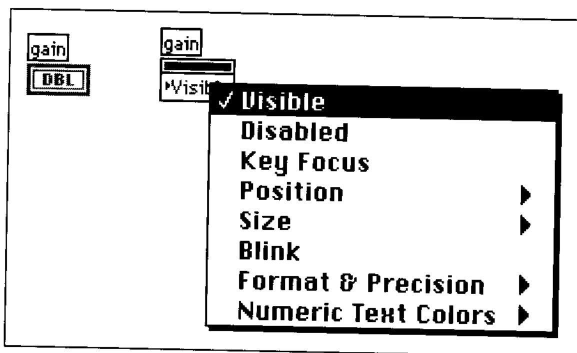
Attribute Nodes

With attribute nodes, first implemented in version 3 of LabVIEW, you can start making your program more powerful and a lot more fun. Attribute nodes allow you to programmatically control the attributes of a front panel object: things such as color, visibility, position, numerical format, etc. The key word here is *programmatically*, that is, changing the properties of a front panel object according to an algorithm in your diagram. For example, you could change the color of a dial to go through blue, green, and red as its numerical value increases. Or you could selectively present the user with different controls, each set of them appearing or disappearing according to what buttons were pressed. You could even animate your screen by having a custom control move around to symbolize some physical process.

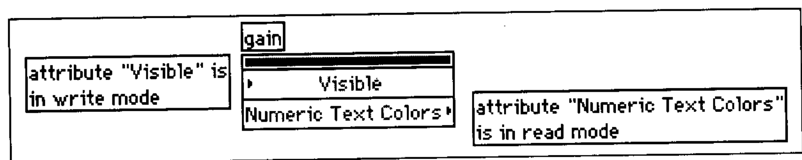
To create an attribute node, pop up on either the front panel object or its terminal, and select **Create➤Attribute Node**. A terminal with the same name as the variable will appear on the diagram. To see what options you can set in a variable's attribute node, click on the node with the Operating tool or pop up on the node and choose **Select Item➤**. Now you have the choice of which attribute or attributes you wish to select. Each object has a set of *base attributes*, and sometimes, an additional set of attributes specific to that object.



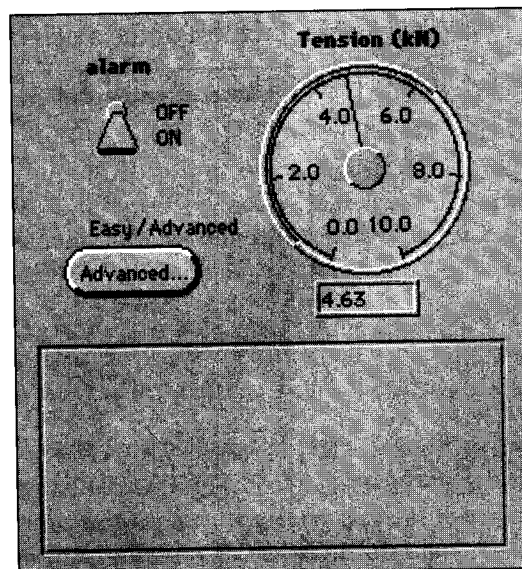
Just like with local variables, you can either read or write the attribute of an object. To change the mode of an attribute, pop up on it and select the **Change to...** option. The small arrow inside the attribute node's terminal tells you which mode it's in. An attribute node in write mode has the arrow on the left, indicating the data is flowing into the node, *writing* a new attribute. An attribute node in read mode has the arrow on the right, *reading* the

current attribute and providing this data. The same analogy we used for locals—read mode works like a control and write mode works like an indicator—holds for attribute nodes.

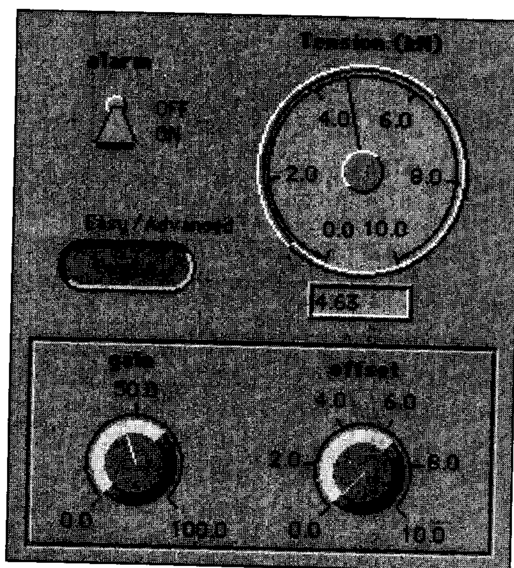
An interesting feature of attribute nodes is that you can use one terminal on the block diagrams for several attributes (but always affecting the same control or indicator). To add an additional attribute, you can use the Positioning tool to *resize* the terminal and get the number of attributes you need, in the same way multiple inputs are added to functions like **Bundle, Build Array**, etc. The following figure shows two attributes on the same terminal for the numeric control gain.



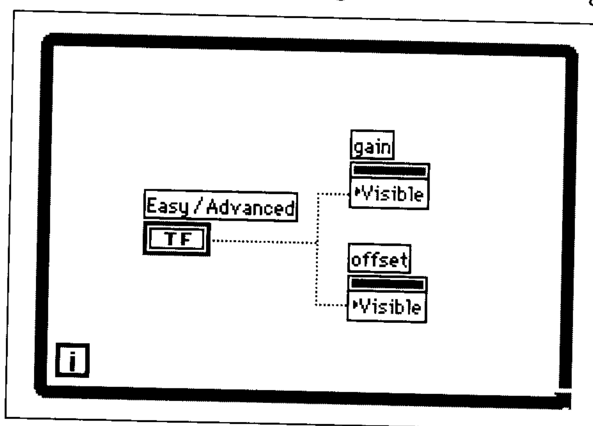
Let's look at a simple example. Suppose you wanted to have a front panel that would hide certain specialized controls except for those occasions when they were needed. In the following front panel, we see a tension gauge and a Boolean alarm switch. We include a button that says Advanced..., hinting at the possibility that if you pressed it, some really obscure and intricate options will pop up.



In this example, we've included two more controls, gain and offset, which are made invisible by setting their attribute nodes' option "Visible" to FALSE unless the button is pressed. If the Advanced... button is pressed, then ta-dah! The two knobs become visible.



The entire block diagram is encompassed in a While Loop like the one shown next to make the button control the visibility of the two knobs and thus give the "pop-up" effect. For simplicity's sake, we've hidden most of the block diagram in the following figure.



Often you will want to use more than one option in an object's attribute node. Remember, instead of creating another attribute node, you can select several options at a time by enlarging the

terminal with the Positioning tool (much like you enlarge cluster and array terminals). You will see each new option appear in sequence; you can later change these if you like by clicking on any item with the Operating tool or popping up on them and choosing **Select Item➤**.

What do each of the base options in an attribute node refer to?

- **Visible**: Sets or reads the visibility status of the object. Visible when TRUE, hidden when FALSE. This is often a better choice than coloring an object transparent, since transparent objects can accidentally be selected.
- **Disabled**: Sets or reads the user access status of a control. A value of 0 enables the control so the user can access it; a value of 1 disables the control without any visible indication; and a value of 2 disables the control and "grays it out."
- **Key Focus**: When TRUE, the control is the currently selected key focus, which means that the cursor is active in this field. Key Focus is generally changed by tabbing through fields. Useful for building a mouseless application. See Chapter 13 for more information on Key Focus.
- **Position**: A cluster of two numbers that respectively define the top and left pixel position of the front panel object.
- **Size:** A cluster of two numbers that respectively define the height and width in pixels of the entire front panel object.
- **Blink:** When TRUE, the front panel object blinks.
- **Format and Precision**: Sets or reads the format and precision attributes for numeric controls and indicators. The input cluster contains two integers: one for format and one for precision. These are the same attributes you can set from the pop-up menu of the numeric object.
- **Color**: Depending on the type of object, you may have several color options. The input is one of those neat color boxes that sets the color of the text, background, etc., depending on the object.

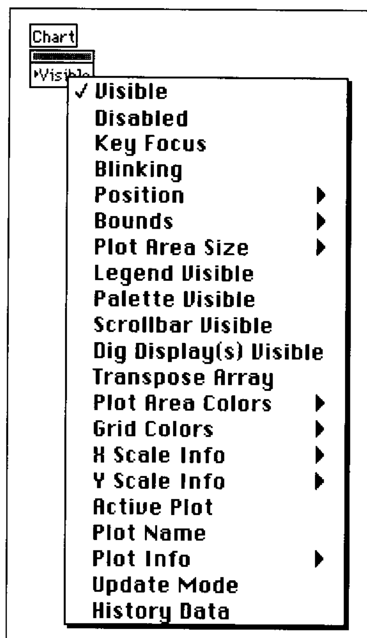
The Help window really is helpful when using attribute nodes. If you move the cursor onto the terminal of an attribute node, the Help window will show you what the attribute means, and what kind of data it expects. You can also pop up on the attribute node terminal and choose **Create Constant** to get the cor-

rect data type wired up immediately—this comes in very handy when the input is a cluster.

Almost all controls or indicators have the base attribute options. Most of them have many more, especially tables and graphs, which have as a many as 73 options! We won't even begin to go into most of these options, because you may never care about many of them and you can always look up the details in the manuals. The best way to learn about attribute nodes is to create some to go with your application and to play around with them. You'll find that attribute nodes are very handy for making your program more dynamic, flexible, and user-friendly (always good for impressing your nontechnical manager).

Another example

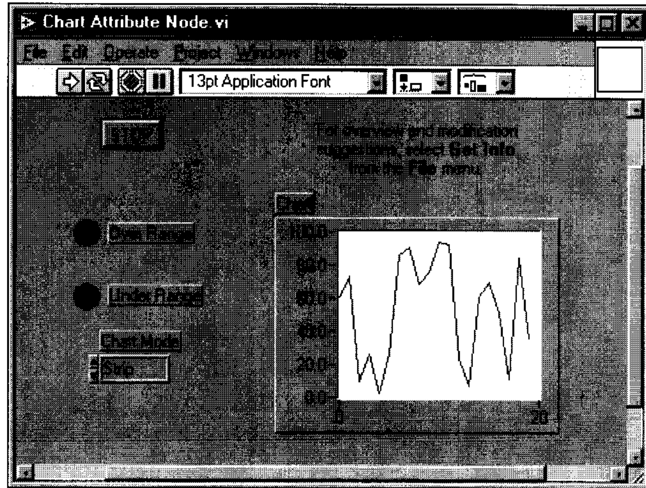
Graphs and charts have zillions of options in their attribute nodes, as you can see by clicking with the Operating tool on the terminal of a chart's attribute node.



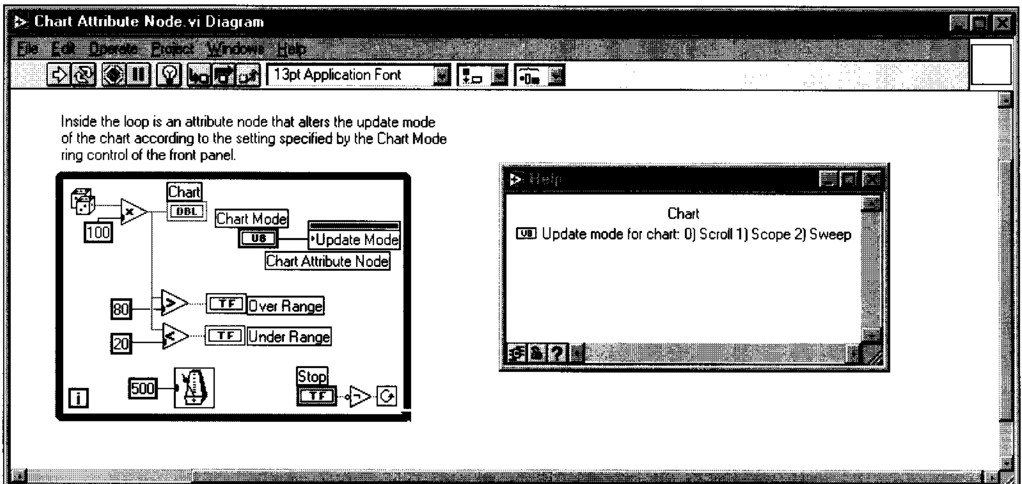
Some attribute nodes have several components (such as the Position attribute, which is a cluster of the left and top pixel coordinates). These attribute nodes, called compound attributes, are identified by the arrows indicating they have submenus. The

fact to notice is that you can access the compound attribute as a whole, or just select one of its components.

This next example, which can be found in the attribute node examples in the full version of LabVIEW in `examples\general\attribute.llb`, shows just one of the many aspects of a graph you can control programmatically. **Chart Attribute Node** lets you programmatically select one of three display types for the chart: Strip, Scope, or Sweep (if you need to, review Chapter 8, *Charts and Graphs*, to see what they do).

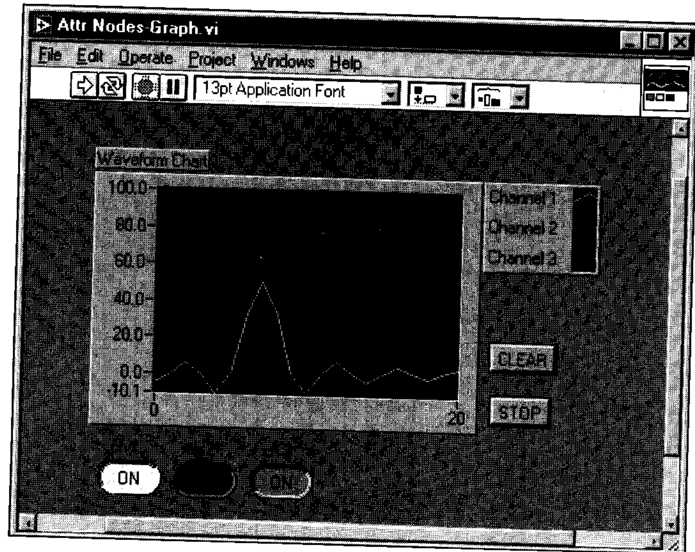


You can select the chart mode and watch it change even while the VI is running. The way this is done is through the **Update Mode** option on a chart's attribute node.

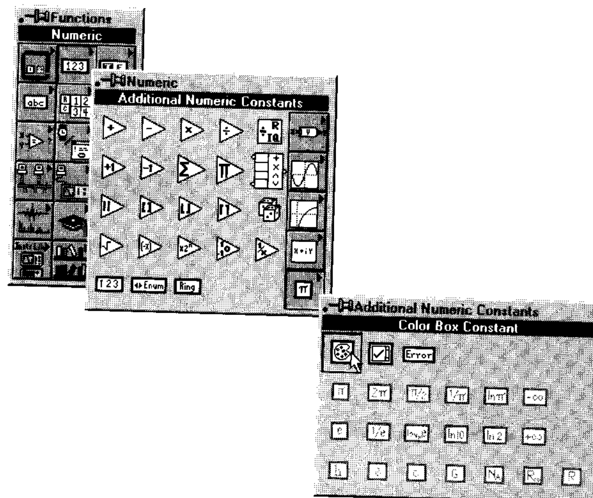


Activity 12-4

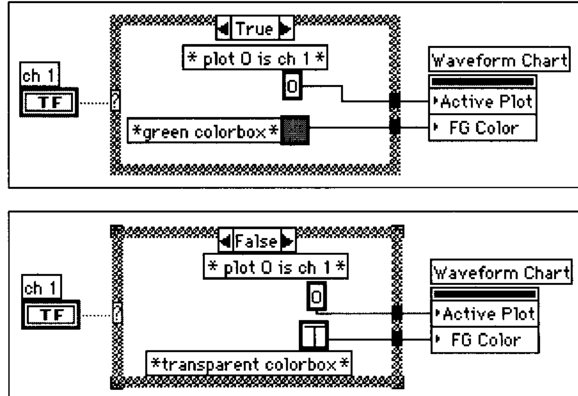
1. Write a VI that graphs three channels of data (either real-time data through a DAQ board, or random data). Let the user turn on or off any of the plots with three buttons. Since this is a chart and not a graph, the data will accumulate along with the old data.
2. Add a "CLEAR" button that clears the chart.



*LabVIEW includes a "transparent" color. To set a color attribute, you can create a cluster of **Color Box** constants. The **Color Box** is actually a numeric type, found buried deep in the **Numeric➤Additional Numeric Constants** palette. Pop up on a **Color Box** constant to select its color (remember the "T" is the transparent color).*



When using a multiplot graph or chart, you can only affect the attributes of one plot at a time. The plots are numbered 0,1, ...,n for attribute node purposes. A specific attribute, called **Active Plot**, is used to select the plot for the attributes you are modifying or reading. In this activity, you will need to build some case statements like the following:

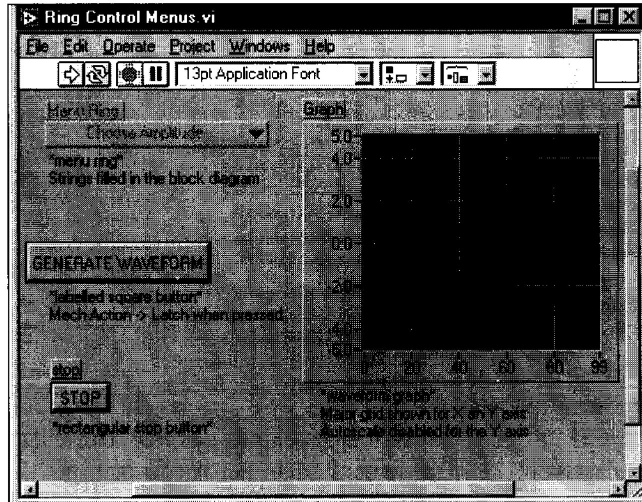


*To clear a chart, use the **History Data** attribute of the chart. Then wire an empty array to this attribute node.*

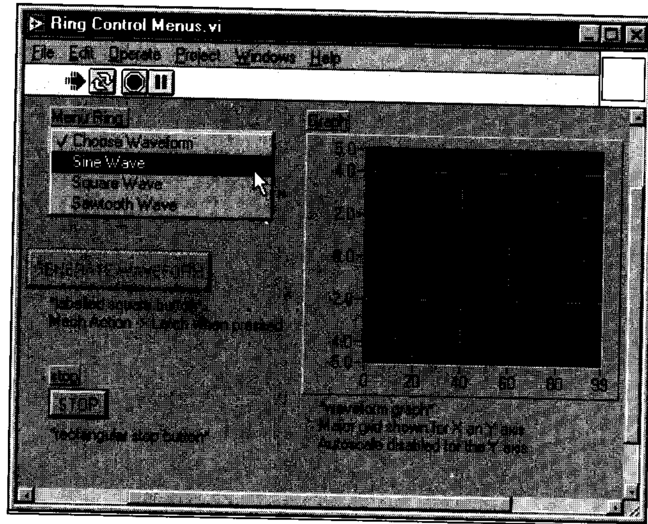
3. Save your VI as **Attr Nodes-Graph.vi**.

Activity 12-5

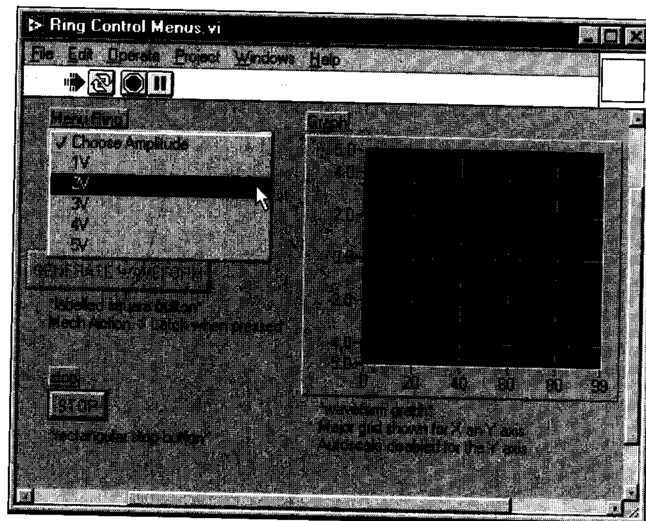
Ring controls, along with attribute nodes, can create some powerful interactive VIs. Examine the following example, called **Ring Control Menus**, which is in CH12.LLB.



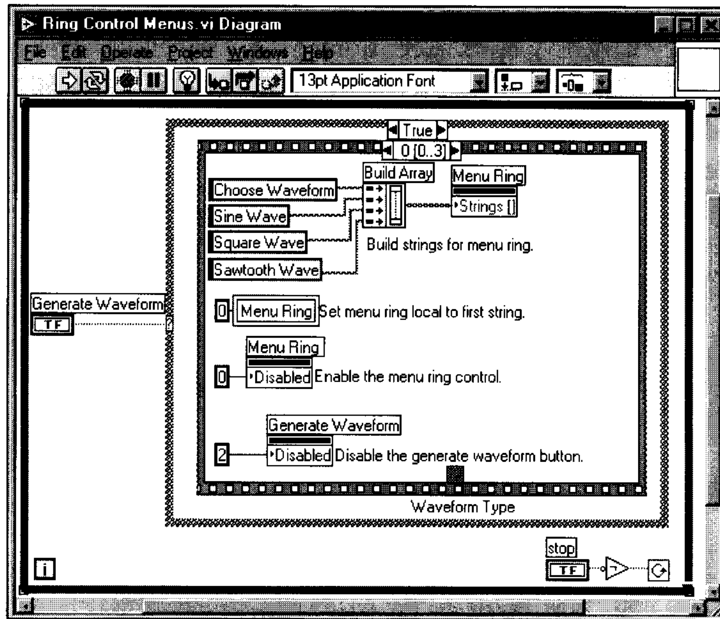
A pull-down menu ring, initially empty, will be used to query the user for the type of output. Once GENERATE WAVEFORM is pressed, the ring control will become enabled and say "Choose Waveform" and present several options: sine, square, or sawtooth wave.



Next, the same ring control *will change* to let the user choose the output amplitude. Finally, the waveform is generated at the DAQ board and graphed.



Here's a peek at part of the block diagram—check out the software solution for more detail.



Other LabVIEW Goodies

A "miscellaneous" category is always hard to avoid—like the ever-growing one in our budgets. Many miscellaneous functions are located in the **Advanced** palette. We'll cover some of the functions in that menu as well as some others that don't quite fit anywhere else but nevertheless can be quite useful.

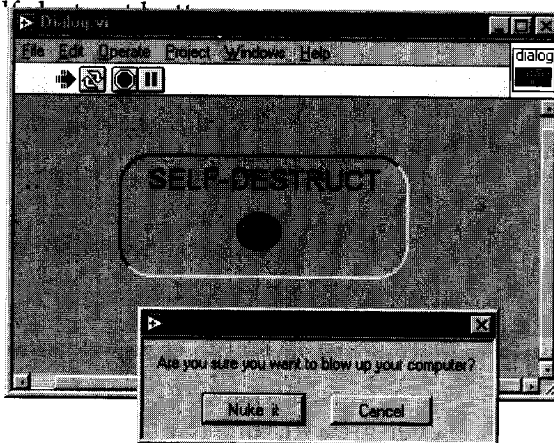
Dialogs

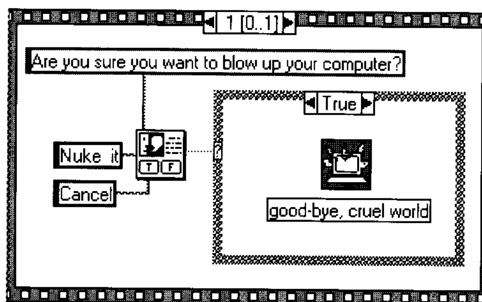
LabVIEW carrying on a real conversation with you? For now, you can make LabVIEW pop-up dialog windows that have a message and often some response buttons, like "OK" and "Cancel," just like in other applications. We've mentioned dialog functions before, but you may find the review and additional information useful.

The dialog functions are accessible from the **Time & Dialog** palette. LabVIEW provides you with two types of pop-up dialog boxes: one-button and two-button. In each case, you can write the message that will appear in the window, and specify labels on the buttons. The two-button dialog function returns a

Boolean value indicating which button was pressed. In any case, LabVIEW halts execution of the VI until the user responds to the dialog box. The dialog box is said to be *modal*, which means that even though windows (such as other VIs' front panels) will continue to run and be updated, the user can't select any of them or do anything else in LabVIEW via the mouse or keyboard until he or she has dealt with the dialog box.

As an example, suppose you wanted to add a dialog box to confirm a user's choice on critical selections. In the following example, a dialog box is presented when the user presses the computer's se





LabVIEW also provides dialog boxes for reading and saving files. These are discussed further in Chapter 14, *Communications and Advanced File I/O.*

Dialog boxes are a nifty feature to make your application have the look and feel of a true Windows or Mac executable, but don't overdo them. Remember, LabVIEW programs already provide

the interactive graphical interface, so having unnecessary dialogs popping up can make your application more cumbersome and cluttered. It's best to reserve the dialogs for very important notifications and/or confirmations of a user's choice. Also, note that LabVIEW does not let you have more than two response buttons in a dialog box. For more elaborate "dialogs," you're best off writing a subVI whose front panel contains the dialog options you'd like. You can make this subVI pop open and even customize its window appearance (see the next chapter on how to do this).

Occurrences

The *occurrence* functions, **Generate Occurrence, Wait On Occurrence,** and **Set Occurrence**, all have kind of scary-sounding names—which is probably why we know hardly anybody who uses these functions. Nevertheless, occurrences are a powerful programming tool. You can use occurrences to force one or more parts of the block diagram to wait for something (an occurrence!) without using messy While Loops or globals. They can also speed up your program by saving processor time that would otherwise be spent examining conditions in loops. The occurrence functions are accessible from the **Occurrences** subpalette of the **Advanced** palette.

Occurrences are somewhat like Booleans—they're in one of two states: set or waiting to be set. This is what each of the functions do:

Generate Occurrence

Generate Occurrence creates an *occurrence refnum* (a magic number LabVIEW uses to keep track of the occurrence functions) that you pass on to the other occurrence functions.

Wait On Occurrence

Wait On Occurrence forces the part of the structure that contains it to halt execution until the occurrence is set. You can optionally wire a timeout to this function.

Set Occurrence

Set Occurrence triggers the occurrence; that is, it changes the "state" of the occurrence passed to it. When this function is executed, all other **Wait On Occurrence** functions that are wired to the *same* refnum detect it and allow the structure they're in to continue, regardless of what part of the block diagram these structures are in.

Occurrence functions operate on the part of the block diagram structure they're contained in, such as a particular frame of a Sequence Structure or a particular case of a Case Structure. In this next example, an occurrence is set if the continue? Boolean becomes TRUE (presumably a button available to the user). When the Sequence Structure at the right reaches frame 2, it does not continue to the next frames until the occurrence is set. Essentially, it allows just *part* of the Sequence Structure to run parallel to the While Loop, while the other part (after frame 2) must wait until the While Loop has concluded or the continue? button is pressed.

Generate Occurrence
continue?
TF
True
Set Occurrence
exit
TF
2 [0..3]
This frame and the next ones do not execute until the occurrence is set.
Wait on occurrence

Note how both the **Set Occurrence** and **Wait on Occurrence** functions require an input from **Generate Occurrence**. You always need to provide this **Generate Occurrence** because in the case of multiple occurrences, the **Set Occurrence** and **Wait on Occurrence** need to know which one you're talking about. In other words, the **Generate Occurrence** doesn't really generate anything except the occurrence refnum used by the other functions. Another important detail you should notice in the above example is the second **Set Occurrence** wired to execute after the While Loop is finished. Think about what could happen if we left this out.

*When using occurrence functions, always remember to wire a **Set Occurrence** as an "escape" in your block diagram for all possible outcomes. Check your program to make sure it won't wait forever at a **Wait On Occurrence** event.*

Often you can achieve the same results using local variables as you would with occurrence functions. The advantage of occurrence functions is that they are slightly more memory- and time-efficient.

Saying "NO" harshly

LabVIEW gives you a couple of functions to abort the execution of your code immediately: **Stop** and **Quit LabVIEW**. Like the occurrence functions, these functions are also in the **Advanced** palette.

Stop Function

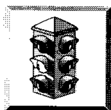
Stop has a Boolean input, that when TRUE (which is the unwired default input), halts execution of the VI and all its subVIs, just as if you had pressed the Abort button on the Toolbar.

Quit LabVIEW Function

Quit LabVIEW does just that, when its input is TRUE (also the default input). Be careful with this one!

Some of us reminisce about a third function that was included back when LabVIEW was in version 2.2: **Shutdown**. On certain Macintosh models, this function would effectively turn off the whole computer, monitor and all. We never understood how it could be useful, but it was fun to leave a colorful VI running with one big button that said "DO NOT PRESS THIS BUTTON" on someone else's Mac, and then just wait to see who would be too curious. The button was, of course, connected to the operating system's Shutdown function. It was a good way to amuse ourselves with the less experienced LabVIEW users.

On a serious note, however, you should use these functions with caution. It is considered bad programming etiquette (in any language) to include a "hard stop." Strive to always have a graceful way to exit your program.



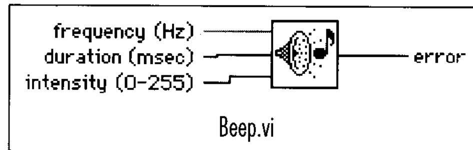
*Make sure your VI completes all final tasks (closing files, finishing the execution of subVIs, etc.) before you call the **Stop** function. Otherwise, you may encounter unpredictable file behavior or I/O errors, and your results can be corrupted.*

Sound

A very nice or very obnoxious feature to have in many applications is sound, depending on how you use it. An audible alarm

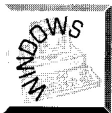
can be very useful in situations where an operator cannot look at the screen continuously during some test. A beep for every warning or notification, however, can be downright annoying. So use sounds with some thought to how often they will go off.

LabVIEW gives you access to the operating system's beep, through the **Beep** function. **Beep** is located in the **Advanced** palette.



Beep.vi

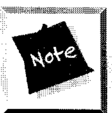
In the MacOS, you can set the frequency, intensity, and duration of the sound.



In Windows, these inputs are ignored and only the standard system beep is used. However, LabVIEW has the ability to call *.WAV sounds. The full version of LabVIEW includes an example VI called **Play Sound.vi** (you'll find it in `examples/dll/sound/plysnd.llb`). This VI calls external system code (see next section) that makes use of a Windows multimedia sound DLL.

Calling Code from Other Languages

What happens if you already have some code written in a conventional language (such as C, Pascal, FORTRAN, Basic) that you'd like to use? Or if for some reason you just miss typing in all those semicolons in your familiar text-based code? LabVIEW does give you some options for interfacing with code from other languages. If you are thinking about writing *all* your code in C or C++, you should check out LabWindows/CVI® (available from National Instruments), a programming environment very similar to LabVIEW; the main difference is that C code replaces the graphical block diagram. But if you'd like to (dare we say it?) actually have *fun* programming, then stick to LabVIEW and use conventional code only when you have to.



You may want to skip this section if you aren't familiar with writing code in other programming languages such as C, or if you don't expect to need to interface LabVIEW and external code.

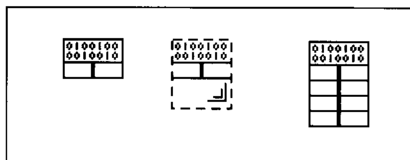
You have two options for calling external code in Windows: you can call a Dynamic Link Library (DLL) using the *Call Library* function, or you can call the externally-generated executable code directly by using the *Code Interface Node (CIN)*. Both of these functions are accessible from the **Advanced** palette. Under Windows 3.1, you can call 16-bit DLLs, and under Windows 95, you can call 32-bit DLLs. LabVIEW 4 now also allows you to call code from LabWindows/CVI front panels by converting the C functions into subVIs.

In the MacOS, you can also use the Code Interface Node (CIN) to call executable source code. On Power Macintosh systems, you can use the Call Library function to communicate with Code Fragment libraries. On the older 680x0 Mac systems, unfortunately you cannot communicate with the Apple Shared Library (ASL), like you can with DLLs on Windows systems.

Using a CIN involves basically the following steps:

1. Place the CIN icon on the block diagram (from the **Advanced** palette)

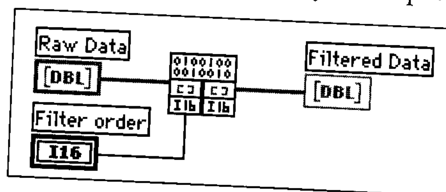
2. The CIN has terminals for passing the inputs and outputs from the code. By default, the CIN has only one pair of terminals. You can resize the node to include the number of parameters you need.

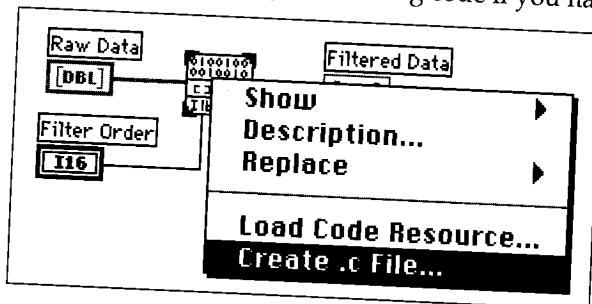
   By default, each pair of terminals is an input-output: the left terminal is an input, and the right terminal is an output. However, if a function returns more outputs than inputs (or has no input parameters at all), you can change the terminal type to **Output-Only** by popping up on that terminal and choosing this option.

3. Wire the inputs and outputs to the CIN. You can use any LabVIEW data type when wiring the CIN terminals (of course, it *must* correspond to the C parameter data type for the function you're calling). The order of the terminal pairs on the CIN corresponds to the order of the parameters in the code. In the following example, we call a CIN that filters the Raw Data input array and passes the

output into the Filtered Data array. Notice also how the terminals, once wired, indicate the type of data you are passing.



4. Create a `.c` file by selecting this option from the pop-up menu. The `.c` file created by LabVIEW, in the style of C programming language, is a template in which you write the C code. With a little effort, you can paste in your existing code if you have it.



5. Compile the CIN source code. This step could be tricky, depending on what platform, compiler, and tools you are using. You have to first compile the code with a compiler that LabVIEW supports, and then, using a LabVIEW utility, modify the object code so LabVIEW can use it.

6. Load the object code into memory by selecting **Load Code Resource...** from the pop-up menu. Select the `.lsb` file you created when you compiled the source code.

Once all these steps are completed successfully, you're calling C code in the CIN as if it were a subVI, with one major exception: calls to CINs execute synchronously, meaning that, unlike most other tasks in LabVIEW, they do not share processor time with other LabVIEW tasks. For example, if you have a CIN and a For Loop at the same level in the block diagram, the For Loop halts execution until the CIN finishes executing. This is an important consideration if your timing requirements are tight.

One important final fact about CINs: They are most definitely NOT portable across platform. If you compile your LabVIEW code with a CIN on a Power Mac and then try to run it under Windows,

it won't work. The way around this is to rebuild the CIN using a C compiler for the platform that LabVIEW is running on. At press time of this book, the C compilers compatible with LabVIEW are:

Windows 3.1	Watcom C
Windows 95/NT	Microsoft Visual C++, Microsoft Win32 SDK C/C++ compiler
MacOS	THINK C (ver. 5 or above), Symantec C++ (ver. 8 or above), Metrowerks CodeWarrior, MPW from Apple
Solaris	Sun ANSI C compiler
HP-UX	HP-UX C/ANSI C compiler

There are many complex issues involved in LabVIEW communicating with external code. Since many of these are highly dependent on the processor, operating system, and compiler you're using, we won't attempt to go any further into discussing CINs or Call Library functions. If you'd like more details, contact National Instruments and request their application notes on this subject, or look at the *Code Interface Reference Manual* included in the LabVIEW manual set.

Fitting Square Pegs into Round Holes: Advanced Conversions and Typecasting

Remember *polymorphism*, discussed early in this book? It's one of LabVIEW's best features, allowing you to mix data types in most functions without even thinking about it (any compiler for a traditional programming language would scream at you if you tried something like adding a constant directly to an array—but not LabVIEW). LabVIEW normally takes care of doing conversions internally when it encounters an input of a different but compatible data type than it expected at a function.

If you're going to develop an application that incorporates instrument control, interapplication communication, or networking, chances are you'll be using mostly string data. Often you'll need to convert your numeric data, such as an array of floating-point numbers, to a string. We talked about this a little in Chapter 9. It's important to distinguish now between two kinds of data strings: *ASCII strings* and *binary strings*.

ASCII strings use a separate character to represent each digit in a number. Thus, the number 145, converted to an ASCII string, consists of the characters "1," "4," and "5." For multiple numbers (as in arrays), a delimiter such as the <space> character is also used. This kind of string representation for numbers is very common in GPIB instrument control, for example.