elements as you want, by growing it with the positioning tool. You can also negate any incoming element. For example, you could configure the Compound Arithmetic function to behave as the Subtract function, so that these two icons are equivalent:  . The little circle shown on the lower input of the Compound Arithmetic function here is created by popping up on that terminal and selecting **Invert**; when adding numbers, this inversion circle takes the negative of the incoming number.

It would be tempting to describe all such useful functions here, but this is only one book and should not weigh 50 pounds.

1.6 SIMPLE DAQ

Since this is a DAQ book, let's create an extremely simple DAQ VI. This section assumes you have a DAQ device installed with analog input, and that MAX (Measurement & Automation Explorer from National Instruments) is installed and recognizes your DAQ device. If you launch MAX and expand the Device and Interfaces item, as in Figure 1-31, you should see your DAQ device with a number next to it. Set your Device Number to 1, if it is not already. In my case, my DAQ device is an AT-MIO-16DE-10 and my device number is one. Select your device from MAX and test with the Test Panel button, shown in Figure 1-31, to see analog input.

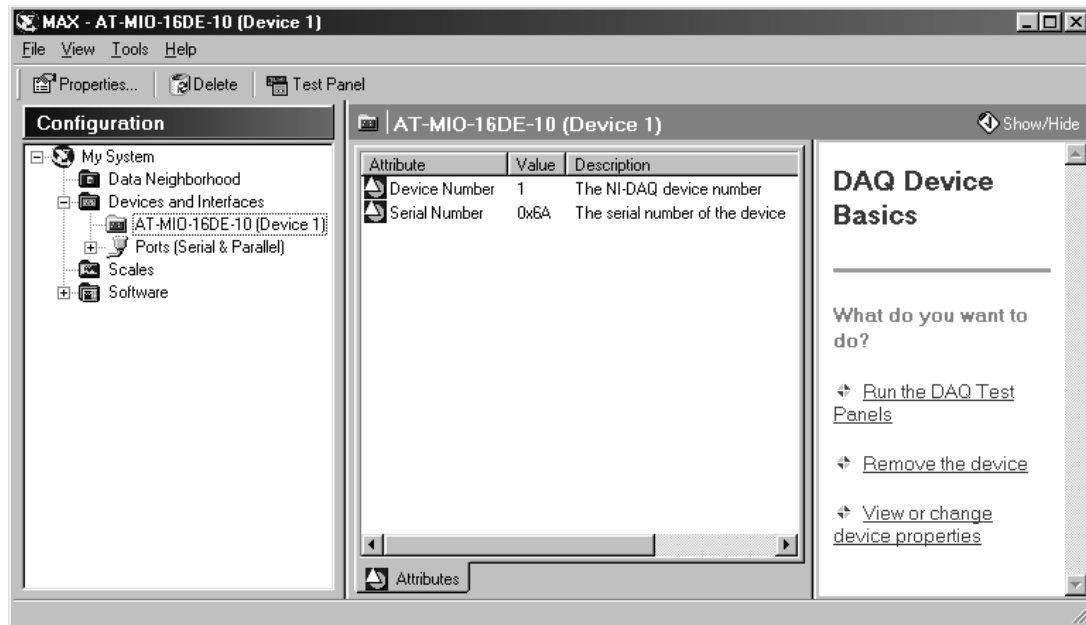


Figure 1-31

The "Devices and Interfaces" screen of MAX is shown here.

Build the VI shown in Figure 1-32, starting with the front panel, using these tips:

1.6 Simple DAQ

43

1. The Waveform Chart is found in the **Controls»Graph** palette. Pop up on the chart's Plot Legend (labeled Plot 0) and select the chart type shown, which is little squares connected with lines. The chart can accept the Waveform data type, which was covered in Section 1.2.3, but its simplest data type is a single number.
2. The Stop Button is found in the **Controls»Boolean** palette.
3. The While Loop is the big gray rectangle shown in Figure 1–32 on the block diagram, and it is found in the **Functions»Structures** palette. Loops will be covered in detail later, but they basically are rectangles in which all functions are executed repeatedly. To draw the While Loop, select it, left click the block diagram where you want one corner of the While Loop, and drag your mouse to the diagonally opposite corner.
4. Pop up on the little green square inside the While Loop and set it to **Stop If True**. This will be explained later in this chapter.
5. The box labeled “AI ONE PT” is found in the **Functions»Data Acquisition»Analog Input** palette. Be sure to choose `AI Sample Channel.vi`, *not* `AI Sample Channels.vi`.
6. The box with the metronome is called “Wait Until Next ms Multiple” function, and is in the **Functions»Time & Dialog** palette. Pop up on its left terminal and select **Create Constant**, then type in 250 before clicking anywhere.

Run the VI. If you get an error and you cannot figure it out, you'll find sources of help in the preface of this book. If you do not get an error and you see data on the screen that might look something like Figure 1–33, you have successfully performed DAQ with LabVIEW!

We must wait until Chapter 3 before covering DAQ in LabVIEW extensively, but I decided to let you know right away how it's done. Technically, we also performed DAQ with MAX when we tested the analog input channel, but LabVIEW greatly expands your ability to do just about anything you want with the data.

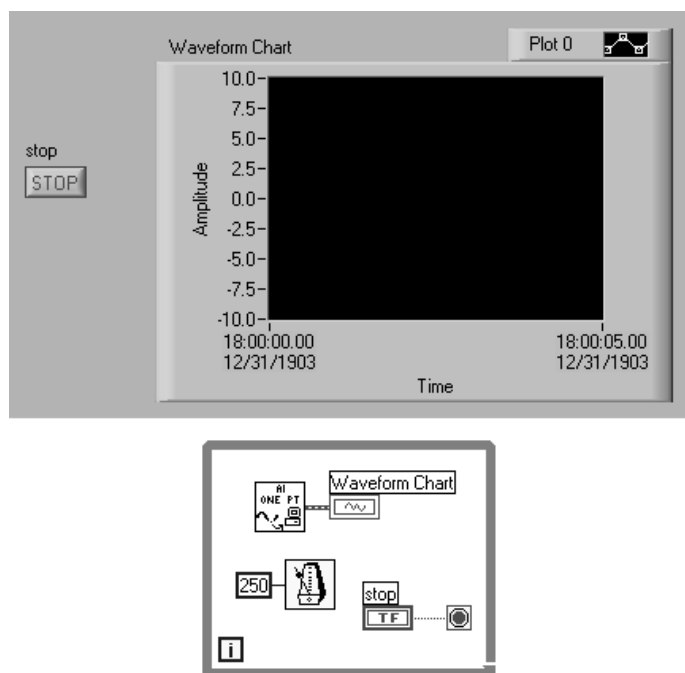


Figure 1-32
A very basic DAQ program is shown here.

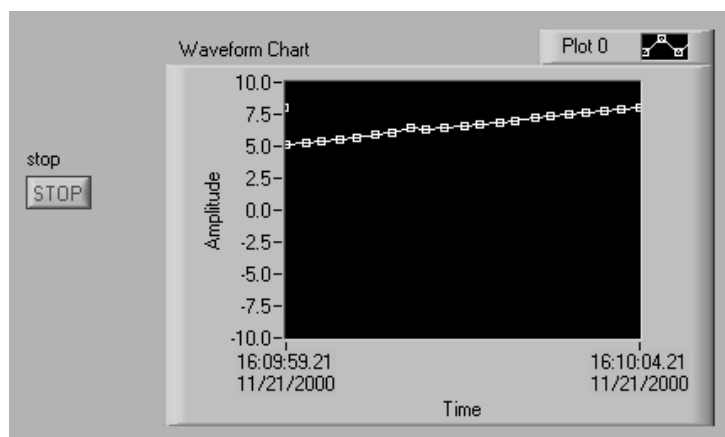


Figure 1-33
Example of a successful DAQ.

1.7 DECISIONS, DECISIONS

As of this section, we begin our dive into those aspects of LabVIEW that make it a real programming language, rather than just a fancy presentation. In every program, at some level, the program must make a decision. For those of you who have programmed before in text languages, this is sometimes called a *conditional statement* or an *if/then/else statement*. Decisions must be based upon conditions, so we will be using the conditional functions in the **Functions»Comparison** palette. We will often use the Case Structure, soon to be described.

1.7.1 A Simple Example

With that said, let's start building a VI to illustrate decisions using conditional functions. Create a new VI. Use the Digital Control from the **Controls»Numeric** palette and the Round LED indicator from the **Controls»Boolean** palette to create the front panel shown in Figure 1-34. Do not worry about the exact appearance of anything outside the main gray area throughout this book, unless instructed.

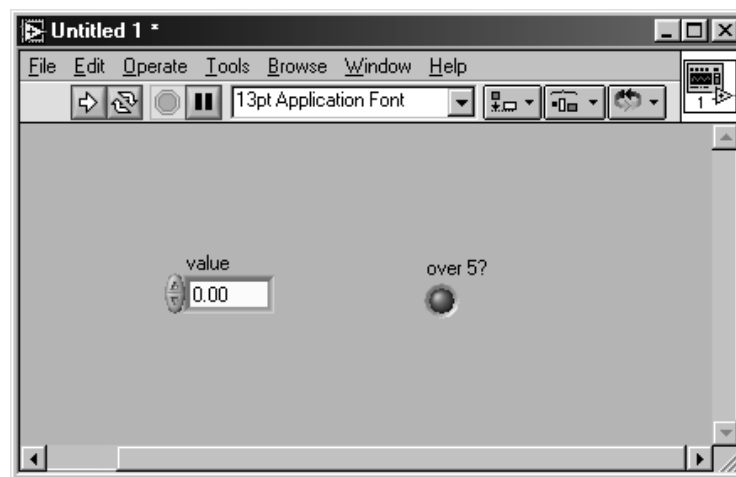


Figure 1-34
Creating a new VI's front panel.

Next, move on to the block diagram and wire it as shown in Figure 1–35 (if you have trouble, read the paragraph after Figure 1–36).

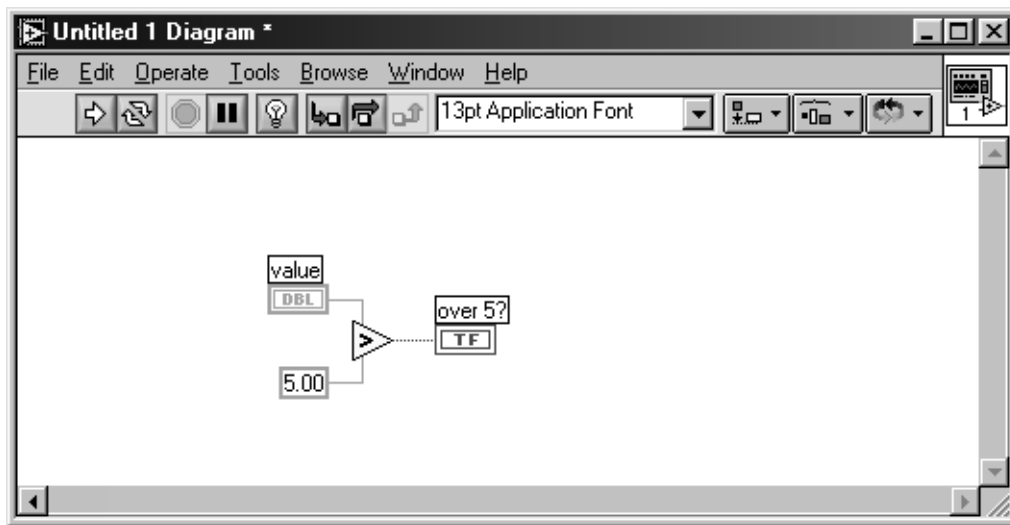





Figure 1–35
Creating a new VI's block diagram.





To save space in the rest of the book, we will be showing only the inside areas of the front panels and block diagrams of our VIs, and sometimes the icon/connector (described later) when it's important. Let's redisplay the above VI more efficiently, as in Figure 1–36.




Figure 1–36
"Shorthand" for showing both a front panel and block diagram, previously seen in Figures 1–34 and 1–35.

If you've successfully built this VI, skip this paragraph. Otherwise, the value and over 5? objects should be there when you first see the block

diagram. The box with the 5.00  and the Greater Than function  can be created from the **Functions** palette. The box with the 5.00 is a Numeric Constant from the **Functions»Numeric** palette. If you immediately type in 5.00 after it's dropped, you'll have a number with the DBL data type. Otherwise, if you had just typed 5, your number would have the I32 data type; you can change that data type to DBL by popping up on the 5.00 and selecting DBL through its **Representation** menu item. The Greater Than function comes from the **Functions»Comparison** palette. If you have difficulty wiring, review Section 1.2.8, "Wiring in Detail." To start from scratch on the block diagram, select everything (drag a rectangle around it with the Positioning tool ) and delete it with the <Delete> key—only the terminals of front panel objects will remain.

With the VI successfully built (the **Run** button will be unbroken ) , go back to the front panel, type a 6 into the value box, then hit the **Run** button . You should see the over 5? box lighten. This light color means True, and in the case of this VI, it specifically means that "6 is greater than 5" is True. Next, type a 4 into the value box, hit that same **Run** button, then you should see the over 5? box darken. This dark color means False, or 4 is *not* greater than 5. Finally, go to the block diagram, hit the **Highlight Execution** button  so that it turns yellow and has little lines emanating from it , then watch the block diagram after you hit the **Run** button so you can see in detail what's happening. Notice how this *execution highlighting* slows down your VI, so click it again to turn it off, because we'll soon run it at high-speed.

After this section, we will learn how to build a *loop*. In software terms, a loop is a means of repeatedly doing a similar thing. We could build a VI, using a loop, that monitors some data by repeatedly reading that data, and possibly acting on the data if it meets certain conditions. For now, go to the front panel again, and we will see what the **Run Continuously** button  is about. In general, this is a bad button for beginners, because it can lead to great confusion—but it's nice to know about it.

Go ahead—push the button and see what happens to the buttons on the tool bar (see Figure 1-37).





Figure 1-37
The result of pushing the Run Continuously button.

This is a sign that you are in continuous run mode. We have effectively created a loop, since the VI runs repeatedly. While this VI is continuously running, go to the front panel and change value to different values, some greater than 5, and some less than 5.



*If you type in the numbers rather than using the little up/down arrows on the **value** control, LabVIEW doesn't use the new value until you hit the <Enter> key or click elsewhere. This little weirdness applies to most text-entry objects in LabVIEW, except the String Control, which you can configure via a pop-up menu item to have LabVIEW read the string data as you type.*

As you change value to different numbers above and below 5, you should see the over 5? box change from True to False (while you are in continuous run mode).

The **Run Continuously** button, which normally looks like this , should now be darkened . Click it, so as to stop continuously running, then go to the block diagram.

A more elegant way to create a loop in LabVIEW is to use the built-in looping structures; either the While Loop or the For Loop. These will be presented in a later chapter.

Next, we will modify our VI further to illustrate more basics of conditional programming. The front panel will soon look like Figure 1–38.

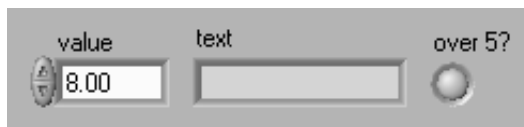


Figure 1–38
Building an example front panel.

To create this, drop a String Indicator on the front panel and label it `text`. The String Indicator is found in the **Controls»String & Path** palette.

1.7.2 The Select Function

Next, modify the block diagram as in Figure 1–39.

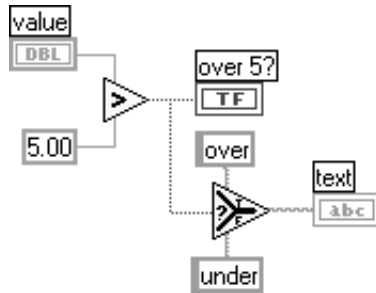


Figure 1-39
Building an example block diagram.

The new items are found in the palettes described in Table 1.8.

Table 1.8

Item	Appearance	Palette
String Constants (after you've typed "over" and "under")	over, under	Functions»String
Select function		Functions»Comparison

After wiring the VI as shown in Figure 1-39, go to the front panel and run an experiment similar to what you did before, using the **Run Continuously** button and entering values greater than, then less than 5 while the VI is running. As you should see in the text indicator, the Select function, presented here, selects either the data on its top wire or its bottom wire, depending on whether the data on its middle wire is True or False, respectively. To clarify this process, look at the block diagram with execution highlighting on.

This Select function is one of the two major ways to make a decision in LabVIEW. The other is the Case Structure.

1.7.3 The Case Structure

On the block diagram we've been having fun with, we'll soon replace the Select function with the Case Structure, which will do almost exactly what

the Select function did. Later, we'll compare the Case Structure to the Select function.

Select the Select function with the Positioning tool  (see Figure 1-40).

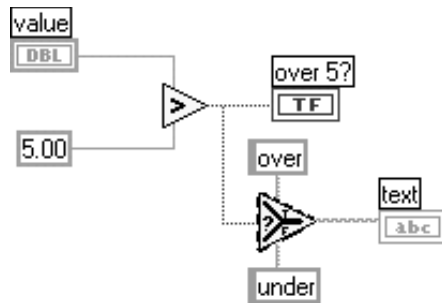


Figure 1-40

The Select function is illustrated.

Delete it with the <Delete> key, delete any bad wires remaining with <Ctrl-B>, then arrange your objects as in Figure 1-41.

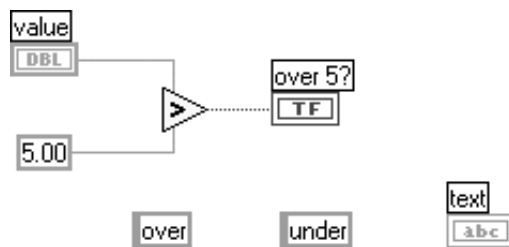



Figure 1-41

A few items from Figure 1-40 are deleted.

Select the Case Structure from the **Functions»Structures** palette (called "Case" there), and while your cursor still looks like this, , drag an outline exactly where you see the Case Structure in Figure 1-42—as soon as you release your mouse button, the Case Structure will be created there. Pop up on the Case Structure (its wall, not the interior) and select **Make This Case False**, so it looks like Figure 1-42, thus our True/False logic will be correct.

1.7 Decisions, Decisions

51

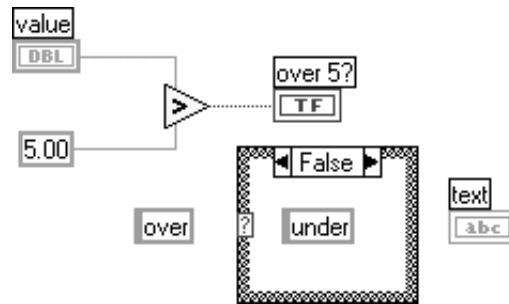




Figure 1-42

The VI from Figures 1-40 and 1-41 is being modified.

If you need to adjust the position and/or size of the Case Structure, you can use the Positioning tool to drag any of its corners.



It's very easy to accidentally hide other objects behind structures. The hidden object is still there, but you cannot see it. Simply move the structure if you think there may be something behind it.

By default, a Case Structure drops with two cases, False and True. Click one of the little left/right arrows   to see the True case, notice that it's empty, then look at your False case again by using these same arrows. Wire the under String Constant through the Case Structure's wall to the text indicator, as shown below. Notice a *tunnel* is created, as in Figure 1-43.

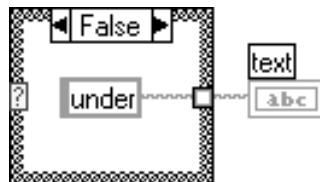


Figure 1-43

A Case Structure tunnel.

A Case Structure tunnel allows a wire to pass through the wall of the structure; we will see similar tunnels on loops soon. A Case Structure tunnel must be wired from *all* cases within the Case Structure. If it is not, it then

looks like a white rectangle, and the VI will not run. When it is correctly wired from all cases, it will change to a solid, non-white color. First, wire the True case *incorrectly*, as in Figure 1–44 (first use the left/right arrows to get to your True case, then move the over String Constant into this case):

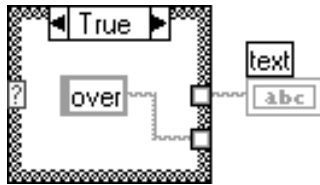



Figure 1–44
The True case wired incorrectly.

If you wanted the VI to run, you would have wired the over constant to the existing tunnel, not to another place on the Case Structure's wall. This is a common mistake, especially when those two white tunnels are almost on top of one another—you will think you have one tunnel that's wired from both cases that should be a solid, non-white color, when in reality you have two, so it looks like one that's white.

Let's wire it correctly now. First, the little rectangle  on the left wall of any Case Structure is the all-important selection terminal—the value it receives determines which case is executed. Delete the wire coming from the over constant, and wire your diagram so it looks like Figure 1–45 (notice I've shown both cases here, while you can only see one case at a time).

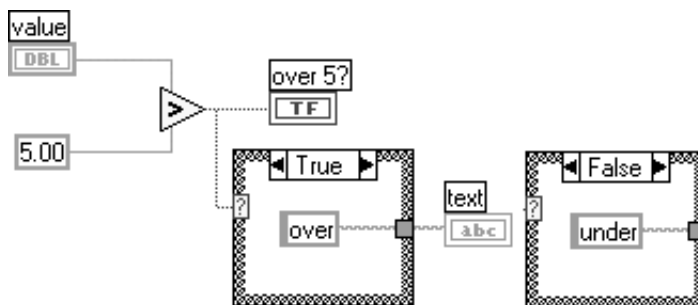





Figure 1–45
The True case wired correctly.

1.7 Decisions, Decisions

53

We have used the Case Structure to do exactly the same thing the Select function did previously. So why use the Case Structure? Its major advantage is that the objects in the cases that are not executing will not be executed. Although it won't make much of a difference in this example, some LabVIEW code may take significant time to execute. A secondary advantage is that the Case Structure saves some block diagram real estate. This could also be considered a disadvantage in that the objects in the cases that are not showing are hidden. To illustrate a case not executing, build the following block diagram, following these tips:

1. Take care *not* to hit the **Run Continuously** button  while building this one.
2. The new object inside the rectangle here is the One Button Dialog function, found in the **Functions»Time & Dialog** palette.
3. You can move from one case to the other by clicking on the little left/right arrows on the top of the Case Structure  . See Figure 1-46.

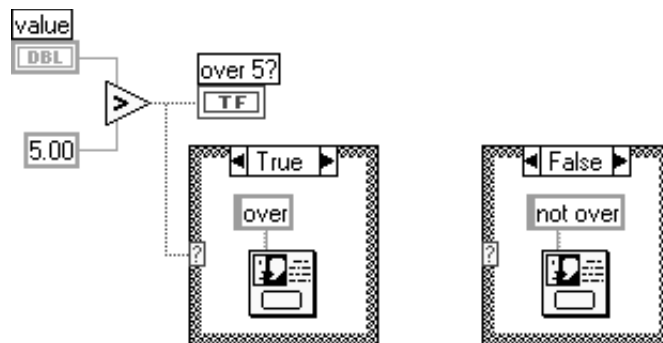


Figure 1-46

Move from one case to the other by clicking on the left/right arrows on the Case Structure.

As before, you will not see both cases of the Case Structure at the same time, as is shown above. But in this book, it is most convenient to portray all Case Structures' cases in the same figure, since you cannot very well click on the pages of this book and expect to switch cases. If this happens, you have discovered my super-secret magic edition of this book!

When you run the above VI, it should be clear that only one of the two cases executes per run, since you only see one dialog box pop up. To clarify


this VI further, show both the block diagram and the front panel (there are a couple of **Windows»Tile...** menu items that do this automatically). Turn on execution highlighting , run the VI from the block diagram, then you can easily see what's happening.

Figure 1-47 shows an example of an *integer* wired to a Case Structure, as compared to the *Booleans* we've seen before. Booleans only allow two cases, but integers allow multiple cases:

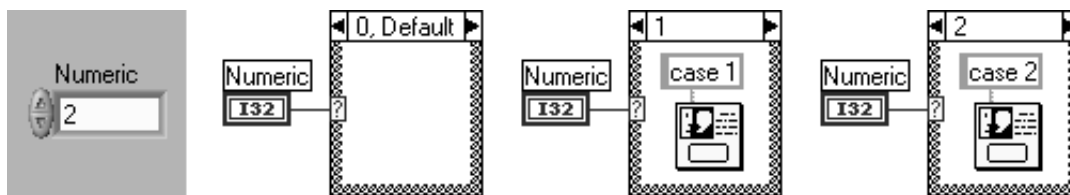




Figure 1-47

An integer wired to a Case Structure.

To create the additional case, pop up on the Case Structure and select the appropriate menu item. If you're clever, you'll create case 1 then duplicate it by popping up on the wall of the Case Structure with case 1 showing, selecting the Duplicate Case item, then editing the text of the newly created String Constant.

Note that every Case Structure (except those with Boolean inputs) has a default case that executes when none of the other cases' conditions are met. Case Structures with Boolean inputs should generally have exactly two cases. Here is a summary of the basic features of the Case Structure:

1. Case Structures allow a choice of executing one of two or more portions of LabVIEW code, based upon incoming data.
2. Depending on the data that reaches the selection terminal , only one of the cases will be executed.
3. You must be careful to click on the wall of the Case Structure, not inside it, if you want to pop up on it.
4. The box in the top center with the two arrows lets you move from one case to another; pop up on the wall of the Case Structure for many more options.
5. You may pass a Boolean, integer numeric, string, or Enum (detailed soon) to a Case Structure's selection terminal .

6. Every Case Structure (except those with Boolean inputs) has a default case, which will execute if none of the other cases are a match to the input.
7. You may have two or more cases in a Case Structure. Although one case is possible, it is pointless. Case Structures with Boolean inputs use only two of their cases.

The Enum Constant object is found in the **Functions»Numeric** palette. It is a *ring*, and as with all LabVIEW text rings, you can type any number of distinct text values, and use the pop-up menu or <Shift-Enter> to add new values. The Enum is especially useful with the Case Structure, because it displays its text values at the top of each case—but its data type is really numeric, not text.

To summarize, decisions in LabVIEW are fundamentally handled in two different ways: (1) by the Select function, for very simple two-way decisions, and (2) by the Case Structure, for nearly all other decisions. Very seldom has any useful LabVIEW program been written without the Case Structure, as decision-making is fundamental to all but the most simple of programs.

1.8 The Sequence Structure

In LabVIEW, there are only two ways to control the order of events: dataflow (wiring nodes together) or the Sequence Structure. If you were to build the block diagram shown in Figure 1-48, you would have no guarantee that message 1 would appear before message 2.



Figure 1-48

This block diagram allows no control over the order of events.

The fact that one block diagram node is placed to the left of the other has nothing to do with the order of execution. Since these two nodes are not wired to one another, LabVIEW may arbitrarily choose to pop up either box before the other. The Sequence Structure, found in the **Functions»Structures** palette, has *frames* that act like the frames of a film; the first frame (number 0)

executes, then the second frame (number 1), then the third frame (number 2), and so on. The Sequence Structure may have one or more frames, but not zero frames.

Create a new VI. Drop a Sequence Structure on the block diagram (like you dropped the Case Structure earlier), then create the message 1 String Constant (in the **Functions»String** palette) wired to a One Button Dialog (in the **Functions»Time & Dialog** palette) as shown in Figure 1-49 within the Sequence Structure. Next, pop up on the Sequence Structure (like any structure, you must hit its edge, not its innards), so you can add a second frame with identical contents.

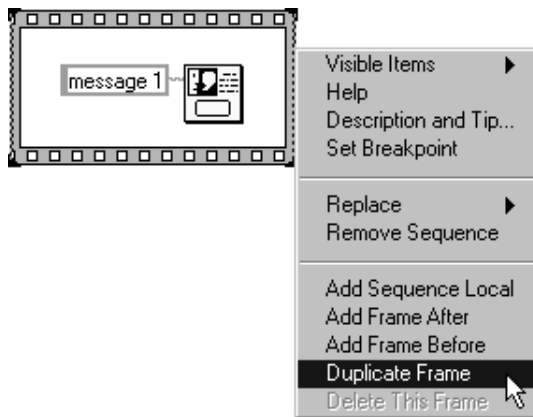




Figure 1-49

Use the pop-up on the Sequence Structure to add a second frame with identical contents.

Move to the second frame if needed by clicking the little right arrow  with the Operating tool , then change the string in the second frame of your sequence to message 2, so your two frames look like Figure 1-50.

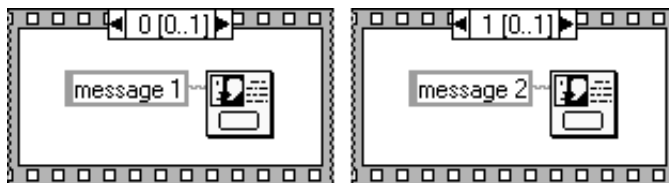


Figure 1-50

Two frames of a Sequence Structure are shown here at once, although you'll only see one at a time on a block diagram.

1.8 The Sequence Structure

57



Sequence Structure frames are always ordered 0, 1, 2, and so on, unlike Case Structure cases, in which you may skip numbers.

You are now guaranteed to have message 1 pop up before message 2, because the frames of a Sequence Structure always happen in order. Why did we wait until now to introduce the Sequence Structure? We recently introduced the One Button Dialog function, which is not usually wired to other parts of the block diagram, so it is inherently more difficult to control exactly when it executes in relation to the rest of the block diagram. Most of the other functions in LabVIEW are usually wired to one another, and wiring *also* controls the order in which nodes occur, thanks to dataflow. Since it's usually unnatural to wire these One Button Dialog functions to other parts of any block diagram, a Sequence Structure is often appropriate with these functions.

One common use for the Sequence Structure is timing sections of the block diagram. First, let me introduce the Get Date/Time In Seconds function, whose Help window contents are shown in Figure 1-51.



seconds since 1Jan1904

Get Date/Time In Seconds

Returns the number of seconds that have expired since 12:00 am, Friday, January 1, 1904 Universal Time.

Figure 1-51

The Get Date/Time In Seconds function Help window.

This function gives you a special number used often in LabVIEW to represent the current time and date. This was described in Section 1.2.3 as LabVIEW's standard Time & Date format. Why not always represent the time and date with a string? Computers manipulate numbers much more efficiently than strings. When it comes time to display this time and date to the user, *that's* when the number is converted to a string.

Build the VI shown in Figure 1-52, using these tips:

1. Create the front panel Digital Indicator `elapsed time` at the very end, by popping up on the Subtract function's output and selecting **Create»Indicator**.
2. As with the Case Structure, we are showing multiple frames of a Sequence Structure. There is really only one Sequence Structure in the block diagram.
3. That little box on the bottom wall of the Sequence Structure is a Sequence Local, produced by popping up on the bottom wall and selecting **Add Sequence Local**.
4. The two new functions shown are found in the **Functions»Time & Dialog** palette.

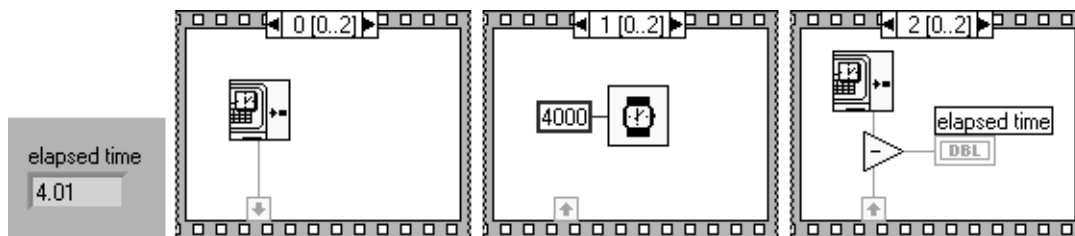



Figure 1-52

The Sequence Structure is used for timing purposes.

Run this one with execution highlighting , keeping in mind this will lengthen your `elapsed time` a bit. Even without this, your timing will likely be quite inaccurate unless you're using a real-time operating system, in which case it will be less inaccurate.

1.9 ARRAYS, LOOPS, GRAPHS, AND CHARTS

The last section focused on how to make decisions; we now shift gears and begin to talk about manipulating and displaying arrays of data.

1.9.1 Arrays: A Quick Summary

In general, even outside of LabVIEW, an array is a group of objects of the same type. In LabVIEW, an array is a group of objects having an identical data type. For example, you could have an array of numbers

[8, 6, 7, 5, 3, 0, 9]

or an array of strings

["remora", "tick", "leech", "Democrat", "telemarketer", "parasite", "lawyer"]

but not an array of *both* number *and* string data types. A *cluster* can handle different data types, but we'll just have to wait for the cluster.

It is helpful for most people (me, for example) to think of the array objects as being in a row, as in Figure 1-53, rather than scattered about at random as in Figure 1-54.

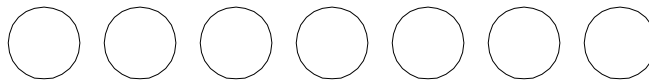


Figure 1-53

An array of generic objects, circles in this case, is shown.

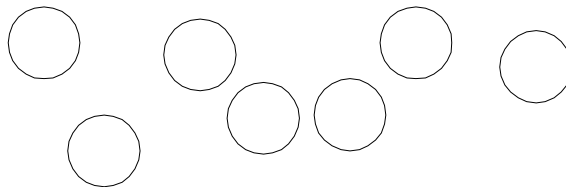


Figure 1-54

The same array from Figure 1-53 is shown, but scattered about—this can be confusing.

This is especially true when thinking about 2D arrays, which could be portrayed as in Figure 1-55.

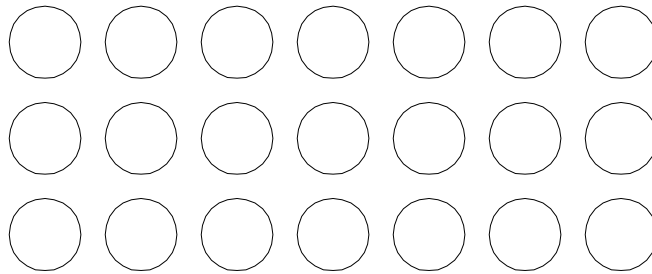


Figure 1-55

A two-dimensional array of generic objects, circles in this case, is shown.


To efficiently handle the arrays within LabVIEW (or any programming language), each element of any array is assigned an index. Taking for example our 1D (meaning one-dimensional; 2D will mean two-dimensional) array of numbers above—the indexes (also correctly called indices) would be as follows:

The array: 8, 6, 7, 5, 3, 0, 9

Its indexes: 0, 1, 2, 3, 4, 5, 6

Whenever you say “element N,” then “N” is implicitly the index of that element, *not the element itself*. So in this array, element 0 is 8, element 1 is 6, element 2 is 7, and so on. This array has a size, or length, of 7.

Now, let’s start using arrays in the context of LabVIEW. They can be used on the front panel or the block diagram. We’ll build the numeric array shown above, and soon illustrate the tricky parts of these front panel arrays. Create a new VI, closing any others you might have without saving them. Drop an Array onto the front panel from the **Controls»Array & Cluster** palette. You should see the image shown in Figure 1-56 on your front panel.

You should also see this on your block diagram: . This is an empty array, so we need to make it an array of something before we can use it. We could make an array of almost any LabVIEW data type, like strings or Booleans, but we’ll stick to numbers for a while, since they make the best demonstration. To make this an array of numbers, simply drop a Digital Control into the large gray square portion of the front panel array. As soon as you drop the Digital Control into that large gray square area, the array shrinks itself around that control to look like Figure 1-57.

1.9 Arrays, Loops, Graphs, and Charts

61

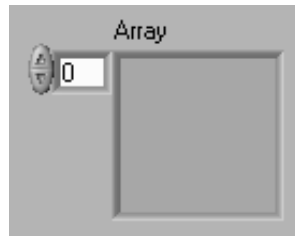


Figure 1-56

An array dropped onto the front panel from the Array & Cluster palette.

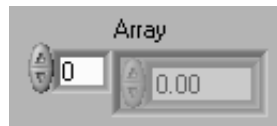


Figure 1-57

An array of numerics is shown on the front panel.



Arrays are sensitive about where they're clicked. If you pop up on an element inside the array, the pop-up menu refers to that element, not to the array. You might think you were popping up on the array, and wonder why array-relevant menu items were not present in the pop-up menu.


I've used a drawing program to blacken the rectangular area of the array in Figure 1-57 that is not relevant to the array where you pop up or click—this black area is relevant to the array's *element* (see Figure 1-58).



Figure 1-58

You'll probably never actually see this in "real" LabVIEW, but the "element" portion of a front panel array is drawn in black.

These front panel arrays can be confusing, so follow these instructions carefully.

1. With the Positioning tool , wiggle your cursor over the lower right-hand corner of the *element inside the array*, not the array, until the cursor looks like Figure 1–59.

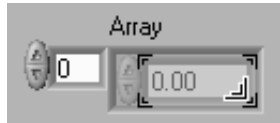


Figure 1–59

The cursor is placed over a corner of an *element* of the array, not the array itself.

2. While the cursor still looks like that, *drag* that corner of the array's element to the left a bit so your array looks like Figure 1–60.

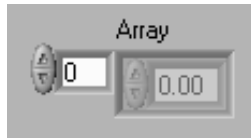



Figure 1–60

The cursor shown in Figure 1–59 was dragged to the left a bit to shrink the size of the array's element.

3. With the Positioning tool , wiggle your cursor over the lower right hand corner of the *array*, not the element inside the array, until the cursor looks like Figure 1–61.

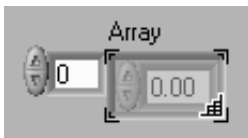




Figure 1–61

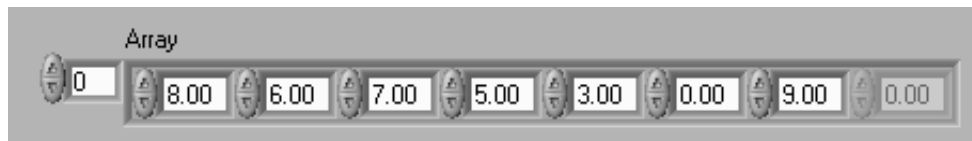
In contrast to Figure 1–59, the cursor is placed over a corner of the *array*, not its element. Very slight mouse movement will differentiate these two cursor appearances!

4. With the cursor looking like it does just above, drag the array corner to the right so you have eight elements showing, as in Figure 1–62.

**Figure 1-62**

If the cursor shown in Figure 1-61 were dragged to the right, more elements would appear.

5. Switch to the Operating tool  or the Labeling tool , and enter the values as shown in Figure 1-63.

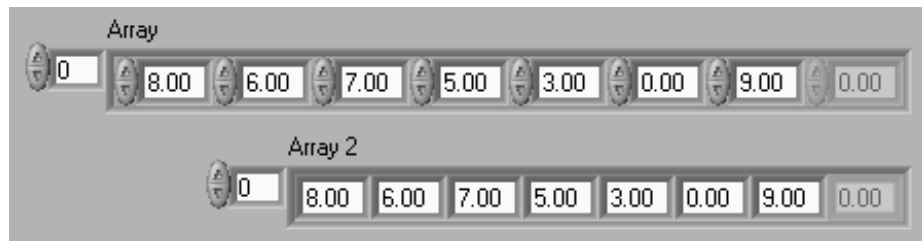
**Figure 1-63**

The empty array from Figure 1-62 is now filled in.

6. Make these values default, as you'll use them throughout this chapter, by using the array's **Data Operations** pop-up menu item.

You have just created your first array! Now let's use it.

To get a quick idea of the things you can do with arrays, clone this array just below itself. You must take care to click on the array, not the element inside, as illustrated earlier in Figure 1-58. After you have the clone, Array 2, change it to an indicator (by popping up on it *or* one of its elements), so that you have the image shown in Figure 1-64.

**Figure 1-64**

A "control" array is contrasted to an "indicator" array.

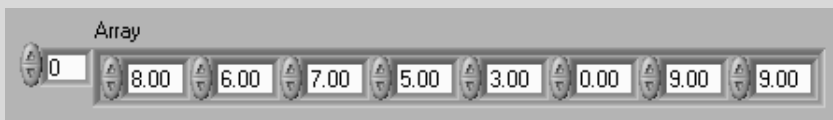


Suppose you were to try to grow the array to show eight elements with the element selected instead of the array. In other words, if your cursor looked like this instead of this when you did the drag to show eight elements, you might wind up with something like this:



When this happens, which it will eventually if you continue to use LabVIEW, just use the undo feature, then go back and do it right.

Suppose you accidentally typed in eight elements instead of seven, like this:



If you noticed this mistake right away, you could undo it, but if you didn't notice it until much later, you could use the array's **Data Operations»Empty Array** pop-up menu item to empty the array, then type the numbers in again. If the array were much larger, you could use the array's **Data Operations»Cut Data** pop-up menu (read the LabVIEW manual or Section 1.9.4 to find out how it works).

Now, on the block diagram, arrange and wire your two array terminals as shown in Figure 1–65 (hint: you can wire *first*, then pop up on the middle of the wire, then select the Increment function from the **Insert»Numeric** palette).



Figure 1–65

This will increment each element of the array.

Looking at Array 2 on the front panel, run the VI and notice all the elements of Array are incremented by one and placed into Array 2, as shown in Figure 1–66.

1.9 Arrays, Loops, Graphs, and Charts

65



Figure 1-66

Each element of the upper array in Figure 1-64 are incremented by one.

Once you get this working, save the VI as `Increment 1D Array.vi`.

Note When you're starting out, it is easy to drop a numeric front panel control and think it's an array. The terminals look similar on the block diagram, but the array's terminal has square brackets while the numeric control's does not. Compare the terminals, which I've explicitly labeled here:

Array

Numeric

The same confusion can apply to array versus numeric indicators, as well as controls. Fortunately, their front panel objects look very different:

Array

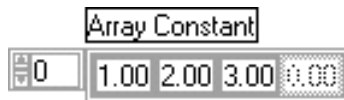
Numeric

Notice how these array wires (as in Figure 1-65) are **two** pixels thick rather than one—this is indicative of a 1D array. Also, notice the square brackets around the DBL data type; this is also indicative of an array. If you had an

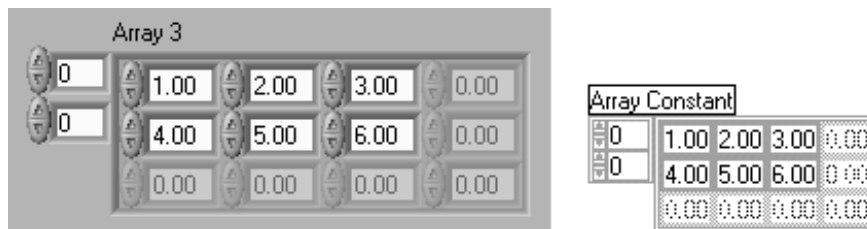
empty array, it would look like this: , which is useless until you drop some sort of control or indicator into it on the front panel.

Arrays can also be created as constants on the block diagram; the Array Constant is in the **Functions»Array** palette. You drop objects into this Array Constant on the block diagram much as you did on the front panel, so it might look like Figure 1-67.

You may want to use 2D arrays. Popping up on an array control on the front panel, or an array constant on the block diagram, you will see an **Add Dimension** menu item. Figure 1-68 shows examples of 2×3 2D arrays (2 rows by 3 columns of data); one is on the front panel, the other is a constant on the block diagram.

**Figure 1-67**

An array on the block diagram is shown, in contrast to the previously-shown front panel arrays.

**Figure 1-68**

Two-dimensional arrays are shown on the front panel and the block diagram.

I've shown extra empty array elements on either dimension in the arrays in Figure 1-68 just to prove they are 2×3 . Without these elements showing, we might be looking at subsets of much larger arrays.

There are many array functions at your disposal in the **Functions»Array** palette. Take some time to experiment with them as they're described in this section; many of them should come in quite handy for even your most basic programs. If you have shelled out an insane amount of money for your version of LabVIEW, rather than just a large amount, you may have "advanced analysis" functions, which are prepackaged statistical, filtering, and mathematical functions for your arrays. If available, these advanced analysis functions would be found in the **Functions»Mathematics** palette.

Here are some of the more commonly used array functions (whose Help panels I've copied directly from LabVIEW) with which you should become familiar. There are quite a few of them, but they are very fundamental to any LabVIEW program, so you should certainly experiment with them as you read about them. To do this easily, create a new VI per experiment, drop the array function being described, then pop up on the various terminals, selecting **Create Control** for the inputs and **Create Indicator** for the outputs. This technique will automatically create a front panel object with an appropriate data type.

1.9 Arrays, Loops, Graphs, and Charts

67

Remember that as you're working through this book, you may close all VIs after each section if you want, or your screen will become quite cluttered.

First, the Array Size function, shown in Figure 1-69, tells you how many elements there are in an array.



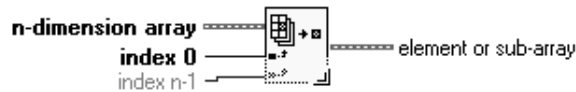
Array Size

Returns the number of elements in each dimension of input. Input can be an n-dimensional array of any type.

Figure 1-69
The Array Size function.

As an example, if you called this function on the array [8, 6, 7, 5, 3, 0, 9], it would return 7, because there are seven elements in this array.

The Index Array function, shown in Figure 1-70, can pick out any element of an array. This is taken care of automatically by passing a wire through a For Loop, provided you want to access the elements in order. Here's LabVIEW's description of the Index Array function:





Index Array

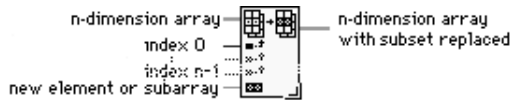
Returns an element or subarray of array at index. Leaving an index unwired will slice out a cross-section of the array along that dimension.

The function will automatically resize to show one index per input array dimension, but you may resize by hand to show more outputs to access multiple elements in one operation. If all indices for the second or subsequent outputs are unwired, function will auto-increment the indices for the previous output in a sensible manner.

Figure 1-70
The Index Array function.

When you drop this function from the **Functions»Array** palette, it defaults to a 1D array data type . It can be grown to handle additional dimensions with the Positioning tool , as seen in Figure 1-70. As an example, if

you called this function on the array [8, 6, 7, 5, 3, 0, 9], and you passed in a 0 for the index; it would return 8. If you passed in a 1 for the index, it would return 6, a 2 for the index would return 7, and so on.



Replace Array Subset

Returns a portion of array starting at index and containing length elements. Array can be an n-dimensional array of any type. Function will automatically resize to contain as many index/length inputs as dimensions in array.

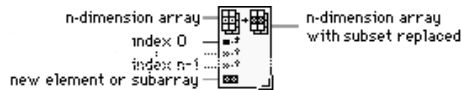
Figure 1-71

The Array Subset function.

The Array Subset function, shown in Figure 1-71, lets you pick out a subset of an array (a portion of the input array, if you want).

For example, if you had the array [8, 6, 7, 5, 3, 0, 9] and you wanted to pick out the subarray [7, 5, 3], you would need to specify an index of 2 (because 7 is the element having an index of 2) and a length of 3.

The Replace Array Subset function, shown in Figure 1-72, is fairly self-explanatory, wherein you can choose any subset *or element* of an array and replace it with something else.



Replace Array Subset

Replaces the element or subarray in array at index. If an index is unwired, it describes a slice of the array along that dimension. For example, to select a whole column in a 2D array to be replaced, wire the column index but leave the row index unwired. The base type of the new element or subarray must be of the same type as the input array. Function never resizes original input array; subarrays which are too large to fit will be truncated.

The function automatically resizes to show one index input for each dimension in the input array, but may also be resized by hand to show multiple element/indices sets. This allows multiple portions of an array to be replaced in one operation.

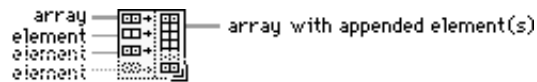
Figure 1-72

The Replace Array Subset function.

The Build Array function, shown in Figure 1-73, is often useful for building arrays. Be warned that it is slow for building very large arrays element-by-element.

1.9 Arrays, Loops, Graphs, and Charts

69



Build Array

Build an n dimensional array out of n and $n-1$ dimensional inputs (elements or subarrays). Inputs are concatenated in order. If all inputs are the same (n dims), you can popup and select 'Concatenate Inputs' to concatenate them into a longer n -dim array, or deselect it to build an $n+1$ dim array.

Figure 1-73

The Build Array function.

Be sure to review all the array functions—they are really quite powerful. Arrays are inherent to most DAQ, as will be seen in Chapter 3, so the more you know about these functions, the more power you have.



The Sort Array function is especially powerful because, given an N element array, it sorts in order $N \cdot \log(N)$ time, not N^2 . This is a huge timesaver for large arrays, should they need sorting.

1.9.2 Loops: A Quick Summary

Any time you want to perform the same action, or nearly the same action, repeatedly, loops are the way to get this done. Loops often work on array elements one at a time.


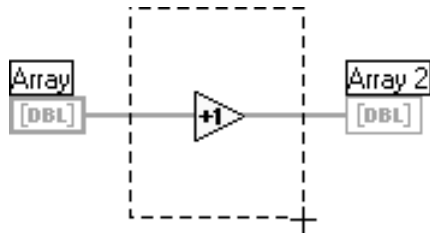
Open your Increment 1D Array.vi from the last section, if it's not already open. On the block diagram, spread out the three objects about an inch apart horizontally, then select the For Loop from the **Functions»Structures** palette so that your cursor looks like this: . With that cursor, drag a For Loop around the Increment function, but not around the front panel terminals, as in Figure 1-74.

Figure 1-74



A For Loop is being drawn around the Increment function.

Then, release the mouse button so you have the image shown in Figure 1-75.

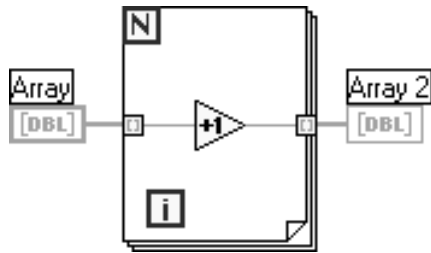




Figure 1-75

Once the mouse button is released from Figure 1-74, a For Loop is drawn.

First, run the program with [8, 6, 7, 5, 3, 0, 9] in your front panel Array control, so you can see that it works, then I'll explain what's going on.

Before you added the For Loop, the Increment function took care of *indexing* the array, meaning it *stepped through each array element* to perform the math. The For Loop explicitly indexes the arrays with its tunnels, which are the little boxes  where the array passes through the wall of the For Loop. Run the VI from the block diagram with execution highlighting on , and you'll see indexing in action. You wouldn't be able to see this without the For Loop, as the indexing would happen in one step inside the Increment function.

The For Loop causes everything inside to be executed N times. In this case, N is determined by the size of our input array, which is seven. That means that everything inside the loop happens seven times. For each iteration, 0 through 6, that particular element of the input and output arrays are operated upon. The first iteration of the loop takes the 8 from element 0 of the input array, and puts the resulting 9 into element 0 of the output array. The second iteration takes the 6 from element 1 of the input array, and puts the resulting 7 into element 1 of the output array. This continues for all seven

iterations, until the output array has been completely built. Close your `Increment 1D Array.vi` without saving it.

I will now describe the parts of the For Loop. The blue box with the **N** is the For Loop's *count terminal*, and the blue square with the **i** is the *iteration terminal*, which starts at 0 and increments per loop iteration. If you don't have any indexing array tunnels, as shown in Figure 1-75, you can use the count terminal to determine how many iterations the For Loop will make by wiring an integer to it. If you do have indexing array tunnels coming into the For Loop, the count terminal will tell you how many times the loop will iterate. Let's build a VI to demonstrate these two For Loop terminals.

Use the **File»Save As...** menu item to save this VI as `Count Terminal Demo.vi`. Delete the front panel array control (controls can only be deleted from the front panel) and the increment function, then hit <Ctrl-B> to delete all broken wires, so you're left with the image in Figure 1-76.

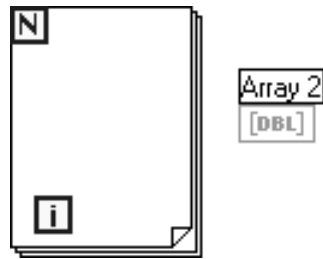


Figure 1-76

A block diagram is being built with only a For Loop and the terminal from a front panel array.

We will now build a new VI, as shown in Figure 1-77. Pop up on the count terminal **N** and select **Create Control**—label it **N**. Add the Multiply function in the middle right of the For Loop, then pop up on the upper left terminal of the Multiply function and select **Create Constant**, giving it a value of 3. Save this VI again (as `Count Terminal Demo.vi`), then run it from the front panel with **N** set to 0, 1, 2, then 3. You should see now that the number of iterations of the loop is determined by **N**, and the size of the output array as built through an indexing tunnel is determined by the number of iterations of a For Loop.

The iteration terminal starts at 0 on the first iteration, becomes 1 on the next, 2 on the next, and so on. Run this VI with execution highlighting while watching the block diagram if you're not clear on how it works. As it is a 32-bit integer, the iteration terminal will roll over to -2,147,483,648 right after

2,147,483,647, then count right up to zero again and keep repeating this cycle. Keep this in mind if you think the loop will ever get this far!

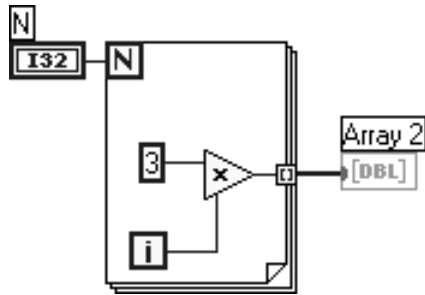



Figure 1-77


The block diagram from Figure 1-76 continues to be built.

Notice how the wire changes from one to two-pixels-thick as it passes through the right wall of the loop. This is consistent with the idea that each element of the array is being operated upon per iteration of the loop.

A For Loop must have either its count terminal  wired, as in Figure 1-77, or an indexing array coming into a tunnel, as shown in Figure 1-75. If the For Loop has both of these wired, the number of iterations will be the smaller of these two numbers: the number received by the count terminal or the size of the array coming into the indexing tunnel. With multiple incoming indexing array tunnels, the For Loop uses the smallest-sized incoming array to determine its number of iterations.

The other type of LabVIEW loop is the While Loop. Open your `Increment 1D Array.vi` from the last section, whose block diagram should look just like Figure 1-65, then build the block diagram shown in Figure 1-78, noting that you will need to pop up on the array tunnels and change them to indexing tunnels.

By default, passing a wire through a While Loop's tunnel does not index an array, while a For Loop's does.

Notice the broken **Run** button (because that new object inside  is unwired). To fix it, add the objects shown in Figure 1-79 to the block diagram (notice the resizing and moving around of objects, so things don't look too crowded).

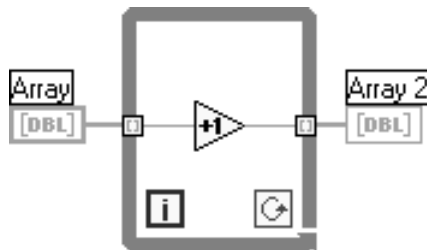


Figure 1-78

A While Loop can have indexing tunnels like a For Loop.

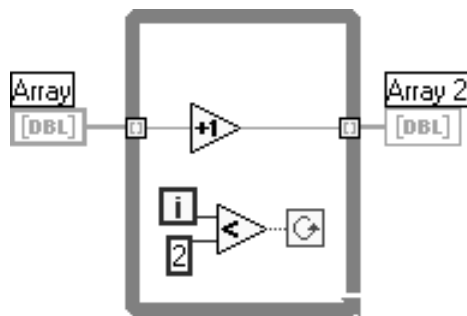

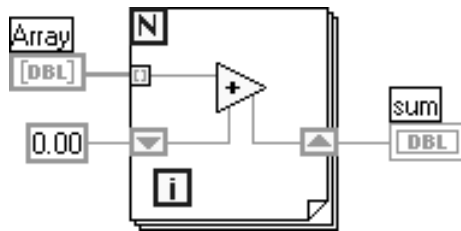


Figure 1-79

The While Loop block diagram from Figure 1-78 is now changed to a working program.

Run the program, and see that now your output only has three elements! This is because the While Loop iterates only until its *conditional terminal*  is False. On iterations 0 and 1, the condition “ $i < 2$ ” is True. On the third iteration, when i is 2, the value hitting the conditional terminal is False, so the loop stops there. If you have any trouble understanding this, run the loop with execution highlighting on.

Shift registers are usually the preferred way of passing data from one iteration of a loop to the next. They are created by popping up on a vertical wall of a loop, then selecting the **Add Shift Register** menu item. They work with both For Loops and While Loops. Figure 1-80 is an example of how you could sum the elements of an array using a shift register.

**Figure 1–80**

An example of how to sum the elements of an array using a shift register.

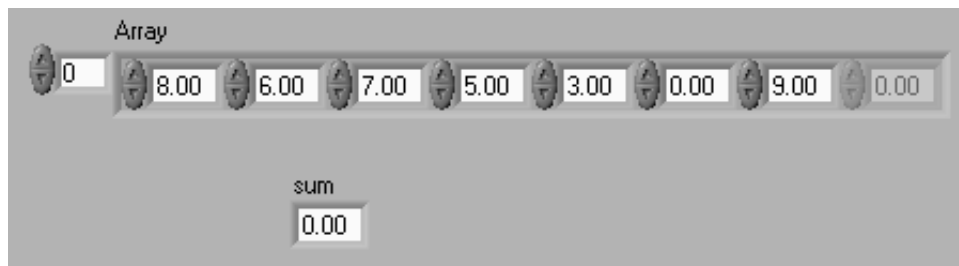
Although the shift register above appears to have a box on both sides of the loop, think of it as just one unit, because it can only hold one number at a time.



You can grow the left side of a shift register downward so it holds more numbers, but we won't cover that in this book, as it's not often used in DAQ.

Here's how that shift register works: Whatever number shows up in the right box at the end of one iteration of the loop will come out of the left box on the next iteration. The initial value of the shift register is determined by whatever is wired to the left box; in Figure 1–80, it starts out at 0.00.

Build the VI shown in Figure 1–80, and run it with execution highlighting on, and the array as shown in Figure 1–81. In case it is not obvious by now, sum is a Digital Indicator. Before running, your front panel might look something like Figure 1–81.


**Figure 1–81**

The block diagram from Figure 1–80 could have this front panel.

1.9 Arrays, Loops, Graphs, and Charts

75

Unless you're an advanced user, you probably won't want to leave the left shift register unwired. If you do, the shift register actually remembers its number from one run of the overall VI to the next! This should seem weird, since so far, VIs have not been able to remember anything between runs. This memory feature can lead to great confusion for beginners, because the VI may behave differently every time you run it, since the uninitialized shift register may contain different values at the beginning of each run; so unless you know what you're doing, initialize all shift registers. Other common ways to remember numbers from one VI run to the next are by using local or global variables, covered later, or in files (on disk), covered even later.

The above was just a concocted example to clearly demonstrate the shift register. To sum the elements of an array, you would normally use the Add Array Elements function, , oddly found in the **Functions»Numeric** palette, not in **Functions»Array**.

A 2D array can be manipulated or created by putting one loop inside the other with indexing tunnels. Figure 1-82 is the block diagram and front panel of a simple example of creating such an array.

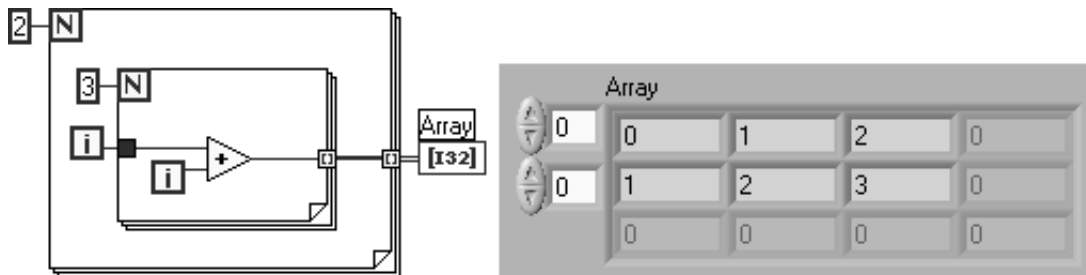




Figure 1-82
This VI illustrates a two-dimensional array.

Note the difference between an indexing tunnel  and a non-indexing tunnel . Arrays are sometimes smart enough to index when you want them to and not index when you don't, whenever you pass a wire through their walls. But if you wanted to force indexing one way or the other, you could pop up on any loop's tunnel and do so, as seen with the While Loop previously.

I generated the 2D array on the front panel by popping up on the far right tunnel  and selecting **Create»Indicator**.

Build such a VI, then run it with execution highlighting turned on. On the first iteration of the outer For Loop, the wire entering the left indexing tunnel from the left is a simple number, not an array. After the inner For Loop executes three times, the 1D array [0, 1, 2] appears on the wire between the two tunnels, and fills up the first row of the 2D output Array. The second (and final) iteration of the outer For Loop places the 1D array [1, 2, 3] on the wire between the two tunnels, which fills up the second row of the 2D output Array.

A good way to build a really large array in LabVIEW is to first create a full-sized array of zeros or NaNs, then replace each element with the data. This circumvents the inherent slowness of building a large array element-by-element with the Build Array function. This VI, although it could perform its duty without the shift register or Replace Array Subset function, demonstrates how to replace each element of an array with the numbers 0, 1, 2, and so on. Build the VI shown in Figure 1-83, using the Replace Array Subset function on the block diagram and popping up on the *wall* of the For Loop to create the shift register, and save it as Simple Array Shift Register.vi.

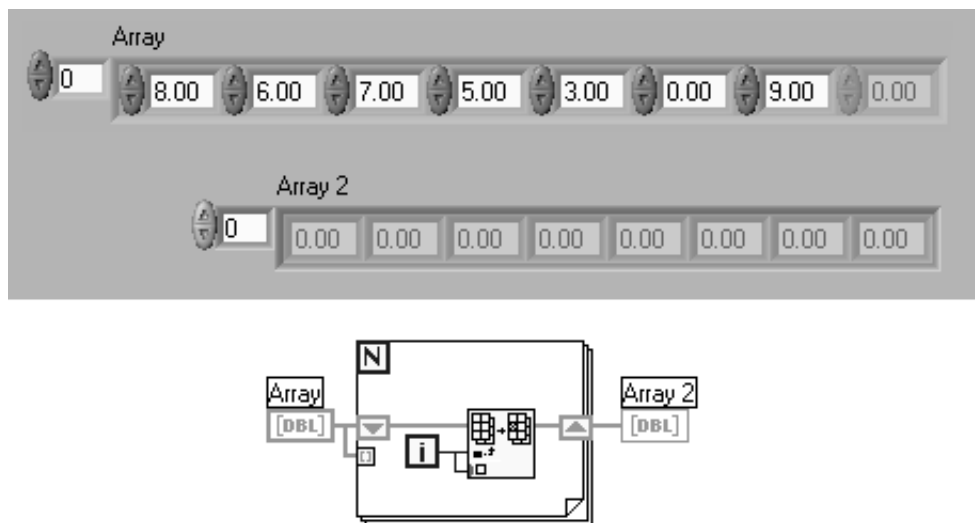


Figure 1-83

This VI illustrates the Replace Array Subset function and a shift register.

1.9 Arrays, Loops, Graphs, and Charts

77

Shift registers are especially helpful when you pass an entire array into them.



If you're an experienced programmer who's new to LabVIEW, you might think this is a bad thing, as you're visualizing many numbers being moved around every iteration of the loop; but such is not necessarily the case. If LabVIEW can perform its array-in-a-shift-register duties without moving the array, it will, as in the VI shown in Figure 1-83.

The contents of the shift register after each iteration of the For Loop is illustrated here:

[0, 6, 7, 5, 3, 0, 9]

[0, 1, 7, 5, 3, 0, 9]

[0, 1, 2, 5, 3, 0, 9]

[0, 1, 2, 3, 3, 0, 9]

[0, 1, 2, 3, 4, 0, 9]

[0, 1, 2, 3, 4, 5, 9]

[0, 1, 2, 3, 4, 5, 6]

It would be impossible to use execution highlighting to see these values, since execution highlighting will not show arrays.

Suppose we had not used the shift register, but wired the array directly through the For Loop wall, then disabled indexing on the two tunnels. As soon as we disable indexing, the For Loop doesn't know how many times to iterate, so we could add another tunnel whose only purpose is to give the For Loop a count. Our block diagram would look like Figure 1-84.

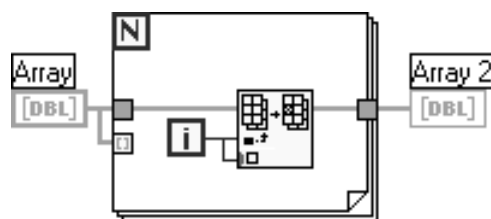


Figure 1-84

A nonsense VI, illustrating why the Shift Register of Figure 1-83 is useful.

If we ran this nonsense VI, it would not do the same thing as our last VI because the same input array, [8, 6, 7, 5, 3, 0, 9], would appear on the upper left tunnel on each iteration of the loop. For this reason, only the last element of the incoming array would be replaced in “Array 2.” Remember that the shift register is the best way to pass any information from one iteration of a loop to the next, and that includes entire arrays.

Exercise 1.1

Brace yourself: This exercise covers quite a bit of material.

Suppose you have an array of seven 32-bit integers, [8, 6, 7, 5, 3, 0, 9], on a front panel. Shown in Figure 1–85 is a front panel in which the elements of such an array control are reversed and placed into an array indicator; I’ve changed the array elements’ representation to I32, a 32-bit integer, so you don’t see any decimal points.



Figure 1–85

Two front panel arrays with I32 elements.

The simplest way to reverse the order of elements of this array would be to use the Reverse Array function, so that you get the above result from the very simple block diagram shown in Figure 1–86.

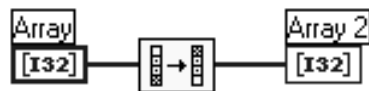


Figure 1–86

The Reverse Array function is illustrated.

But suppose LabVIEW did not have the Reverse Array function. Could you build a LabVIEW VI that would reverse the array, as shown in Figure 1–85, without using the Reverse Array function? Try to do this in LabVIEW before reading the following hints.

1. Use a For Loop that iterates once per array element.
2. Although you could use a shift register, use tunnels on the For Loop—this will simplify your VI.
3. Use arithmetic that can perform addition and subtraction.
4. You'll want to index the array, either with an indexing tunnel or the Array Index function.
5. You will need to determine the Array's size, either with the Array Size function, or the count terminal in conjunction with an indexing tunnel.

Two Solutions

There are many ways to solve this problem! Figure 1–87 shows the easiest solution to understand, in my opinion (or least difficult, if you're brand new to LabVIEW).

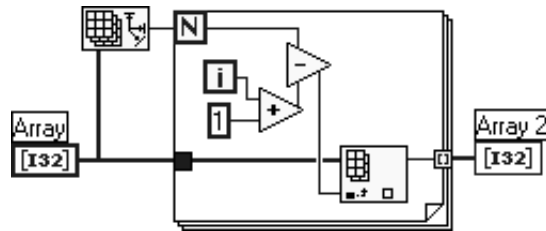


Figure 1–87

One solution to this exercise is shown.

The Array Size function is used to tell the loop to iterate seven times here, since our input array has seven elements. For each array iteration i , where i ranges from 0 to 6, the Index Array function picks the element number $(7-i-1)$, which is the element needed to build a reversed array in the tunnel on the right. The Add Function and Subtract function are responsible for computing $(7-i-1)$. Figure 1–88 shows another way to solve the problem, optimized for size, not clarity.

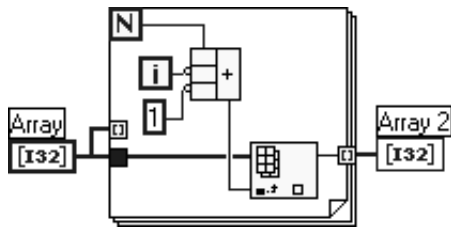


Figure 1–88

A second solution to this exercise is shown.

Both of the solutions above may be described with the picture in Figure 1–89, where the input array is on top and the output array underneath.

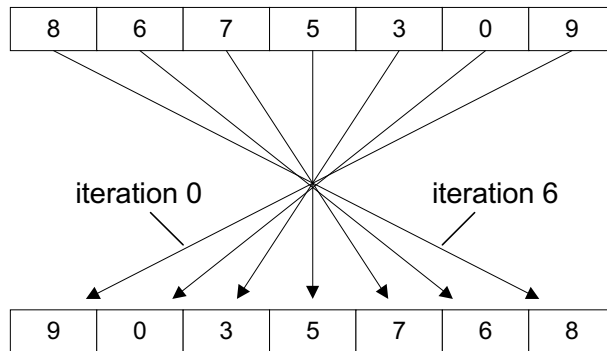


Figure 1–89

A graphical illustration of the underlying data manipulation of this exercise is shown.

While I was building this VI, I found it helpful to wire a 1 (a Numeric Constant) to the For Loop's count terminal, then run the VI with execution highlighting to make sure my math was right. I did this again with a 2 instead of a 1. Obviously, with just one tiny math error anywhere, this VI probably won't work.

1.9.3 Graphs and Charts: A Quick Summary

Now that we have, I hope, a general understanding of arrays and loops, it's time to discuss LabVIEW's graphs and charts. Create a new VI with an array

1.9 Arrays, Loops, Graphs, and Charts

81

of Digital Controls as shown in Figure 1–90 (you can drag an array from one of last section's front panels), and a Waveform Graph (found in the **Controls»Graph** palette; ignore the plot on the graph for a moment).

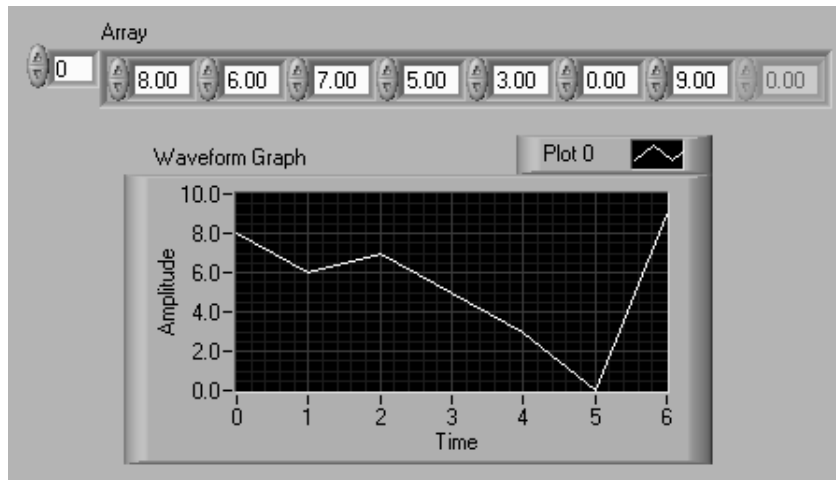


Figure 1–90

A Waveform Graph (with an arbitrary plot) is shown.

On the block diagram, wire the array directly to the Waveform Graph, as shown in Figure 1–91.




Figure 1–91

A block diagram used to show the graph in Figure 1–90 is shown.

Run the VI, and you should see something like the plot in Figure 1–90 on your front panel. I've shrunk my graph a bit, so my numbers on the axes might be different than yours. With graphs, you have little control over the exact spacing of these numbers.

The entire array is sent from the numeric array to the graph. You can change some of the numbers in your array and run it again to verify that it works.

This item, **Plot 0** , is the graph's *Plot Legend*. Pop up on it, and pick the point style shown in Figure 1–92.

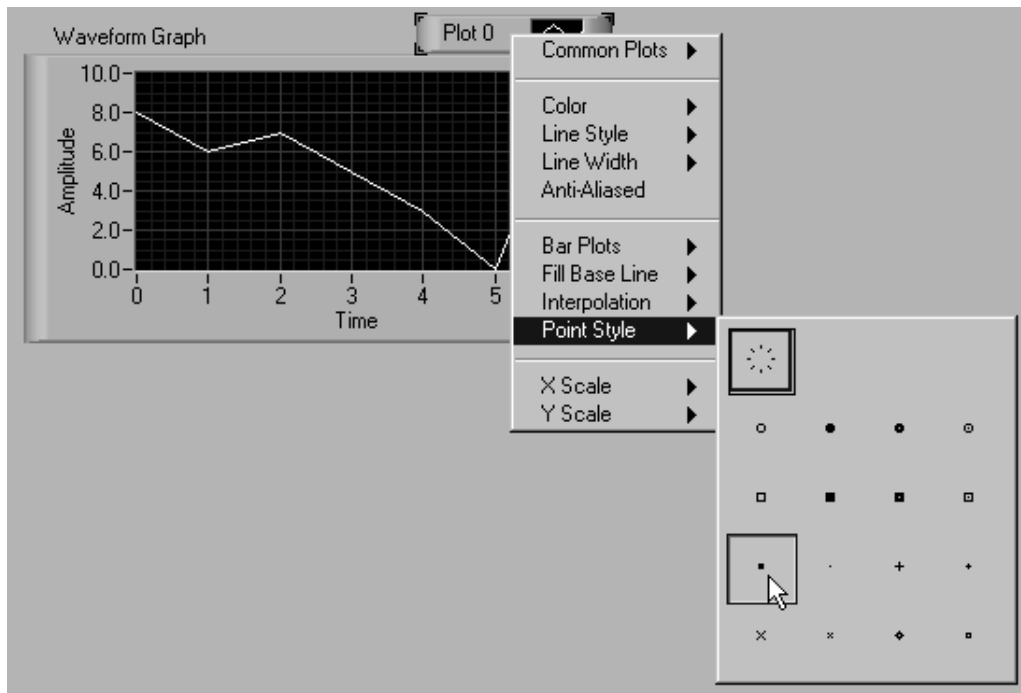


Figure 1-92

My favorite point style is this one—it is as small as possible while still being easily visible.

As Figure 1-93 shows, you can now easily see each individual point.

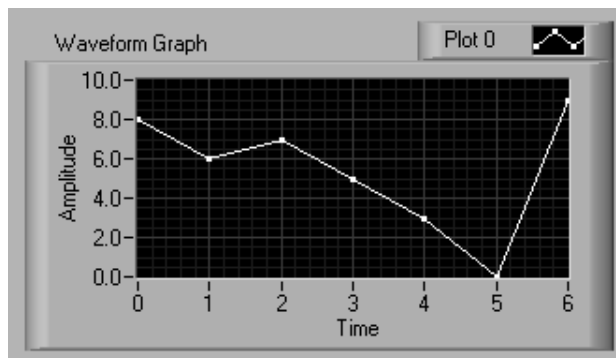


Figure 1-93

The point style chosen in Figure 1-92 is perfect for most applications.

1.9 Arrays, Loops, Graphs, and Charts

83

A chart is very similar to a graph, except it is fundamentally designed to take one point of data at a time rather than an entire array. There are many exceptions to this rule, but that's the basic idea.

Go to your block diagram and add the items shown in Figure 1-94 by dragging a new For Loop across the wire and inserting the Wait Until Next ms Multiple function from the **Functions»Time & Dialog** palette.

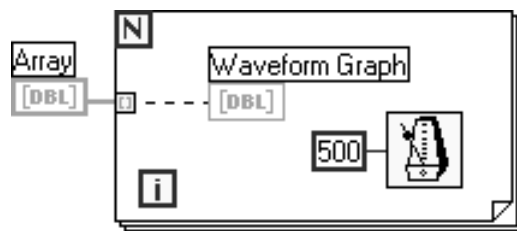


Figure 1-94
Preparing to build a simple DAQ application.

The Wait Until Next ms Multiple function is designed to work inside loops to make them run at the given rate. Realize that unless you're using a real-time LabVIEW setup, you cannot count on this timing to be accurate, since most operating systems are not truly real-time.

On the front panel, replace the Waveform Graph with a Waveform Chart by popping up on it and using the **Replace** menu item, and finding the Waveform Chart in the **Controls»Graph** palette. By doing this, you do not need to rewire your block diagram, and the broken wire you see above will fix itself. This is because a Graph does not accept simple numbers, whereas a chart does. The graph, as mentioned above, is geared towards arrays, not numbers.

Run the new VI, and notice how the points are sent to the chart one at a time. If you did not have the timing function in the block diagram, the For Loop would run full speed, and unless you're running on a 1 MHz computer or slower, all of the points would appear to show up on the chart at the same time.

1.9.4 Arrays and Loops: Details and Features

Front Panel Issues

The box with the 0 in it, towards the upper left in Figure 1–95, is the array's *index*. It specifies the index of the first array element shown (always the closest element to the index)—in this case, it means that the 8.00 has index 0. Both arrays shown in Figure 1–95 contain *exactly the same data*, but if you didn't notice that the index was 3 in the second array, you might think that its data was simply 5, 3, 0, 9.

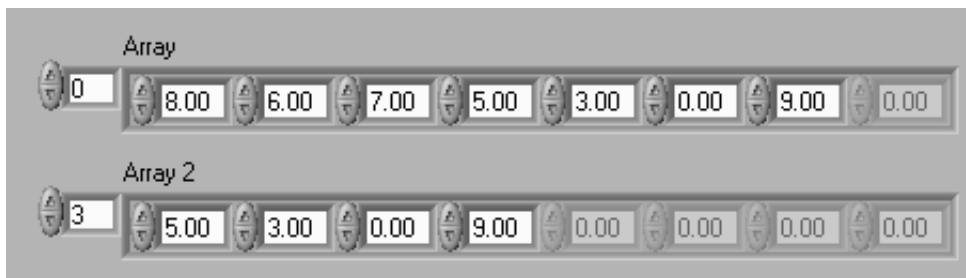


Figure 1–95

The index of an array can be very confusing when not zero.

You may cut, copy, or paste sections of the array by popping up on it and using the **Data Operations** menu, the same menu you would use to empty the array. You can select the entire array, or a subset of the array. This menu can be tricky to understand at first. In order to select a subset of an array, you must define your selection by using the array's index. First, change the index and select the start of your selection, then change the index and select the end of your selection. When you paste, the paste begins at the element shown by the index.

Block Diagram Issues

Speed is often an issue when you're using arrays, because arrays can contain an arbitrarily large amount of data. As mentioned previously, a very common mistake is to build a big array element-by-element, using the Build

Array function. Judging from its name, I can see why a beginner would choose to use the Build Array function to build an array. Here's what can happen. Suppose you're brand new to LabVIEW and you want to create an array of 1,000,000 elements (make the 1,000,000 proportionally larger for the faster computer you probably have, long after I'm typing this). Which of the methods shown in Figure 1-96 do you think is faster?

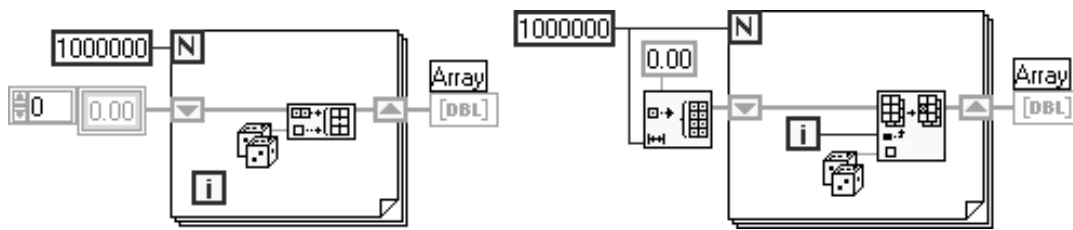


Figure 1-96

Two ways to create an array of 1,000,000 elements

On my Pentium II, 400 MHz computer, the method on the left takes about 11 seconds, compared to 2 seconds for the method on the right! This is because the array is repeatedly being grown, and this is slower due to the underlying memory management mechanism. This problem is not specific to LabVIEW, but to all programming languages. The Build Array function is often fine for smaller arrays of a few hundred elements or less, but be aware of this speed issue with large arrays.

Complicated arrays can also slow things down, particularly when you have array elements which themselves contain arrays (or strings). In general, the more complex your data type, the more likely it is to decrease your execution speed.

Graphs and Charts: Details and Features

Much of what is said about graphs in this section will also apply to charts. It's cumbersome to always say "graphs or charts," so this section will often just say "graphs" when referring to either a graph or a chart. A chart is really just a special type of graph.


There are quite a few other issues that we will briefly touch upon concerning graphs. First, realize they are *polymorphic*, meaning you can wire different


data types to them. LabVIEW's help utilities document the specific permissible data types per graph. A sneaky way to see at least one of your graph's data types is to go to the block diagram, pop up on the graph's terminal (even if unwired), then select **Create»Constant**.


Table 1.9 describes the basic types of graphs.

Table 1.9 Basic Types of Graphs

Graph Type	Description
Waveform Chart	Fundamentally displays one point, or group of points, at a time; has internal memory called <i>history</i> .
Waveform Graph	Displays an entire array or arrays of points with equally spaced x-values; has no internal history.
XY Graph	Displays an entire array or arrays of points with arbitrarily spaced x-values; has no internal history.
Intensity Chart	Takes a 1D array of data points in which each data point corresponds to a rectangular area of the chart; has history.
Intensity Graph	Takes a 2D array of data points in which each data point corresponds to a rectangular area of the graph; has no history.
Digital Waveform Graph	Useful for displaying digital data, which is a type of DAQ signal to be discussed in a later chapter.
(many others)	Will vary widely with future versions of LabVIEW, I'm sure.

Suppose you were acquiring data in some experiment, one point every second, and displaying it on a chart. Since we have not yet discussed how data is acquired, we'll simply use LabVIEW's random number generator  to give us simulated data, where each data point will be in the range 0.0 to 1.0. Go ahead and build this VI in LabVIEW, as it's being described, using these tips:

1. Begin by dropping a Waveform Chart, and shrinking it as shown in Figure 1-97.
2. That's a label on the front panel below the chart, which you can get by simply selecting the Labeling tool  and typing no autoscaling, which will be described soon enough.

3. The random number generator is found in the **Functions** »**Numeric** palette.
4. The square wired to the **stop** button's terminal is the While Loop's conditional terminal, , shown in Section 1.9.2, but we have inverted its logic by popping up on it and changing it to **Stop if True**. In other words, the loop stops if this terminal receives a Boolean value of True, instead of False as it did before.

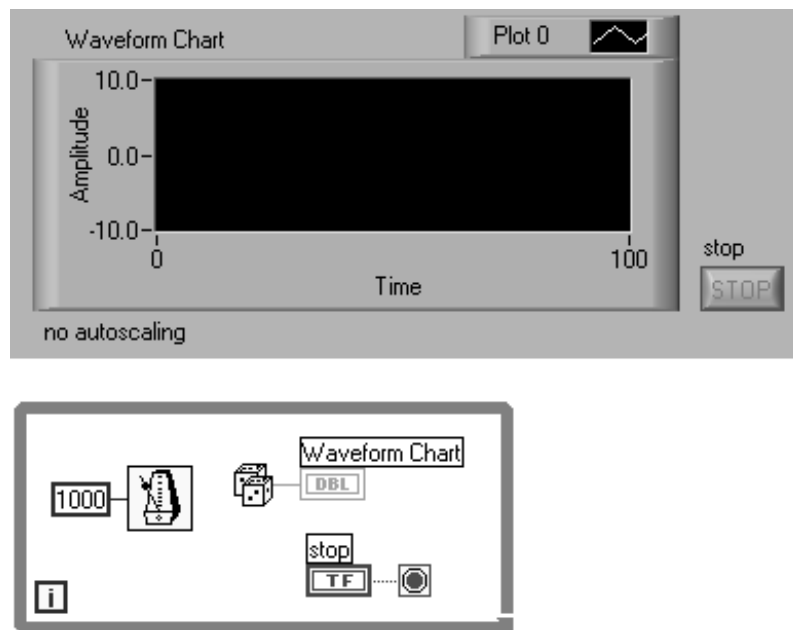


Figure 1-97

This is the format of the simplest LabVIEW DAQ application, but here we simulate voltage with a random number.

If this were done in a DAQ context, you might want to pass the Waveform data type instead of a numeric data type, as in Figure 1-32.

Pop up on the **stop** button on the front panel and notice that its **Mechanical Action** is set to one of the *latching* types. Latching is only useful in loops. Latching means that once the button is read from the block diagram, it returns to its default value on the next iteration of the loop. This is necessary to pop the button back up after you press it. The logic shown above keeps the

value on the conditional terminal False for each loop iteration except the one on which the `stop` button is pushed.

The 1000 wired up to the Wait Until Next ms Multiple function causes the loop to wait one second (1,000 milliseconds) between iterations.

Now run the VI for about 10 seconds, so you see something like Figure 1-98 on your front panel.

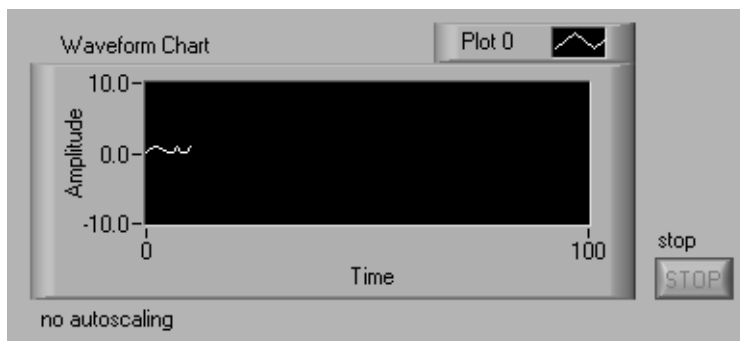


Figure 1-98
11 data points with no autoscaling.

This is a chart of about 10 points, but it's a bit tough to see this data as it's compressed into a tiny portion of the chart. First, let's expand the data on the y-axis. There are two ways to do this: You can choose the Operating tool (or the Labeling tool), and type in numbers like 2.0 and -2.0 on the top and bottom of the y-axis, as shown in Figure 1-99, and the chart will rescale a bit.

When you rescale a graph axis manually like this, it will reduce your confusion if you *only* type into the end points; leave the middle points alone. To see the x-axis a bit more clearly, type in a 12 (using the proper tool) where you see the 100, so you get the image shown in Figure 1-100.

Currently, it's not easy to see exactly where the data points are. Let's fix that—pop up on the chart's legend, `Plot 0`, and select my favorite point style, as shown in Figure 1-92, so the plot looks like Figure 1-101.

1.9 Arrays, Loops, Graphs, and Charts

89



Figure 1-99

11 data points from the previous figure where we manually set the y-axis.



Figure 1-100

11 data points from the previous figure where we manually set the x-axis.

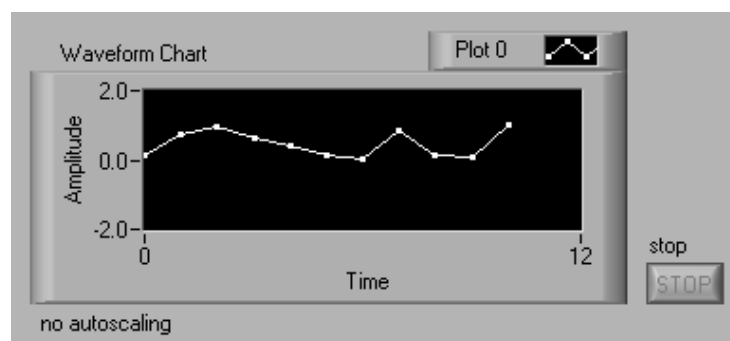


Figure 1-101

11 data points from Figure 1-100 where we set the point style.

Another popular feature of graphs is autoscaling. When autoscaling, LabVIEW automatically scales a graph's axis so you can most efficiently show the data. One way to set autoscaling is to pop up on the graph itself (not the legend, label, or many other parts you'll soon see), and select autoscaling. Turn autoscaling on for the y-axis of your chart (pop up and look in the **Y Scale** menu), then change the label below the chart, so you see something like Figure 1-102.

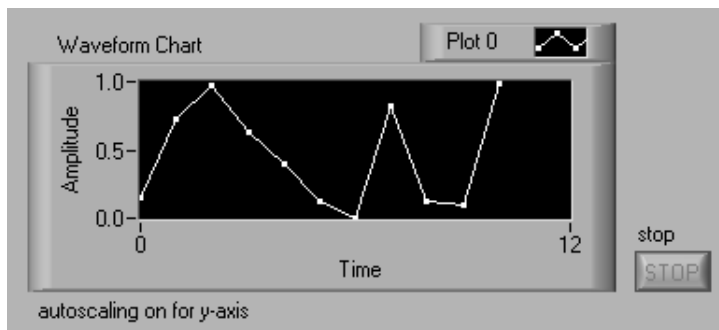


Figure 1-102

11 data points from the previous figure with autoscaling on the y-axis.

Delete your free label concerning autoscaling, because we're about to show autoscaling in a different manner. Pop up on this chart, select the **Visible Items»Scale Legend** option, and you'll see the little box shown in Figure 1-103, called the *scale legend*:

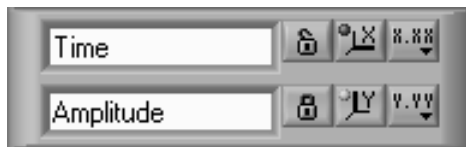




Figure 1-103

The graph's scale legend.

For each axis, you can set autoscaling by using the appropriate buttons per axis. See how the little light next to the picture of the y-axis **Y** is on (the **Autoscale Now** button), and the little padlock just to its left is closed? That means that autoscaling is turned on for the y-axis. Autoscaling is not cur-

1.9 Arrays, Loops, Graphs, and Charts

91




rently on for the x-axis, as its little icons indicate. You can turn it on quite easily from this scale legend by clicking the x-axis' padlock . If you had clicked the x-axis' **Autoscale Now** button, which looks like this , the autoscaling would have occurred at the time you clicked the button, but it would not remain on as if you had pushed the padlock button.

You can control other axis properties, such as numeric precision or grids, by popping up on the X.XX and Y.YY buttons.


Next, select the chart's **Visible Items»Graph Palette** item to show the palette in Figure 1-104.



Figure 1-104
The graph's palette.

First, unlock both axes with the padlock buttons from the scale legend, so they appear like this: . Next, click the little magnifying glass icon, and experiment with zooming. If you lose sight of your data, you can always get it back by clicking the two **Autoscale Now** buttons , .

Clear your chart's data by popping up on it and selecting **Data Operations»Clear Chart**.

The next major topic will involve showing multiple plots at once. First we'll look at charts, then graphs. We now use the *cluster* data type before its "official" section of this chapter. Using the VI we've been building, drop the Bundle function  from the **Functions»Cluster** palette, and build the block diagram shown in Figure 1-105.

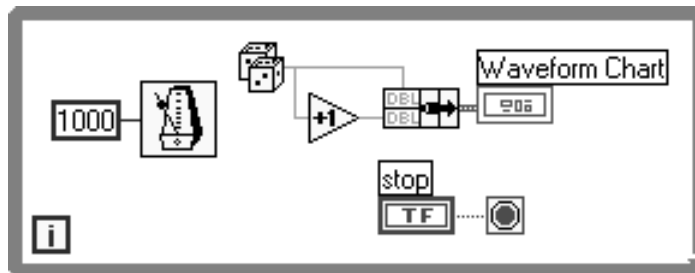



Figure 1-105
The Bundle function can be used with a Waveform Chart to create multiple plots on one chart.

The Bundle function creates a cluster data type, which the Waveform Chart can accept. Turn autoscaling on for both axes by closing both little padlocks  and run your VI about 10 seconds; you should see something like Figure 1-106 (your waveforms will vary).

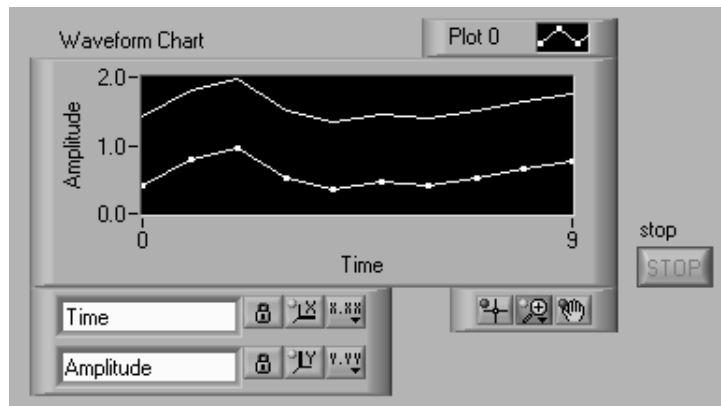


Figure 1-106

Multiple plots are shown on one chart, as created by the block diagram of Figure 1-105.


Finally, drag the Plot Legend over to the right of the chart, and using the Positioning tool , grab the legend's lower right-hand corner, and drag it down a bit, as in Figure 1-107.



Figure 1-107

A chart's legend must be grown to customize multiple plots.

Pop up on your new Plot 1 on this Plot Legend, give it a hollow square point style and a green color (gray in this book), then your chart should look something like Figure 1-108.

1.9 Arrays, Loops, Graphs, and Charts

93

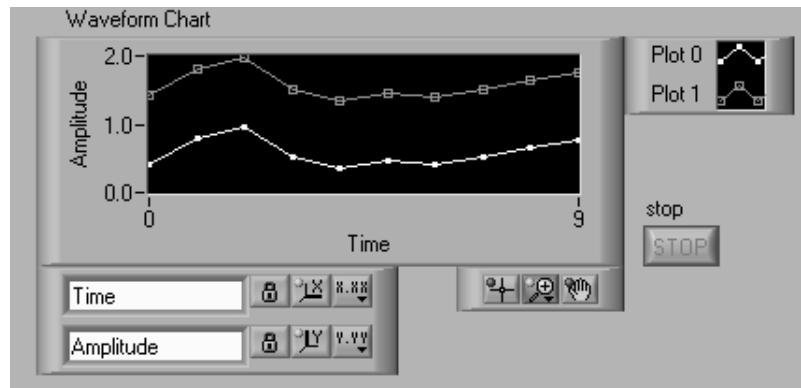


Figure 1-108

Plots are customized on this chart.

When you're building VIs and you have just one plot, you'll probably want to keep the legend hidden. But when you have more than one plot, as in Figure 1-108, you can type your own descriptive plot names where you see Plot 0 and Plot 1. This chapter is long enough as is, so we won't be doing this here.

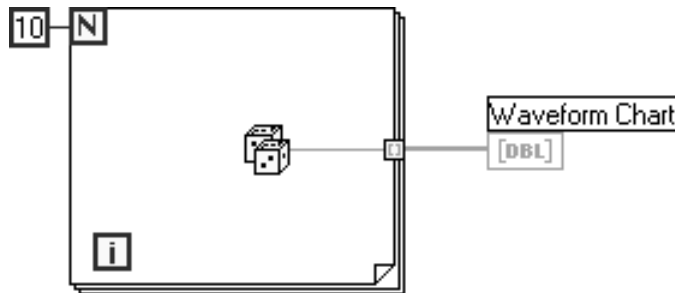
Save this VI now as `Two Plot Chart.vi`, as we'll use it in the debugging section later on.

Time for graphs! Save this VI now as `Two Plot Graph.vi`, ensuring that we don't overwrite our `Two Plot Chart.vi`. On the front panel, delete the stop button.

On the block diagram, get the Positioning tool, drag a big rectangle around everything, and delete it (only the front panel chart, Waveform Chart, should remain).

Next, build the block diagram shown in Figure 1-109.

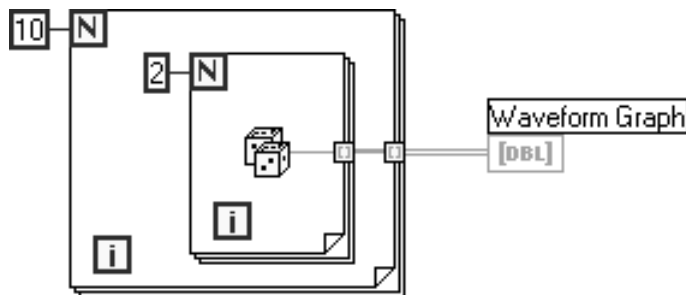
On the front panel, make sure the chart's autoscaling is on. Run the VI a few times, and you see how the chart, being polymorphic, can also accept arrays of data. How? The chart displays them as if they were sent sequentially, one at a time. This might be useful information if you ever need to speed up your charts' charting.

**Figure 1-109**

A 1D array of random numbers is created.

Graphs are quite similar to charts, except they take all of the data at once rather than one point (or points, as in our last example with the Bundle function), at a time. Graphs cannot take a simple numeric data type—they need an array (or something more complex). Go to your front panel, pop up on the chart, and *replace it* with the Waveform Graph. Notice how the **Run** button is *not* broken! This means that both the graph and the chart can accept a simple array of numbers. However, they handle them differently, as you can see by running the VI a few times. Save the VI now with its new name, so it earns its name with the word “Graph” in it.

Notice how the graph, unlike the chart, has no history, or memory, of past data points. But the graph can accept more complex forms of data than the chart.

**Figure 1-110**

A 2D array of random numbers is created.

1.9 Arrays, Loops, Graphs, and Charts

95

Let's try to produce two plots of 10 data points. Go again to your block diagram, and make it look like Figure 1-110 by dragging a For Loop around the Random Number function, then wiring a new numeric constant with a value of 2.

Run the VI, then see the sort of nonsense on the front panel shown in Figure 1-111.

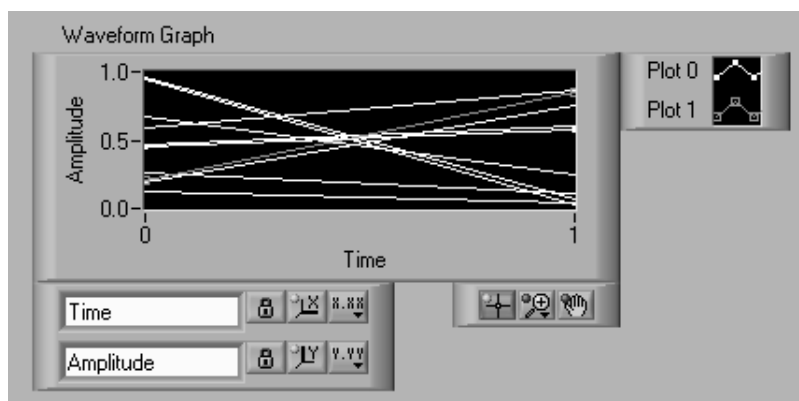


Figure 1-111

A nonsense graph, 10 plots of two points.

Pop up on the graph, select the **Transpose Array** menu item, then see something more logical looking, like Figure 1-112 (your exact waveforms will almost certainly vary, since these are random numbers).

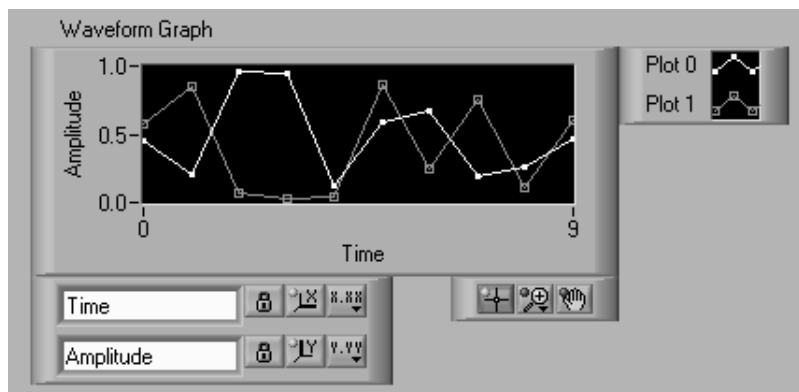


Figure 1-112

Better graph, two plots of 10 points.

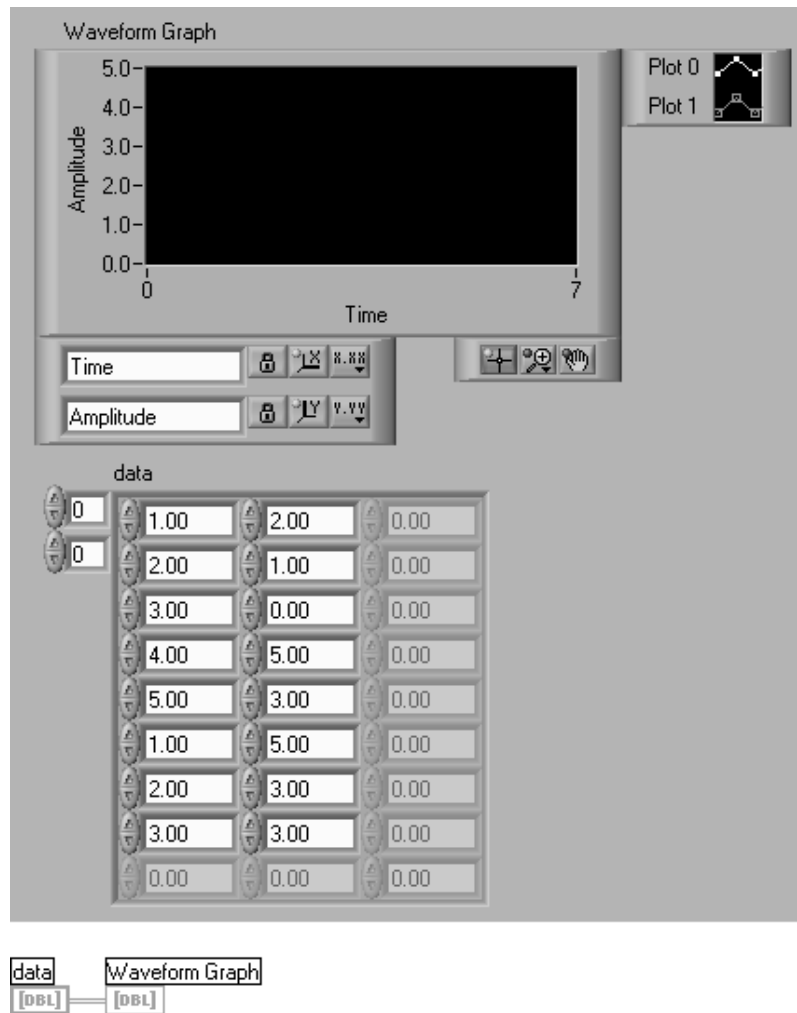



Figure 1-113
A 2D array is wired directly to a Waveform Graph.

What we are plotting here is a 2D array of numbers. In the nonsense graph above, we had 10 plots of two numbers. By transposing the array, we're showing two plots of 10 numbers.

To make sure you understand just how a 2D array of numbers gets to a graph, get the Positioning tool , then drag a rectangle around everything on the block diagram so that it's all selected. Hit your <Delete> key to erase

1.9 Arrays, Loops, Graphs, and Charts

97

everything, so that the only object left on the block diagram is the Waveform Graph's terminal. Now go to the front panel, grow your graph vertically somewhat as shown in Figure 1-113, then create a 2D array of Digital Controls labeled `data`, so that you have the VI in Figure 1-113.

After you run this VI, the data on your graph should appear as in Figure 1-114, which you should be able to easily correlate to the numbers in `data`. Change one number at a time slightly (keep them roughly in the ± 10 range) to see how the graph represents its data.

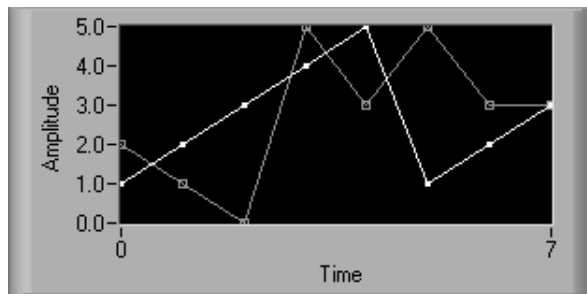


Figure 1-114

The numbers shown in the previous figure are drawn by the Waveform Graph as two plots.

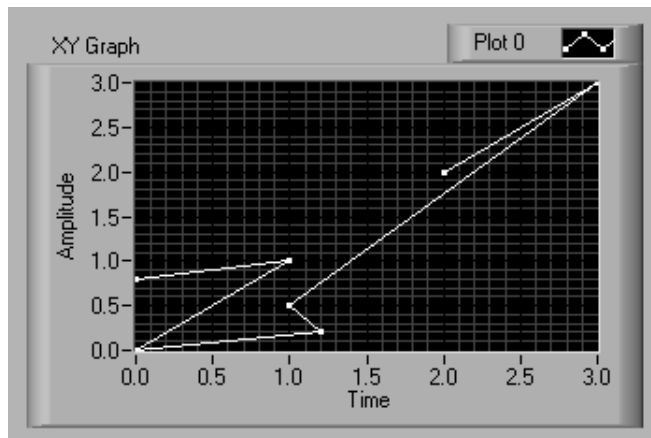


Figure 1-115

An XY Graph allows arbitrary x-coordinates, unlike a Waveform Graph or Waveform Chart.

The final sort of *very* useful graph to be described here is the XY Graph, but we need to cover **clusters** before we get into that. As a little preview, we can display any string of (x, y) coordinates, where unlike the Waveform Chart and Waveform Graph, the x-coordinates do not need to be evenly spaced.

An XY Graph example is shown in Figure 1–115.

1.10 SUBVIs

Every programming language on the planet supports calls to subroutines, right? So does LabVIEW, which is the topic of this section.

1.10.1 Overview

Just poking around on the **Functions** palette, you can see that LabVIEW has tons of built-in functions. But what if LabVIEW doesn't have the function you want? If you can build it in LabVIEW as a VI, then you can make your *own* function from it, a *subVI*, which can appear on the block diagram of another VI. The two VIs are referred to as the *caller* (the parent VI) and the *subVI* (called by the parent). For example, the exercise in which we reversed the elements of an array could have been encapsulated into a single icon and used just like the Reverse Array function in any number of other VIs. You will generally not see the front panel of a subVI when it is used, just as you do not see block diagrams nor their functions when they're used. There are two major reasons you should use subVIs: (1) you are performing the same tasks at more than one place in your program, and (2) your block diagram has become too large. Let's review both cases.

The first reason to use subVIs is to perform the same set of functions at different places in your LabVIEW program. Having been a programmer for over 20 years, I see that the most common and time-wasting programming mistake is having redundant code. This is a subtle and seemingly harmless mistake, but it is by far the most damaging. If you are a text-based programmer, and you have ever found yourself copying more than a few lines of text and using them as they are, you are guilty of this mistake. In LabVIEW, if you find yourself using the same few functions in the same way in more than one place in your program, you are also guilty of this mistake! Why is this so

damaging? Other than the obvious reason of having too much code, if you later come back to modify one section of code, you might forget to modify the other, when the two pieces of code should be acting identically. In the real world, this happens all the time. Code sits around for years (or even days), then the original programmer, or some other person, will fix one section of code without realizing that the other also needs fixing.

The other reason to use subVIs is that your block diagram is too large.

1.10.2 An Example to Build

Our objective here is to build a function, like the Add function, which calculates the hypotenuse of a right triangle. See Figure 1-116.

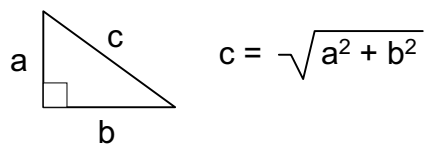



Figure 1-116
The Pythagorean Theorem.

The three parts of any VI are the front panel, the block diagram, and the icon/connector. The icon/connector has been largely ignored until now, but we must discuss it here, as it is the only interface between a subVI and its calling VI. The icon/connector consists of an icon (a little picture you can draw) and a connector pane (a pattern of terminals to which you can connect wires). Examples are shown in Figure 1-17. With the connector pane, we get to choose from a variety of terminal patterns to effectively define our own function. Given that our function has two inputs and one output, we will use this pattern: 

To build a subVI, here are the steps you should follow:

1. Build a VI that you will use as a subVI, using front panel controls for data input and indicators for data output.
2. Pick a pattern for the connector pane, and assign only the necessary front panel controls and/or indicators to the connector pane's terminals. With the connector pane showing, pop up on all

input terminals and select **This Connection Is»Required**. *Always do this for subVI input terminals, unless they really must be optional, as it will save debugging time later when you forget to wire an input terminal.*

3. Create an icon for your VI.
4. Save the VI.

Let's make our hypotenuse subVI following these four basic steps:

1. Build a VI that you will use as a subVI, using front panel controls for data input and indicators for data output. Create a VI with the front panel and block diagram shown in Figure 1-117. Make sure it works by setting *a* to 3.00 and *b* to 4.00, then run it—*c*, of course, should be 5.00.

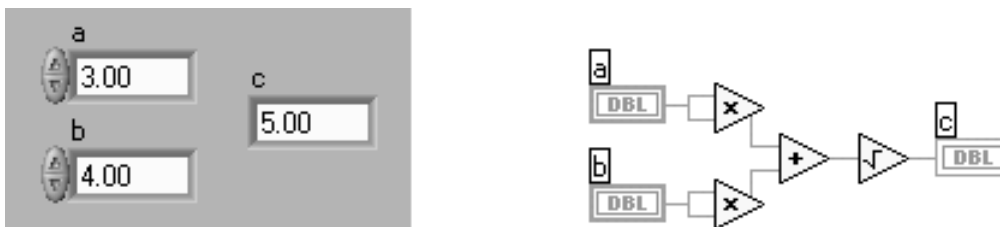







Figure 1-117
A simple VI for hypotenuse calculation.

2. Pick a pattern for the connector pane and assign only the necessary front panel controls and/or indicators to the connector pane's terminals. The icon/connector (the little square in the upper right-hand corner of your front panel or block diagram) will be showing either the icon (by default) or the connector pane. We want it to show the connector pane, so pop up on the icon/connector and select the **Show Connector** menu item. If you do not find this item, you will see the **Show Icon** menu item instead, which means the connector pane is already showing.
Pop up on the connector pane again, and select this pattern: . To associate the front panel objects *a*, *b*, and *c* with these three terminals, you must use the Wiring tool , then click on sections of the connec-

tor pane immediately before (or immediately after) you click on the corresponding front panel object.



Notice that when you hit the <Tab> key to cycle through the tools on the front panel, the Wiring tool  is not one of them. You could explicitly show the Tools palette and select the Wiring tool, but a quicker way to get the Wiring tool is to select the Operating tool , then click on any terminal in your connector pane. For this to work, the icon/connector must be showing the connector pane, not the icon.

For our example, Figure 1–118 shows a rather speedy order in which you could click on the various areas to quickly assign your connector pane terminal (the numbered black squares represent six separate mouse clicks with the Wiring tool ).

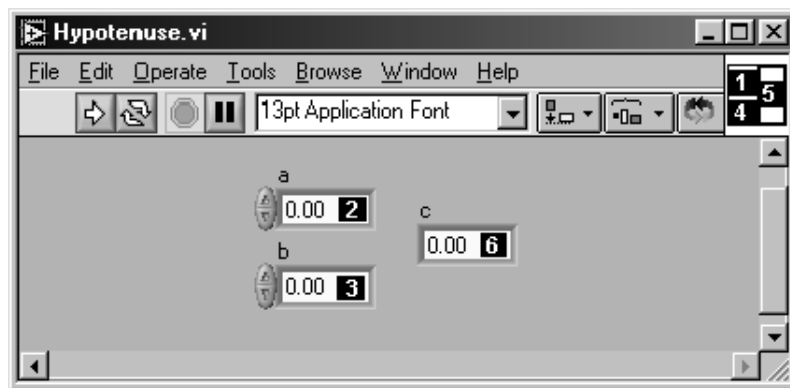



Figure 1–118

An efficient clicking sequence (white-on-black numbers drawn in by me).

3. Create an icon for your VI. When your subVI appears in the block diagram of another VI, you may control its appearance by creating an icon for this subVI. Pop up on the connector pane and select the **Show Icon** menu item if it's there. Now double-click the icon, and up pops the icon editor. You have the option here of making a picture or text. Years of experience tells me that you're better off with a text-only description, no matter how clever you think your picture is, so let's make some text. First, using the

selection tool , select everything just inside the outer rectangle of the large editing window, as in Figure 1-119.

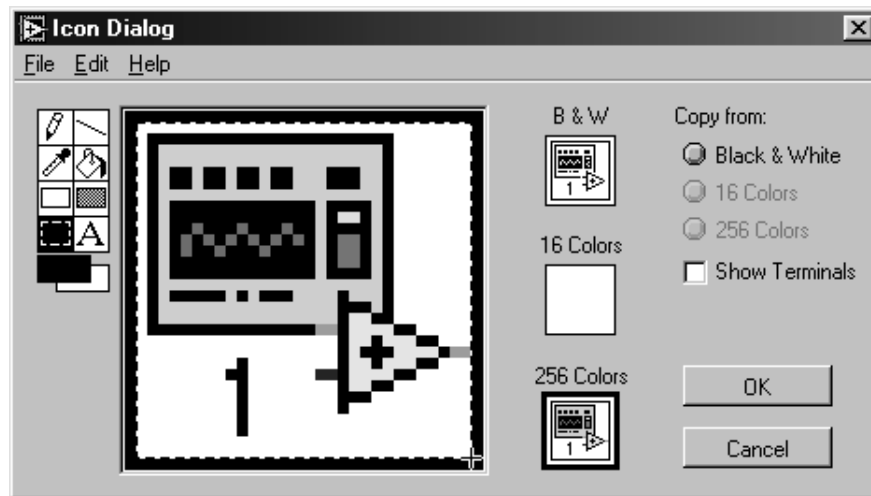


Figure 1-119

The inside of a LabVIEW VI's default icon is selected in preparation to erase it.

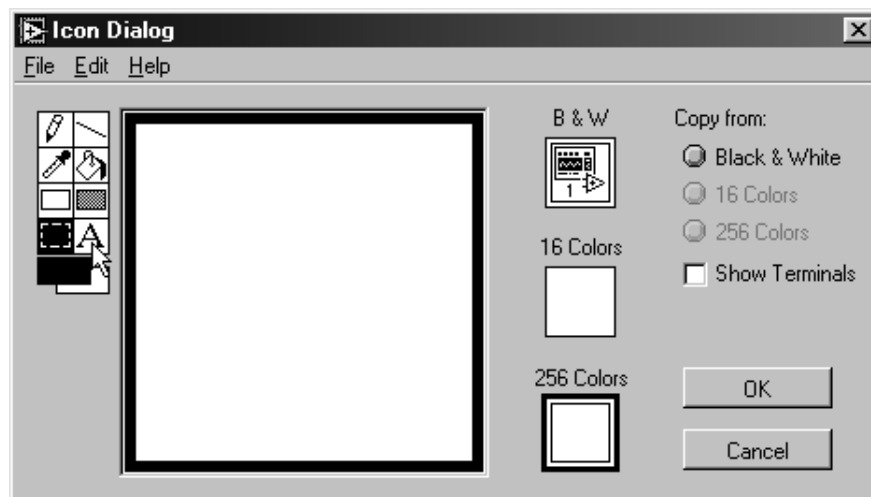



Figure 1-120

As hoped for, the inside of an icon is erased.

Next, hit the <Delete> key, so you have just the outer rectangle remaining, as in Figure 1–120.

Now you must type some text in that square. It's important to use the right font here, so double-click the text tool  and select the **Small Fonts** font, size 10. If you manage to exit LabVIEW without crashing, this will be your default font until it's changed again here, using this version of LabVIEW on this computer. Your objective is now to produce the icon shown in Figure 1–121 (or something that looks like it).

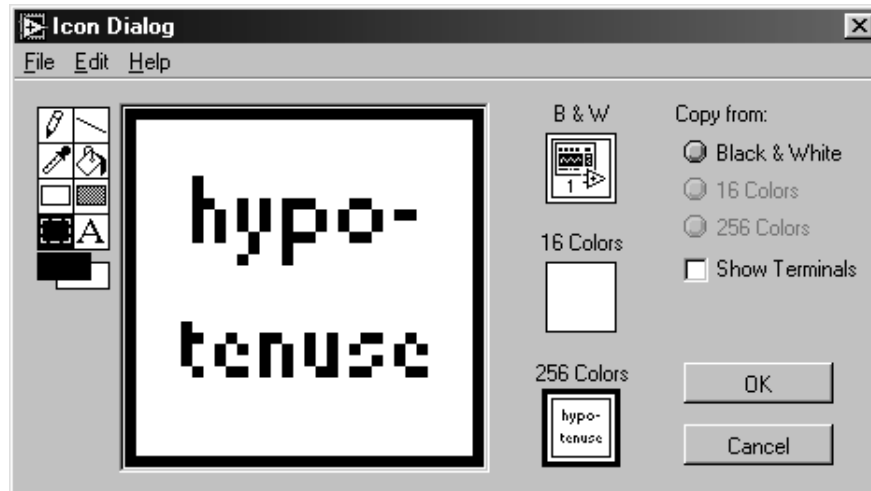



Figure 1–121

Small Fonts, size 10, is an excellent choice for icons' text.

Using the text tool, click where the lower left corner of the h is (do h's have corners?), and start typing. If you make a mistake typing or with your initial cursor position, you can use the <Backspace> key to erase your typing. You can use the selection tool  to select and drag your text around in this main editing screen if your positioning is a little off. Once the icon looks good, make sure nothing is selected on your icon (no marching ants), then copy its smaller image from one of the boxes to the right by clicking on said box and hitting <Ctrl-C>. If anything had been selected in the large icon, it would have been copied—but we want the whole image. Next, paste the image into the other two boxes

by clicking on them, one at a time, then hitting <Ctrl-V>. Or, you could use the items under “Copy from:”.

4. Save the VI. Save this VI as `Hypotenuse.vi`.

The subVI has now been created. Next, we’ll use this VI in another VI, which we’ll call the *calling* VI. Create a new VI. Select the **Functions»Select A VI...** button, shown in Figure 1-122.



Figure 1-122

This lower left icon is one way to drop a subVI into a block diagram. It will pop up a file dialog box to let you select that subVI.

Using the file dialog box that will pop up, find your `Hypotenuse.vi`, select it, and drop it into the block diagram. On your calling VI, create front panel controls and indicators just like those on `Hypotenuse.vi`, and wire them up to all three terminals of your subVI. Running your new VI, you can see that it actually performs the function of `Hypotenuse.vi` by calling it as a subVI; this is our first subVI in action. Save your new VI as `Hypotenuse Caller.vi`, as we'll be using it to demonstrate debugging techniques later. It should look something like Figure 1-123.

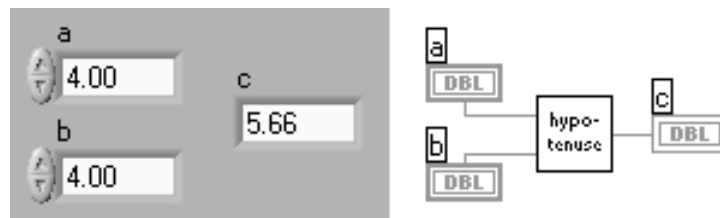


Figure 1-123

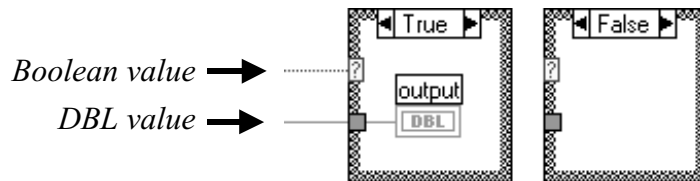
Your `Hypotenuse Caller.vi` should look something like this.

In this simple example, we don't really need this subVI. But if you wrote a program that needed this hypotenuse calculation in more than one place, or if a block diagram had become crowded, the subVI would have been appropriate.

1.10.3 SubVI Details

If you have a subVI with an output that does not receive any data because it's in a Case Structure case that does not get executed, the subVI passes the default value of that output to the calling VI. For example, suppose the False case were executed in the subVI's block diagram shown in Figure 1-124. The *default value* of the output indicator would be passed to the calling VI, which may or may not be desirable.

If you are using many subVIs in a LabVIEW program, and you want to get a graphical overview of what VIs are calling what subVIs, use the **Browse»Show VI Hierarchy** menu item then hit <Ctrl-A>. At the very minimum, the names of a subVI's inputs and outputs, as wired on its connector pane, will appear in its Help window (toggled with <Ctrl-H>). You can also specify additional information for its Help window, as described in Section 1.3.3.

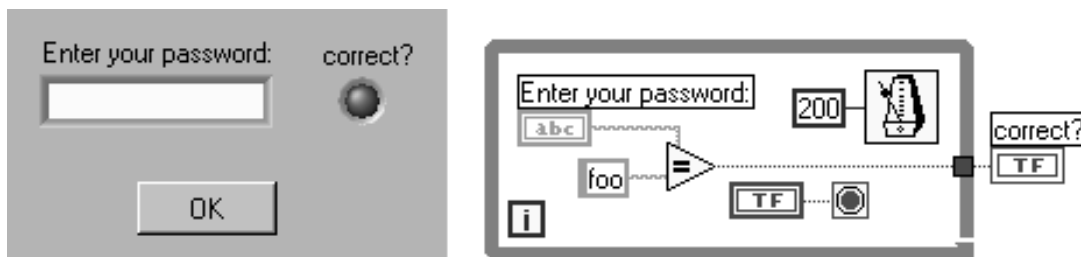
**Figure 1-124**

What happens if the `output` indicator is not executed, yet it's the terminal of a subVI?

1.10.4 Custom Dialog Boxes

Suppose you wanted a pop-up dialog box that's a little more complex than the ones LabVIEW gives you (the One Button Dialog and Two Button Dialog functions). For example, suppose you want a window to pop up and ask the user for a password. We will build such a dialog box as a VI, then make it a subVI. This is one you could likely use in your real applications.

Create a VI with the front panel and block diagram shown in Figure 1-125 (make sure your **OK** button has a latching mechanical action; this button comes ready-to-go from the **Controls»Dialog Controls** palette).

**Figure 1-125**

A simple VI to be used as a subVI dialog box for entering a password.

Don't forget to change the conditional terminal to **Stop If True**. Try the VI by running it a few times, typing in a password, then hitting the **OK** button. Notice that the Boolean indicator `correct?` is True when you've entered `foo` as the password; otherwise, `correct?` is False. Once you're happy with

how it works, you can pop up on the String Control and select the **Password Display** menu item so that prying eyes cannot see what you're typing, unless they have a keyboard-monitoring device (hardware or software) or they're watching your fingers very carefully.



Make sure the **OK** button is associated with the <Return> key via its **Advanced»Key Navigation...** pop-up menu item. The <Return> key on your keyboard may have the word "Enter" on it. If you got the **OK** button from the **Controls»Dialog Controls** palette, it should already be set that way. This allows you to hit the <Return> key after you've typed the password, which is more convenient than clicking it.

To use this as a subVI, you must connect the correct? indicator to a terminal in the connector pane. Just one terminal is all you need in this connector pane. Make its icon say "password." Save this VI as `Password.vi`.

Now, how do we make this VI pop up when it's used as a subVI? Use the **File»VI Properties...** menu item to open the **VI Properties** window. Set the **Category** menu ring to **Window Appearance**, then click the **Dialog** radio button, as shown in Figure 1-126.

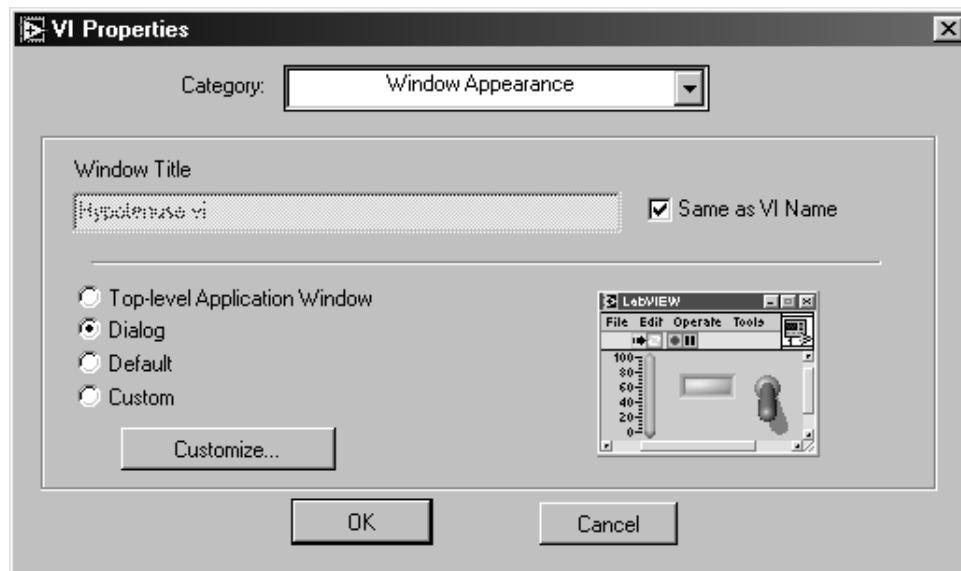


Figure 1-126

This **Dialog** button makes it easy to create a subVI dialog.

This will cause the VI to pop up like a dialog box when called as a subVI, then close itself later. If you want more control over your dialog box, you can click the **Custom** radio button, then use the **Customize...** button. I prefer these settings, shown in Figure 1-127, which are similar to the setting resulting from the **Dialog** radio button, because they don't allow the user to close the dialog box without using the **OK** button.

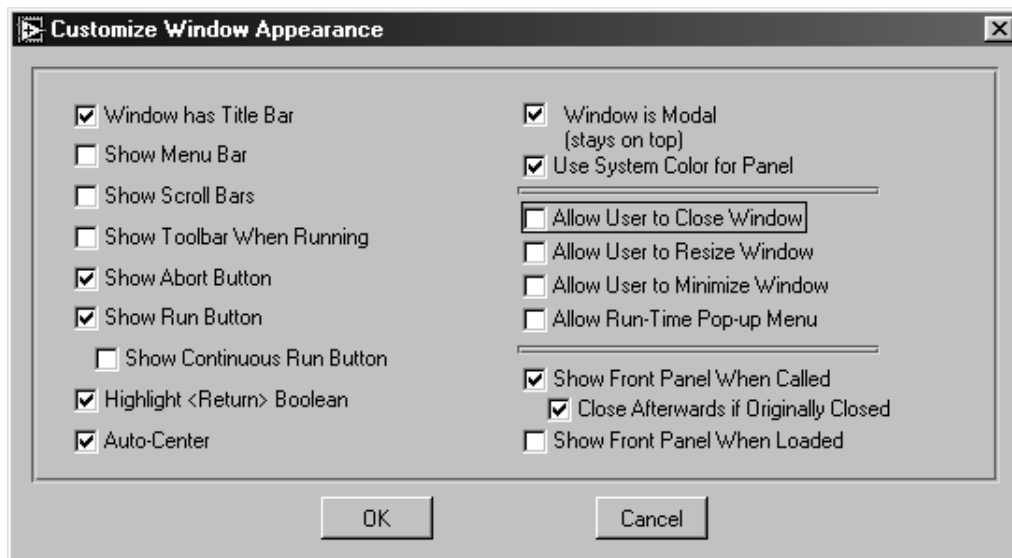


Figure 1-127

It is better if the user is *not* allowed to close subVI dialog boxes. Otherwise, it is possible to “hang” the entire program.

Next, create a calling VI with nothing on the front panel and with the block diagram shown in Figure 1-128.

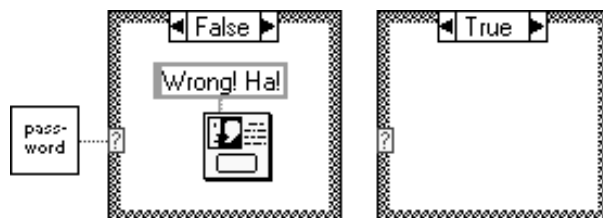


Figure 1-128

A very simple VI tests our password subVI.

Run this VI, and you'll see your newly created dialog box in action. Notice that you must select your password entry string when the password dialog box pops up, and the last thing you typed is still there! This is highly unacceptable. We will be improving `Password.vi` later in this chapter when we cover the necessary topics, thus making it fit for actual use. Save the calling VI as `Password Caller.vi`.

1.10.5 The Create SubVI Menu Item

This menu item is so incredibly useful, it deserves its own section. Its usefulness ranks right up there with `<Ctrl-Z>` and `<Ctrl-F>`, in my opinion.

Create a VI with the front panel and block diagram in Figure 1-129, using these tips:

1. Do *not* run this VI until instructed.
2. This **OK** button comes from the **Control»Dialog Controls** palette.
3. Be sure to select the Two Button Dialog function, not the One Button Dialog function.
4. The Format Into String function is in the **Functions»String** palette.

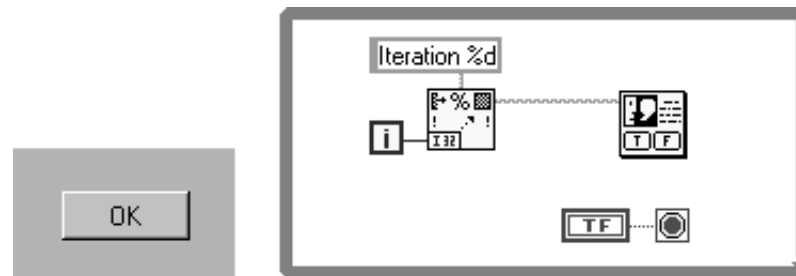


Figure 1-129

We begin to build a safe message box, although this version is anything but safe.

Save this VI as `Safe Message Box.vi`.

If you were to run this, which I do not recommend, you might not be able to easily stop it, because it pops up the same dialog box over and over so quickly that you cannot press the **OK** button. This could force you to `<Ctrl-Alt-Delete>` your way out of LabVIEW, or if you're not terribly bright, you might reboot your computer! We can use a safety mechanism here. Since the

box that pops up has two buttons, we'll call one **Stop VI** so we can push this button to stop LabVIEW's execution, if desperate. Modify your VI as in Figure 1-130 (the little stop sign stops the VI, and can be found on the **Functions»Application Control** palette).

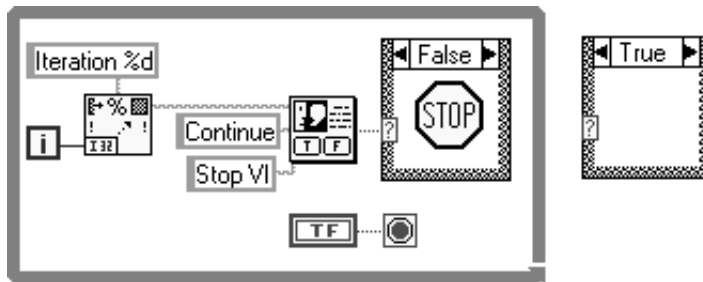


Figure 1-130

This version is much safer than the one shown in Figure 1-129.

You can create a subVI from this and use it in place of all of your One Button Dialog function calls just to make sure you never fall into the trap of the never-ending pop-up box described a bit earlier. The easiest way to change this piece of code into a subVI is to select the generically useful code by dragging a selection rectangle around it as in Figure 1-131 to select the images shown in Figure 1-132.

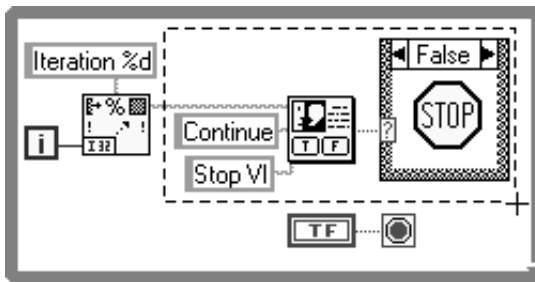


Figure 1-131

We really need only the selected items to create a safe dialog box for generic use.

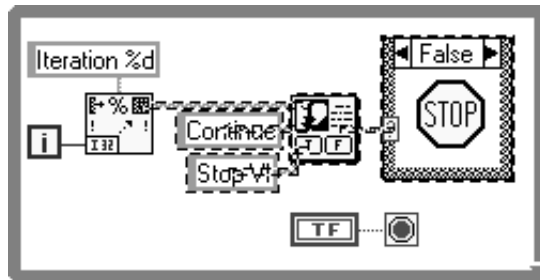


Figure 1-132

Make sure you only have these items highlighted when building this.

With the selection shown above, select the **Edit>Create SubVI** menu item. Whenever you do this, you should immediately open your newly created subVI, give it a nice icon, and save it. For this subVI, put the text `safe dialog` on its icon, and save it as `Safe Dialog.vi`. Your new block diagram should look something like Figure 1-133.

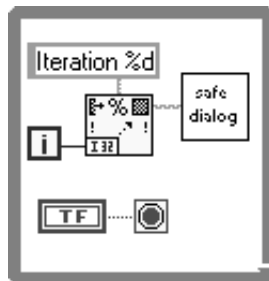


Figure 1-133

Finally, our safe dialog box appears on the block diagram of a calling VI.

Note how we didn't need to fool around with any wiring or the connector pane when creating this subVI; the **Create SubVI** menu item did that dirty work for us! This is even more convenient when multiple wiring connections are involved. On the other hand, this automatic wiring often doesn't give you exactly the connector pane you might want (too many terminals from unused front panel items are often created for you), so you may want to set up the connector pane manually as you wish.

1.11 DEBUGGING TECHNIQUES

If you never make any mistakes, you may skip this section. Otherwise, read on. LabVIEW has many built-in debugging tools, and I have quite a few tricks up my sleeve for you as well. First, we'll cover LabVIEW's debugging tools.

1.11.1 LabVIEW's Debugging Tools

One of LabVIEW's most commonly used debugging tools is the probe. Close all VIs you may have open, then open your `Two Plot Chart.vi` from Section 1.9.4, pop up on the wire connected to the Random Number function, then select the **Probe** menu item, as in Figure 1-134.

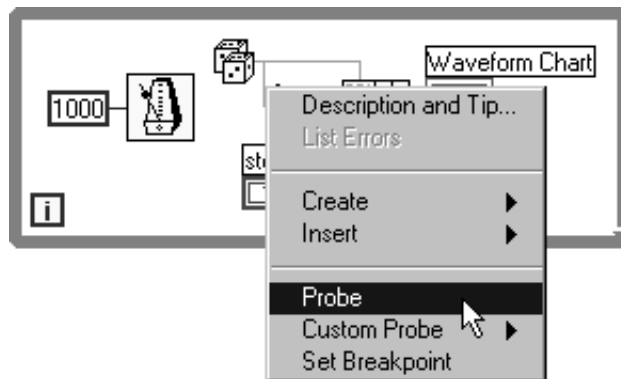


Figure 1-134

If you pop up on a wire, you can create a *probe*.

Up pops a temporary little probe window, which looks something like Figure 1-135, even when the VI is running.

This probe will display the current value of the wire being probed as the VI is running. You could have many probes on different wires.

Beyond the probe, this whole string of buttons on your block diagram's tool bar (see Figure 1-136) is devoted to debugging.

1.11 Debugging Techniques



113



Figure 1-135
Here's a simple numeric probe.



Figure 1-136
Various other debugging buttons exist on the block diagram's tool bar.

Let's start with the **Highlight Execution** button , which is responsible for turning on execution highlighting. We've already used this several times in this chapter. Execution highlighting is most useful with numeric or string data types, as it can show the values moving along the wires in the block diagram when the VI is running, when it is activated, like this  (see Figure 1-137).

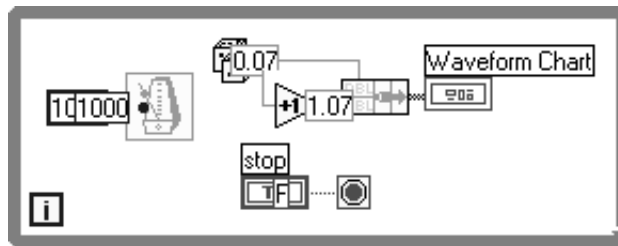







Figure 1-137
Execution highlighting can be very useful with simple data types when you can see the values.

What about those other buttons? The one on the far left  is the **Abort Execution** button. When you have a VI running that you cannot stop by conventional means, like a Stop Button, this button might stop your VI.

I find the **Pause** button  to be of limited usefulness. If you have this button pressed, the VI will pause when it is run. This is more useful if your VI is a subVI.

This set of buttons is a bit more useful: . They allow you to step into subVIs, step over subVIs, and step out of subVIs, respectively. These buttons treat structures, such as the Sequence Structure, the For Loop, the While Loop, and the Case Structure, like subVIs. To see these buttons at work, you can open your `Hypotenuse Caller.vi` and press the leftmost button  a few times, and you'll see yourself stepping into the `Hypotenuse.vi` subVI. If you had been pressing the middle button instead , you would not have stepped into `Hypotenuse.vi`.

1.11.2 Other Debugging Techniques

Sometimes, you may need something a little more powerful than the probe. Suppose, for example, you want to look at eight elements of a 200-element array. If you place a probe on the array's wire, you will only see one element at a time, as shown in Figure 1-138.

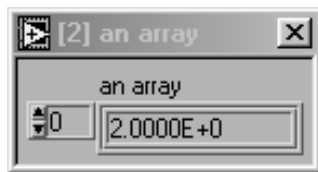


Figure 1-138

A probe only allows you to see one element of an array.

In this case, you might want to place an array of numeric indicators on the front panel, and wire these to your array in question, as in Figure 1-139.

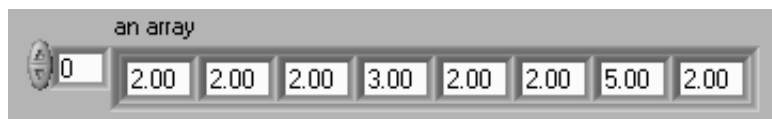


Figure 1-139

If you want to "probe" multiple elements of an array, try a front panel array indicator like this.

You can later delete this front panel indicator when you're sure your VI works correctly.

It is sometimes helpful to pop up a dialog box with debugging information only under special conditions. When you can, use the `Safe Dia-`

`log.vi` you created earlier inside a Case Structure so that the dialog pops up *only* under your special conditions, and pass in a helpful string. You can make such a string even more helpful by using the Format Into String function, as in the pseudo-VI in Figure 1-140.

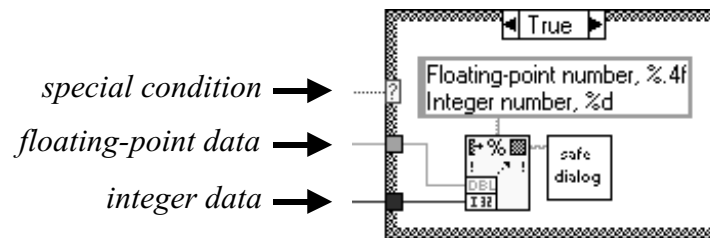


Figure 1-140

This pseudo-VI only pops up a box when a special condition is true.

Final debugging tip—you will usually have a main loop in your program. Divide this loop's iteration terminal by 100 and display the remainder on the front panel, so you can immediately tell when your VI has unexpectedly stopped (crashed). See Figure 1-141.

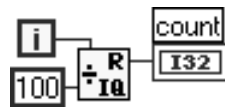


Figure 1-141

Displays a different two-digit number per loop iteration so you know the loop is looping and your VI has not crashed.

Tuck the `count` Numeric Indicator in an obscure corner of your front panel. The remainder business ensures your count will always be two digits or less, so the `count` indicator on the front panel may be physically small.

1.12 CLUSTERS

There are two basic means of grouping objects in LabVIEW: arrays and clusters. An array is a variably sized group of objects having the same data type,

whereas a cluster is a fixed-size group of objects, which may have different data types. For example, an array of numbers may change in size from four numbers to eight numbers while the VI is running, but a cluster of four numbers must always have four numbers while the VI is running. Unlike the array, a cluster could contain any assortment of data types, as in Figure 1-142.

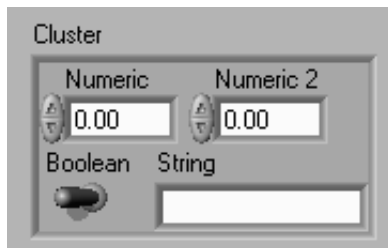






Figure 1-142

A cluster can contain arbitrary data types.

1.12.1 Using Clusters

In LabVIEW, to *bundle* means to group objects into a cluster. We will build the cluster shown above, then use it to show clusters' features. Create a new VI, then drop a Cluster onto the front panel from the **Controls»Array & Cluster** palette. Size it so that it's a bit larger than the one shown in Figure 1-142, then create and drop the objects inside, as shown. Now pop up on the cluster (be sure to get its edge, not the inside), and select the **AutoSizing»Size to Fit** menu item. Save this VI as `Cluster Example.vi`.

Cluster objects are always ordered. To see this order, pop up on the cluster (its edge) and select its **Reorder Controls In Cluster...** menu item. The screen will then look something like the screen in Figure 1-143, perhaps with a different order.

Click on the little boxes with the numbers in them  and change the order to something other than what you have. When you're finished, click the **OK** button  in the tool bar if you want to keep the new order; otherwise, click the **X** button  to revert to the original cluster order. You will also see this on the tool bar: **Click to set to** . You can start out with a different number than zero for your first click, but this box will increment after each click until all cluster objects have been counted. Finally, order them as shown in Figure 1-143.

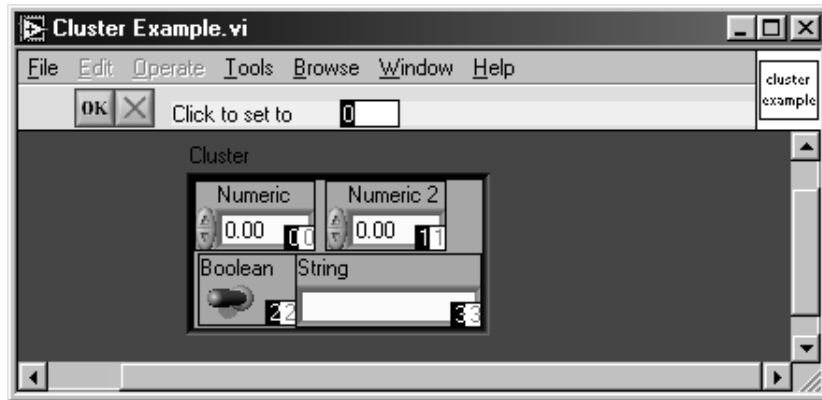


Figure 1-143

This screen is seen when a cluster's elements are reordered.



A similar ordering can be done to control the <Tab> order of front panel objects via the **Edit»Set Tabbing Order...** menu item.

Next, go to the block diagram. Drop the Unbundle function from the **Functions»Cluster** palette. Wire the cluster to the left half of the Unbundle function. Notice how the Unbundle function automatically grows itself to four elements. It knows to do this by following the wire to the cluster of four elements (see Figure 1-144).

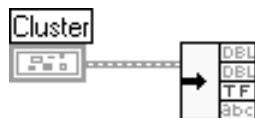


Figure 1-144

The Unbundle function gives you access to the elements of a cluster.

Now, let's use the Unbundle function's outputs. On your front panel, horizontally clone your cluster (<Ctrl-Shift>-drag an *edge* of Cluster with the Positioning tool) to the right of the original Cluster, then make the new Cluster 2 an indicator. Now, type the numbers shown in Figure 1-145 into Cluster (still a control).

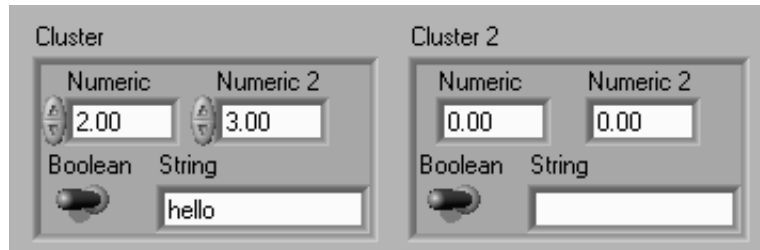


Figure 1-145

The numbers to be typed into Cluster.

Build the block diagram (shown in Figure 1-146) from this front panel, using these tips:

1. The Bundle function is in the **Functions»Cluster** palette.
2. The far right wire must be connected to the Bundle's rightmost terminal, not the center one.
3. The Not function is in the **Functions»Boolean** palette.

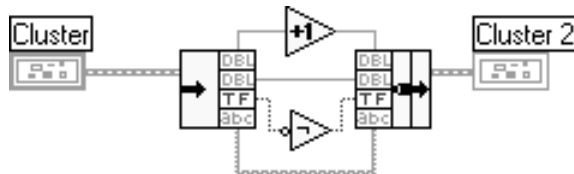


Figure 1-146

The elements of a cluster are manipulated with the Bundle and Unbundle functions.

Run the VI, and you should get the image in Figure 1-147.

Notice how cluster element 0 Numeric was incremented, and the Boolean value was toggled, which should make sense as you look at the block diagram. What's that center terminal on the Bundle function doing? Nothing, in this case, but you could clean up the block diagram a bit by wiring it as in Figure 1-148.

1.12 Clusters

119

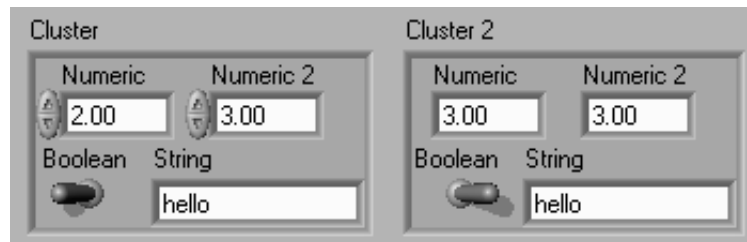


Figure 1-147

This is the result of manipulation as shown in Figure 1-146.

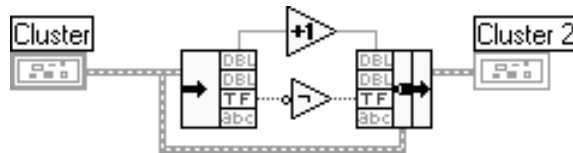



Figure 1-148

The middle terminal of the Bundle function is used to produce the effect of Figure 1-146.

The center terminal of a Bundle function allows you to leave any number of its other inputs unwired by providing the Bundle function with default values (and data types) of the incoming cluster's unwired elements.

In the beginning, there was the Unbundle function with its corresponding Bundle function. But the LabVIEW developers thought, "Hey, let's get fancy." This happens quite a bit. So along came the fancy Unbundle By Name function and its corresponding Bundle By Name function. These functions use cluster elements' labels (names) to identify them. We'll now see how to use these "By Name" alternatives, as they're very helpful. Pop up on the Unbundle function, select **Replace**, and replace it with the Unbundle By Name function. Similarly, replace the Bundle with its corresponding Bundle By Name function. After the replacements, you will need to move things around quite a bit with the Positioning tool , and you will likely need to delete lots of wires (if not all of them) then rewire everything, so you get the image in Figure 1-149.

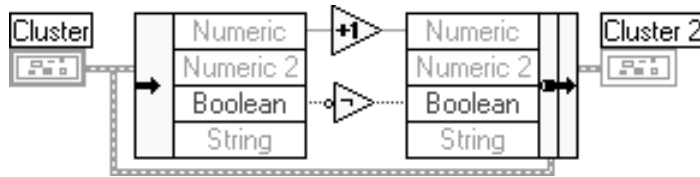



Figure 1-149

The Unbundle By Name and Bundle By Name functions produce the effect of Figure 1-146.

Notice how we have a broken **Run** button  at this point. This can be remedied by removing all unwired elements from our new bundling and unbundling functions (pop up on each unwired element, and select **Remove Element**), so you get the image in Figure 1-150 (after some repositioning of wires and other items).

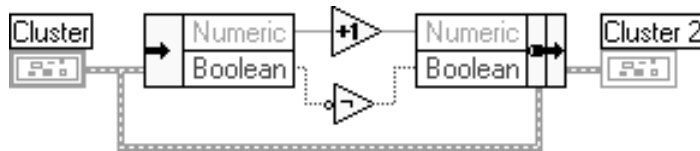


Figure 1-150

The middle terminal of the Bundle By Name function produces the effect of Figure 1-146.

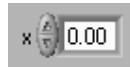
This should work just fine, if there are no broken wires hiding anywhere. Notice you can pop up on the elements of the Bundle By Name and Unbundle By Name functions and change them to a different element quite easily (such as `Numeric` to `Numeric 2`). Left-clicking with the Operating Tool also does this.

The last point to make about the “By Name” functions is this: The Bundle By Name function requires that you wire its center terminal, whereas the regular Bundle did not require this, provided you had *all* its inputs wired.

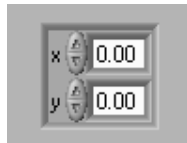
1.12.2 Other Cluster Uses


Clusters are very handy for passing complex data structures to charts and graphs. Back in the section on graphs, we didn’t go into the XY Graph because we had yet to discuss clusters in depth. Build the VI shown in Figure 1-151, using these tips:

1. First, drop a Digital Control, shrink it a bit horizontally, and label it x . Drag the label to the left, so that it looks like this:





2. Clone this control below itself, label the clone y , drop a cluster, drag the two Digital Controls into the cluster, and select the cluster's **AutoSizing»Size to Fit** menu item, so that it looks like this:



3. Drop an array, drag the cluster into the array, grow the array as shown in Figure 1-151, then enter the values shown. This is an array of clusters of two floating-point numbers. We can pass this particular data type to an XY Graph. The NaN stands for Not a Number, and causes the corresponding point to not be drawn on a graph or chart.
4. Since you've gone through the trouble of entering this data, pop up on the array (not an element of the array), and using the **Data Operations** menu item, make its current value default.
5. On the front panel, create an XY Graph. Any other type of graph will not accept the data type we've created, so make sure you get the right graph. Pop up on the graph's Plot Legend, and change the line width to its widest. Pop up again on the graph and disable autoscaling for *both* x- and y-axes. Add the free label, using the Labeling tool , saying autoscaling off for both axes. Make sure the axes are both ranging from 0 to 10.
6. Wire the two terminals together on the block diagram. Your VI should now look like Figure 1-151 (see Table 1.4 to see why the color of your wire is brown).

Now, go back to the front panel and run the VI so you can see the power of XY Graphs. Save your VI as `XY Graph Example.vi`.

One final point about clusters—they are quite handy for passing information into subVIs. If you find yourself with too many inputs and/or outputs on a subVI, and don't want a connector pane that looks something like this  or this , group some inputs and/or outputs into a cluster. This technique can lead to sloppiness, unless the clusters you create can be used throughout your VIs and subVIs, in which case this technique can lead to efficiency.

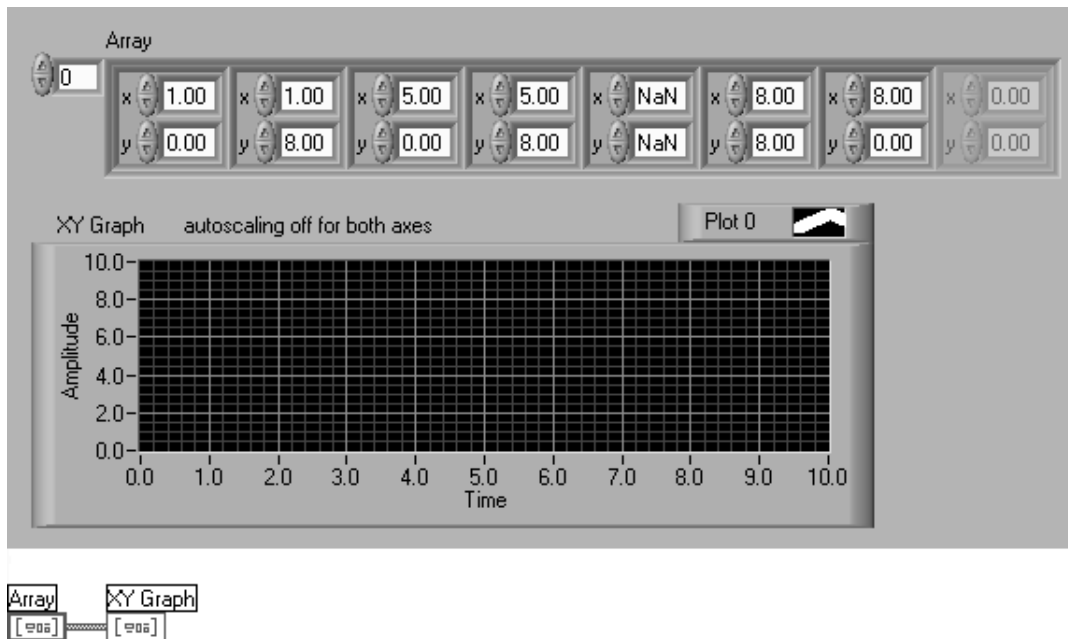


Figure 1-151

This VI illustrates a data format for the XY graph.

When I worked at NI on the LabVIEW team, those two crowded connector patterns in the previous paragraph were created to please a particular NI employee named Monnie, thus were internally known as the *Monnie Pleaser* and the *Super Monnie Pleaser*, respectively.

1.13 FILE I/O

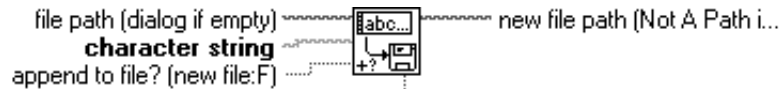
Please make sure you understand the material in Appendix A before reading any of this section.

File I/O means file input and/or output, where a *file* is a group of bytes residing on some form of nonvolatile memory—*nonvolatile* means that the data is retained when the computer's power is turned off. As of the writing of this book, this nonvolatile memory usually comes in the form of rotating magnetic disks, which are typically called hard drives, hard disks, or floppy

disks. But the important point is not where a file resides, but the fact that it stores nonvolatile data.

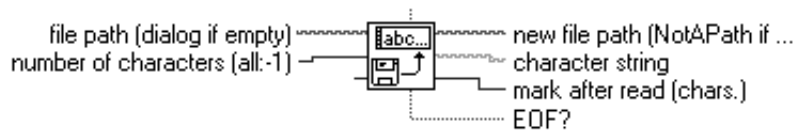
1.13.1 High Level File I/O

Let's go ahead and use LabVIEW's high-level file I/O VIs (used like functions, but are actually subVIs) to read and write files. Interruption: Please read Appendix C now if you have not already done so; it applies to many file I/O VIs if you've changed versions of LabVIEW on your computer. Descriptions of the two most useful high-level file I/O VIs, copied directly from LabVIEW's Help window, are shown in Figure 1–152.



Write Characters To File.vi

Writes a character string to a new byte stream file or appends the string to an existing file. The VI opens or creates the file beforehand and closes it afterwards.



Read Characters From File.vi

Reads a specified number of characters from a byte stream file beginning at a specified character offset. The VI opens the file beforehand and closes it afterwards.

Figure 1–152


The two most useful high-level file I/O VIs (from LabVIEW's Help documentation).

The **file path** wire shown above has a *path* data type that we have not yet discussed. The path is very similar to the string data type, except it is only used to specify the location of a file. On a PC, path data will look like C:\Folder 1\Folder 2\Filename.txt. On other platforms, the delimiters are often characters other than the backslash. Many characters, such as :, /, \, <, >, ?, *, |, and " are not allowed in file names.

The **character string** wire shown in Figure 1–152 is the data being written to (or read from) a file. It is a group of bytes, but not necessarily readable text.

All other inputs and outputs may be studied at your leisure, as the objective here is to have a quick tutorial. In this section, we will only be reading or writing the entire contents of a file. Later, we will see how to read and write parts of a file.

Create a new VI with the front panel and block diagram shown in Figure 1–153, using these tips:

1. For the front panel strings, the left one is a control, and the right one is an indicator. Create one of the strings first, set its font and appearance, then clone it to create the other string. To do this, drop a string, grow it, then show its scrollbar. Use the text ring in the tool bar  to set the font inside the string controls. In order to do this, you must select all of the string's text (if any), and set it to Courier New font, size 14, with the text ring. Since there's probably no text in your newly-created strings, simply place the cursor inside the strings when setting their fonts. Courier New (and Courier) are *monospace* fonts, meaning that all characters have the same width.
2. If this were not an example VI, you might want to hide the buttons' labels on your front panel, but show them on your block diagram, since the buttons' captions are almost the same as their labels. Take care to keep the labels consistent with the captions if you do so, as this is breaking my rule of "no redundant code!" But I suppose redundancy in a *very* localized context is sometimes helpful. The front panel labels are shown here for clarity.
3. Be careful not to confuse your Path control with your String control. Most functions with a terminal expecting a String wire will not accept a Path wire, and vice versa. LabVIEW text usually says "File Path" where this book says "Path."
4. The Path control has been grown a bit to accommodate long paths. Your path should indicate a real directory on your hard drive (likely not the one you see below). It should not initially specify a real file name; otherwise, you'll be overwriting one of your own files! Path controls in LabVIEW support drag-and-drop from the operating system.
5. The subVIs shown on the block diagram are the two described above, found in the **Functions»File I/O** palette.

1.13 File I/O

125

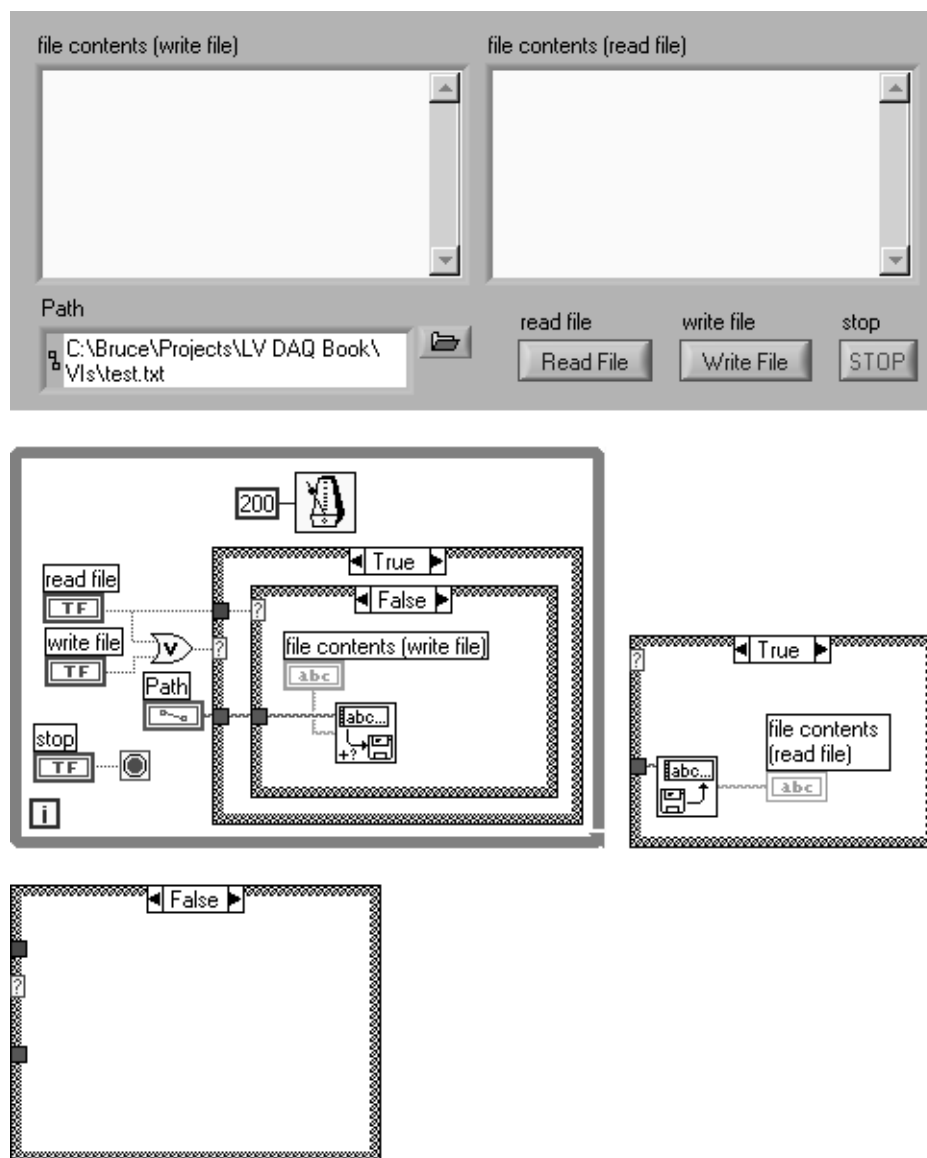


Figure 1-153

A simple, but flexible VI illustrates high-level File I/O.

First, notice the 200 ms delay in the loop from the Wait Until Next ms Multiple function. This is a politeness issue. I will often sneak such a delay into our While Loops without an explicit explanation for you. This technique effectively leaves more time for other programs to use the computer's processor. This 200 ms delay also translates to a response time of about 200 ms, so you may prefer 100 ms or 50 ms, if you notice this tiny delay. I don't.

Run this VI; first write a file (make up a *new* file name), then read the same file back in. This illustrates how easy it is to read and write entire files in LabVIEW. If you end your file name with the .txt extension, you could easily use any text editor to read the file independently from LabVIEW.

We'll do a quick overview of some of the more useful high level file I/O VIs, then go into more detail in the next section.

Save this VI as High Level File IO.vi, as we will be using it later when we discuss local variables. So that you don't lose your path information you have on the front panel, pop up on that path and make its current value the default value, then save the VI again. If you want this path to stay useful, leave that path alone on your disk until you finish this chapter.

Have a look at the **Functions»File I/O»File Constants** palette, shown in Figure 1-154.

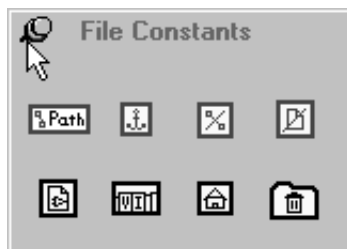




Figure 1-154
The **Functions»File I/O»File Constants** palette.

There is a Path Constant in the upper left, allowing you to specify your path on the block diagram, just like constants of other data types. The Current VI's Path constant  is very useful, as it allows you to programmatically access data in the same folder as your VI. The Not A Path constant  is often used as an error condition to be returned from some file I/O functions.

Let's move up one palette level and look at the **Functions»File I/O** palette, shown in Figure 1-155.

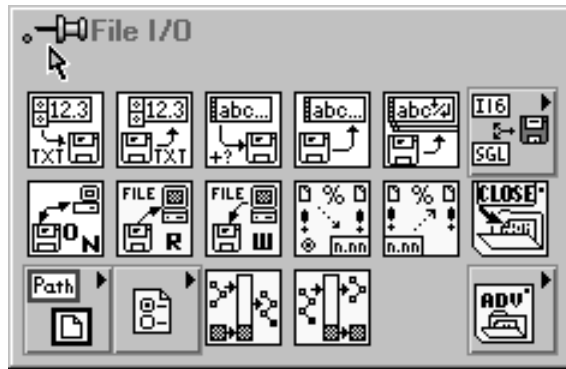



Figure 1-155
The **Functions»File I/O** palette.

To programmatically build paths, such as this sequence

```
C:\
C:\Folder 1
C:\Folder 1\Folder 2
C:\Folder 1\Folder 2\Filename.txt
```

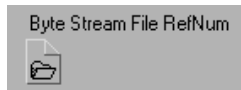
indicates you can use the Build Path function . Conversely, to strip names from the ends of paths, you can use the nearby Strip Path function



Other items in this palette are built for spreadsheet files (only ASCII spreadsheet files, such as tab-delimited or comma-separated files, in spreadsheet lingo). Still other items work only with binary files, where bytes are interpreted as the I16 or SGL data types. Such files would appear as garbage if you opened them with a text editor.

1.13.2 Low Level File I/O

Internally, whenever any file read or write takes place on any computer, a *file reference number* is created. The basic LabVIEW data type for this is the Byte Stream File RefNum, which I'll simply be calling *RefNum* in this chapter, even though LabVIEW has many types of RefNums. Here's what it looks like on a front panel:



On any computer running any program, the basic internal sequence of events for reading a file is this:

1. Open the file in *read mode*. This creates a RefNum.
2. Using this RefNum, read the file or a portion of the file.
3. Using this RefNum, close the file.

Writing a file is very similar—just substitute the word *write* for *read* in the sequence above.

In general, if a file is opened for writing, it cannot be opened again for writing, neither by LabVIEW nor by any other program; the operating system enforces this rule. However, the same file can be opened simultaneously for reading.

When a file is opened, and we have a RefNum, there is also a *file position* associated with this file. LabVIEW's low-level file-reading function allows you to set this position when doing reads. Suppose we want to read the middle four bytes of a file containing the text `Hello!`. Figure 1-156 how this is done in LabVIEW, using a valid path I have on my computer:

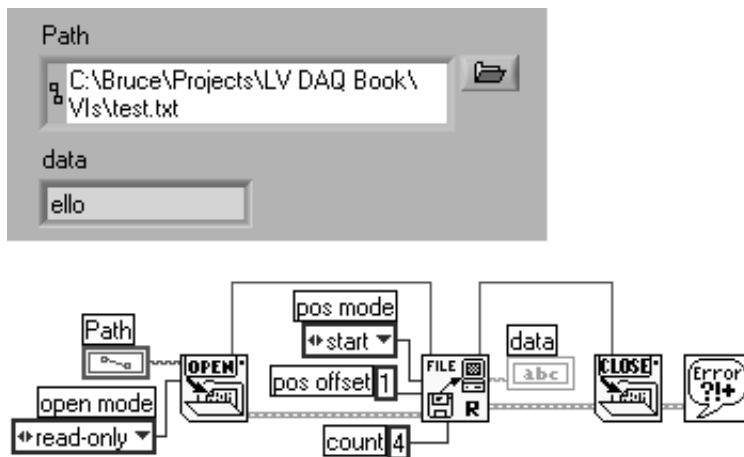


Figure 1-156
An example of low-level File I/O.

See how the four file functions are strung together by the wire on the bottom? This wire is the error cluster, designed to be used in this daisy-chaining fashion. It's often used whenever a group of error-prone functions are called in sequence, so that whenever an error occurs somewhere along the line, it's passed all the way to the error-handling function at the end of the daisy-chain. This helps to pinpoint the source of the error. Use this error cluster technique in your programs wherever possible!

Trying not to bog you down with details, the picture in Figure 1-156 shows the low-level file-opening, file-reading, and file-closing functions (to be named later in a table). A 4 is passed to the `count` parameter, thus instructing this function to only read four bytes. A 1 is passed to a the `pos offset` parameter of the file-reading function, so we start with the letter `e`. A 0 into `pos offset` would have started reading from the first byte, thus giving us `He11` (I swear that was accidental) instead of `e11o`.

LabVIEW offers you many other powerful file I/O functions listed in Table 1.10. Anytime you see "string" in the description column below, realize you can convert data from *any* format to a string, and vice versa, so this "string" can handle *any* of your file I/O needs. Such conversions will be detailed in Chapter 6 (you will read about the Type Cast function and string "flattening" functions). Table 1.10 provides a summary of all file I/O functions that I've actually used in my numerous LabVIEW projects, where the "functions" that are actually VIs (higher level "functions") are shown in monospace font.

Table 1.10 A Summary of File I/O Functions





Name	Icon	Description
Write To Spreadsheet File.vi		Writes a string to a spreadsheet file (only ASCII files, such as tab-delimited or comma-separated files, in spreadsheet lingo).
Read From Spreadsheet File.vi		Reads a string from a spreadsheet file (only ASCII files, such as tab-delimited or comma-separated files, in spreadsheet lingo).
Write Characters To File.vi		Writes a string of characters to any file.
Read Characters From File.vi		Reads a string of characters from any file.

Table 1.10 A Summary of File I/O Functions (continued)












Name	Icon	Description
Open/Create/Replace File.vi		Opens, creates, or replaces a file, and returns a RefNum. The RefNum may be used for reading, writing, and other low-level file operations.
Read File		Given a RefNum, reads any number of bytes from anywhere in a file; returns a string (or user-specified data type).
Write File		Given a RefNum and a string (or user-specified data type), writes any number of bytes to anywhere in a file.
New File		Given a path, creates a new file and opens it, returning a RefNum.
Open File		Given a path, opens a file and returns a RefNum.
Close file		Given a RefNum, closes a file. You should close all files you have opened when you're finished with them.
Build Path		Given a path and a string, appends the string to the path, returning a larger path.
Strip Path		Given a path, strips the last string from the end the path, returning a smaller path.
Read From I16 File.vi		Given a file composed of I16s, reads them to an array of I16s without explicitly using the string data type.
Write To I16 File.vi		Given an array of I16s, writes them to a file without explicitly using the string data type.
Read From SGL File.vi		Given a file composed of SGLs, reads them to an array of SGLs without explicitly using the string data type.

Table 1.10 A Summary of File I/O Functions (continued)















Name	Icon	Description
Write To SGL File.vi		Given an array of SGLs, writes them to a file without explicitly using the string data type.
File Dialog		Pops up a file dialog box, allowing you to select a file or folder. Returns a path along with some other information.
Flush File		Given an open file that has some data buffered in memory due to previous file writes, this physically moves the data to the disk.
File/Directory Info		Given a path, returns information about a file or folder, such as the file size, the modification date, whether the path refers to a file or folder, etc. This can be used to determine whether a path refers to an existing file.
Volume Info		Given a path, returns information on the size of the volume (the floppy disk, the hard drive, etc.), and how much free drive space exists. This is useful for monitoring disk space, so you'll know when you're about to run out.
Move		Given two paths, moves a file or a folder and its contents from one location to another.
Copy		Given two paths, copies a file or a folder and its contents from one location to another.
Delete		Given a path, deletes a file. The file cannot be open when you attempt this deletion.
List Directory		Given a path to a folder (a directory), lists the files and folders within.
New Directory		Given a path, creates a folder (a directory).
Path To Array Of Strings		Given a path, returns an array of strings containing the volume name, folder names, and ending file name. All of these items are optional in a path.

Table 1.10 A Summary of File I/O Functions (continued)

Name	Icon	Description
Array Of Strings To Path		Opposite of the Path To Array Of Strings function, above.
Path To String		Given a path, converts its data type to a string.
String To Path		Given a string, converts its data type to a path.

1.13.3 File Formats: ASCII Versus Binary

Technically, data formatting isn't really a part of any file manager, but you should understand at least the basics of data formatting if you want to program with file I/O. Remember the simple file containing the text `Hello!?` We could think of the file as illustrated in Figure 1-157, where each box represents one byte.

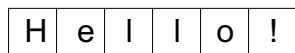


Figure 1-157

ASCII data illustrated as one byte per rectangle.

This data is saved in ASCII (sometimes called *text*) form. Often in LabVIEW, the need arises to store numeric data to files, then read it back later. Suppose we had a spreadsheet full of data. To simplify this example, our spreadsheet will contain only two rows and two columns:

```
1.234, 2.345
10.678, 11.555
```

Figure 1-158 illustrates how your spreadsheet application might save its numeric data in CSV (comma-separated value) form, which is also ASCII; thus, any text editor can read this data. CSV file names should end with

1.13 File I/O

133



.csv, so your spreadsheet application can automatically recognize it. Each box represents one byte:

1	.	2	3	4	,	2	.	3	4	5	\r	\n	1	0	.	6	7	8	,	1	1	.	5	5	5	\r	\n
---	---	---	---	---	---	---	---	---	---	---	----	----	---	---	---	---	---	---	---	---	---	---	---	---	---	----	----

Figure 1-158
28 bytes of ASCII data.

Notice the `\r` and `\n` characters here; together, they correspond to the *end of line* (or *return*) character on a PC, which is what you get when you hit the <Return> (or <Enter>) key on your keyboard in a text editor. Oddly, it takes two bytes (on a PC) to represent this character. Non-PC platforms may use just one of these two bytes to represent the *end of line* character. Table 1.11 lists some technical details on these characters:

Table 1.11 End of Line Characters

Name	Icon	Backslash Symbol	Hexadecimal Value	Decimal Value
carriage return		<code>\r</code>	0x0d	13
line feed		<code>\n</code>	0x0a	10

Suppose we were really concerned about disk space or numeric precision, and wanted to save these same four numbers more efficiently. We could use the SGL type (32-bit floating-point number). On disk, we could save the file as in Figure 1-159, where each box represents four bytes (32 bits).

1.234	2.345	10.678	11.555
-------	-------	--------	--------

Figure 1-159
16 bytes of binary data as four groups of four bytes.

This data is saved in *binary* form. Unlike the previous two examples of ASCII data, each byte does not correspond to a human-readable character. A

text editor could make no sense of this data. If you're curious, the internal bit patterns of these 32-bit numbers are loosely explained in Appendix A.

Reading and writing data in either ASCII or binary form is conceptually straightforward, but can be tricky to implement in real life, regardless of your programming language. So, which is better for file I/O, ASCII or binary? ASCII files are easier to understand for most people, because any text editor can read them, and people can more easily visualize the individual bytes as characters. If ASCII files are put in the right form, many spreadsheets can read them as well. Binary files are more efficient in terms of disk space, numeric precision, and processing speed. In other words, binary files produce smaller, more precise data files that can be processed by the computer more quickly.

1.13.4 Formatting Data for File I/O

This section assumes that you'll be reading or writing a 1D or 2D array of numbers, as this is the situation in most LabVIEW applications. I almost always find it easiest to convert my data into string format if it's not already, then use this string with LabVIEW's file I/O functions. Some of LabVIEW's high-level file I/O functions hide the string data type for you, by showing you only numeric data types like the I16 and SGL data types. With the low-level file I/O functions, you can manipulate data in non-string formats by specifying a different data type in the Open File function's **datalog type** input. This trick will not be discussed further in this book, but you may prefer this approach.

I will first cover in detail the Format Into String function (in the **Functions»String** palette), as it is often useful. Suppose you wire it, as shown in Figure 1-160, to a String Indicator on your front panel.

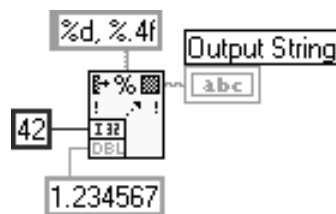
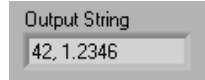



Figure 1-160
The Format Into String function is very useful.

When you run this, you get the following text in your front panel String Indicator:



Here's how the Format Into String function works: The percent character is special, and the letter after the % tells the function to format either a number, string, or path in a particular way. In our example, the %d tells the function to format an integer (d = decimal), and the %.4f tells the function to format a floating-point number with four digits of precision (notice the rounding of the 5 up to 6 in the number displayed, 1.2346). This weird syntax is a result of the C programming language's printf function. For a technical description of what can follow the %, get any reference book on the C programming language and look up the printf function. Some minimal Web searching on "printf" is likely to give you enough useful information so you don't really need a book; look for examples, as they are extremely helpful.

In the example shown in Figure 1-160, if you were writing this string to a file and you wanted to add an *end of line* character to the end of this line, you would want to use the Concatenate Strings function to add an End of Line constant , as shown in Figure 1-161.

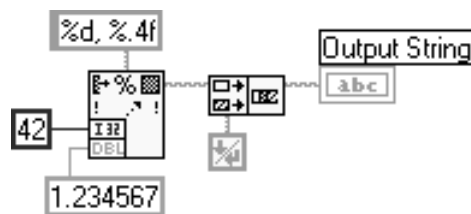

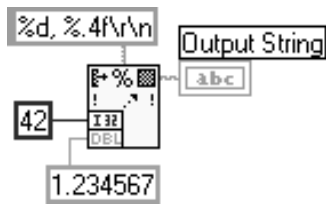


Figure 1-161

The Format Into String function uses a platform-independent End of Line constant.

If you're really clever, you can do the above without the Concatenate Strings function! This End of Line constant  translates into the carriage return and/or line feed character, depending on the platform. On the PC, it translates to the carriage return followed by the line feed. On other platforms, it translates to just one of these two characters. If you knew for a fact that your program would never be run on non-PC platforms, you could have added the *end of line* characters directly to the "format string" constant, and skipped the string concatenation function, as in Figure 1-162.

**Figure 1-162**

The Format Into String function now uses a carriage return and line feed regardless of the platform.

Here is an overview of LabVIEW's arsenal of data formatting functions useful for file I/O (and other things, of course). There's a bit of overlap with Table 1.10. The functions described in Table 1.12 are found at various locations under the **Functions»File I/O**, **Functions»Strings**, and **Functions»Advanced»Data Manipulations** palettes.






Table 1.12 Data Formatting Functions: Useful for File I/O

Name	Icon	Description
Write To Spreadsheet File.vi		Writes a string to a spreadsheet file (only ASCII files, such as tab-delimited or comma-separated files, in spreadsheet lingo).
Read From Spreadsheet File.vi		Reads a string from a spreadsheet file (only ASCII files, such as tab-delimited or comma-separated files, in spreadsheet lingo).
Write To I16 File.vi		You can use this, described in Table 1.10, or the Flatten To String function (below) instead, as it's more generic.
Read From I16 File.vi		You can use this, described in Table 1.10, or the Unflatten From String function (below) instead, as it's more generic.
Read From SGL File.vi		See Write To I16 File.vi, above.
Read From SGL File.vi		See Write To I16 File.vi, above.

1



Table 1.12 Data Formatting Functions: Useful for File I/O (continued)

Name	Icon	Description
Type Cast		VERY USEFUL: (advanced) Useful for converting binary data to ASCII, and vice versa; works only with data types of limited complexity.
Flatten To String		VERY USEFUL: Changes any data type into a string, no matter how complex.
Unflatten From String		VERY USEFUL: Undoes what the Flatten To String function did.
Swap Bytes		Some equipment or computers use data with byte-swapped 16-bit integers—this will swap bytes inside each integer. For example, a Macintosh computer stores 16-bit integers with the most significant byte first, while a PC stores the least significant byte first.
Swap Words		Same idea as the Swap Bytes function, only 16-bit integers within 32-bit integers are swapped.

Exercise 1.2

Write a VI that generates and saves to disk a pair of floating-point random numbers once every 500 ms for 10 seconds (exact 500 ms timing is not really possible unless you have a real-time setup, so just get close). The disk file should be specified by a Path Control on the front panel and named `fake-data.dat`. The disk format should be CSV, like this:

0.0037,1.5194

0.4615,1.7076

0.7643,1.3877

The first random number of each pair should be in the range 0 – 1, and the second in the range 1 – 2. The numbers should have four digits of precision.

All of these hints apply to the solution shown, and may not be relevant should you build this VI another way.

1. You can base this VI on a For Loop with the constant 20 wired to its count terminal.
2. Create a 1D array within each iteration of the For Loop, and pass it through a tunnel to create a 2D array coming out of the For Loop.
3. Pass the 2D array to Write To Spreadsheet File.vi. Be sure to wire proper constants to this subVI.

Solution

Figure 1–163 shows one of many ways to implement this VI.

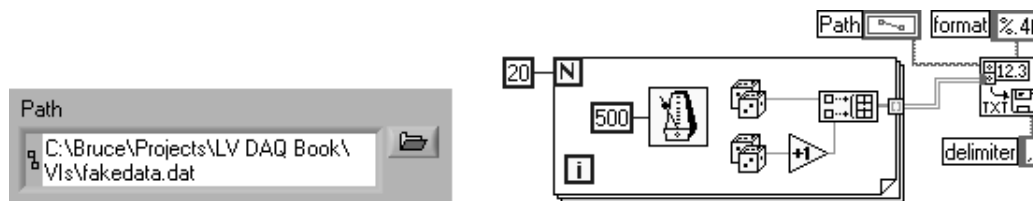


Figure 1–163

One way to implement the VI in Exercise 1.2

1.14 DEVIATIONS FROM DATAFLOW: LOCAL AND GLOBAL VARIABLES

Rule #1 for locals and global variables: *Avoid them.* The only times I still get confused in LabVIEW are when a local or global variable is involved. LabVIEW existed for quite a long time without them, and most people were quite happy with it (except for the fact that LabVIEW had no Undo capability back then). Local and global variables will often be called locals and globals in this section, for the sake of brevity.

1.14.1 Local Variables

A local variable is simply a copy of the data in a front panel control or indicator. Here is an example of why you might want to use a local. Suppose your VI must have two While Loops running simultaneously, and you wanted to stop them both with the push of one button. Before reading past the end of this sentence, see if you can explain why this VI won't work as described. See Figure 1-164.

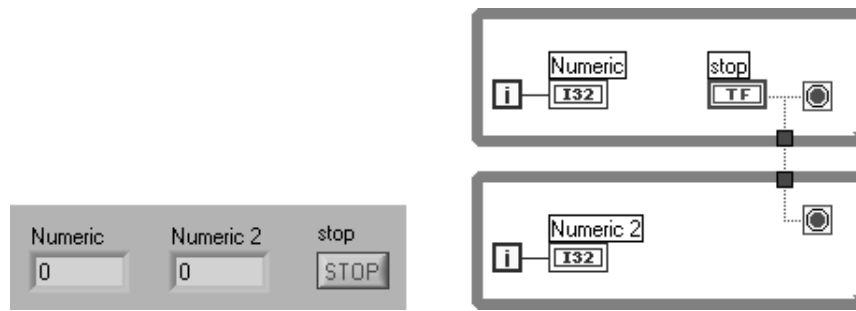


Figure 1-164
A tricky VI.

Why does this not work as described? The top While Loop never finishes until the `stop` button is pressed. The rules of dataflow view both loops as big nodes. Data cannot flow from the top loop to the bottom loop until the top loop has stopped, so the loops are never running simultaneously. Can you figure out a way to have the `stop` button stop both running loops?

Unless you are experienced with LabVIEW's locals or globals already, or are another Einstein, you probably could not figure this one out. One slick way to do what we want would be to use a local that refers to the `stop` button. Build the VI in Figure 1-164, saving it as `Locals Example.vi`, and verify that it does not work as described. Now, pop up on the `stop` button's terminal on the block diagram, and select **Create»Local Variable**. Clone this local so you have two locals. Make one a *Write Local*, and leave the other one as a *Read Local* (this is analogous to controls vs. indicators), by popping up on the local and selecting the **Change To Write** or **Change To Read** menu item if needed. Wire your diagram as in Figure 1-165.

1.14 Deviations from Dataflow: Local and Global Variables

141

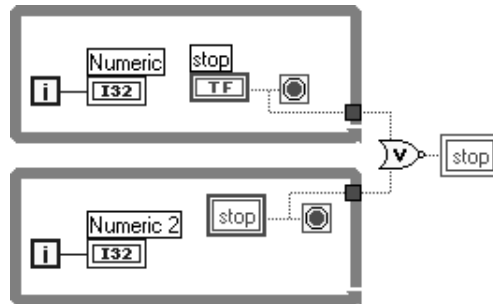





Figure 1-165
Another tricky VI with locals.

Notice the broken **Run** button . You will need to pop up on the **stop** button on the front panel and change its mechanical action to something that's non-latching. Since buttons with latching cannot have locals, we must pop the front panel button up ourselves with that far right local in the block diagram in Figure 1-165. We are writing a False to the **stop** button there, which pops it up.

Open `Password Caller.vi`, and run it twice with the correct password, `foo`. Notice that the second time the password dialog box pops up, your password is showing! This is not a very good feature, especially from a security perspective, so let's fix it. Go to the block diagram, double-click `Password.vi` (with the Positioning tool  or Operating tool ) , then open its block diagram. Pop up on the `Enter your password: string`'s terminal, and select the **Create»Local Variable** option. Make sure it's a Write Local (its border should be one pixel thick, not two), as we'll be writing data to it.

Modify `Password.vi` as in Figure 1-166 (that new box is an Empty String constant).

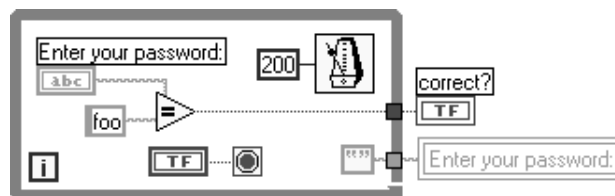


Figure 1-166
Locals make our previously created password subVI more useful.

This will automatically empty the `Enter your password:` string after the VI has closed. Save `Password.vi`, and close it. Now, run `Password Caller.vi` a couple of times, and notice that your password string is empty when the password dialog box pops up *after* the first run.

Exercise 1.3

Open `High Level File IO.vi` that you created earlier, and delete the large String Indicator on the right, as well as any broken wires. Change the remaining string's label to `file contents`. Use this String Control both for writing data to your file and reading data back. Use a local to do the latter. Your front panel can be simplified as in Figure 1-167.

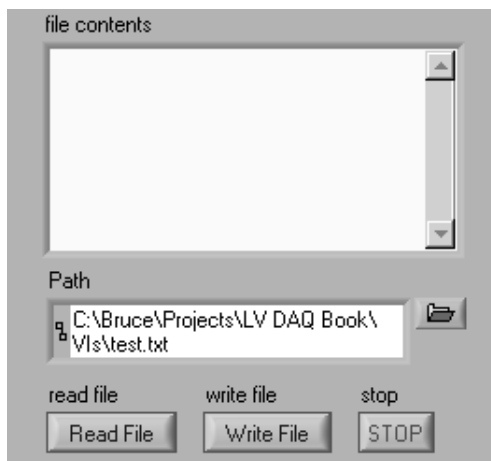


Figure 1-167

The VI shown in Figure 1-153 can be simplified to this.

Solution

Simply create a local variable (a Write Local) for the `file contents` string control, then wire this local to where your deleted String Indicator used to be, as shown in Figure 1-168.

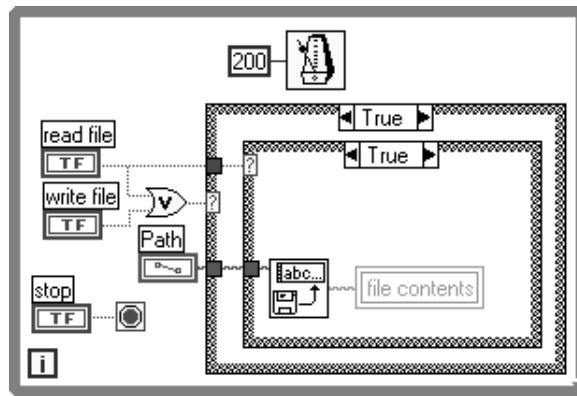


Figure 1-168




The VI shown in Figure 1-153 can be simplified to this by using a local.

Save this new, improved VI again.

Bonus: You can modify this VI further to read and write binary files, since the string is capable of viewing hex data, should you ever have the need.

1.14.2 Global Variables

Global variables (globals, for short) are somewhat similar to locals, in that the same piece of data can be referenced from multiple places within a block diagram. However, globals exist on their own special front panel, unlike locals, which always require a preexisting front panel control or indicator. More importantly, any global can be used in any number of VIs, unlike locals, which are restricted to just one VI. If you really must use the global, you can find it in the **Functions»Structures** palette. Here is a step-by-step guide on how to create and use a global:

1. Drop a Global Variable on the block diagram from the **Functions»Structures** palette. The empty global looks like this: 
2. Double-click this global (with the Positioning tool  or Operating tool ). You are now looking at your new global's very own front panel. A global's front panel is special because it has no block diagram.

3. Drop any control, such as a simple Digital Control, on this front panel. Be sure to label it (for example, `global 1`); otherwise, you cannot use it.
4. Save the global as `Globals.vi`.
5. Back on the block diagram from step 1, pop up on your global and select the name you chose for its label. If you chose the name `global 1`, your global should now look like this on the block diagram:



I find it convenient to have all the globals for any given project in the same front panel. I always call it `Globals.vi` out of habit, as this habit always makes it easy for me to find my projects' globals. Be careful not to use globals from this `Globals.vi` in other projects, though. If you copy VIs from one project to another that references these globals, you should also copy your `Globals.vi`.

Time for the mandatory warning.

1.14.3 Why You Should Avoid Locals and Globals

Locals and globals are dangerous! From the lingo of digital hardware designers comes the term *race condition*, which means two events will occur in random order, and the proper behavior of the system requires that one of these events occurs first—therefore the system is unreliable.

It is very easy to accidentally produce a race condition whenever you're using globals. The example in Figure 1-169 may work perfectly, in which one loop generates 50 points of data at 10 Hz, and the other loop charts those 50 points.

However, this VI is unreliable; there is no guarantee that the loops will stay synchronized! The bottom loop could easily miss some data.

In general, locals and globals are safe to use whenever you can guarantee that their data is valid whenever they're being read. For example, I will write a value to a global only once, before I ever read it—then I'll read it at multiple points in my program. I must take extra care *not* to read it before I write to it the first time! Things can get confusing when your locals or globals are inside a loop, where one loop reads and the other writes to the same global (as shown Figure 1-169), so don't call me if you get stuck with locals *or* globals!

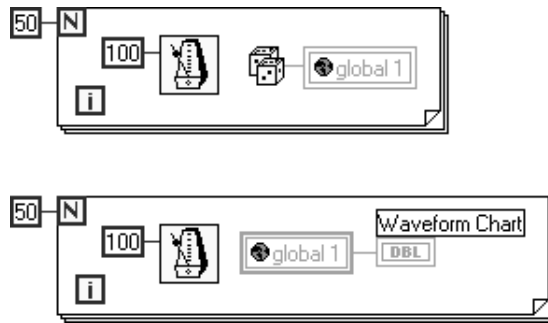


Figure 1-169
Global variables used incorrectly.

The other downside to locals and globals is that an extra copy of their data is generated for every Read Local or Read Global (*read* here means data is read from them by the block diagram, like a control). This can degrade performance if the local or global data is a large array or other type of large data (containing many bytes).

1.15 PROPERTY NODES

Property Nodes, formerly called Attribute Nodes, are one of my favorite features in LabVIEW. With them, you can programmatically make a front panel object invisible, change its color, move it around, and so on. Different types of front panel objects have different types of properties you can change. Most objects allow you to change their visibility, and most objects, like the graph, have properties that are specific only to their object's type. For example, a graph might allow you to change a plot's color, but a numeric would not, since it has no plots.

Create the front panel objects shown in Figure 1-170 on a new VI (a Boolean and Numeric control).

To create a Property Node, simply go to the front panel object, or better yet, its block diagram terminal, pop up on it and select **Create»Property Node**. Do this now with the Numeric control, and find the new Property Node on the block diagram. To begin with, the Property Node should look like the item on the left in Figure 1-171.

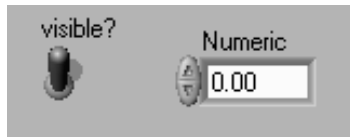


Figure 1-170
A Boolean and Numeric control.

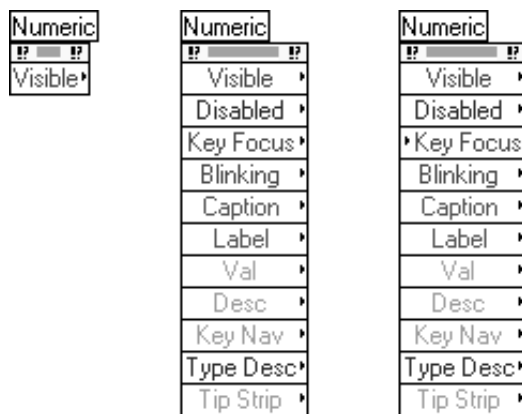



Figure 1-171
These are Property Nodes showing various properties of Boolean and Numeric controls that you can read or write.

The middle Property Node in Figure 1-171 is a result of growing the left Property Node, thus showing more properties. When a Property Node is showing multiple properties, it executes them from top to bottom.

The right Property Node in the figure is a result of popping up on the **Key Focus** property and selecting the **Change To Write** menu item. Property Nodes let you know whether data is going in or coming out by means of the little black arrows, shown in the figure. This is unlike many other one-terminal block diagram nodes, like front panel objects' terminals and local variables, that let you know this with a one-pixel or two-pixel thick border.

You can pop up on any Property Node to change the property shown. Take a look at a few different properties by using the Operating tool  and left-clicking the Property Node with the Help window showing—the Help window will describe each property. Notice that different properties have different data types.

1.16 Printing

147

As an example, let's make a Digital Control invisible. Wire your block diagram as in Figure 1-172, then run the VI a few times with the `visible?` control in different states.



Figure 1-172

A simple illustration of the Property Node.

Open `Password Caller.vi` again, then open `Password.vi` from its block diagram. Open the block diagram of `Password.vi`. Pop up on the `Enter your password: string`'s terminal, and select the **Create»Property Node** option. Modify `Password.vi` as in Figure 1-173.

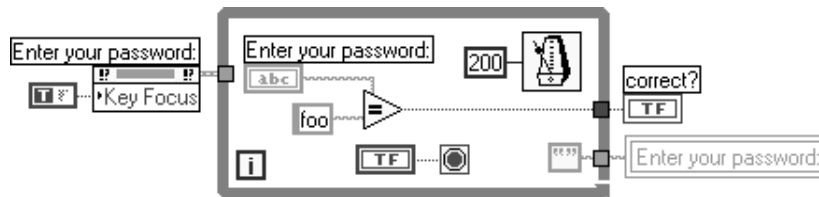


Figure 1-173

Our previously-created password VI is refined once again.

Save `Password.vi`, close it, and try it out now from `Password Caller.vi`. Notice that you can now immediately start typing your password when the dialog box pops up, because the `Key Focus` property places your cursor on the `Enter your password: string`.

1.16 Printing

Printing in LabVIEW can be very tricky. What works fine on one printer might not work so well on another. By now, most of these issues have been resolved, but be aware of this potential problem. When I worked on the LabVIEW development team, I was very glad that printing was not my department!

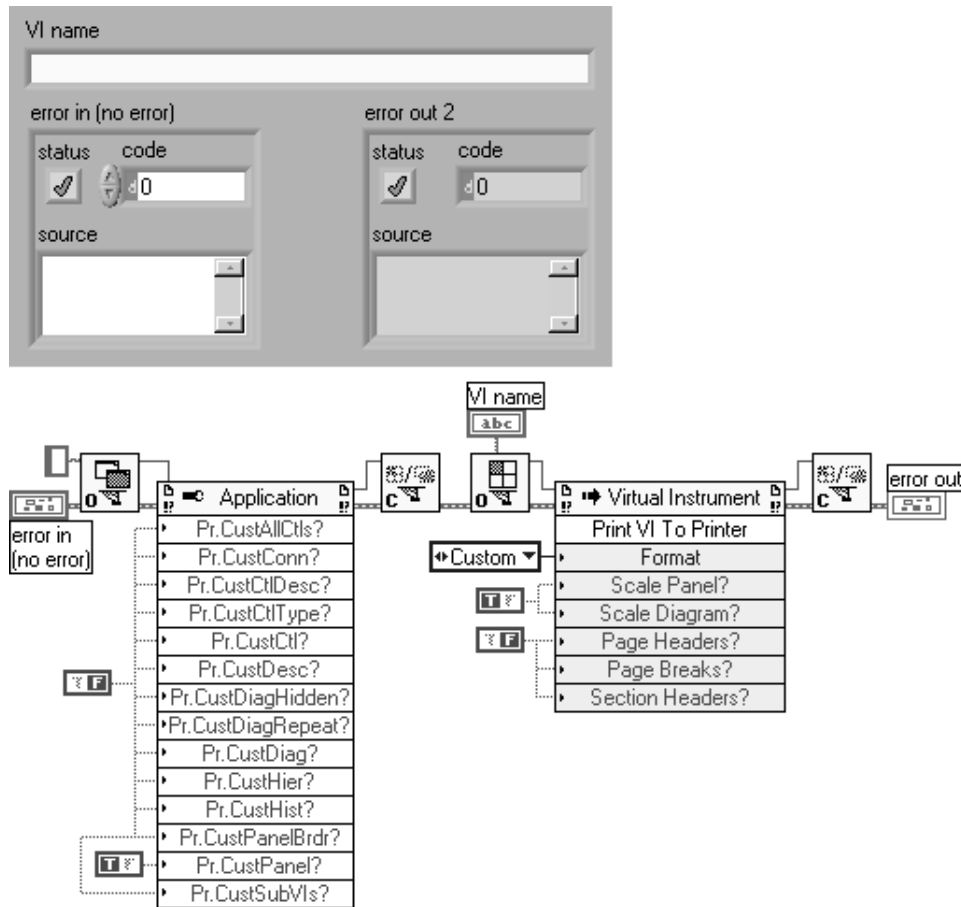


Figure 1-174

Property Nodes are useful for customizing printing.

Probably the simplest way to print your results in LabVIEW is to send your results to a subVI that is dedicated to printing. This subVI will be called your *print VI* hereafter. As you're developing your application, you can save time (and paper) by making this VI a dialog box that pops up and displays your results rather than actually printing them. Then, when you're ready to complete your application, change the print VI so that it no longer pops up, but so that it has its **Operate »Print at Completion** property set. Other handy printing tips:

1. Color the background (of the front panel) of your print VI white. This speeds up the printing, as the printer doesn't print any ink where it sees white.
2. If you have objects on the front panel of your print VI that you don't want printed, which you probably will in order to get data to your print VI, use the Property Node to make them invisible just before printing, or simply hide them with their terminals' **Hide Control** menu item.

If you want to get rid of the header information printed across the top of your page, you might need to get fancy and tweak the printing properties, using some advanced functions, as in Figure 1-174. The new functions shown here can be found in the **Functions»Application Control** palette. The two large nodes are the Property Node (different than our previously discussed Property Node) and the Invoke Node.

Finally, LabVIEW has some built-in report-generating functions found in the **Functions»Report Generation** palette.

1.17 Finding Objects in LabVIEW

LabVIEW has a very nice *find* feature. Hit <Ctrl-F>, and up pops a box like the one in Figure 1-175 (possibly with something other than the Add function shown).

This powerful utility lets you find objects, like functions or subVIs, or even text. In general, keep the **Search Scope** ring to **All VIs in Memory**, as shown. Sometimes you will want the **Include VIs in vi.lib** box checked, other times not. `vi.lib` is a folder in LabVIEW's directory that contains many of LabVIEW's built-in VIs.

When **Search for:** is set to **Objects**, as it is shown in Figure 1-175, you can quickly find any VI in memory by popping up on whatever **Select Object:** icon is showing and selecting the **VIs by Name...** option.

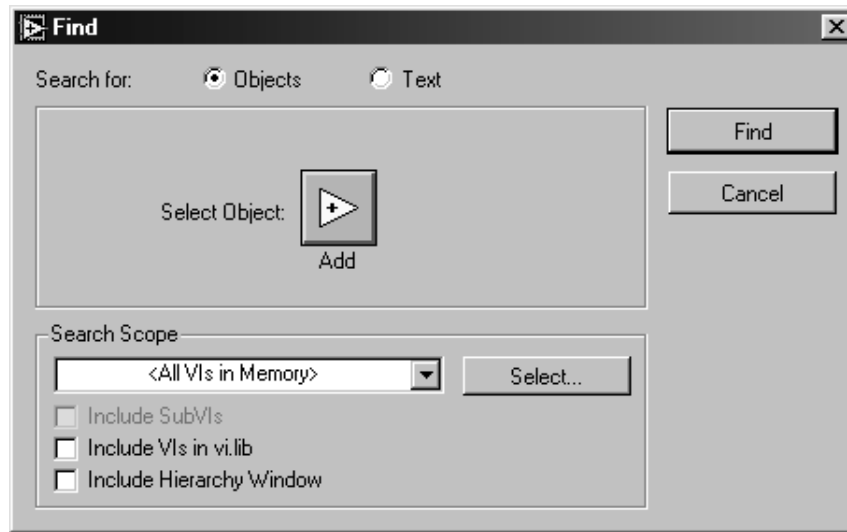


Figure 1-175
LabVIEW's find feature.

1.18 LEARNING MORE

As you might have guessed by now, you cannot learn everything about LabVIEW in one chapter of one book. This was just to prepare you for doing some useful LabVIEW/DAQ work—if you've thoroughly learned the material in this chapter, you're ready.

For more detail on LabVIEW in general, read a more in-depth LabVIEW book, such as the excellent *LabVIEW for Everyone*, described in Section 1.1.

After you have a good working knowledge of all information in this chapter, you may want to take a LabVIEW class, sponsored by NI. I've enjoyed teaching dozens of LabVIEW classes when I worked there (and afterwards). Unless you're a genius, you had better know some LabVIEW basics before you set foot in any of these classes! Becoming familiar with this chapter and working through the examples will adequately prepare you for the basic LabVIEW class.

Close any VIs you might have open with the **File»Close All** menu item, as we don't want our computer screen to become too cluttered.