

PHY 406 - Microprocessor Interfacing Techniques

LabVIEW Tutorial - Part I

Beginning at the Beginning

Introduction

One of the main objectives of this course is to teach you how to gather data using computers and some of the pit-falls and prat-falls which can happen during such an operation. As an adjunct to that it is important that you also do some practical data gathering and try to work through some of the issues on your own. Knowledge that you have wrested from the universe for yourself is often more meaningful than that which you have been handed ready-processed.

The backbone of the practical work in this course is the use of the National Instruments “LabVIEW” package. This package is specifically designed to permit you to quickly implement a computer-controlled data gathering and analysis system which can be extensively customised to suit your needs. It is a very capable package, but is probably unlike anything you have met before. There is therefore going to be a steep learning curve ahead of you before you will be able to be proficient in such work. I wish I could make it otherwise, but I cannot. These notes are designed to make life as easy as possible and to get you started as quickly as possible. In order to make that happen I have made a few restrictions and a few assumptions which are:

- ▶ That you are familiar with the concepts of computer systems, such as files, filestore, directories, printing, etc (The notes will help you with the peculiarities of this system)
- ▶ That you are reasonably familiar with a “GUI” (Graphical User Interface) and with the use of the keyboard and the mouse.
- ▶ That where there are two ways of doing things, I will show you one way - you can look at the full manuals to show you other ways of doing the same thing
- ▶ Where there is a system-specific issue I will not discuss other systems
- ▶ This is a course in data-gathering, not in pretty LabVIEW programming - what counts is getting the job done, not (at least within reason!) the elegance of the implementation

The Philosophy of LabView

LabVIEW is an entirely graphical language which looks somewhat like an electronic schematic diagram on the one hand and a 1950's vintage style electronic instrument on the other - these are the concepts of the *block diagram* and the *front panel*. LabVIEW is heirarchical in that any *virtual instrument* that you design (any complete functional unit is called a *virtual instrument* and is almost always referred to as a “VI”) can be quickly converted into a module which can be a sub-unit of another VI. This is entirely analagous to the concept of a procedure in conventional programming.

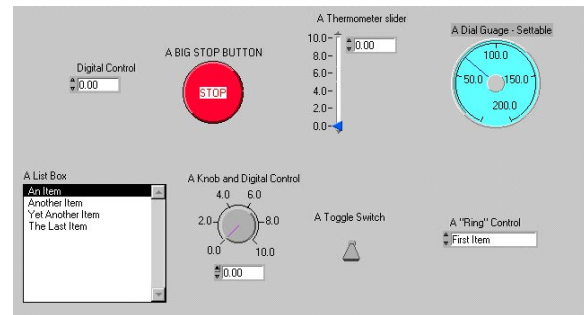
LabVIEW is also designed to be extendible. You can add modules through various

means. A manufacturer of an interface card or an instrument may provide you with a LabVIEW driver which appears as a VI representing the card and its functionality in the LabVIEW environment. You can also write a LabVIEW module using LabVIEW and present it as a VI to be used in other programs (re-usable code) or you can also write modules which interface with LabVIEW in other languages such as C and C++. These are known as “sub-VIs” and are no different from VIs except that the interface has been defined to the next level. Sub-VIs in C or C++ are very useful if you have a complex numerical procedure to perform on the data which is not covered in a standard LabVIEW routine. Since scientists are rather partial to complex numerical procedures, this can be a very useful property in our context.

Basic Concepts of LabView

As suggested above, there are two “faces” of any LabVIEW VI. They are the *block diagram* and the *front panel*.

The *front panel* is the face that the user of the system sees. It contains *controls* and *indicators*. LabVIEW has a very rich selection of both (you can even design your own) and this permits a wide range of options to the designer. This is a demonstration of a few of the controls.

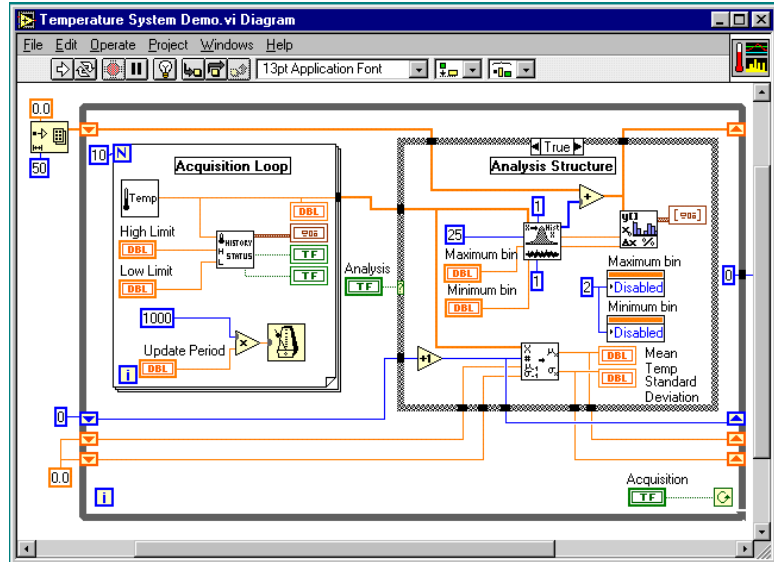


A *control* can take many forms. Many of the forms are themselves “pictures” of real controls used on real instruments - rotary knobs for example. Others are strictly digital in concept. All controls have some form of visual feedback to show the user what state they are in. This helps enormously as you do not have to make explicit allowance to show the state of the controls in your design. A second extremely useful property of controls is that you can specify how they are to react if the input given is unsuitable. To give a specific example - if a control should have an input range of 0 to 10 in integer numbers, you can specify what should happen if the value 3.5 is given or -1 or “zero” as a character string. Since a great deal of time can be consumed in “bullet-proofing” a user interface against these sorts of problems, this can be a big timesaver.

Indicators take a large number of forms. Again some are “pictures” of real indicators - lights and meters. Some are more designed for the computer screen. The concept of indicator also includes graphs and charts which is a second major timesaver as you do not have to design any of these elements explicitly.

By intelligent design of the *front panel* of a VI it is fairly simple to produce a simple clean design for the user.

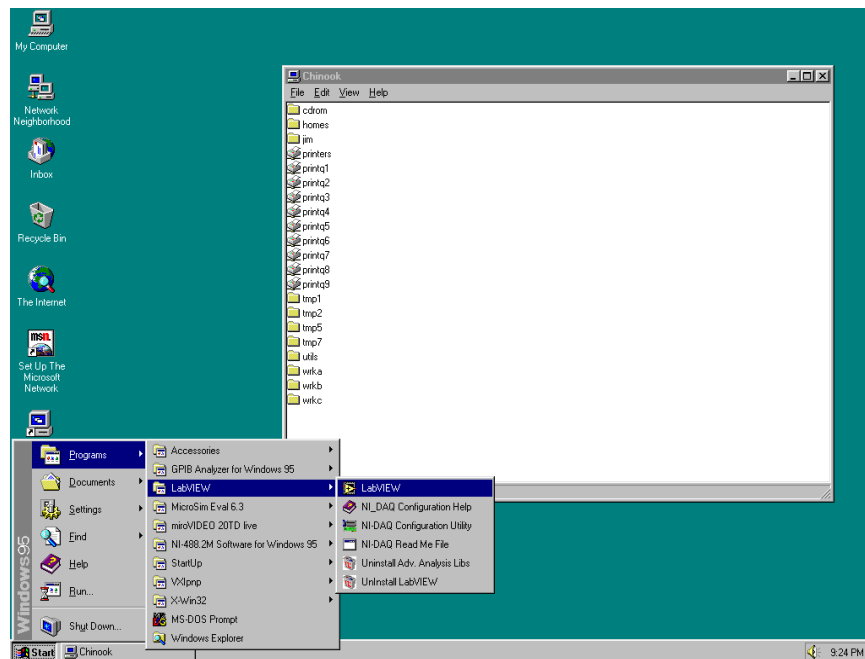
The *block diagram* of the VI is almost the “backside” of the *front panel*. It shows how all the controls and indicators fit together as well as the hidden modules where all the work gets done. It looks somewhat like an electronic schematic diagram and is at least conceptually wired up in the same way. Like a real piece of instrumentation, it is easy for the wiring to look very complex and untidy. One of the major issues in LabVIEW programming is to allocate the timing and ordering of operations. In a conventional programming language this is handled by the order of the statements along with the use of various loop constructs (FOR, WHILE, etc). LabVIEW works in exactly the same way, but the way in which you specify the ordering is more subtle. The concept in LabVIEW is “dataflow” - any item executes when all it's inputs are available. This implies parallelism (or at least pseudo-parallelism). The standard execution is left-to-right because inputs are generally on the left of an item and outputs on the right, but this is a convention, not a requirement. Looping and ordering is handled by structures which look like books with a number of pages or the frames of an old-fashioned cine film.



A Tour of the Interface

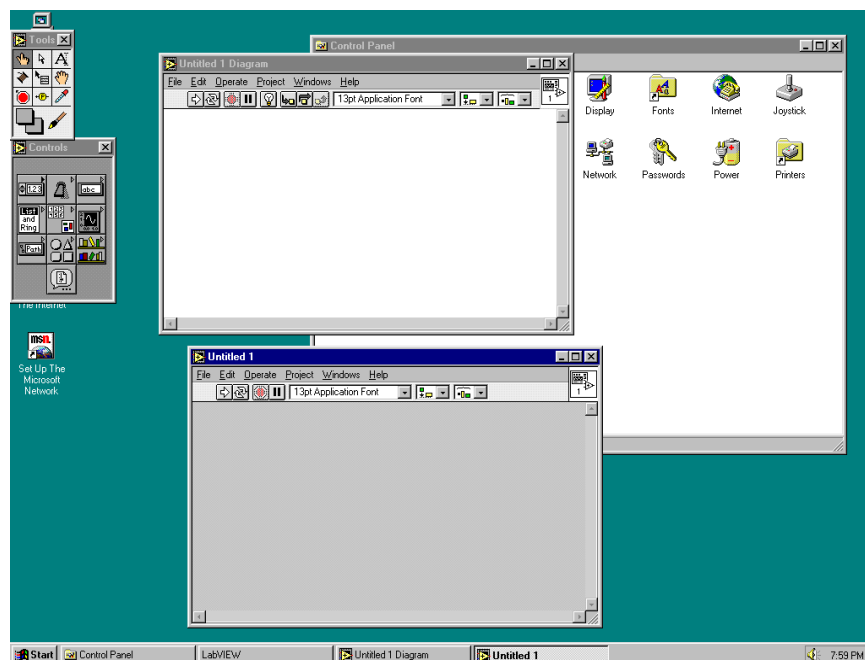
What follows now is a very brief tour of the LabVIEW interface. We are not going to go into all the places possible, but just look at the most common parts. You should read through this section whilst sitting in front of a computer so that you can verify that these notes are correct and you understand them.

First we invoke LabVIEW by the path **Start>>Programs>>Labview>>Labview**. (If there is a LabVIEW icon on your screen you can also start LabVIEW by double-clicking on it).







When we use the notation **Start>>Programs>>Labview>>Labview** we are asking you to follow a series of menus starting from the first item which should be visible in the current window. You can see the entire tree for this operation on the screen below.

Your screen will look something like this (you might have to move a couple of windows around to see everything):






Let's concentrate first on the window with the grey background - the *front panel*. The window is blank (no panel designed yet) with seven icons along the top of the window and a large one in the top right. The seven icons are used to design a front panel and the large one represents the entire front panel when we are making or modifying a sub-VI.

The seven icons divide into two groups. The first four control the running of the VI:

Icon	Meaning	Explanation
	RUN	Run the VI once. VIs, like conventional programs, do not repeatedly run unless you tell them to. The RUN button changes appearance when the VI is actually running.
	RUN REPEATEDLY	Run the VI over and over. Unless you are debugging a VI, this is not a recommended way of repeating any but the simplest of VIs. There are much better ways of doing this using LabVIEW constructs
	STOP	STOP (unceremoniously) the current VI
	PAUSE/ CONTINUE	Press once for pause, again to continue

The last three icons are used in building front panels

Icon	Meaning	Explanation
	FONTS	Controls the fonts (size and style of type) used for the front panel
	ALIGNMENT	Controls the alignment of groups of controls and indicators - useful for getting things in straight lines and columns
	DISTRIBUTION	Controls gaps between things - useful for getting things uniformly spaced.

Above the line of icons (pictures) there are five text menu items:

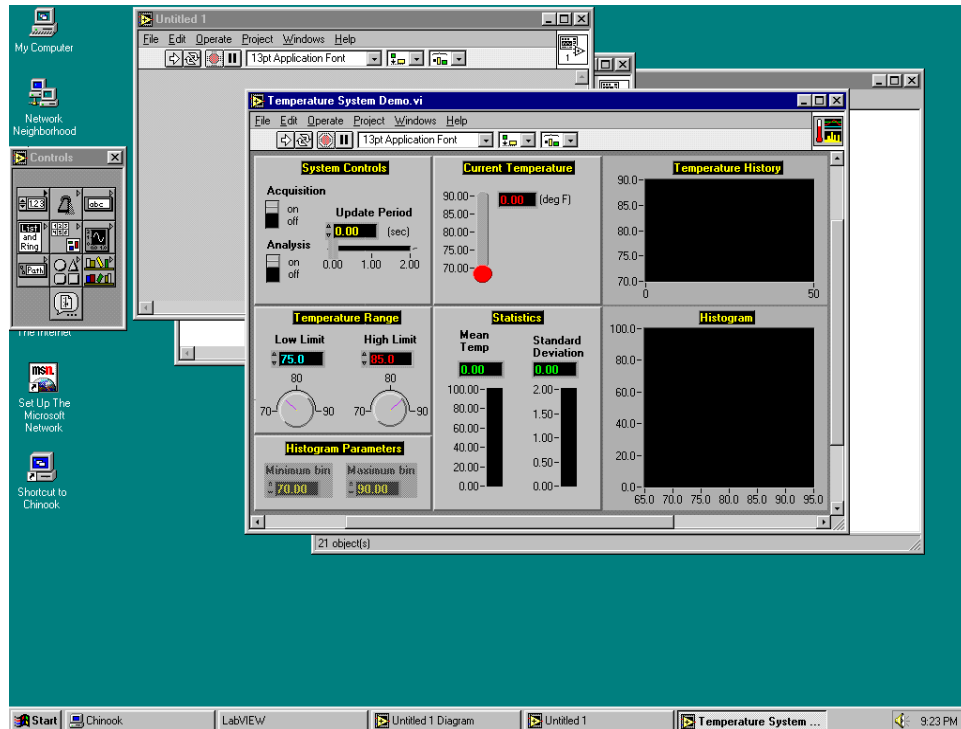
FILE	Is the menu which permits you to do file operations. In this context VIs live in files and therefore this is where you load and save VIs
EDIT	Is the edit menu and contains the commands for editing elements
OPERATE	Operations involving the running of the VI
PROJECT	Operations involving the make-up of the VI - including subVIs
WINDOWS	Select which windows are visible
HELP	One of the most useful areas is the help area. There are several ways of getting help on an item or concept and we will deal with these a lot.

FOLLOW ME

We will now do something very simple - load and run a VI. The VI is a standard example from National Instruments and is a “fake” of a temperature measuring system. The point of this operation is to show you a few of the things that make up LabVIEW and to familiarise you with simple LabVIEW operations.

Select **File>>Open** - double click on “Examples”, then on “Apps” then on “tempsys”. Tempsys is a library of VIs - within the library are a number of items - find the one called “Temperature System Demo.vi” and double-click on it. A box will now appear showing you the loading process and some diagnostics which don’t concern us here.

You should now have a screen which looks like:



On the upper left of the window are a set of controls: Two slider switches which have two states, “on” and “off” - these are boolean controls. There is another control which has a digital and analog part. The digital part is a panel with a number in it and two arrows for raising and lowering the value. Digital inputs like this can usually be changed by either clicking on the arrows or by over-writing the number in the display. The analog part is a slider which can be dragged with the mouse to change the value.

Below that panel in the middle left is a second pair of controls which have a similar digital section, but have rotary controls rather than sliders. Personally I don’t like rotary controls in LabVIEW because I find it hard to rotate the knob using the mouse, I much prefer the sliders.

In the middle of the screen are two panels with indicators on them. Both indicators have a digital and an analog section - the digital section looks pretty much like a control without the arrows to change the value. The analog section of the indicator is a “thermometer” display.

The right-hand side shows some of the most powerful LabVIEW indicators - charts and graphs. These are graphical records of the output and can be displayed in a number of forms which we will discuss later.

There are a few things on the front panel. On the bottom left are two controls which are “greyed out” - this means that they cannot be accessed (changed) at the moment. There are also a couple of other indicators which don’t show up yet.

Now we will run the VI. First, I must explain what it is supposed to do. The VI simulates reading a temperature at a rate determined by the “update period”. It displays the current temperature and also computes the mean and standard deviation of the last 10 readings. It displays a “strip chart” of the history of the temperature readings with the current high and low limits shown and a histogram of the readings. Finally it activates an under- and over-temperature warning.

Before we run the VI I want to show you one important thing- how to print things. There are two procedures depending upon what exactly you want to print:


Print the Contents of the VI

- 1) Select **File>>Print**
- 2) You will be presented with a selection menu which asks what you want to be printed

Print an Image of the Entire Screen

- 1) If required, maximise the VI by pressing the maximise button (second from top right)
- 2) Press the “PrintScreen” button
- 3) Reset the size of the VI by pressing the restore button (second from top right)
- 4) Open the paint program **Start>>Programs>>Accessories>>Paint**
- 5) Import the clipboard **Edit>>Paste** - at this point you can edit the picture to take out bits that you don’t want - but you’ll need to learn a bit about the paint program to do that.
- 6) Print the picture **File>>Print**.

I am assuming that you now have a paper copy of the VI in your hand - sorry it’s not in colour, but funding is still tight!

Press the RUN icon at the top left  Notice that the icon changes to the running form and the front panel starts to update. This VI does not end because it contains an infinite loop as we shall see later.

Using the operating tools we can slide, twist, poke and over-write the controls. To slide and twist knobs and switches put the operating tool on the control and use the left mouse button. To poke a digital control, put the tool on the appropriate arrow and click (the amount the control changes can be varied as we shall see later). To over-write a value, double-click on the old value and then type the new one in. Please try changing a few controls and observe the effect. Notice that the analog controls (sliders and knobs) naturally show you the limits of the input, whereas the digital controls don’t

When you have finished playing, stop the VI by sliding the acquisition switch to “off”. The VI may not stop immediately because it has to finish what it is doing. This is a much better way of stopping a VI than pressing the “STOP” on the toolbar because it permits a clean end to the VI operation, the STOP button can leave unfinished business around.

Now we are going to look at the “backside” of the *front panel* - the *block diagram*. Select **Windows>>Show Diagram**. Notice that the block diagram doesn’t replace the front panel - both are visible at the same time, useful in development operations.

The first thing that should strike you about the block diagram is that it is a bit more complex than the front panel! However most equipment looks nicer on the outside than the inside!

The next thing is that there are three shaded boxes - a big outer one and two inside it. These are the looping boxes. The outside one is a “while” loop which continues until a boolean condition becomes false. The left-hand inner one is a “for” loop which executes once for every iteration of the while loop (these loops are therefore “nested” as the diagram suggests) and the right-hand one is an “if-else” condition (not a loop). The “true” part is showing at the moment. Click on the arrows by the word “true” at the top middle of this box to swap to the “false” side which is a lot less crowded than the “true” face. Click on the arrow again to swap back.

The “wires” connecting the parts and some of the boxes are in different colours. LabVIEW uses different coloured wire to tag different types of variable. Green (dotted) is boolean, blue is integer and brown/orange is double precision. Notice that the brown/orange come in two thicknesses - the thin lines represent scalars, the thick lines arrays.

Since LabVIEW operates left-to-right it is clear that within each iteration of the outer while loop, the left-hand for loop runs completely through once and then the right-hand if-else clause is executed.

By the way if you are having trouble swapping between the windows, arrange them so that they are offset slightly along a diagonal line with enough offset to show a little bit of the other window area even when it is not selected. Then you can click on this small area to swap between windows. If the overall screen is large enough, you can arrange them so that they don’t overlap and then you can see both at once, but this isn’t always practical.

You will see a number of boxes with single coloured lines around then with a number inside. These are constants. The boxes with two lines, outer thick - inner thin, are controls - the “backside” of the things on the front panel. Double-click on the one marked “update period”. The display swaps to the front panel and you can see that the control marked “update period” is highlighted there. These are the front and back side of this control. Move back to the block diagram.

The boxes with double thin lines are the indicators. Try a similar selection process on the one marked “DBL” in the top right of the for loop. It corresponds to the thermometer and digital value in the current temperature area.

The squareish boxes mostly in black-and-white are very significant. They are also VIs. Try double-clicking on the one in the middle of the for loop (left-hand inner box). It will pop up another front panel. The “controls” on this front panel are in fact controlled by the input wires to the VI and the indicators feed their output through the attached wires elsewhere. Notice that there are three controls and three indicators. Close this window and see that there are three inputs (left-hand) and three outputs (right-hand) to this VI. This is the “procedure” or “subroutine” of LabVIEW. Each of these sub-VIs can be independently made and then strung together to form a more complex whole.

Now close both the front panel and block diagram - you shouldn't have changed anything, but if you did please ask the system to ignore the changes. (Just say no!).

Now we are going to build a very simple VI. In order to make sure that we all start from the same point, please close LabVIEW now by closing the blank windows. There will be a final little box which says “There are no VIs open” and you should choose “Quit” from this.

Summary

So far we have:

- ▶ Introduced you to the basic philosophy of LabVIEW, a graphical processing language. It uses the concepts of a *front panel* and a *block diagram* to show you the user and designer views of the VI (*virtual instrument*)
- ▶ Given you a tour of a moderately complicated VI which contained *controls*, *indicators* and sub-VIs
- ▶ Shown you how to start LabVIEW and load a VI


PHY 406 - Microprocessor Interfacing Techniques

LabVIEW Tutorial - Part II

A Simple Example

A Very Simple Example

A very long time ago (or yesterday, depending upon your age) it was suggested that if you could write a program to say “hello world” on the teletype (this was a long time ago!) you had probably done about 90% of the required work in learning the programming language. At a minimum you had mastered the basics. I am going to help you do the same thing in LabVIEW but the example is: $2 \times 3 = 6$! This will take some time because I want to go into some details about things, so bear with me and please actually do the example!

Open LabVIEW. Swap to the front panel screen (grey background). The first thing we will need is the *tools palette* which is the available toolkit for making things. Select **Windows>>Show Tools Palette** and a tools palette should appear (it may already have been there - don't worry about that!). We will use most of the tools in the palette but at the moment we just need the selection tool which is the top middle one. Click on selection tool  to use it (you may need to click twice - once to activate the window and once to get the tool. I am now in the habit of doing two clicks to pick up a tool to make sure I get it - that's two single clicks, not a double click)




In order to do this example we will need two controls (for the “2” and the “3”) and an indicator for the “6”. Since the two inputs will be variables, you will be able to do all sorts of multiplications! We select a control by using the *controls palette* (**Windows>>Show Controls Palette**)

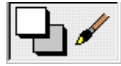


To use a palette is a bit tricky (at least I find it so!) There are ten panes on the control palette. You need the top left one. Click on it once and then drag the mouse directly onto the new pop-up window. This window shows some of the possible controls which supply numeric (as opposed to boolean, text, etc) inputs. The one we want is again in the top left. Click on it - and don't worry when the panel disappears!. Move the mouse to the front panel window - at which point a dotted outline appears - and position the control on the front panel. Click again and the control appears.

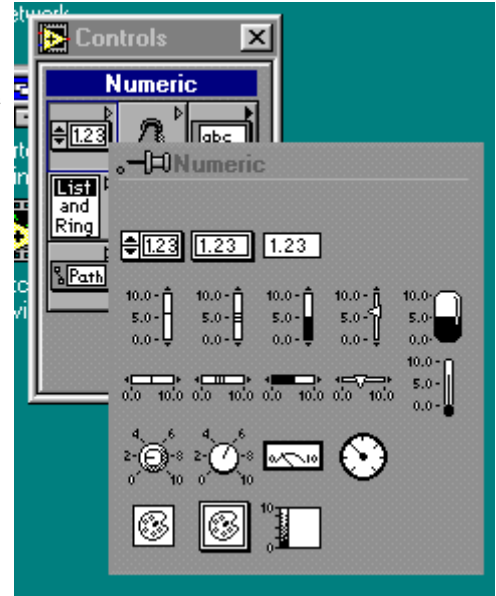
The control comes up with a small text panel ready for text input. **DO NOT PRESS ENTER.** Type in some text like “var a” and then press ENTER. This adds a label to the control on both the front panel and the block diagram. Now try selecting the label and moving it. Notice

that the control stays put. Move the label back. Now do the same to the control. Notice that the control and label move as one. The point is that moving a label moves it in relation to the control. Moving the control retains the relative spacing between the control and the label. You will therefore need to be careful which one you move.

Now I am going to show you how to “embellish” a control by changing the colours. I have no particular reason for this - but it’s fun! However it will help you to get some experience with the tools and it will also (I hope) give you some satisfaction in customising your front panels. The two tools you need for this are the colour pick tool  - third rank



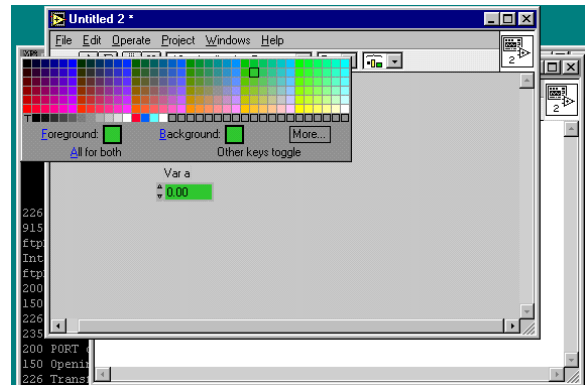
which is the bottom panel.




There are many pieces of the control which you can colour and I don’t want to do all of them, but here are two:

Get the colour pickup tool and click once on the background of the VI. Then get the paintbrush and place the tip at the background to the text label. Click once and the background changes to the same colour as the general background and disappears.

Now, still using the paintbrush, go to the number and click on the *right* mouse button and hold it down. A colour panel appears and by sliding around the colours you can change the background of the number - I generally pick white as a background, but that’s my preference.



To make the second control we will duplicate the control (thus keeping our colour changes and so on) and then modify it. Pick up the selection tool and select the control (remember that’s the *control*, not the *label*). Then do a Ctrl-drag (hold Ctrl and left mouse button down and drag the control to the new location). This places a new copy of the control on the front panel, but it is identical to the original - with the exception that the label changes. In order to distinguish it from the first one we will change the label. We do this using the text tool which is the upper-left tool . The pointer changes to a text tool and we can go and click on the control label and change it.


Next we need an indicator - which we get from the controls palette (which therefore should be the controls *and* indicators palette). Just for fun we’ll use a vertical slide type

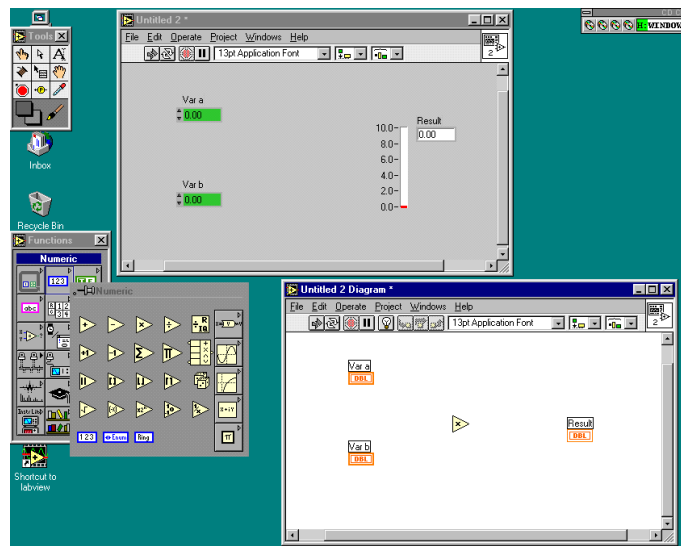
indicator which is on the same panel as the control you got before. (Watch out there are several things which look similar - the one you want is called “vertical fill slide”) When you have placed it, labelled it and coloured it, notice that there is a problem - this is a *control* and we need an *indicator*. No problem!. Using the select tool, select the control, press the *right* mouse button and the first menu item is - *change to indicator*. Notice that when you select that, the little arrows on the digital display disappear - you don’t need them for an indicator.

Now with a bit of judicious selecting and moving you can give the front panel a nice appearance. Try selecting the two controls (Using the select tool, start at upper left and drag the mouse across both controls to lower right whilst holding the left mouse button down) and then aligning them vertically using the alignment tool from the top of the window.



Now swap to the block diagram. You will see the backside of your two controls and an indicator. You can independently move these around to please yourself on this side (remember to move the control/indicator, not the label only!). An important aspect of LabVIEW programming is the concept of *dataflow programming* - things happen as the data for them are available, and by convention things are generally arranged to flow from left to right. Since the two input variables to our problem are available simultaneously, they should align vertically to the left and the result then appears to the right.

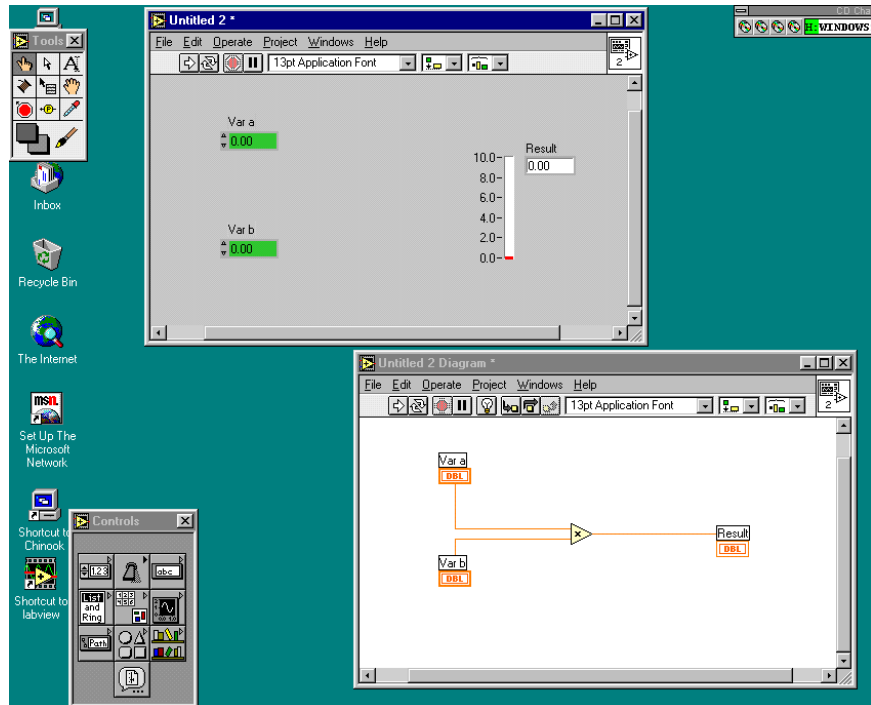
Between the inputs and the outputs we need a multiplication function. We get these naturally enough from the *functions* palette, **Windows>>Show Functions Palette**. Use **functions>>numeric** and get the multiply function. Put it in the middle of the block diagram.

Now we need to wire the diagram up. This is done with the spool of wire  on the *tools* palette so select that tool and then go to the first input control. The *DBL* box will flash as you pass over it. To lay a wire, push the left mouse button while the box is flashing and hold it down until you reach the upper left point of the multiply symbol which will also start to flash and show a label *x*, then lift the mouse button and then click it. The wire should turn brown/orange. If it stays dotted, press **Ctrl-B** (remove bad wires) and try again. It is slightly tricky to get a wire to lay down properly, but watch for the upper left of the symbol to flash, then release the mouse button and the wire should stay there. Do the same for the lower input and the output indicator.



OK- you’re done - now your screen should look (somewhat) like this:

Go to the front panel and run the VI. Since the two inputs default to 0.0 - the answer is 0.0. To actually do the question asked, you need to change the two input values to 2.0 and 3.0. Make sure the VI is stopped (there is zero chance that it hasn't, but at this stage you should not change things while the VI is running) by making sure that the *start* indicator is in the "not-running" state which looks like this . Now use the *operating tool*  to change the values - either by clicking on the up arrow on the appropriate control or over-writing the value. Run the VI again and you should get the right answer!



Finally all this is no use unless you can save your work. To do this select **File>>Save As** and save the file in an appropriate directory under an appropriate name. (Important - you must add the ".vi" extension to the file - LabVIEW won't do that for you). Notice that this is slightly different to the way we loaded the thermometer example above, in that case we used a file which was a library of VIs and then loaded a single VI from the library. However I did say that I would only show you one way of doing any one thing when there might be many ways of accomplishing the desired end.

Please note that you should save all your own files in your own home directory which is the *F-Drive* on this system. When you get the file selection menu - press the up icon until you get to the screen which permits you to choose the F-drive and then select it. Within that drive, you can do what you like! Do not leave anything on the local drives of the PC - these drives may be cleaned out without any notice whatsoever.

Data Input and Display

Now let's try a few variations. One of the very strong points of LabVIEW is its ability to control the way in which input is authenticated and displayed. All of these are controlled by the pop-up menu which comes up when you click on the *right* mouse button when over the appropriate item. Some of the most useful things you can do are:

Change the default value	Select Data Range and then alter the <i>Default</i> value.
Change the rate at which the variable changes when you click the “up” and “down” arrows	Data Range and then alter the <i>Increment</i> value.
Change the allowable range of the data	Data Range and change the <i>Maximum</i> and <i>Minimum</i> values.
Change what happens at and beyond the limits	Data Range then select from “If value is out of range” - the options are: <i>Ignore</i> Don't apply any limit checking <i>Coerce</i> If the value is out of range, reduce it to the nearest limit value. The increment/decrement arrows will not exceed the range <i>Suspend</i> If the value goes out of range the VI is suspended or will not run - you can also add an error message.
Change the way in which the data are presented	Format & Precision and make appropriate changes.

Try changing a few of these to see how it all works.


You can also change the range of the thermometer display simply by changing the maximum value on the display with the operating or text tools

Fun Stuff (aka Customisation)

Some more things you can do to brighten up the front panel are to add labels, colours and decorations.

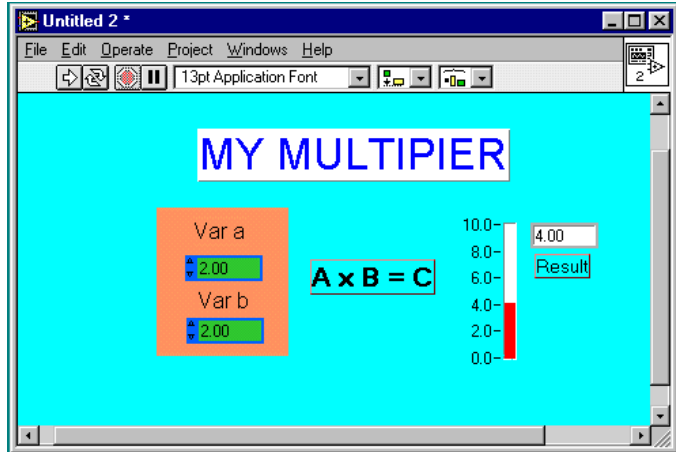
Labels are added with the text tool and any text item can be changed by selecting it and then using the font menu at the top of the screen.

Colours are added with the colour tool as I've already explained - you might already have accidentally discovered that you can change the background colour as well as any other colour by using the paintbrush away from a control or indicator.

Decorations are accessed through the decorations item on the controls menu  and can be independently coloured. Objects may be sized by selecting them, then dragging a corner to resize it. You will also need to cope with *layering* - which item lies on top of another - since only the top one of a stack of objects is visible. Hence if you have a decorative panel behind a control,

that is exactly what you need - a decorative panel *under* a control. Layering is controlled from the **Edit** menu and any selected item can be brought up a layer, down a layer, to the front or to the back. Usually it is only necessary to bring things to the front or push them to the back.

Anyway after I had played around a bit my front panel looks like this - which it must be admitted is unique!



Summary

- ▶ You can create your own VI which used *controls, functions and indicators*.
- ▶ You can load and save VIs
- ▶ We can alter the way in which data are displayed on controls and indicators.
- ▶ We can alter the default values and the rate of change of a variable
- ▶ We can alter the limiting values of a variable and the action when a limit is reached
- ▶ We can change the size, colour, font etc of text
- ▶ We can add decorations to the screen

Exercise

Try changing your VI to compute a quadratic (ax^2+bx+c) with controls for a,b,c and x and a sensible front panel layout.

PHY 406 - Microprocessor Interfacing Techniques

LabVIEW Tutorial - Part III

Going Forward and Round and Round

While Loops and Switches

We are now going to progress with our LabVIEW by learning how to loop a VI and stop it in a civilised manner. The loop we will use is a “WHILE” loop which runs round and round as long as a control variable is “true”.

In order not to tax things too much, we will re-use the VI we made last time and put it in a loop so that we can keep changing the inputs and see the outputs changing. We will also adopt the practice of having a nice switch to stop the VI when we are ready.

If you are continuing from the previous example, then you already have the right screens in front of you, if not restart LabVIEW and bring that practice VI up again.

Some thoughts on modifying things so that you add to them and getting them saved properly. I have found that if you want to take a VI and then upgrade it to be saved under a new name, the best thing to do is to save it under the new name now. The reason for this is that I have a habit of saving things from time to time as I reach “plateaus” of functionality or get more worried about how much I will lose if the system crashes or the power dies. If I have the VI saved it under the new name now, I only have to “save” from there on - I don’t need to remember to “save as” the first time - which I always forget and then overwrite the original VI. So if you agree with my philosophy, now is the time to save this VI under a new name!

We now need to put all this in a loop on the block diagram. Using the select tool, find the “while” loop under **functions>>structures**. Start by moving to the upper left of the diagram and the holding the left button down drag the loop out until it encloses everything. (This is the only time you can do this - you can’t arbitrarily “enclose” components at any other time - they should be created either inside or outside the loop structure).

Notice the two squares in the lower corners. The left-hand *I* is a variable which counts (from 0) the number of times the loop has executed. The right-hand is the boolean control variable. If the input is “true” then the loop executes, if it is “false” then it stops at the end of the current iteration - ie cleanly.

Now let’s learn another LabVIEW shortcut. Right-click on the loop and select the “create indicator” option - a digital indicator pops up on the front panel. This is a really quick way to create new indicators when you need them. You can move the indicator around and label it by selecting the indicator and then selecting **show>>label** from the pop-up menu (right-hand mouse



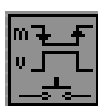
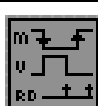
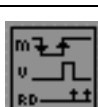
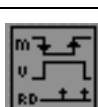
button).

The same thing can be done with the loop control, use the pop-up menu to create a control. Now using the operating tool you can set the control to “on” and run the VI. The loop indicator should show that it is running round and round very fast. You can now dynamically alter the input variables and the output changes. When you push “off” - the VI stops (run arrow changes and stop button dims). Now try to restart the VI by pushing “run”. Nothing happens because the push-button control still says “off”. To run again you have to change the pushbutton to “on” and start the VI - inconvenient. What we need is a smarter pushbutton.

Now you might think that boolean switches are fairly simple, but that is not true. There are a number of essential questions to ask:

- ▶ Do you want the switch to latch a new value or just react when you “push” it?
- ▶ Do you want this to happen when you push the mouse button or when you release it?
- ▶ Do you want to be sure that LabVIEW will read the switch in it’s new state?

This leads to six different types of switch:

Icon	Latch?	Action on Press or Release?	How many times will LabVIEW read it?
	YES	PRESS	Many times
	YES	RELEASE	Many times
	NO	PRESS/RELEASE	Maybe never (released too quick) Maybe many times (released “slowly”)
	NO	PRESS	Once - revert when LabVIEW reads
	NO	RELEASE	Once - revert when LabVIEW reads
	NO	PRESS	At least once, maybe more - revert when LabVIEW reads after button released.

The one most suitable for our purpose is the non-latching, revert after LabVIEW reads -


the fourth in the list. Change the pushbutton to that type from the pop-up menu item **mechanical action**. You should also change the default state to “on” (manually set the pushbutton “on” and then set that as the default state using the pop-up menu).

Now when you run the VI it should only require you to press the run button to start it.

From now on, I will assume that you are going to save your VI at appropriate times. It is a good idea to save things to a new name every time you reach a satisfactory state of operation. This is more than paranoia - it is a sensible policy to make sure you don't lose an afternoon's work because of “finger trouble”.

It's a bit static isn't it?

The world of video games makes us like movement and this VI is a bit static, only a counter (which might by now have reached a very high value!) Counting upwards. Time to add a bit of action. First start changing one of the input numbers about 10 times a second, then.... Well maybe there's an easier way.

LabVIEW is very rich in functions. What we are going to do is to change one of the input variables to the output form a random number generator. First delete the control by selecting it on the front panel with the select tool, then pressing **Delete** (you can't delete a control from the block diagram that way. Go back to the block diagram and remove the broken (bad) wire by **Ctrl-B**. Now find the random number generator on **function>>numeric** (it looks like a couple of fuzzy dice ) and put it on the block diagram in the while loop and wire it up with the wiring tool to the multiply operator. You might wonder how the generator knows when to “produce” a new value. In effect Labview asks every component whether it has a value available. The digital control says “yes” and produces the value of the control, the random number generator says “yes” and produces a new random number.

Now when you run the VI, the thermometer should “shiver” very satisfactorily assuming that you have the right scaling.

It would be nice to have a record of the output over time - a graph or a chart. LabVIEW distinguishes between a graph in which all the data are available together (as in an array) and a chart when only one value is available at a time (as in a strip-chart recorder). We will add a chart to the VI. At this stage you might want to expand the VI by enlarging the front panel window (drag on right-hand edge with left mouse button) until there is a space about as large as the rest of the window spare on the right. Now select **controls>>graph>>waveform chart** and place the chart on the right of the front panel. Add the label before pressing **return**.

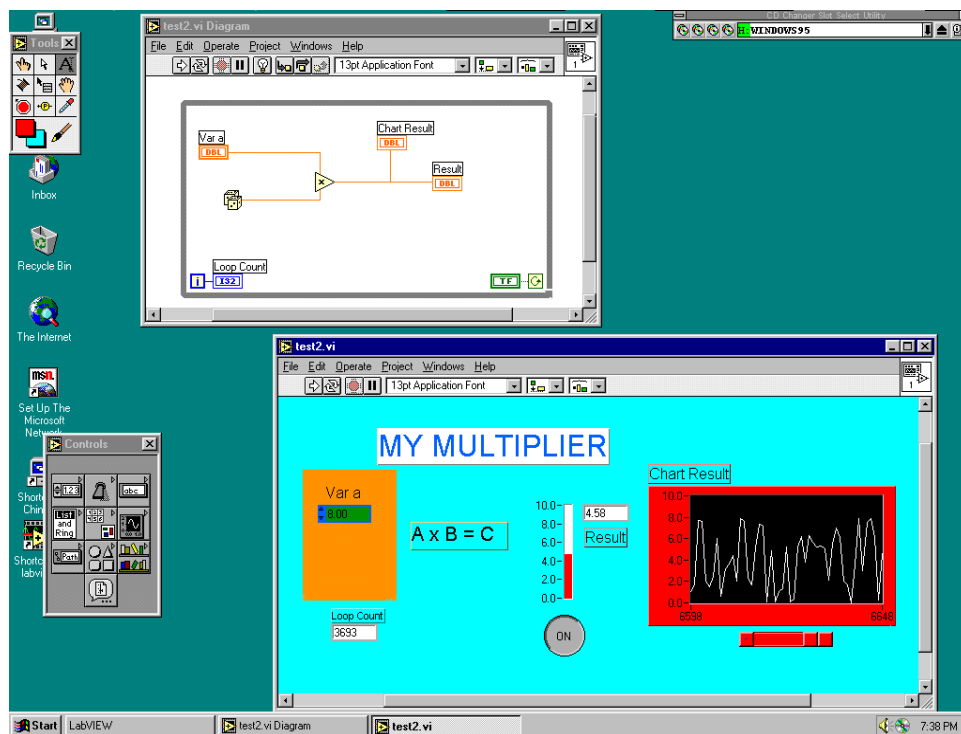
The chart can be wired up in parallel with the thermometer on the block diagram by starting with the wiring tool at the wire between the multiplier and the output indicator and wiring

to the chart from there - one output can feed any number of inputs of the same type. Make sure that the indicator is inside the confines of the while loop.

Now when you run the VI, you see a chart of the last 50 values and the chart presents a running window of the last 50 values. From the pop-up window you have control over a large number of parameters. A new set is the set that controls the rolling window. You can control the number of values visible by changing the horizontal scale and control how these are spread out by changing the size of the box by selecting it and dragging the size out horizontally. You can also review the history of the values using a storage capability and the horizontal scale with a scrollbar. By default 50 values are shown and 1024 values are stored. By activating the scrollbar from the pop-up menu **show>>scrollbar** you can use a simple scrollbar to run through the stored values (You probably need to colour the scrollbar white to see the parts properly, for some reason the default values don't show up too well). You can also change the length of the stored dataset from **Chart History Length**

Those of you who have sharp eyes will notice that the VI is running distinctly slower with this chart in place. There is no such thing as a free lunch and more displays equals more processing required equals more time per loop equals fewer loops per second. Something to be borne in mind for the future.

My design now looks like this:



Too Fast!

Now that you can see how fast the VI is going, you might decide that it is too fast. A new value every 250mS would be just fine. There is a way to time a loop and that is to insert a timing element. From **functions>>Time & Dialog>>Wait Until Next ms Multiple** get the little picture of a metronome, place it in the loop and use the pop-up to generate a constant for the input and set the value to 250 (the value is in mS). Move the constant over and wire it to the input (left-hand) side of the timer. Your VI should now loop four times a second.

Summary

- ▶ While loops have an iteration count and a boolean control variable
- ▶ Pushbuttons can be of six different types depending upon the action required.
- ▶ The random number generator creates a new random number between 0 and 1 every time it is interrogated.
- ▶ Charts are used to display the sequence of outputs from a process and are highly programmable.
- ▶ LabVIEW may be “slowed down” by timing units

Exercise

Instead of the random number generator, use the **Digital Thermometer.vi**. From the tutorial functions - use the Boolean “mode” input so that you can have the display in either Fahrenheit or Centigrade - add a third option for Kelvin. (You will find it useful to call up the help (**Ctrl-h**) on the Thermometer to see how to wire it up).

PHY 406 - Microprocessor Interfacing Techniques

LabVIEW Tutorial - Part IV

Data Representations and Help

Data Representations

You should have on your diagram three colours of wire and three data types

- ▶ Orange/Brown - with DBL boxes
- ▶ Blue - with I32 boxes
- ▶ Dotted Green - with TF boxes

LabVIEW supports a large number of data types and their conversion. The major types of numeric variable are:

Abbreviation	Type	Storage Length (Bytes)	Minimum Value	Maximum Value
EXT	Extended Precision Real			
DBL	Double Precision Real	8		
SGL	Single Precision Real	4		
I32	Long Integer	4		
I16	Word Integer	2	-32766	32767
I8	Byte Integer	1	-126	127
U32	Unsigned Long Integer	4	0	
U16	Unsigned Word Integer	2	0	65535
U8	Unsigned Byte Integer	1	0	255

Since similar arithmetic operations can be performed on each of these types, the arithmetic functions, the controls and the indicators must be able to cope with these. To achieve this, three different techniques are used: changing type, implicit type conversion and explicit type conversion.

To see an example of the first and the second, use the pop-up on the input control to the multiplier in your example to change the representation. Notice that the wires are orange/brown for float and blue for integer/unsigned. Notice also that if the input types to the multiplier are not

the same type, then a black spot on the input indicates and implicit type conversion - the types of the inputs are changed to compatible types.

To see an example of explicit conversion, let's change the type of the output to a Word Integer. Remove the wires from the output of the multiplier to the chart and indicator by selecting them with the select tool and **delete**. Use **Ctrl-B** to clean up the broken wires. Now select **functions>>numeric>>conversion>>To Word Integer** and place the converter opposite the output of the multiplier. Complete the wiring from multiplier to input of converter and then to the chart and the indicator. The chart automatically changes to an I16 when you connect to it, the indicator stays at DBL and acquires a conversion dot to indicate the conversion. You can change the type of the indicator by using the pop-up and the **representation** item to change to an I16 in which case the conversion dot disappears and the type is consistent throughout.

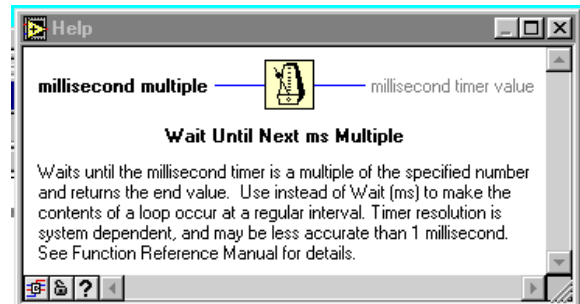
There are many times in LabVIEW programming where you will have to be aware of the data type and ensure that the type conversions are being handled correctly.

Help - I Need Somebody!

There are two forms of help in LabView - LabVIEW's help to you, the designer, and your help to the user.

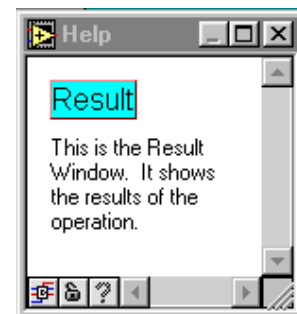
LabVIEW will help you in many ways. There is a built-in Help system (under **Help>>Online Reference**) which will give extensive information about LabVIEW,

There is also a more specific help system which can tell you about individual items. This help system is available under **Help>>Show Help**. To use it, activate the option by selecting it and then put the select tool on any item. For a system item you get a little summary of the item, for example the one for the millisecond timer looks like this:



If this help system is activated, then you will get help on any item that your tool is on. This can be useful, but frankly I find it annoying to have on all the time, so I use it sparingly and use the short-cut key **Ctrl-H** to toggle it on and off when I need it.

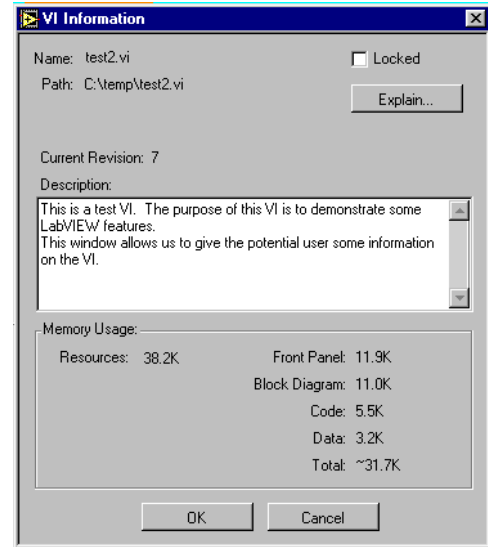
If you place the tool over a control or indicator, you will notice that it comes up with the label of the control and the information "No description available". You can change that by using the pop-up and selecting **Data Operations>>Description**. This will give you a simple window into which you can type some useful description of the item. The when you (or someone else) asks for help on that item, the description appears. Here's a simple message that I put in for the "Result" thermometer:



These facilities are extremely useful.

Documentation is the curse and salvation of many systems and a little bit of time spent in documenting things at an early stage (like before making the VI!) Will save an enormous amount of time later. Remember that the person you may be writing the help file for is a grumpy version of yourself, five years older, and working your second double shift in a row!

Finally there is a space for an overall description of the VI. This is found under **Windows>>Show VI Info**. In this area you can write an overall description of the VI and its functionality. Again this is useful as “unlosable” documentation for the VI.



Summary

- ▶ Data can be of several types - LabVIEW will convert between them but you might need to intervene from time to time
- ▶ Help can be easily written for any control and for the overall VI - do it!

Exercise

Create a blank “while loop” with an indicator for the iteration count. Time the loop with a watch and find out the limiting rate for a blank loop. Using the timer, as above, but by varying the interval, find out the fastest timed loop iteration rate which actually executes at the right rate.

PHY 406 - Microprocessor Interfacing Techniques

LabVIEW Tutorial - Part V

Loop Variables??

Thanks for the Memory!

One of the things that we often need to do is to refer to values in a *previous* loop iteration. Thus we might want to average the last four values of the random number generator. To do this we need a shift register. A shift register takes an input in one iteration and delivers it as an output into the next iteration. So why is it called a shift register? Well because it is a little more elaborate than that. The shift register can be as long as you like and values are shifted down one element at a time. So there is one input and many outputs.

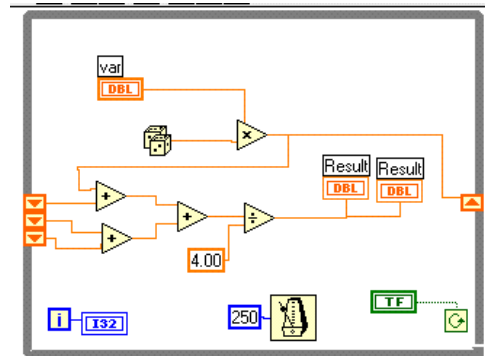
Suppose we have a shift register with four elements which contain 75,99,63,27. A new value of 93 is pushed into the shift register and everything moves down one. The shift register now has 93,75,99,63. At the next iteration 45 is pushed in and the register now has 45,93,75,99 and so on.

We can generate a shift register by popping-up on the right-hand side of the while loop and selecting **add shift register**. Two new icons pop-up on the right (input) and left (output) of the loop. Go to the left one and pop-up - select **add element** - do it again and you now have three elements which go in time order down the page, newest on top.

Re-arrange your VI to show the average of the last four values of the random number generator. The result should look something like this:

Run the VI and you should get the expected result.

It isn't going to take a rocket scientist to realise that doing any significant arithmetic in LabVIEW is going to be tedious. Evaluating any complex expression will involve lots of elements and will rapidly get tedious. Fortunately there is a better way.



Under **structures** there is an item called a **formula node**. This is designed to make things easy. Remove the multipliers and dividers for the averaging and stretch a formula node box into the space.

The formula node box can have inputs and outputs which must be named like variables in an algebraic expression - which they are. You then type the formula in the box in terms of the variables. Create inputs x0, x1, x2, x3 and output y on the formula node box by using the pop-up

on the edge. Then using the text tool type the formula $y = (x_0 + x_1 + x_2 + x_3)/4$; into the box (don't forget the semi-colon, it's important!). You may need to re-arrange things a bit at this point. Now run the VI again and all should be well. Although in this example, the two ways are roughly as bad (or good) as one another, for more complex formulae, the formula box will win. More importantly, the formula box is clearer and clarity is worth a lot.

More Loops

Now I want to introduce you to another loop and thereby start to introduce the concept of arrays. The problem we will tackle is to present the average and standard deviation of a number of the values produced by our VI. We could do this by averaging them explicitly, but that would get tedious and there are better ways. We need to group the numbers into an array and then analyse them. We will delete all the shift register and averaging code to make things simpler.

The "For" loop is the construct to use, and proves very handy because it automatically puts some outputs into an array. In order to see this, proceed as follows. Using the select tool find the for loop in **functions>>Structures** and drag it over the entire VI inside the while loop leaving the loop counter and the termination switch outside. A for loop has two numbers associated with - the Termination count (N is the upper left corner) which is the total number of times the for loop runs before terminating and the iteration count (i in the lower left corner). Use the pop-up to attach a control (outside the loop) to the Termination count.

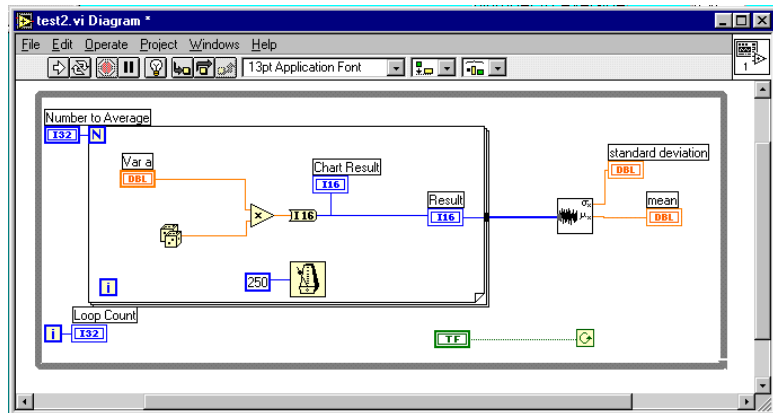
At this time it might be worth pointing out that if you place a control on either panel you can use **find control** on the pop-up to locate it on the other panel - very useful as LabVIEW's idea of positioning is often "interesting".

Now you want to stretch out the while loop to take some more stuff and you will probably want to expand the window at the same time.

Now find **functions>>Analysis>>Probability and Statistics>>Standard Deviation**. This VI takes an array as input and produces the mean and standard deviation as output. You will certainly want a little help with this module as you use the pop-up to create the output indicators (use **Ctrl-H** to guide you). Wire the input to the output of the I16 converter in the for loop. As you do that you will notice two things: first that there is a black square on the boundary of the loop indicating something is happening, second that the line inside the for loop is thin, but outside it becomes thick. That is because the black dot is assembling the individual outputs from each iteration of the for loop into an array whose size is the number of times the loop executes - the

Termination count.

This is most important. Set the Termination count to a number >0 and then run the VI. Try experimenting with numbers >0 . Why not 0? Well, what is an array with 0 elements, and how do you find the mean and standard deviation? This is a guaranteed error situation!



We can cope with this situation in two ways: Limit the data range of the Termination count to be always >0 or install an error handler. I'm going to show you the second way, because you can already do the first.

Summary

- ▶ Shift registers enable us to use the results from previous loop iterations
- ▶ Formula boxes are clearer and more compact for more-than-trivial arithmetic
- ▶ For loops enable us to form arrays quickly for analysis as a group.

Exercise

Use shift register and some additional indicators to display the last three values of the mean as well as the current value.

Use a formula box to modify the output of the random number generator to ax^2 instead of ax as at present.

PHY 406 - Microprocessor Interfacing Techniques

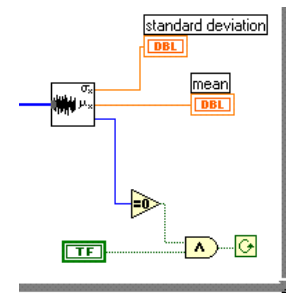
LabVIEW Tutorial - Part VI

Error Handling

Handling Errors

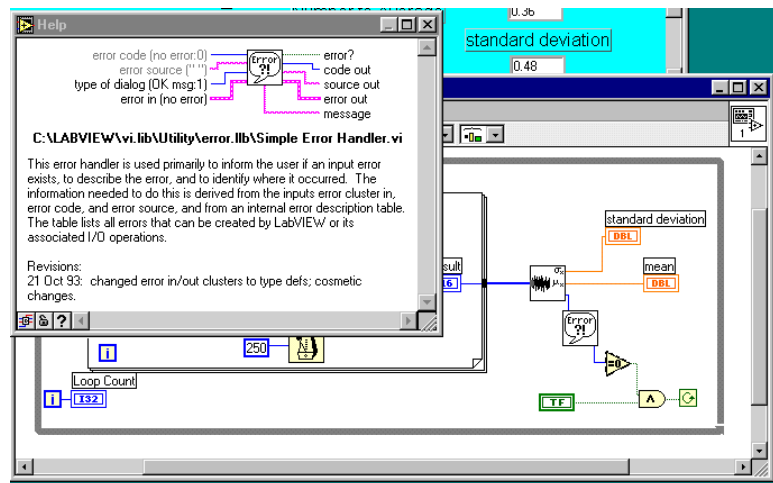
Before I show you how to connect an error message handler into the system it is important to realise that an error handler has to do something with the error or it will happen again. In this case if we find an error (array dimension <1) then unless we either increase the array dimension or abort the program, it will immediately happen again, and again, and again..... It is therefore important to always think about what to do about an error as well as report it. In addition if we are going to stop the program, we must ensure that the error is reported before the program is stopped - ie the data flow must be through the error handler.

The standard deviation box has an error output for exactly this purpose. It is an integer whose value is 0 in normal operation and non-zero if there is an error. A simple error handler therefore would stop the program if the error output was non-zero. Or phrased another way: The program continues if the run switch is on (true) and the error output is zero. The logic looks like this:



This will certainly stop the VI if the error output becomes non-zero, but it won't tell you why it's stopping! What we need is an indicator to tell us what the error is. If you actually put an indicator on the line it will indicate a number (-20003 to be precise) but that still doesn't tell me what the error is. I need that error to be interpreted in plain Canadian. This is the job of the (simple) error handler. Find that on **functions>>Time & Dialog** (bottom left corner) and place it on the diagram.

This function is quite complex as you can see from the help screen (**Ctrl-H** remember!). The only two leads you need at the moment are the "error code" input and the "code out". Remember what I said about dataflow. The error handler must be traversed before the VI is stopped. We therefore wire the error output of the standard deviation analysis to the input and the output to the comparator and abort logic. This ensures the proper dataflow.



Now when you run the VI, try using 0 as the number of elements to average and see what happens. You should get a nice error box which tells you that “Error -20003 has occurred at an unknown location and the probable cause is that the number of samples for analysis is less than one.

A final piece of customisation is to add a text string to tell the system where the error occurred - the error source. After all, in a real VI there might be a number of places that this error could occur and you would like to know which one it really is. You can resolve this by adding a “string” constant for the error source. String constants are exactly what they seem you can find them under **functions>>string>>string constant** or you can use the wiring tool and the pop-up menu to create a string constant input - you’ll need a steady hand for that, things are getting tight in that area! In both cases add an appropriate string to explain wher the error is coming from and the box will reflect that text. Notice the new colour for the string wiring.

Summary

- ▶ The action taken after an error should make sure that the system does not loop on the error - appropriate action may be an abort
- ▶ The dataflow must go through the error handler before the error recovery action is taken

Exercise

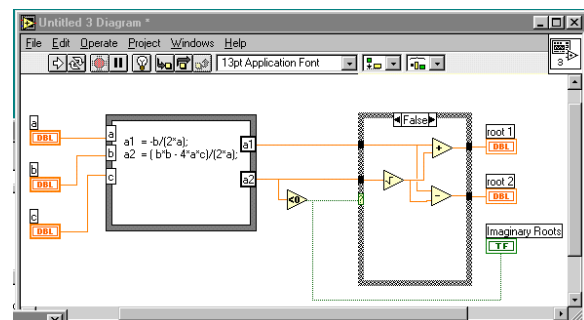
Under **Advanced** there is a **Stop** function. Modify your VI to use it. Is this a good idea? What are the pros and cons of **Stop** vs other means of stopping the VI?

Decision, Decisions!

One thing that we often need to do is to take a different action depending on a decision. “If the number is negative, then don’t take the square root” is a reasonable example and one which will avoid an error if we cope with it properly. In LabVIEW we can use a **case** construct to take care of this. A **case** construct consists of a number of overlaid panels each corresponding to a particular value of the input variable. The simplest case is a true/false decision which can be implemented with a boolean variable, more complex cases involve more states and use an integer control variable.

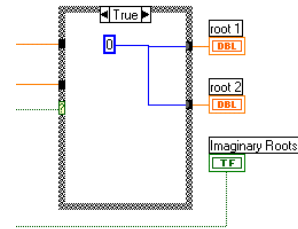
Case constructs must all have the same number of inputs and outputs. All outputs must be driven on all panels of the construct, inputs may be used or ignored at will.

A simple case construct can be made if we try to find the roots of a quadratic equation with coefficients a,b and c - we need to account for imaginary roots. Here is the VI I put together for that. You should try and re-create it. The **Case** structure is under **Structures**.



The other panel of the **Case** structure simply places a zero in each of the outputs. Remember that every output must be driven on every pane - no exceptions.

Here is the other pane of the **Case** structure. Notice that the unwanted inputs are ignored.



Summary

- ▶ **Case** structures allow programs to take different paths depending upon the conditions.
- ▶ All panels of a **case** structure have the same number of inputs and outputs.
- ▶ Unused inputs can be ignored
- ▶ All outputs must be driven.

Exercise

Try writing a VI to compute $\sin(x)$, $\cos(x)$ or $\tan(x)$ depending upon an input control (if you get really good try using a “text ring” control). Your program should also worry about the legality of input.

PHY 406 - Microprocessor Interfacing Techniques

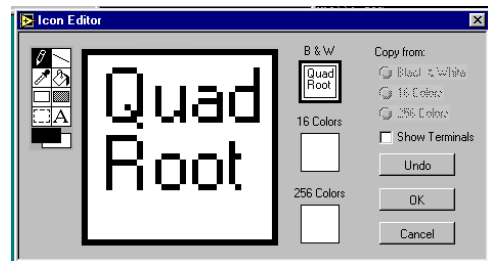
LabVIEW Tutorial - Part VII

Wrapping It Up - SubVIs

Hiding the Hard Work

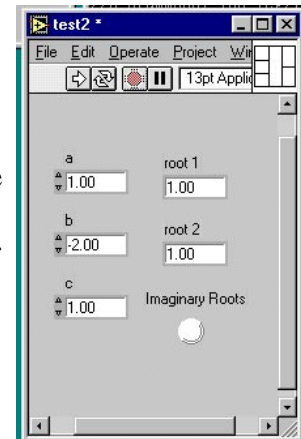
Encapsulation or the enclosing of a body of work into a sub-VI is an important part of serious LabVIEW programming. We are going to encapsulate this VI. To do this we will need to create an ICON and also to define the inputs and outputs required for this VI. When we encapsulate a VI the controls are replaced by the inputs to the sub-VI and the indicators are replaced by the outputs from the VI. It isn't necessary to have all the controls and indicators accessible from outside the sub-VI, but in the case of controls, you had probably have a good idea about what values they default to and whether these are the right values. It is probably better to use constants to prevent changes being made accidentally.

Go to the *front panel* and pop-up on the icon in the top right. Select **Edit Icon**. A little graphical editor should pop-up. I am assuming that you can run something like this. I used the select (dotted box) to clear the icon and then the text tool to write and got this:



When you exit the icon editor you will notice that the icon has changed to your new one. This is the icon which will show when you use this entire VI as a sub-VI.

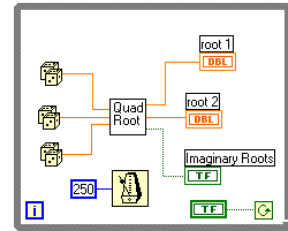
Now pop-up on the icon again and select **Show Connector**. LabVIEW assigns a connector pattern to this VI. The general rule in LabVIEW is that inputs are on the left, outputs are on the right. This connector pattern has only two slots on the right and I want three, so I can change it with the pop-up **Patterns** and get one with three slots on the left and three on the right. Notice also that the tool has changed to the wiring tool. To associate each of the inputs and outputs with a particular sector of the connector pane, click on the front panel item and then on the connector sector. When a connection is established, the sector turns dark.



Now would be a really good time to add descriptions to all the controls and the VI to enable the help facilities to work properly. When you have finished all this, save the VI (don't forget the ".vi" at the end of the name).

Now open a new VI and using the **Select a VI..** option you can put your new sub-VI in place. If you did add the help for it, you can now access all the narratives and your VI will look just like many other parts of LabVIEW

Here's a really simple use of our sub-VI



Summary

- ▶ Sub-VIs can be created from a conventional VI.
- ▶ When used as a sub-VI, inputs replace controls and outputs replace indicators
- ▶ The ICON can be edited to be more realistic
- ▶ The connector pane establishes which inputs and outputs are placed where
- ▶ Help is very, very useful in order to use a sub-VI

Exercise

Construct a sub-VI to compute $x!$ ($x(x-1)(x-2)\dots(1)$) from x as an input. You might want to think about the range and type of input you will accept and an efficient algorithm for computing the result.

PHY 406 - Microprocessor Interfacing Techniques

LabVIEW Tutorial - Part VIII

Arrays and Clusters

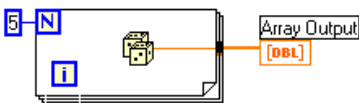
Arrays, Clusters and Conglomerates

LabVIEW has two group constructs: arrays which are N-dimensional arrays of entities and clusters which in any other language would be called “structures”. Both of these are quite useful and are also essential for accessing some of the more powerful LabVIEW functions.

We have already met arrays briefly in an example above. This VI makes up a 1-D array of random numbers 5 long. We examine this array by using the array shell from the **array** controls. Notice that at the moment this has no reality to it - it is just a shell because we don't know what sort of thing to display.

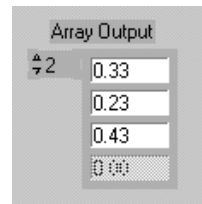


To get further we drop a numeric indicator into the shell - now we know it will be a numeric array (as opposed to an boolean array for example). The shell now looks like this. The left-hand control is for the index - the right-hand value is the value of that index. Since we haven't actually wired the indicator up yet, we don't know whether the index “0” is valid or not (although it should be) so the value is “greyed out” even though I changed the background colour to white.



You can use the select tool to resize the indicator box to display more elements and when you have wired up the indicator to a real array and run the program, the indicator comes to life.

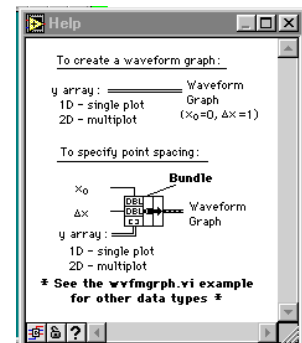
Notice that since the array size is 5 and array indices start from 0, the last element on the list doesn't exist and is therefore still greyed out.



Now let's feed the array to a graph (remember charts are for values which come one at a time, graphs are for arrays). The result should be a simple graph of the value of the random number generator for five values each time you run the VI.

A useful technique to learn at this stage is to “highlight execution” which allows you to run the VI in slow motion and watch the values being produced. This is controlled by the light-bulb icon . Press the lightbulb and run the VI - you can now see each value being produced and how the loop runs. You may need to try it a couple of times to see the full effect. Press the lightbulb again to turn this off.

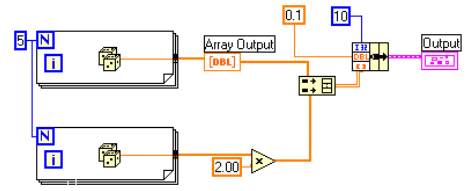
The x-axis of the graph is the array index. It is possible to change that by adding information to the input and for this purpose we need a



cluster. Since we don't have one we are going to have to make one by "bundling" a number of elements together. Look very carefully at the help information for the graph. It shows that we can have up to three elements in the cluster: the initial x, the x increment and the array. In order to use these we must bundle up the array and two other scalars into a cluster.

Choose the **Cluster>>Bundle** function. It appears on the block diagram as a two-element bundler. Use the select tool to resize it to three elements. Wire the output to the graph and the third input to the array. Generate two constants (using the wiring tool pop-up on the terminal and select **Create Constant**). Use 10 for the initial x and 0.1 for the increment. Now when you run the VI you should be able to see that the x-axis has been rescaled appropriately.

More study of the help for the graph indicates that a 2-D array will yield two plot lines. So now we can duplicate the array generator (select the For-loop and do a **Ctrl-drag**). Just to distinguish between the outputs - put a multiplier on one of them to double the array value. The multiplier copes with multiplying an array by a scalar without further intervention. Since we have two 1-D arrays and we need a single 2-D array we need the **Array>>Build Array** function resized to two inputs. We feed a 1-D in each input and a 2-D becomes the output. We can then feed this to the bundle function and then on to the graph. If this all works then the graph should now have two lines on it and an x-scale from 10.0 to 10.4. Here is my block diagram for this.



You might be interested to run this VI under the "execution highlight". Notice that both the for loops execute together. This is a consequence of dataflow programming. Since neither of the loops is waiting for an input, both are permitted to run within the limits of available CPU time. LabVIEW has built-in parallelism.

Summary

- ▶ LabVIEW has both arrays (single and multi-dimensional) and clusters
- ▶ Clusters are similar to structures in other computer languages
- ▶ Arrays can be built automatically or explicitly using array builders. The array builders will combine arrays as well as scalars
- ▶ Clusters are built using the **Bundle** directives (they can also be **Unbundled**)

Exercise

Write a VI to generate a graph of a user-supplied quadratic over a user-supplied range of values.

PHY 406 - Microprocessor Interfacing Techniques

LabVIEW Tutorial - Part IX

Sequences

Sequences

LabVIEW works on the concept of *dataflow* - a function or VI executes when the inputs are available. This is generally OK since we are processing a flow of data, but there are occasions when this is not the case. One of the principal areas where we get into trouble is timing.

In a typical timing experiment things go like this:

- i. Set up the system
- ii. Start the clock
- iii. Send the event trigger
- iv. Wait for the ending event
- v. Stop the clock
- vi. Tidy things up

Now we can debate fiercely whether we should start the clock before sending the trigger or afterwards, but there is a much deeper problem in LabVIEW and that is that there is no obvious dataflow in the pattern. Nothing flows through “start the clock” to get to “send the trigger”. These events (starting the clock, sending the trigger, stopping the clock) could all occur at very different times and in different sequences, but we wish to impose a sequence on them. In LabVIEW this is the job of the *sequence* structure.

In common with the *case* structure, the *sequence* structure has a number of panels lying on top of one another. The difference for the sequence function is that they all execute one after the other.

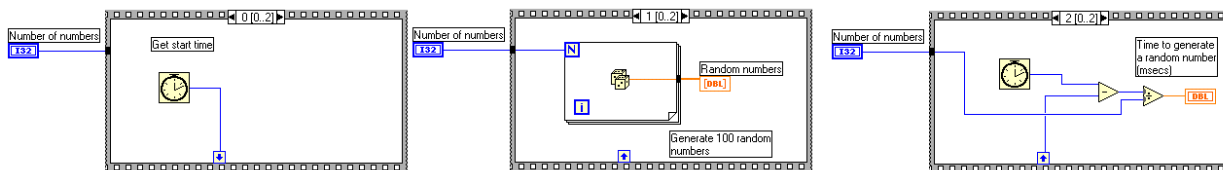
We also need a way of transferring information between the panes as well as to the outside world. Communication with the outside world can be made through *tunnels* which permit a wire to cross the boundary of the sequence. A *tunnel* can come onto or off any pane, but as usual input tunnels can be used in multiple panels and can be ignored at will on other panels, whereas output tunnels must have one and only one connection.

We also need a way of communicating between panes. This is supplied by a *sequence local* which is created anywhere on the perimeter of the sequence panel and can then be fed from an output on one panel to input(s) on other panel(s). *Sequence locals* are created from the pop-up menu on the edge of the sequence. This menu also allows you to add panels before and after the current one as well as to delete panels.

Moving between panels can be accomplished by **clicking** on the arrows at the top of the sequence.

Here is a simple example of a sequence. We wish to find out how long it takes to generate a random number. We need to get the time before we start, generate a lot of numbers and then find the time when we stop - from the difference we can find the time to calculate one random number (plus a bit of “end effect” - but we’ll ignore that at the moment)

Here are the three panels which make up this sequence:



(Note that this diagram is made up from three separate views of the sequence - you would normally only see one of these panels at a time)

The first panel collects the time in msecs (since some arbitrary start which needn't concern us here). This panel transfers the start time to other panels using the sequence variable at the bottom.

The second panel computes the required number of random numbers. It uses the input control to tell it how many numbers to compute and display. This tunnel input is available on all panels, but is only used in two panels

Finally the third panel takes the start time, subtracts it from the end time and divides the result by the number of random numbers calculated to give us the time per random number.

The sequencing of the panels ensures that things are done in the right order.

Many instrument set-up operations require that some things are done before others, and this is another useful place for a sequence. There are other ways of achieving this end, but this is often one of the clearest and simplest.

Summary

- ▶ Sequences can be used to force things to happen in a given order if the dataflow is not sufficient to do so.
- ▶ Sequence locals permit variables to be transferred between the panes of a sequence
- ▶ Tunnels permit variables to be brought in and out of the sequence

Exercise

Make a sequence to measure your reaction time to a light going on. (You will only be able to do this to about 1mS, but the exercise should be instructive)

PHY 406 - Microprocessor Interfacing Techniques

LabVIEW Tutorial - Part X

File Output and Input

Introduction

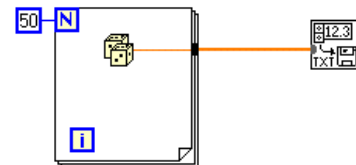
File I/O tends to be complex - simply because there are a myriad of things that you *might* want to do and the software has to be able to let you do (most of) them. LabVIEW can read and write four type of files:

- ▶ **Spreadheet:** These files consist of ASCII (normal) text with **tabs** in suitable places for importing into a spreadsheet. The VIs which control this form write one array to the file at a time.
- ▶ **Character:** These are similar to spreadsheet files in that they consist of ASCII text, but the format is freer.
- ▶ **Binary:** Consist of the data in computer format instead of ASCII. That means that the computer can read them, but you can't! They are more compact than character files and do not suffer from round-off errors. They be trasnlated into ASCII by other programs. Since the data do not have to be translated from binary to ASCII, they are also faster to write. Binary files can be files of 16-bit integers or single-precision reals.
- ▶ **Datalog:** These are similar to binary files except that the data can be more complex types (eg clusters). They must be read in the same format as they are written which constrains them (more or less) to be read by a similar LabVIEW program to the one which wrote them.

This tutorial is going to be concerned with *spreadsheet* files only. If you understand these - then you will be able to understand the LabVIEW elements of the others very quickly.

Writing Arrays

The unique thing about spreadsheet files is that they take either 1-D or 2-D tables of data. This is a simple VI to write a 1-D array of 50 random numbers to a file.



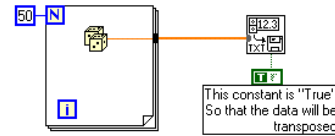
This is a representation of the output you get from this VI:

0.123 0.456 0.789 0.012 0.345 0.678.....

In other words a long line of numbers. This is probably not quite what you were thinking of - I suspect that if you had thought about it at all, then you would have expected a column of numbers, one to a line. However in LabVIEW terms *Rows* come before *Columns* and so we gat a

row of data.

Here is a second VI to make the point clear. It writes a 2-D array of numbers with the second row of the array being just 10x the first row.



The result is:

0.123	0.456	0.789	0.012	0.345	0.678.....
1.234	4.567	7.890	0.123	3.456	6.789.....

Fortunately the LabVIEW people have thought of us and you can *transpose* the array, either 1-D or 2-D so that columns become rows and vice versa. This is accomplished with the *transpose* input set to “true” (These complex functions cry out for you to use the **Ctrl-h** help capability to sort out the terminals) as shown in the diagram.

The output from the second example now becomes:

0.123	1.234
0.456	4.567
0.789	7.890
0.012	0.123
.....	

Writing to Files

Logically there are a large number of things to do with selecting the filename which need to be sorted out.

Do we want to select the filename at run-time or use a fixed filename?

Do we want to select the filename arbitrarily or have a pre-selected one we can over-ride?

If the file exists do we want to

- i) give up
- ii) append the data to the file
- iii) Ask if we should over-write it.
- iv) always over-write the file with the new data

The examples above do absolutely nothing about any of these questions so they use the LabVIEW default. If you construct this VI using **function>>File I/O>>Write to Spreadsheet File** and run it, you will find that it comes up with a graphical selector allowing you to navigate the disk and directory structure of the machine and then **double-click** on a file or type in a new filename. Notice that if the file exists, you are queried as to whether you want to over-write it or not.

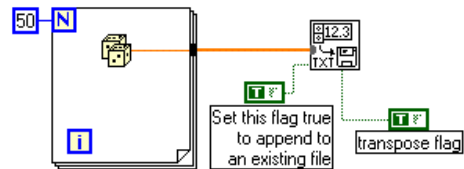
Filenames in Window95 are quite special. Having got rid of the “8+3” naming convention of DOS, you would think that everything was free-form. Well, it is but... There is a very useful property of Windows95 that it can associate a file with a program and then when you **double-click** on the filename it will launch that program with the file in it. The specific example we are going to use here is that we want text files to be launched into the text editor where we can read the contents. The marvelous thing about this is that it is operated by the three-letter extension of the filename. Thus “something.txt” will be recognised as a text file.

Windows95 will, when it recognises a file extension make the file have an appropriate icon (in this case a little notepad) and will display the filename *without* the extension.

The point of this box therefore is that it is probably a good idea when creating spreadsheet files to give them a “.txt” extension and then you can open them easily. But remember that the extension will not show up in the graphical representation of the directory. It will show up in a command-line (MS-DOS) directory listing.

To view your files you should **double-click** on the “My Computer” icon and then follow down the disk and directory structure with **double-click** until you can see your file. **Double-click** on the file will bring it up in the editor so that you can read it.

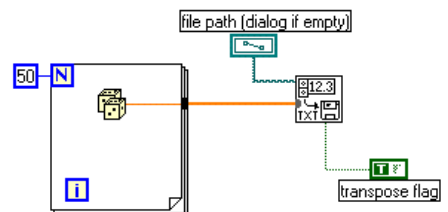
Assuming that giving up is not the option that you want, the next viable one is to append the data to the file. This can be accomplished by the “append” input which defaults to “false” but can be set to “true”. I’ve added it to the “transpose” input here to clarify the presence of both flags.



If the file does not exist then the file is created.

If we want the option of over-writing, we have only to omit the “append” flag or set it false and we will always be queried about existing files when we **double-click** on them.

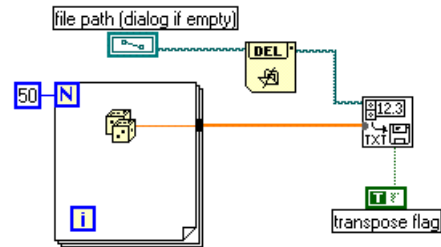
The more complex option is to pre-select a filename. This is simple enough to do by creating a constant or control of the “file path” type (easiest way is to use the wiring tool to **pop-up** on the terminal and create a constant or control). We will select a control. If the input selection is blank, then we will have exactly the same situation as we had before because the file path input defaults to an empty path and therefore a dialog box. However if we now enter a filename we will use that filename.



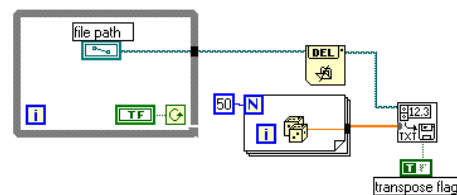
It isn't exactly right because if you play around with it a bit, you will find that it will create a file nicely, but it defaults to the "query before over-write" option whereas we might be more interested in the "over-write whatever" option. The only way of doing that is to delete the file before it starts with the "advanced"

function>>File I/O>>Advanced File Functions>>Delete

. Putting this in-between the control and the write function will cause the file to be deleted if it already exists. Notice the use of "dataflow" programming here as the path is taken as an output from the "delete" function to ensure the execution order is correct.



You might want to ponder (or better still - try out) what this does:

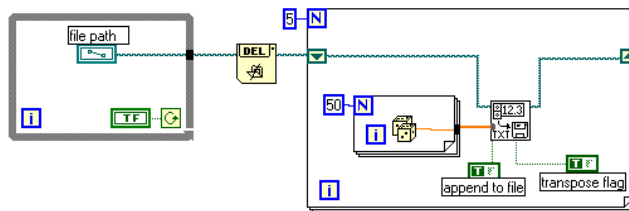


As a small clue - here is the corresponding front panel:



You can obviously get even more sophisticated with selecting a filename, but this walk-through should cover a lot of situations.

The next issue to deal with is the problem of writing several arrays to one file. This can be accomplished by using the fact that the **write to spreadsheet file** function has a path output as well as a path input and therefore you can use the concept of "dataflow" to establish a filename and then proceed to write to it. Here is a concept for writing a series of arrays to a file.



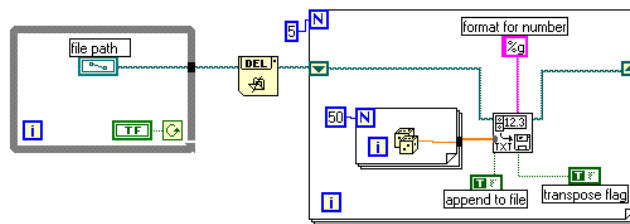
Notice that we have used a shift register to transfer the filename from one iteration of the loop to the next but we have also *initialised* the shift register from outside the loop to give it the correct initial filename.

Formatting the Numbers

You will have realised by now that default way of showing the numbers in the spreadsheet file is as a floating point number with three places after the decimal point. This can be changed by specifying a format with a string. The syntax used is the C-language syntax and for those who don't know it - here are a few useful examples:

- %d - show as an integer number (for I16)
- %g - show as a "general" number
- %fa.b - show as a floating point number with "a" places for the entire field and "b" places after the decimal point.

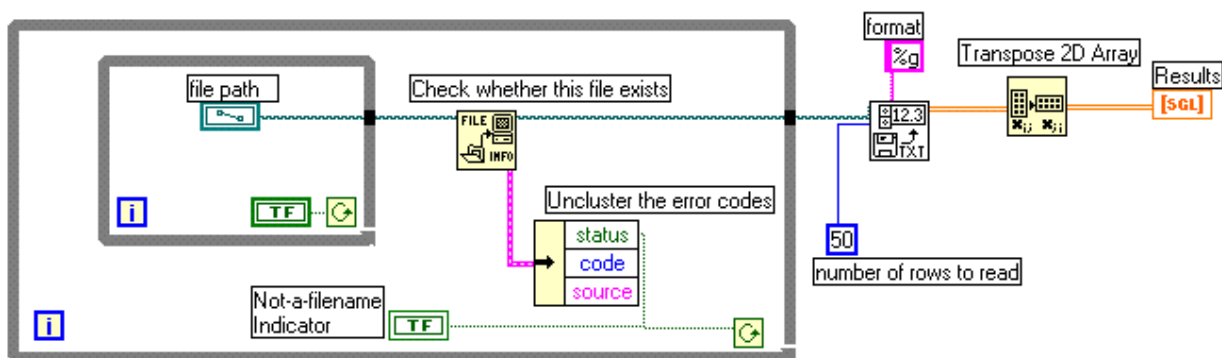
Here is a case where the syntax is used specifically.



The "%g" syntax in this case picks a suitable format for the numbers - in this case it picked "0.123456" as the format instead of the "%.3f" which is the default if you don't supply anything.

Reading Files

Reading files is probably a bit less important in LabVIEW because it is primarily designed for getting data from other places than the filestore. However everything that can be written in LabVIEW can be read. Here is a VI that will read a file of 50 numbers, one to a line, and plot them:



There are several things to note about this VI.

- ▶ Since LabVIEW considers rows before columns. Asking it to read 50 numbers, one to a line, is like asking it to read a 2-D array one wide and 50 deep. To get to a 1-D array, we need to transpose the matrix - 1x50 becomes 50x1.
- ▶ The 50 on the read VI corresponds to the number of rows read, irrespective of the number of numbers along a row.
- ▶ The %g refers to the format to read numbers with. %g will read just about any format - useful if you don't know what's coming!
- ▶ The double loop permits you to do two things: First confirm that the right filename is in the file path control and second, repeat the selection if the file is invalid.
- ▶ The checking of the file validity is done by a call to **function>>File I/O>>Advanced File Functions>>File/Directory Info.**
- ▶ The error cluster is unbundled and the error status (true/false) is used to terminate or repeat the loop and drive the "not a filename" indicator which is a modified "button" on the front panel
- ▶ Dataflow is used to ensure that the file operations are done in the right order.

Summary

- ▶ Spreadsheet file contain numbers separated by **Tabs**
- ▶ Spreadsheet files can map arrays in two different ways - the default is Rows before Columns.
- ▶ Getting the correct filename and getting the correct action for a potential over-write situation requires some care and thought
- ▶ Writing multiple arrays to a file requires using dataflow to move the filename through the various writes
- ▶ Formatting the numbers is simple - follow the C conventions.
- ▶ Reading is as simple as writing, but the file must exist.

Exercise

Write a program to write the a file of 30 lines each line consisting of the integer line number and the factorial (n!) Of the number thus:

1	1
2	2
3	6
4	24
5	120
. . . .	

Why won't I let you go any higher than 30?

Write a program to read a file of numbers at the rate of two a second and display the numbers in strip chart form.