Advanced LabVIEW Functions and Structures

12

Local and Global Variables

Local and global variables are, technically speaking, LabVIEW structures. If you've done any programming in conventional languages like C or Pascal, then you're already familiar with the concept of a local or global variable. Up until now, we have read data from or written to a front panel object via its terminal on the block diagram. However, a front panel object has only one terminal on the block diagram, and you may need to update or read it from several locations on the block diagram or from other VIs.

Local variables (locals, for short) provide a way to access front panel objects from several places in the block diagram of a VI in instances where you can't or don't want to connect a wire to the object's terminal.

Global variables allow you to access values of any data type (or several types at the same time if you wish) between several VIs in

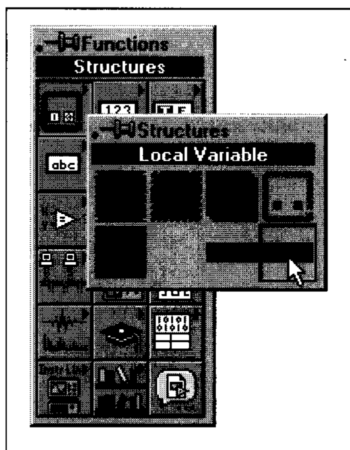
cases where you can't wire the subVI nodes, or when several VIs are running simultaneously. In many ways, global variables are similar to local variables, but instead of being limited to use in a single VI, global variables can pass values between several VIs.

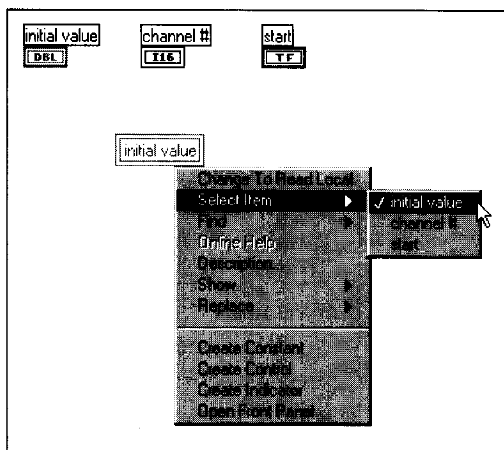
This section will teach you some of the benefits of using locals and globals, as well as show you some common pitfalls to watch out for.

■ Local Variables

Undefined Local

Local variables in LabVIEW are built-in objects that are accessible from the **Structures** subpalette in the **Functions** palette. When you select a local variable object, a node showing a "?" first appears to indicate the local is undefined. By clicking on this node with the Operating tool, a list of all current labeled controls and indicators will appear; selecting one of these will define the local. Or you can pop up on the local variable and choose **Select Item➤** to access this list. You can also create a local variable by popping up on an object's terminal and selecting **Create➤Local Variable.**



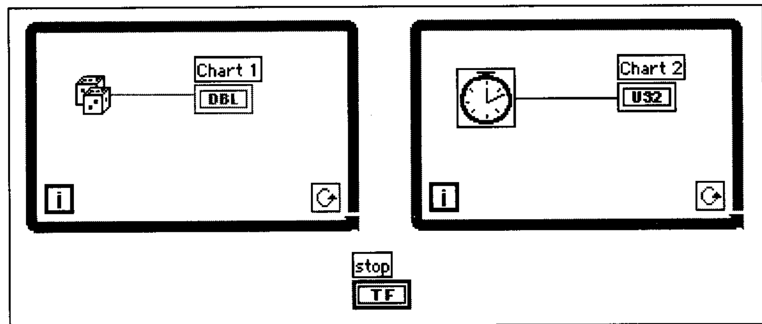


There are at least a couple of reasons why you might want to use locals in your VI:

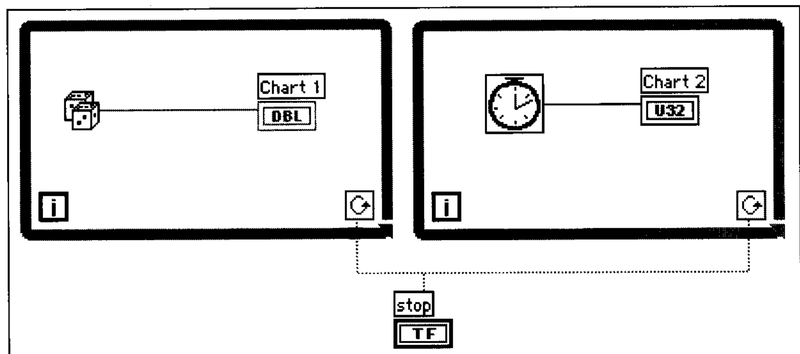
- You can do things, such as control parallel loops with a single variable, that you otherwise couldn't do
- Virtually any control can be used as an indicator, or any indicator as a control.

Controlling parallel loops

We've discussed previously how LabVIEW controls execution order through dataflow. The principle of dataflow is part of what makes LabVIEW so intuitive and easy to program. However, occasions may arise (and if you're going to develop any serious applications, the occasions *will* arise) when you will have to read from or write data to front panel controls and indicators without wiring directly to their corresponding terminals. A classical problem is shown here; we want to end the execution of two independent While Loops with a single Boolean stop control.

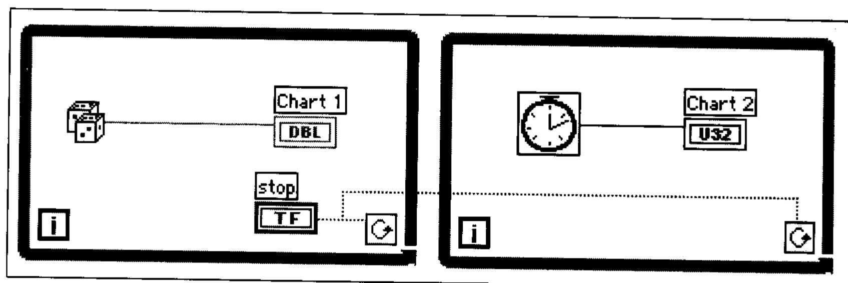


How can we do this? Some might say that we could simply wire the stop button to the loop terminals. However, think about how often the loops will check the value of the stop button if it is wired from outside the loops.



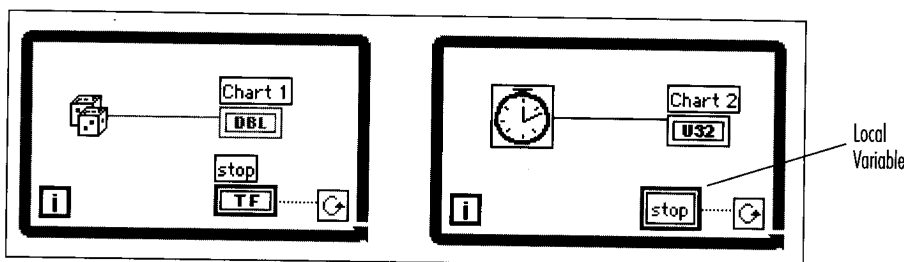
Wiring the stop button from outside the loops to both conditional terminals will not work, since controls outside the loops are not read again after execution of the loop begins. The loops in this case would execute only once if the stop button was FALSE when the loop starts, or execute forever if stop was TRUE.

So why not put the stop button inside one loop, as shown in the next picture? Will this scheme work?

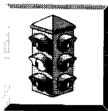


Putting the stop button inside one loop and stretching a wire to the other loop's conditional terminal won't do the trick either, for similar reasons. The second loop won't even begin until the first loop finishes executing—data dependency, remember?

The solution to this dilemma is—you guessed it—a local variable. Local variables create, in a sense, a "copy" of the data in another terminal on the diagram. The local variable always contains the up-to-date value of its associated front panel object. In this manner, you can access a control or indicator at more than one point in your diagram without having to connect its terminal with a wire.



Referring to the previous example, we can now use one stop button for both loops by wiring the Boolean terminal to one conditional terminal, and its associated local variable to the other conditional terminal.



*There's one condition to creating a Boolean local variable: the front panel object can't be set to **Latch** mode (from the **Mechanical Action** option). Although it isn't obvious at first, a Boolean in **Latch** mode along with a local variable in read mode produce an ambiguous situation. Therefore, LabVIEW will give you the "broken arrow" if you create a local variable of a Boolean control set to **Latch** mode.*

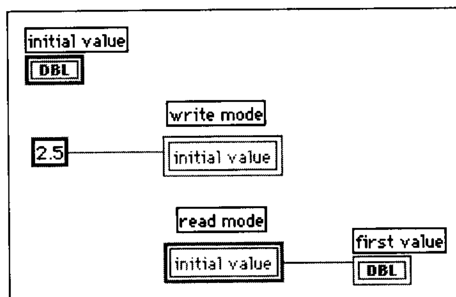
Blurring the Control/Indicator Distinction

One of the really nice features about local variables is that they allow you to *write to* a control or *read from* an indicator, which is something you can't normally do with the regular terminals of an object. Locals have two modes: *read* and *write*. A local variable terminal can only be in one mode at a time, but you can create a second local terminal for the same variable in the other mode. Understanding the mode is pretty straightforward: in read mode, you can read the value from the local's terminal, just as you would from a normal control; in write mode, you can write data to the local's terminal, just as you would update a normal indicator. Just remember this formula for wiring locals:

READ mode = CONTROL
WRITE mode = INDICATOR

Another way to look at it is to consider a local in read mode the data "source," while a local in write mode is a data "sink."

You can set a local variable to either read or write mode by popping up on the local's terminal and selecting the **Change To...** option. A local variable in read mode has a heavier border around it than one in write mode (just like a control has a heavier border than an indicator), as shown in the following figure.



Pay close attention to these borders when you are wiring the locals, to make sure you are in the correct mode. If you attempt to write to a local variable in read mode, for example, you'll get a broken wire—and it may drive you crazy trying to figure out why.

Last but not least, you must label controls and indicators in order to make locals for them. That is, when creating a control or indicator, if you don't give it a name, you won't be able to create a local variable for it.

Activity 12-1

As a simple example, let's say you want to create a knob that represents a timer; the user can set the time, and watch the knob turn as it counts down the time, just like those old-fashioned timers used in the kitchen. Obviously, the front panel object is going to have to be a control since the user will set the time, but it must also be able to accept block diagram values and "turn" accordingly to indicate the passage of time.

1. Build a front panel with a knob labeled `Timer (seconds)` and select the **Show➤Digital Display** option on it. Rescale the knob as shown in the following illustration.

Timer (seconds)
20 40
0 60

2. Create a local variable by selecting it from the **Structures** palette. You will get a local variable icon. Click on this icon with the Operating tool and select Timer (seconds). The local should be in write mode by default

3. Build the following simple block diagram.

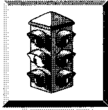
1000
Timer (seconds)
I16
Timer (seconds)
-1
>0
i

In this activity, the shift register is initialized with the value of Timer that was set at the front panel. Once loop execution begins, the shift register's value decreases once per second, and this value is passed to the Timer local variable, which is in write mode. The knob on the front panel rotates to reflect the changed value.

4. Set your knob to a desired time and run the VI. Watch the knob count down to zero!

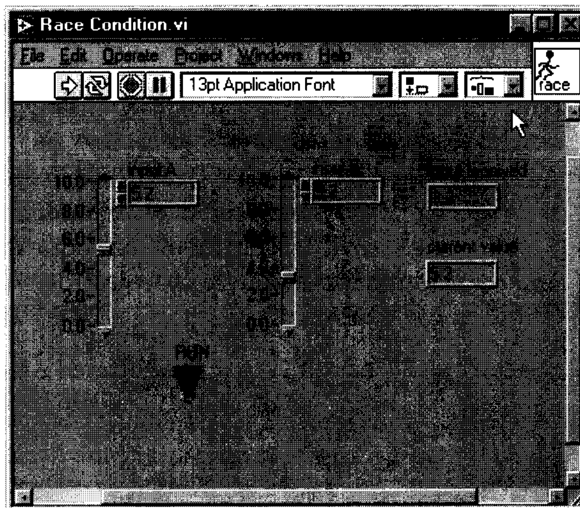
5. Save your VI as **Kitchen Timer.vi**

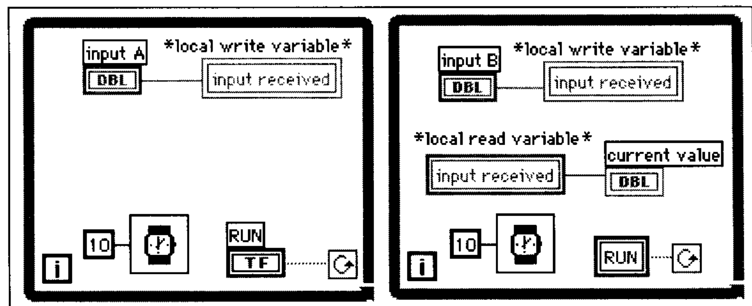
A nice feature to add to this activity would be a sound to alert you that the timer has reached zero, just like the old-fashioned ones (Ding!). Later in this chapter you will see how we can play sounds with LabVIEW.



Locals sound like a great structure to use, and they are. But you should watch out for a common pitfall when using locals: race conditions. A race condition occurs if two or more copies of a local variable in write mode can be written to in an unpredictable order.

There is a hazard to using locals and globals: accidentally creating a *race condition*. To demonstrate, build this simple example.





Notice the two **While Loops** controlled by Boolean RUN and a local variable RUN. However, the local variable input received is being written in the left loop as well as in the right loop. Now, execute this VI with RUN set to FALSE and different values on the two sliders. The loops will execute just once. What value appears at current value? We can't tell you, because it could be either value from input A or input B! There is nothing in LabVIEW that says execution order will be left to right, or top to bottom.

If you run the above VI with RUN turned on, you will likely see current value jump around between the two input values, as it should. To avoid race conditions, one must define the execution order by dataflow, Sequence structures, or more elaborate schemes.

Another fact to keep in mind is that every read or write of a local creates a copy of the data in memory. So, when using locals, remember to examine your diagram and the expected execution order to avoid race conditions, and use locals sparingly if you're trying to reduce your memory requirement.

Activity 12-2

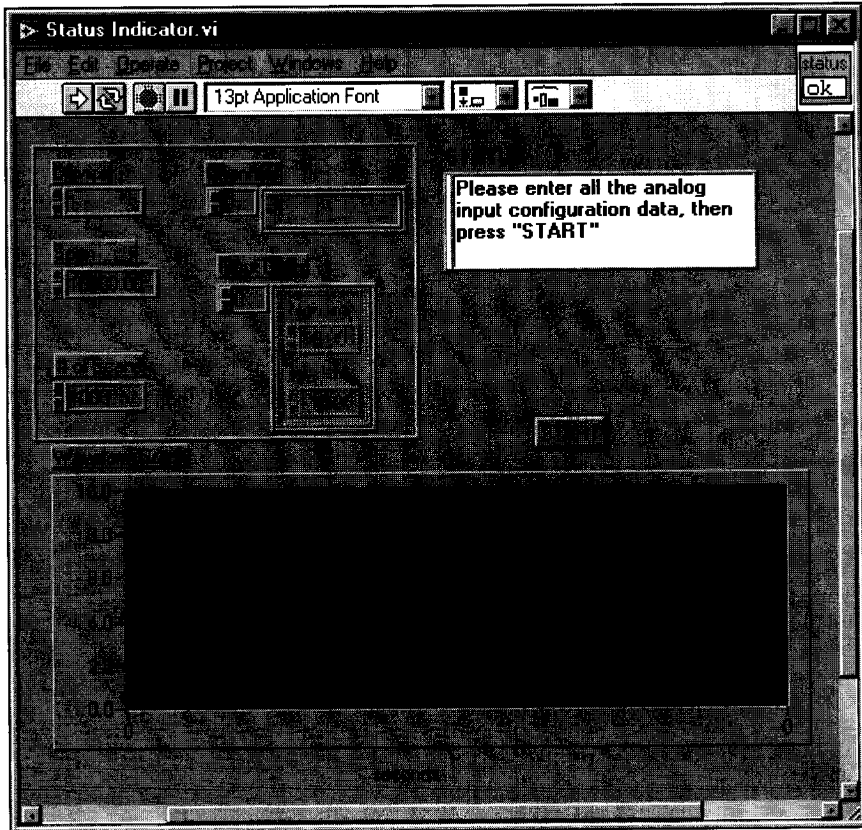
Another case where locals are very useful is in an application where you want a "status" indicator to produce a more interactive VI. For example, you may want a string indicator that is updated with a comment, or requests an input every time something happens in the application.

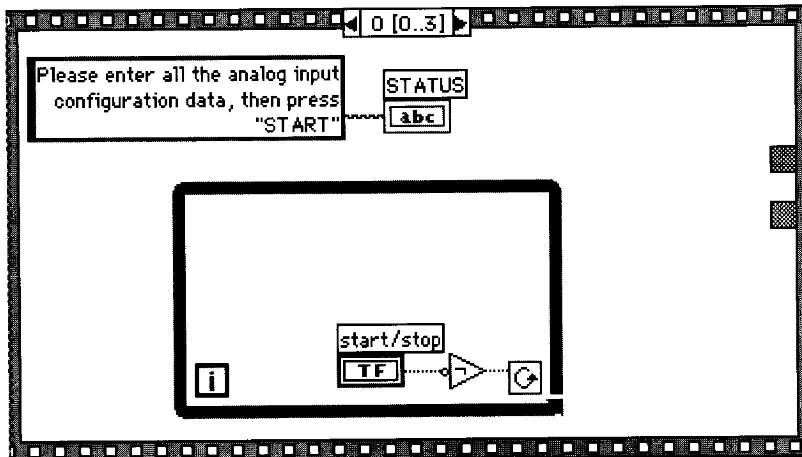
1. Build a simple data acquisition VI similar to one you wrote in Chapter 11 (such as **Buffered Analog In.vi**), but modify it to have a string indicator that tells the user:

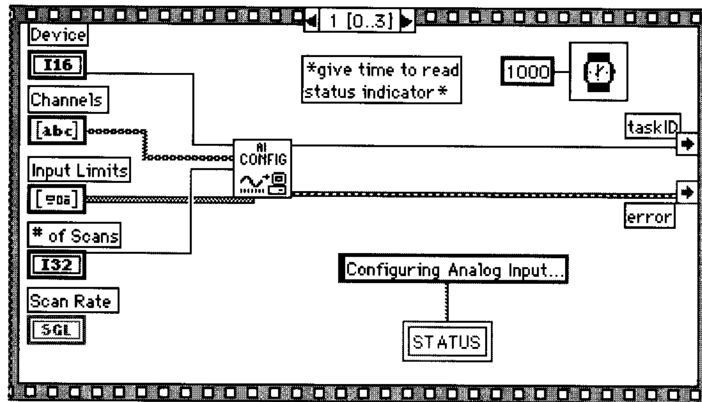
   - *when the program is waiting for input*
   - *when the program is acquiring data*
   - *when the program is graphing the data*
   - *when the program has stopped.*

   To help you get started, the following figures show the front panel and the first two frames of a Sequence Structure.

2. Save your VI as **Status Indicator.vi**.


Status Indicator.vi
File Edit Operate Project Windows Help
13pt Application Font
status
ok
Please enter all the analog input configuration data, then press "START"


0 [0..3]
Please enter all the analog input configuration data, then press "START"
STATUS
abc
start/stop
TF
i

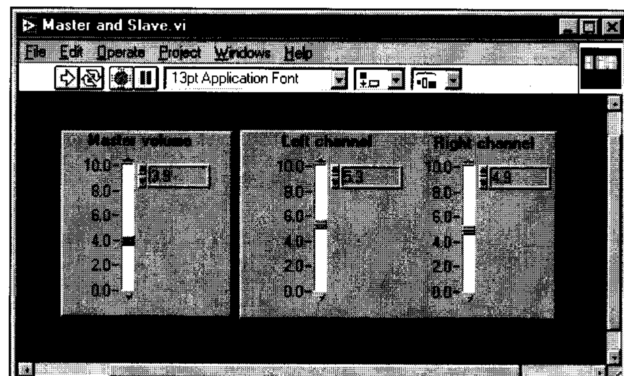


Activity 12-3

In many applications, you may want some type of "master" control that modifies the values on other controls. Suppose you wanted a simple panel to control your home stereo volumes. The computer presumably is connected to the stereo volume control in some way. In the VI shown next, a simulated sound control panel has three slide controls: left channel, right channel, and master. The left and right channel can be set independently; moving the master slide needs to cause increments or decrements in the left and right volumes proportionally.

1. Build the block diagram for the front panel shown below. The fun part about this is that by moving the master slide, you should be able to watch the other two slides move in response.

2. Save your VI as **Master and Slave.vi**.



You will need to use shift registers for this activity.

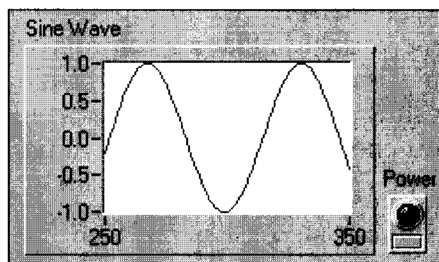
Global Variables

If you've never used globals in your programs before, congratulations! Global variables are perhaps the most misused and abused structure in programming. Globals are more often than not the cause of mysterious bugs, unexpected behavior, and awkward structures. Having said this, there are still a few occasions when you might want to resort to globals.

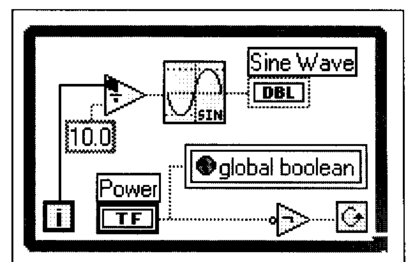
Recall that you can use local variables to access front panel objects at various locations in your block diagram. Those local variables are accessible only in that single VI. Suppose you need to pass data between several VIs that run concurrently or whose subVI icons cannot be connected by wires in your diagram. You can use a global variable to accomplish this. In many ways, global variables are similar to local variables, but instead of being limited to use in a single VI, global variables can pass values between several VIs.

Consider the following example. Suppose you have two VIs running simultaneously. Each VI writes a data point from a signal to a waveform chart. The first VI also contains a Boolean Power button to terminate both VIs. Remember that when both loops were on a single diagram, we needed to use a local variable to terminate the loops. Now that each loop is in a separate VI, we must use a global variable to terminate the loops. Notice that the global terminal is similar to a local terminal, except that a global terminal has a "world" icon inside it.

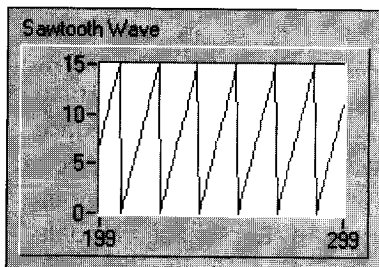
First VI Front Panel



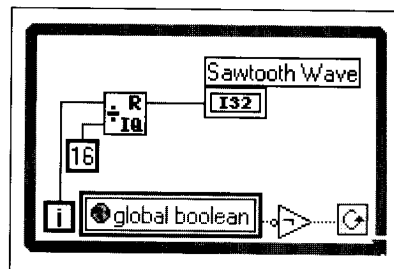
First VI Block Diagram



Second VI Front Panel



Second VI Block Diagram



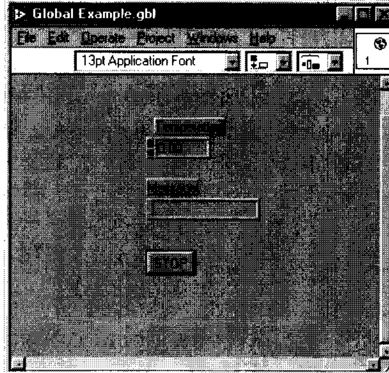
Creating Globals

Like locals, globals are a LabVIEW structure accessible from the **Structures** palette. And like locals, a single global terminal can be in write or read mode. *Unlike* locals, different VIs can independently call the same global variable. Globals are effectively a way to share data among several VIs without any having to wire the data from one VI to the next; globals store their data independently of individual VIs. If one VI writes a value to a global, any VI or subVI that reads the global will contain the updated value.

Undefined Global

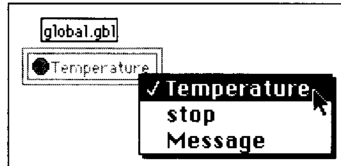
Once the global structure is selected from the palette, the icon shown at the left appears on the diagram. The icon symbolizes a global that has not been defined yet. By double-clicking on this icon, you will see a screen pop up that is virtually identical to a VI's front panel. You can think of globals as a special kind of VI—they can contain any type and any number of data structures on their front panels, but they have no corresponding diagram. Globals store variables without performing any kind of execution on these variables. Placing controls or indicators on a global's front panel is done in an identical fashion to a VI's front panel. An interesting tidbit about globals: It makes no difference whether you choose a control or an indicator for a given data type, since you can both read and write to globals. *Finally, be sure to give labels to each object in your global, or you won't be able to use them.*

A global might contain, as in the following example, a numeric variable, a stop button, and a string control.



Save a global just like you save a VI (many people use a ".gbl" extension when naming globals just to keep track of them). To use a saved global in a diagram, choose **Select a VI...** in the **Functions** palette. A terminal showing one of the global's variables will appear on the diagram. To select the variable you want to use, pop up on the terminal and choose **Select Item➤**, or simply click on the terminal using the Operating tool. You can select only one variable at a time on a global's terminal. To use another instance of that variable, or to use another element in the global, create another terminal for that global (cloning it by <control>-dragging or <option>-dragging is easiest, although you can always **Select A VI...**).

Operating Tool



Just like locals, globals can be in a read or a write mode. To choose the mode, pop up on the terminal and select the **Change To...** option. Read globals have heavier borders than write globals. As with locals, globals in read mode behave like controls, and globals in write mode act like indicators. Again, a global in read mode is a data "source," while a global in write mode is a data "sink."

READ mode = CONTROL
WRITE mode = INDICATOR

Some important tips on using global variables:

1. Always, always initialize your globals in your diagram. The initial value of a global should always be clear from your code. Globals don't preserve any of their default values unless you quit and restart LabVIEW.

2. Never read from and write to global variables at the same time; i.e., where the order of events is undefined (this is the famous "race condition").

3. Since one global can store several different data types, group global data together in one global instead of several globals.

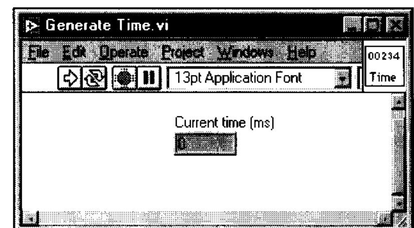
It's important that you pay attention to the names you give the variables in your globals. All the VIs that call the global will reference the variable by the same name; therefore, be especially careful to avoid giving identical names to controls or indicators.

An example

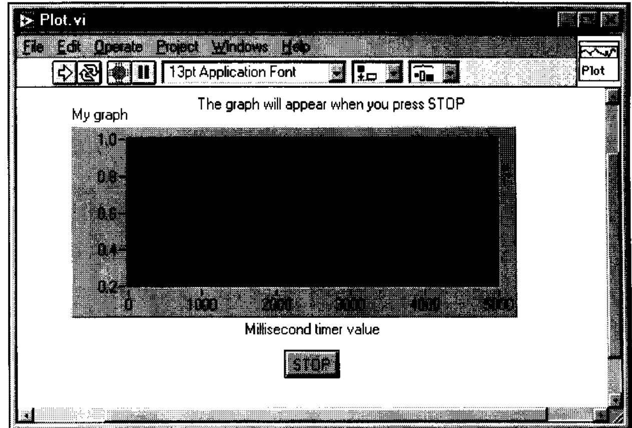
Let's look at a problem similar to the two independent While Loops. Suppose that instead of having two independent While Loops on our diagram, we have two independent subVIs that need to run concurrently.[1]

The subsequent figure shows two subVIs and their respective front panels. These two VIs, **Generate Time** and **Plot,** are designed to be running at the same time. **Generate Time** just continuously obtains the tick count from the internal clock in milliseconds, starting at the time the VI is run. **Plot** generates random numbers once a second until a stop button is pressed, after which it takes all the tick count values from **Generate Time** and plots the random numbers versus the time at which the numbers were generated.

Time

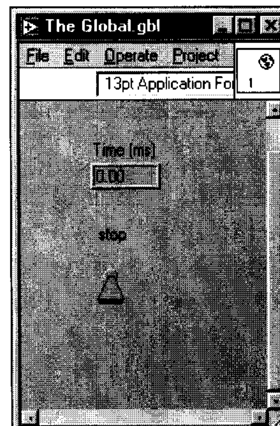


1. Of course, both subVIs can't *really* be executing at the same time on a single-processor machine! LabVIEW does a good job, however, of simulating parallel processes without you, the user, needing to worry about how it does this.

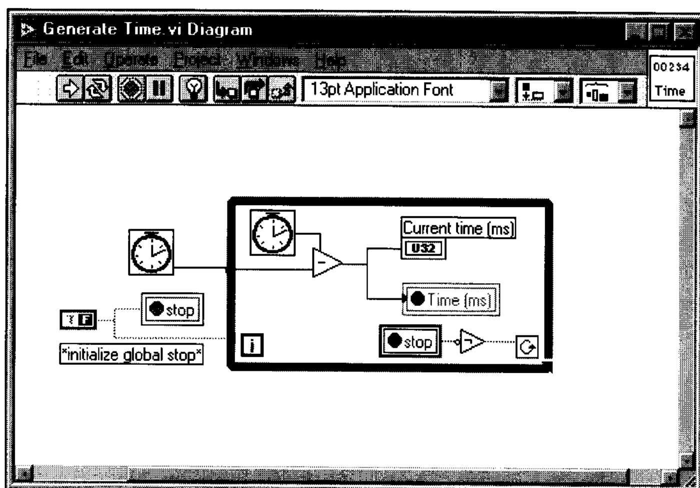


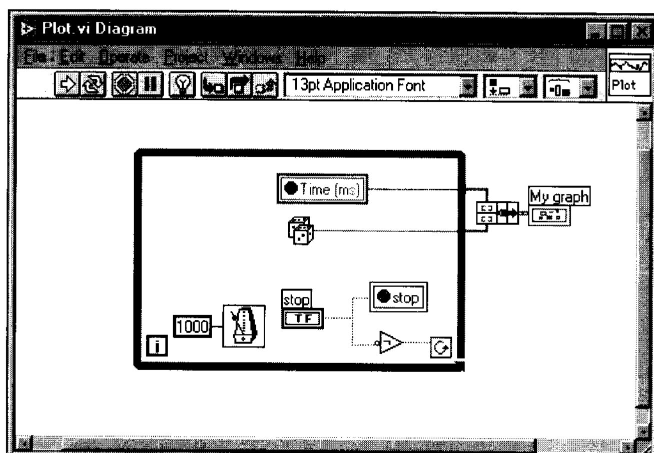
The way these two subVIs exchange data is through the use of a global. We want **Plot** to obtain an array of time values provided by **Generate Time**, and more importantly, we want both subVIs to be stopped by a single Boolean control.

First we create a global with the two necessary variables. Remember, to create a new global, select the **Global Variable** structure from the **Structures** palette, and double click on the "world" icon to define the global's components. In this case, we define a numeric component Time (ms) and a Boolean stop. The name of the global variable is **The Global.gbl**.



Then we use the global's variables at the appropriate places in both subVIs.

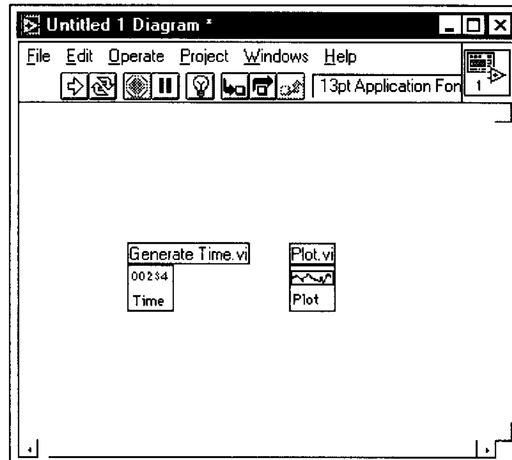




Notice how the stop Boolean variable is used: A stop button from **Plot** writes to the global variable stop, which is in turn used to stop the While Loop in **Generate Time**. Plot's front panel has been configured to open when the subVI is called so the user can see the plot and access the stop button. When the stop button is pressed in **Plot**, it will break the loop in **Generate Time** as well. Similarly, the time values from **Generate Time** are passed to the global variable Time, which is called by the **Plot** VI to build an array.

Hopefully, these two VIs have given you an example of how globals work. We could have avoided using globals for what we wanted to accomplish in this program, but the simple example is good for illustrative purposes.

If you look at the block diagram in the following figure, which calls the two subVIs, you'll see another problem with using globals: There are no wires anywhere! Globals obscure dataflow, since we can't see how the two subVIs are related. Even when you see a global variable on the block diagram, you don't know where else it is being written to. Fortunately, version 4 of LabVIEW addressed this inconvenience by including a feature that searches for the instances of a global.



If you pop-up on a global's terminal, you can select **Find➤Global Definition**, which will take you to the front panel where the global is defined. The other option is **Find➤Global References**, which will provide you with a list of all the VIs that contain the global. For more information about LabVIEW's search capabilities, see the next chapter, *Advanced LabVIEW Features*.