Mechatronic Systems 210

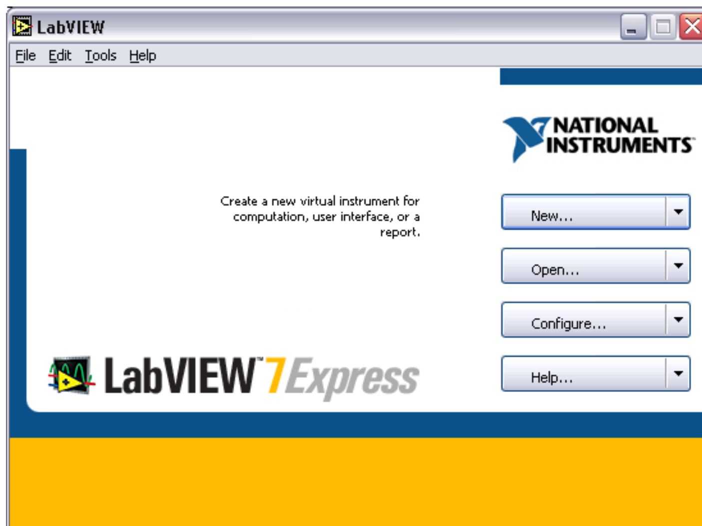
Introducing Labview for Mechatronics Systems Course

This is a guide for your first steps in using the LabView graphics "G" programming language for this unit. The introduction is short, and involves several advanced techniques in structure programmming within G. Why do I throw you "in deep water" right at the start of the course?

You will be learning to use G as a means to an end. The aim of this course is for you to learn about building mechatronic systems and some of the constraints that systems designers face. LabView is a useful tool for this: a tool that is widely used in real commercial systems.

You have the advantage of having learned (or currently learning) Java, a structured high level programming language. This makes it much easier for you to learn G (and hence LabView quickly) and at the same time appreciating some of the high level language concepts we will introduce.

Starting Labview



When you first start LabView you will see a window like this (or a larger one with similar controls). Normally, you will start LabView by double clicking a LabView application file.

Select the downward arrow button at the right hand end of "New…" and choose "blank VI" rather than "choose from…"

A "VI" is a virtual instrument: a simulated control panel, and it comes with two main viewing windows:

a) The front panel, that has a grey background by default

b) The diagram, with a white background, where you construct the "G" program using icon modules and wires.

Shortcut: Use "Ctrl-E" to swap between the front panel and the diagram windows.

The panel and the diagram are initially blank.

Controls and Displays: Exercise 1

In the first step we will construct a simple program that demonstrates:

- Controls
- Displays
- Simple wiring
- A while loop
- Stopping the program using a "stop" button

First, make sure the tools palette is visible: use the "Window" menu above the front panel, and select the "Show Tools Palette" entry. (*Note, you may have to click the "v" at the bottom of the menu to show all entries on the menu to find it*).

The tools palette (left) is important: you will use it all the time. The controls are as follows:

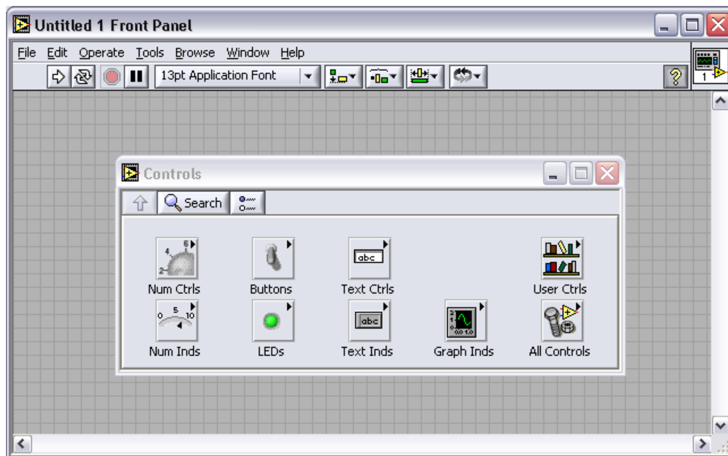
Automatic Tool Select (Green light)		
Operating tool	Arrow: Position, Re-size or Selection tool	Text editing tool
Wiring tool	Object short cut menu	Positioning tool
Set/clear breakpoint	Probe	Colour sampler
Foreground colour	Background Colour	Paint tool

While editing a front panel or diagram, you will use the arrow, text and wiring tools nearly all the time.

Shortcuts: The space bar can be used to swap between the arrow and wiring tool: this is very useful.

The "tab" key can be used to quickly swap between the four most common tools: operate, arrow, text and wiring.

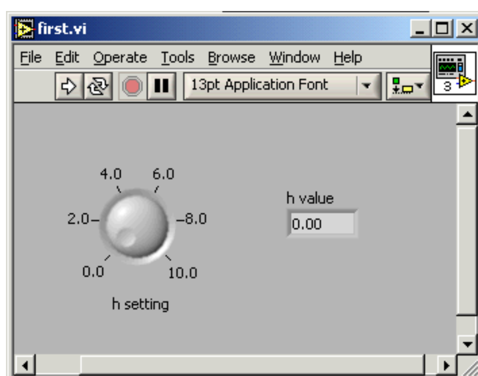
Move the tools palette to one side, and select the arrow tool. On the diagram, right click the arrow: the controls palette will appear. This step take a little practice. The controls palette is a graphic menu: as you move across it, other menus of graphic objects appear.

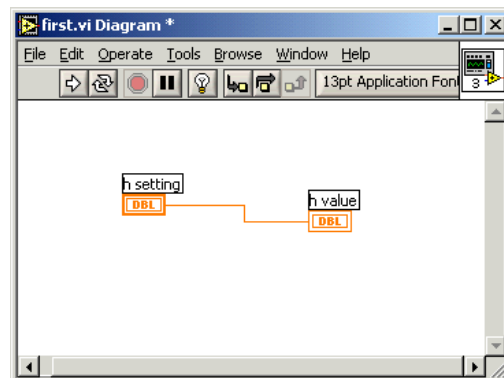


The top left corner is the numeric controls group: go into this and select a knob control. When you do this, the controls palette vanishes and is replaced with a shaded outline showing the size of the control: place it where you want it on the front panel by moving the mouse. Click when you have the right position, and the knob appears. The name “knob” at the top is black – selected text. Type the name of the control immediately, for example, “h value”. This is the control *label* and it can have spaces in the name. Left click somewhere else on the panel and the text will be “frozen” and de-selected. If you forget to type the label you can change it at any stage with the text editing tool. Try this out now and change the label to “h setting”.

You can move the label anywhere you want by selecting it with the arrow tool and dragging it.

Now go through the same steps to add a numeric indicator display to the front panel, called “h value”





Now look at the diagram.

You will see two *terminals* corresponding to the two front panel objects. (*They can be displayed as small rectangular terminals or icons representing the front panel control: right click the terminal and select the appropriate option to change the appearance*).

One terminal has a heavy surround: this is a “source” or “control” terminal that produces a value.

One terminal has a thinner surround: this is a “sink” or “indicator” terminal that accepts a value and displays it.

Next, use the wiring tool to construct a wide from the control terminal “h setting” to the indicator “h value”.

You have now completed your first “G” program: save it using the file menu. The diagram and the panel should look like these pictures.

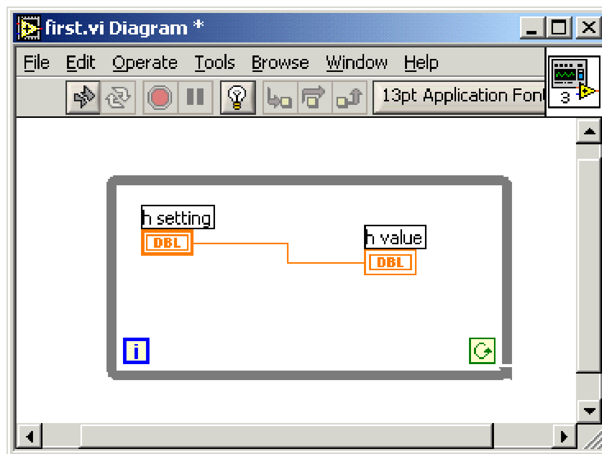
Now, using the operate tool, move the knob to the vertically up position (5.0).

Press the run button (the right arrow on the toolbar).

The program runs and stops immediately, and the numeric indicator will have the value 5.0 (or approximately that) displayed.

However, the program stops immediately, so using the operate tool to move the knob will not affect the numeric indicator unless you start the program again.

How do we make the program run continuously?



For this we will need a while loop, and a method to stop the program when we have finished using it.

Select the arrow tool.

On the diagram window, right click in a clear spot and the functions palette appears. This is larger than the tools palette, and it takes a little while to become familiar with it.

The while loop can be found in the structures group: this is the top left group on the functions palette. Select the while loop just as you selected controls on the controls palette and place it over both terminals. Click first a little to the top and left of the “h setting” terminal, and then click again below and to the right of the “h value” terminal so both are inside the rectangle. After the second click a heavy dark line will denote the while loop:

Now run the program again. It will now run continuously, but it is a very “rude” application. It will hog all the processor time and can only be stopped by pressing the stop sign button. This should only be used when absolutely necessary. We will now add a couple of extra features to make this a well-behaved application.

Select the arrow tool.

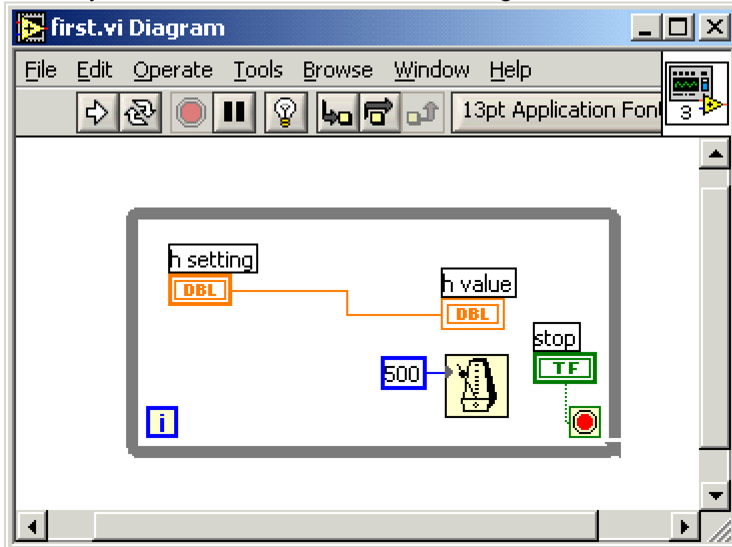
Select the “Wait until next Ms multiple” function” – in the third row of the functions palette under the watch symbol. The icon looks like a little metronome.

Place an integer constant (500) next to it. Right click in a clear spot for the functions palette again and find a blue numeric constant at the bottom left of the “numeric” goup in the top row of the functions palette. When this is placed it is black: type the constant value (500) and click somewhere else to “freeze” the value. The default value is 0: you can use the text editing tool any time to change it if you forget to set it immediately.

Use the wiring tool to join the 500 constant to the "wait until next Ms multiple" function. You must wire to the left hand side of the function!

Next place a "stop" button on the front panel: this is in the boolean group in the top row of controls.

You may need to move the terminal on the diagram – make sure it is inside the loop rectangle!



Now change the green looping terminal at the bottom right hand corner of the while loop to a stop sign. Use the arrow tool, and right click the terminal. You will see a "Stop if True" entry on the short-cut menu that appears. Select this and the terminal changes to a stop sign.

Now wire the "stop" button terminal to the stop sign that controls the while loop, as shown in the picture.

Now the application is well behaved. The loop will only execute once every 500 milliseconds to the processor can do other things while the program runs, and the stop button will terminate the application nicely. Try it out for youself and congratulate youself on completing exercise 1.

LabView Self Help

Now that you have taken your first steps, it is time for you to use LabView's tutorials and activity exercises. Go to the "help" menu and select "View Printed Manuals". Acrobat Reader opens the summary pages. Several of these are very useful and some of these are required reading for this course:

> LabView Development Guidelines
> Application Note: Using LabView to Create Multi-threaded VI's for Maximum Performance and Reliability

Start with "LabView Tutorial" (Note: do not open the "Getting Started with LabView Manual – it repeats what you have already done here). This opens the on-line help at the "tutorial" entry point. Work through the tutorial – some of this will be familiar already. Then work through the following activities: the rest are not relevant at this stage.

Activities: 1,2,3,4,5,7 and 8.

There are lots of LabView examples. To gain an insight into what the examples can demonstrate, try the following one. In the on-line help menu select "Examples…". The on-line help window opens (if was not already open) with the categories of examples shown.

Select "Fundamentals", then "Analysis" then "Measurement Analysis" then "Frequency Analysis" and finally "Power Spectrum". Run the demonstration. It demonstrates the analysis of frequency components in a noisy signal. Look at the diagram and notice how relatively simple it seems to be. Most LabView examples are simple in structure because they are demonstrating built-in LabView features.

LabView was developed mainly as a laboratory data acquisition software environment, and you will see a heavy emphasis on this. It is only in more recent years that it has developed into a control and

mechatronics environment as well, and gained widespread acceptance. Originally developed for Macintosh computers, it is now a multi-platform environment that works on most common commercial operating systems.

By now, you will have some familiarity with the following:

Creating and modifying VI's and sub-VI's, basic debugging tools, some basic program structures, front panel objects (controls and indicators), wiring, and data types. You are ready to be thrown in the deep end….

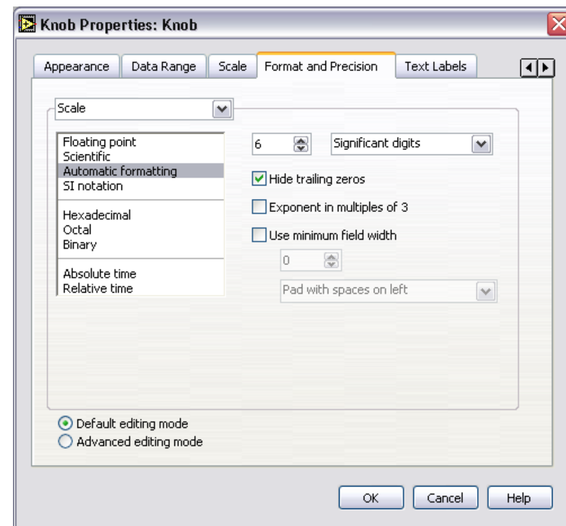
Some LabVIEW 7 Issues

Most of these notes were written for LabVIEW version 6. LabVIEW 7 has a number of additional features but you will find these notes are still completely valid for the new version.

LabVIEW version 7 has an automatic route planner for wiring. It attempts to place wires in free space in the diagram so that they do not overlap. Sometimes this can be very annoying. It can be disabled temporarily by pressing the 'A' key *after* clicking the wiring tool on the starting terminal for the wire. You can also disable this in the "Options" dialog off the "Tools" menu.

The automatic tool selection feature can be useful: select the tool you want if you don't want the automatic tool selection. Select the green light if you want to revert to automatic tool selection.

If you right click on a knob (or other numeric) control and select "Format & Precision" you see the following dialog box:



Changing the number of significant digits can have some unexpected results. If you set the number of significant digits to 1 you may find that successive values on the scale around the knob look very unusual. Change the number of significant digits back to 6 and change the menu setting from "Significant digits" to "Digits of precision". Now you will be able to change the number of digits following the decimal point.

Real Time Issues

To control a real machine, you need software that operates in “real time”. This means that the software has to reliably do what’s needed when it is needed.

To many people, writing reliable “real time” software in the Microsoft Windows environment is an oxymoron: like Militray Intelligence, a self contradiction. Microsoft software is, however, increasing in reliability and is widely used in industry. It is cheap and cost-effective. If you need higher reliability, LabView offers several improved reliability pathways. Many similar software environments for control and automation use similar techniques to LabView so what you learn here will transfer to industrial applications.

The first issue you need to appreciate is that Microsoft Windows is not a real-time operating system. Sometimes it interrupts what you are doing so we need to take precautions and use the system clock to find the actual time and base our calculations on that.

This exercise is based on experience learned over many years of practice in writing and maintaining real time software for mechatronic systems. I have written in machine code, assembler language, Fortran, C, C++ and Pascal. LabView beats them all for most things, especially for multi-threaded applications like those I am about to show you.

Along the way you are going to learn some advanced features in LabView that do not necessarily appear in the text books. For this I have to thank Alex Le Dain and ICON Technologies, the local support company for LabView and National Instruments.

Generating a Sine Wave

To explore this section you need to download some demonstration VI’s. These are located at:

http://www.mech.uwa.edu.au/courses/MS210/demo1.zip

UnZip and extract the VI’s into your own directory before using them. The Word document used to create the PDF document that you are now reading is included with them.

LabView has built-in function generators that are demonstrated in the examples. See, for instance, the “Power Spectrum” example I suggested you look at. These generate sine waves of a given frequency, but it is not so easy to use them for a real-time application.

“First.vi” that we created in section 1 has the metronome function “Wait until next Ms multiple” that ought to make the program run at exactly 500 millisecond intervals. However, despite the fact that a computer is about the most deterministic machine human beings have ever constructed, most computers are anything but deterministic in their timing behaviour, especially with Microsoft Windows and Unix-like operating systems.

There are many demands on the central processor of the computer. Everytime you move the mouse, the computer spends time erasing the image of the mouse pointer by replacing the picture of what lay behind it, and then redrawing the mouse pointer in a new position.

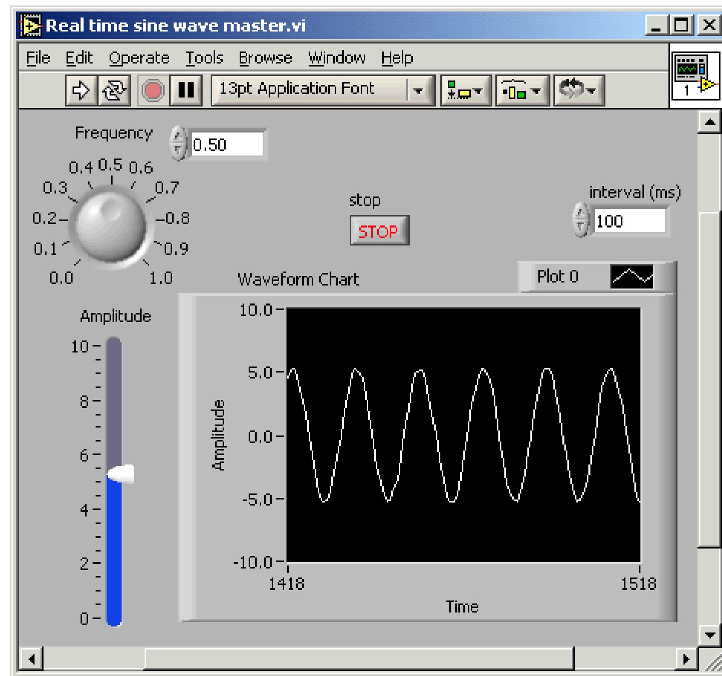
Run the demo “CPU-meter master.vi”. This uses 2 other VI’s to measure the CPU loading: it takes an average reading every 400 milliseconds. Move a window around the screen: you will see a dip in the graph because the processor is working hard to redraw the window many times in different positions as you move it. This means that there is less time for the CPU to do other things.

There are many jobs the CPU has to do. It has to monitor network traffic, the hard disc, the mouse, keyboard, and the clock on the computer. If it is also copying a file, for example, it will do lots more work in s short time. Most systems have activities going on the background that you are not normally aware of.

All this means that the CPU may be busy when it is next time for your program to run: this means your program has to wait until the operating system decides it is time for "your program's turn" to use the CPU. If your program needs to read from, or write to the hard disc, it may be put to sleep while the disc turns to reach the particular position of the data on the disc. Just a millisecond or two, but some other process could perhaps use the CPU while your application is waiting.

Construct this front panel and the diagram that follows. It is supplied for you, but the practice is essential at this stage. Even if you go ahead and run the demonstrations, take the time to come back later and construct the VI's for yourself. All the features are fundamental, so you will have to know where to access them.

- Including digital control/display with knob
- Creating and using shift registers
- Placing a sub-VI
- Altering the size of front panel controls
- Setting default values for controls
- Placing and colouring comments on diagram
- Improving the diagram layout
- Documenting VI's
- Formula node (used in sine generator to calculate sine(x))



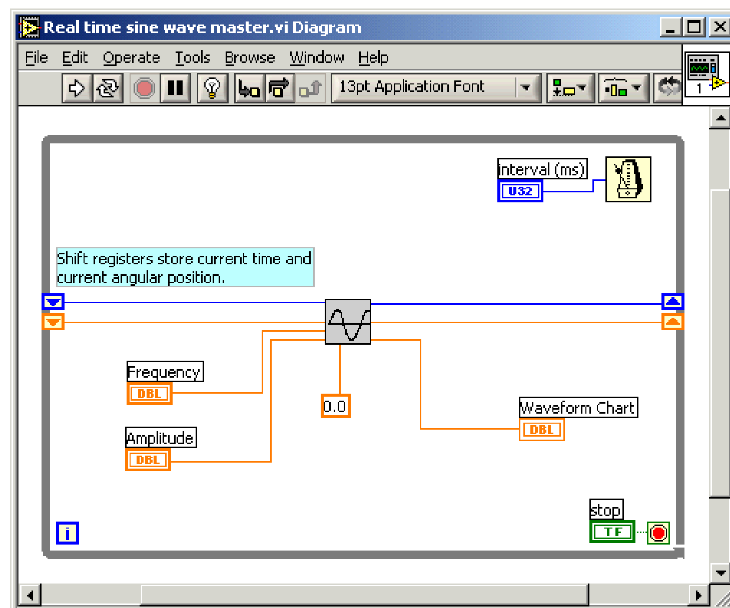
These are all skills you will learn with this example. They will be demonstrated in class.

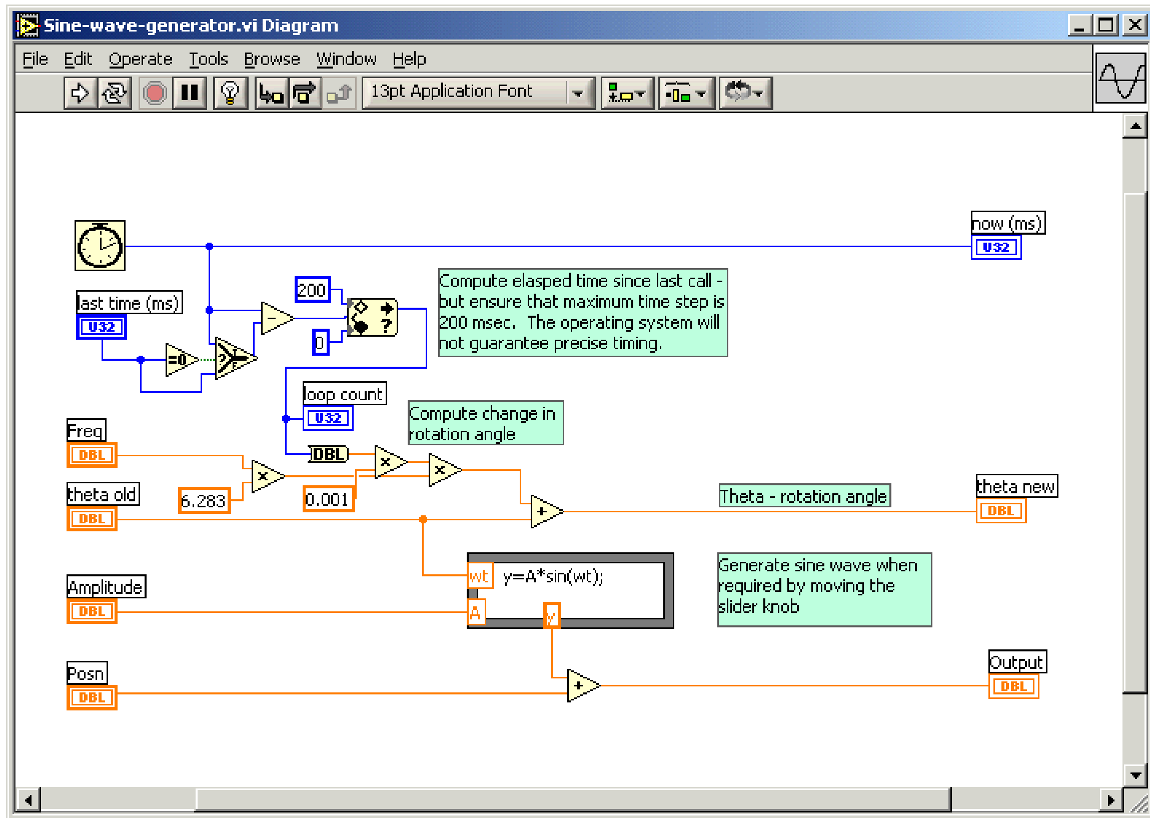
Run theVI. The timing of the sine wave on the display should be accurate.

Now open the sine wave sub-VI: double click on the icon in this window (using the arrow tool).

The next page shows the diagram. This has some special features that I will attempt to explain briefly here, but a fuller explanation will be given in class.

First the timing. The VI reads the current system clock and works out the time interval *since the last time the subVI was called*. This interval can vary as we shall see. It may be quite different from the interval specified by the input to the metronome symbol!





The resulting time interval is “clamped” to ensure the maximum value is less than 200 milliseconds. Sometimes there can be long delays and restricting the maximum value can be a useful trick to avoid some undesireable side effects. These can occur, for instance, if the program is stopped and restarted after some time. Also, the system clock (the millisecond timer) will be reset to zero at midnight each 24 hours: the zero clamp will be activated in this instance. Note that aother function provides a less precise timestamp value that includes the date and time, this is not reset each 24 hours, but cannot measure to millisecond accuracy.

The *thetai value* is advanced by the time interval multiplied by the frequency: here there is a potential error: the value of 2π is only an approximation! This can be likened to a motor driving a crank. When we change the frequency we alter the speed of the crank. The crank, in turn, generates the sine wave. Notice how the sine function is generated within a formula node: here you can insert arbitrary mathematical code that is more conveniently expressed symbolically rather than by using lots of icons.

> Note:
>
> While you are doing this, make sure the *CPU mater master* VI is running. If it is not, the results will not be as obvious.

Now, using the *probe* tool, set a probe on the output from the subtract operator that calculates the time since the last call of “sine-wave-generator.vi”. Run “Real time sine wave master.vi” and watch the probe value. Try different settings for the*interval* control. You may be surprised as the actual time differences measured in milliseconds are often different to the expected values! You can set the *interval* control to a small value, but you will not necessarily get the frequency of execution that you request. This is a decision made by the operating system and you have to live with it…..or change the VI priority.

However, before we do that you need some basic practice.

Exercises

Some mathematical functions

Change the approximate value for 2π to a precise value. On the functions palette, numeric group, at the right hand side, find the trignometric group and the additional numeric constants group. Substitute a precise value for 2π and the sine(x) function in place of the formula node. Don't forget to add a multiply block for the amplitude.

LabView waveform generator

In *Real time sine wave master* substitute the LabView sine function function instead of the call to the *Sine-wave generator* subVI. Calculate the system time by adding the *interval* value to a shift register value each time the VI runs. Use this as an input to the sine function, multiplied by 2π times the frequency. Observe how this introduces two problems. When the frequency is changed, the waveform seems to do strange things. Also, the timing may run slow, especially if the interval value is small.

(Solution is *Simple Sine Wave Master.vi*)

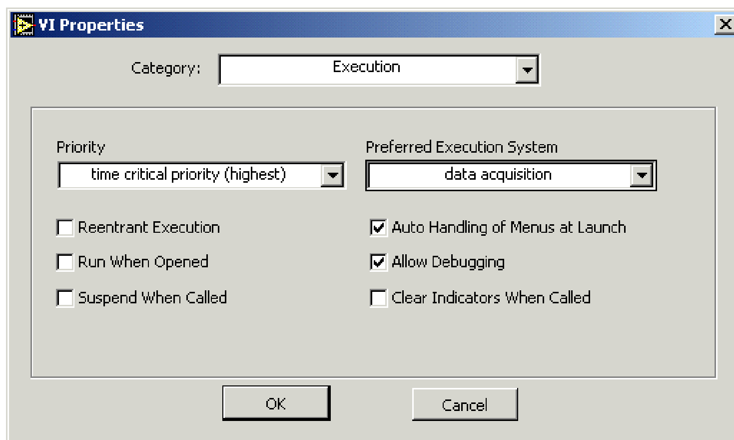
Warning

LabView has two families of functions for generating sine waves: the function generator VI and the waveform group of VI's. These are suitable for generating array of numbers representing a sine wave: they are more efficient than the code I have shown, but are not suitable for real-time operations.

The documentation for these functions can be misleading. Documentation is accurate (as far as I can tell) but I have seen bright students and an experienced engineer misled because they did not fully appreciate the details of the documentation, nor the intended function.

Getting the Priority Right

Return to the *Real time sine wave master* VI. In the "File" menu select "VI Properties". Choose "Execution" from the category menu at the top of the dialogue box. Set the priority to "time critical (highest) and preferred execution system to "data acquisition".



Now re-run the VI with the probe on the output of the sine wave sub-VI subtract function to show the actual interval between successive execution instants. Take the *interval* control down to 20 milliseconds or less and observe the result. Now the VI is running at a much higher priority and the operating system takes

much more notice of the execution time settings! Even though the *CPU mater master* VI is still running in the background, *Real time sine wave master* is getting the priority it needs.

Once you have read this you will see how *CPU meter master* works. It uses two almost identical VI's to measure CPU activity. The first is a high priority one that is only run once, to measure the number of loops the CPU can execute in 400 milliseconds. The second is a very low priority copy that will measure the number of loops the CPU has time left over to run in a 400 millisecond interval. Comparing the two results reveals the relative load on the CPU. Note that the MHz reading on the front panel may be wildly incorrect. I calibrated this on a Windows 98 system running on a 400 MHz laptop. However, I overlooked the fact that the timer function runs much slower in Windows 98 than Windows 2000. So on a Windows 200 system the MHz reading may be 20 times greater than the actual CPU rating. Can anyone correct this?

In the next set of exercises we will deal with inter-process communication: how to exchange data between multi-threaded applications.

Exchanging Data

In reality, multi-thread applications consist of different VI's, different processes.

For instance, a high speed process may be needed to control a fast-changing process, such as a servo-motor controlling a valve. A slower speed process may be needed to update alarms. Another slower speed process may update sensitivity factors that allow optimal control parameters to be adjusted on-line. Finally, there may be a user-interaction process for plant operators to inspect the state of sub-systems, shut down faulty units for maintenance, and so on.

Clearly the fast processes need high priority. These processes usually include data transfers between the plant and the control computer, but then again, only fast changing signals need to be read at high sampling rates.

The VI's *Multi-thread 01 Master* and *Multi-thread 02* provide a very simple example. The second runs at high priority and will probably interrupt the first one. It calculates a sine wave (like the previous example) and puts the results into a global variable. The master VI reads the global variable and displays the results for the operator.

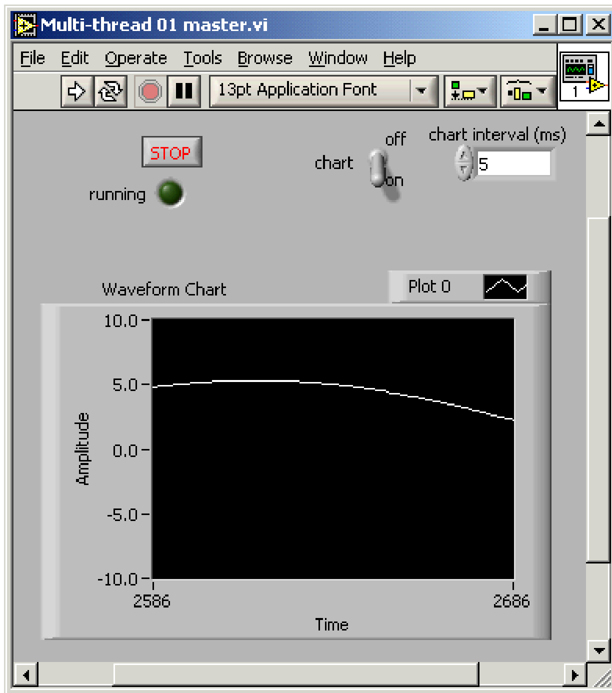
A number of simple features are used here, like global variables, but there are some hidden flaws.

Global variables are used to transfer data between the two VI's.

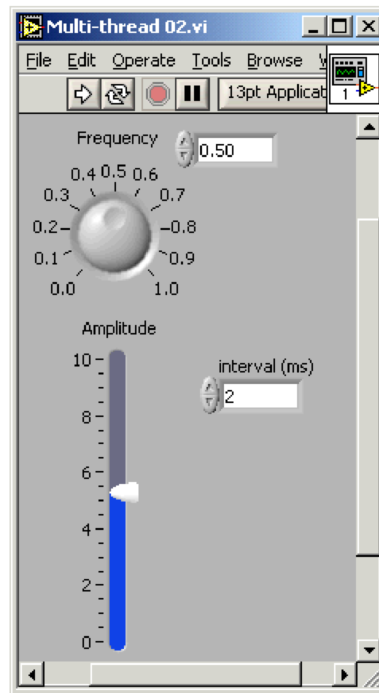
> Note: Constructing global variables is a little tricky, However, as we are about to see, there is little point in learning how to use them as there are much safer ways to exchange data.

Notice how *Multi-thread 01 Master* calls *Multi-thread 02*. Right click on the icon for *Multi-thread 02* in the master diagram. Select "Sub VI Node Setup". A dialogue box opens showing the settings that make the control panel for *Multi-thread 02* appear when the master VI is loaded.

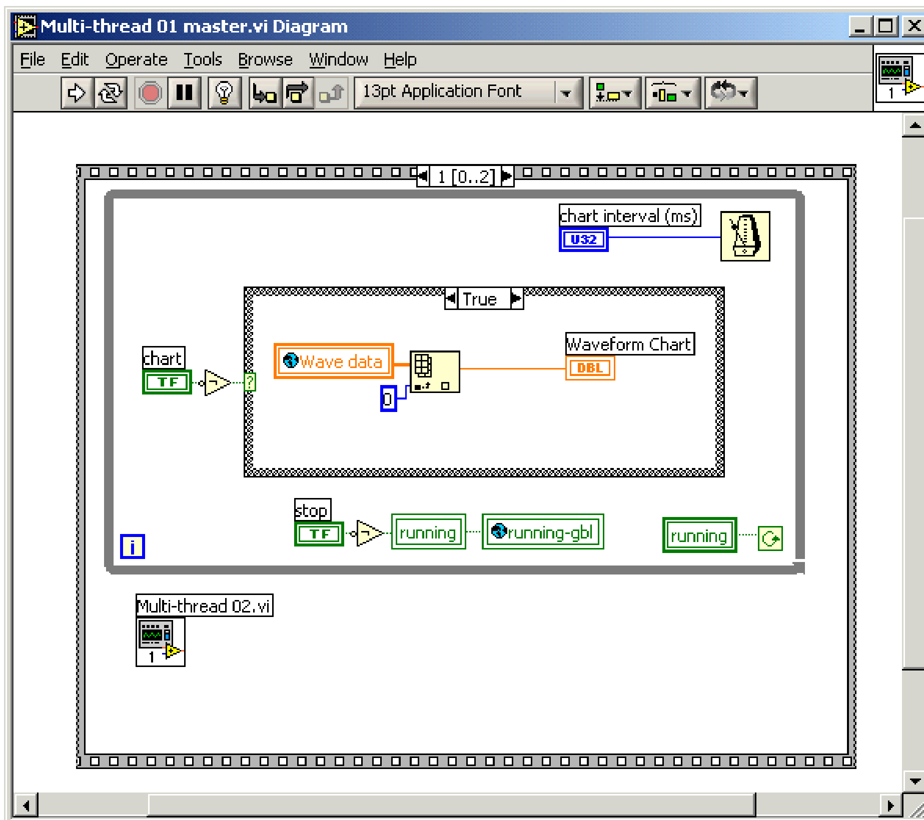
The single STOP button needs to stop all processes simultaneously. The STOP signal (a boolean) is copied to a local variable *running* and the global boolean value *running-gbl* Each while loop then uses this value to decide whether it should continue or not.



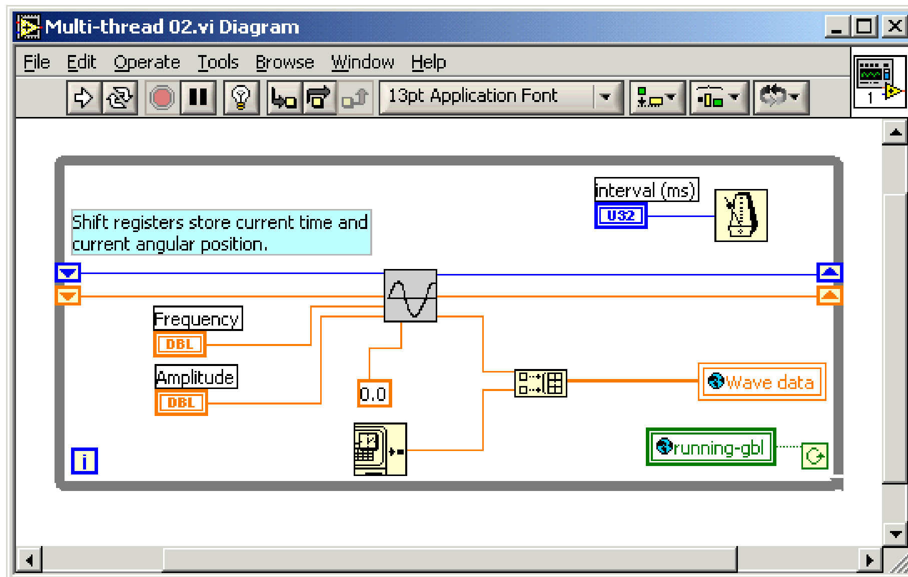
Multi-thread 01 master.vi
File Edit Operate Tools Browse Window Help
13pt Application Font
STOP
running
chart
off
on
chart interval (ms)
5
Waveform Chart
Plot 0
10.0
5.0
0.0
-5.0
-10.0
Amplitude
2586
2686
Time



Multi-thread 02.vi
File Edit Operate Tools Browse
13pt Applicat
Frequency
0.50
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
Amplitude
10
8
6
4
2
0
interval (ms)
2



Multi-thread 01 master.vi Diagram
File Edit Operate Tools Browse Window Help
13pt Application Font
1 [0..2]
chart interval (ms)
U32
True
chart
TF
Wave data
0
Waveform Chart
DBL
stop
TF
running
running-gbl
running
Multi-thread 02.vi



The Flaws

The first flaw is the use of the *Build Array* function in LabView. In *Multi-thread 02* it is used repeatedly to construct an array in which to put the calculated results before these are copied to the global variable. This can cause the system memory to become highly fragmented and must be avoided. This will be explained in class. If you do use arrays, make sure they are constructed to be large enough as early as possible, and once created, are not destroyed. It is difficult to demonstrate this in practice. The real problems occur with more complex examples.

The second, and much more important flaw, is the use of the global variable by both VI's. There is nothing to prevent one VI from interrupting the other VI in the middle of writing (or reading) the global variable data. Once again, I could not actually replicate this behaviour in my simple demonstration, but I have encountered the problem with a slightly more complex LabView program recently.

This is an issue at the heart of *concurrent computing*. To fully explain all the issues takes a full semester course in Computer Science. However, the basic issue can be readily appreciated here.

Both *Multi-thread* VI's have *critical sections*: these are the pieces of code that need the same resources, i.e. the global variables in this instance. The traditional way of handling this problem is to request a *semaphore* from the operating system. LabView has semaphores (see the help index). However, LabView also has a much neater 'unofficial' way to overcome this problem.

This is a technique used by practioners, but not mentioned in any of the official LabView books as far as I know.

The Uninitialized Shift Register (USR)

The principle is simple.

As the application note on multi-threaded applications reveals, VI's can be configured to be *reentrant* or *non-reentrant*. These terms are very important to understand. Reentrant modules are written such that they can be called while in the middle of another call. By default, C++ and Java modules are reentrant: they can even call themselves. However, all data created by the module is local: normally all inputs and outputs have to be in the form of calling parameters. Using global variables in this context is fraught with difficulties.

Non-reentrant modules are very useful because they retain their local data between one call and the next. This is usually very useful in machine control (mechatronics) applications. However, they cannot call themselves, and must not be called by other (interrupting) threads.

LabView has an in-built mechanisms to prevent this from happening. A non-reentrant VI cannot be called while being called from somewhere else. This means it can be used to store local data for safe exchange between VI's in a multi-threaded application.

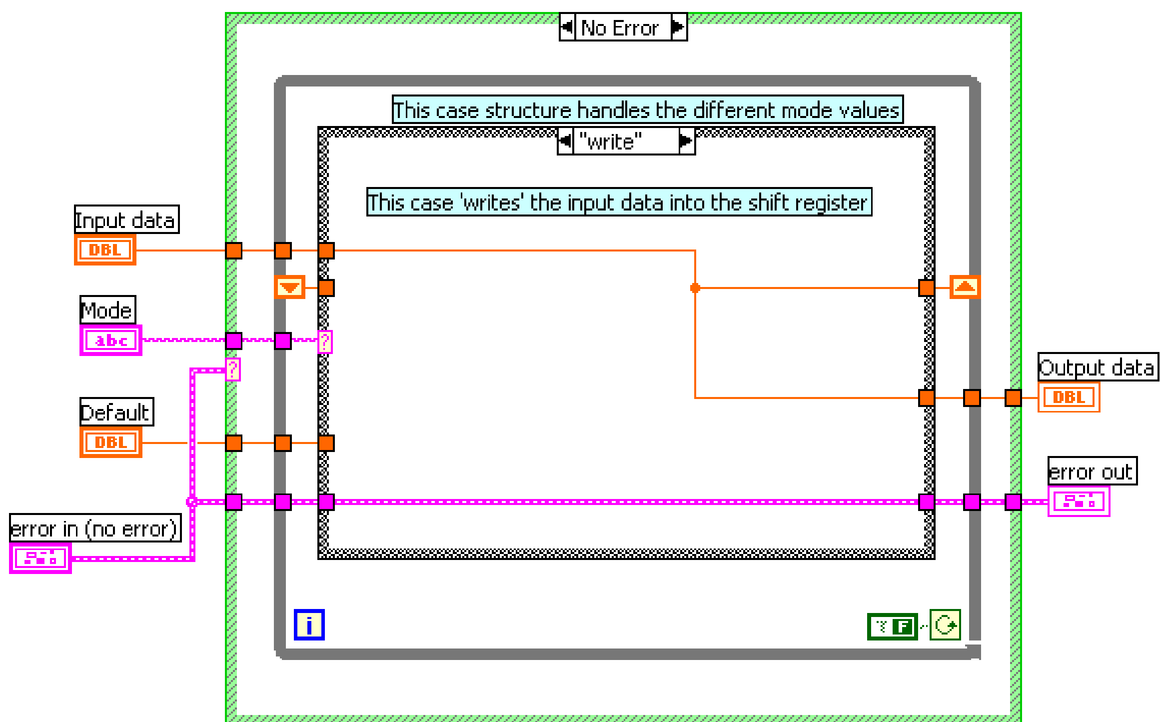
The basic structure is simple. It consists of a while loop that executes once each time the VI is called. A case structure decides whether to:

a) store the input data in the shift register ('write')

b) copy the shift register to the output ('read')

c) initialise the shift register to a default value

The diagram shown below only shows one of each of the cases: explore the source by yourself.



However, it must also be set to be a non-reentrant VI. This is controlled with the same properties dialogue that we used to set the priority. Note how VI's are non-reentrant by default.

The example includes the standard way of handling errors in LabView. If the input error signal already indicates an error (and notice how this can be wired directly to the outer case structure), the VI does nothing.

Exercise

Change the *Multi-thread 01 Master* and *Multi-thread 02* VI's to use this simple USR module to pass data between them. Note that only one 'double' can be transferred: take out the timestamp for the time being.

Using Type-Def's

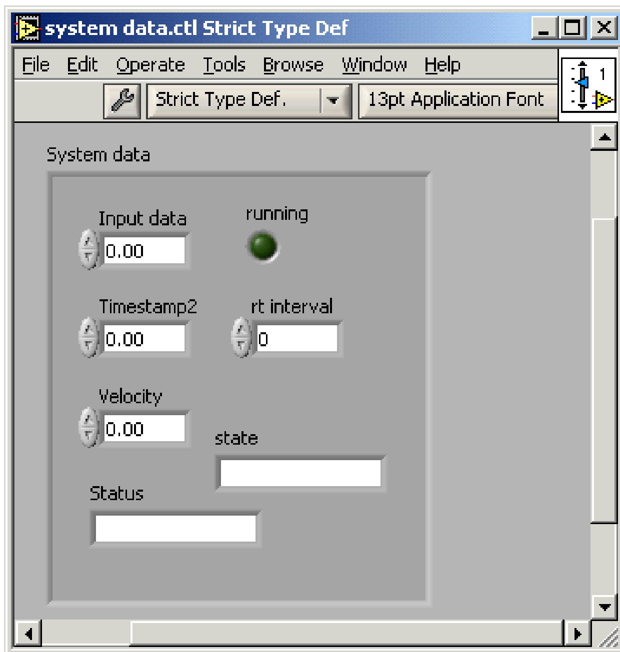
The last stage of this introduction is to show you how to use *type-def's* to handle more communicate several values at one time. Of course, you could substitute an array for the single double precision value in *USR demo*. However, all the elements of an array have to be the same type.

In C, C++ and Java you can construct data types consisting of several elements, each of a different type. In LabView these are called *clusters*. The most common cluster is the error cluster that consists of a string (the name of the function reporting the error), an integer (the error code) and a boolean that indicates whether an error is present.

Make a copy of *USR Demo* and save it as *USR Struct*.

Now, with the panel open, select the *input data* control. Now in the *Edit* menu select "Customize Control". A new window opens with just the control in it.

Place an empty cluster box on the panel, and enlarge the box to make plenty of space. Now put some other controls there, as the following picture shows. The 'control' menu alongside the spanner has been used to select 'strict type def'. It has then been saved as *system data* but LabView makes it a special *.ctl* module.



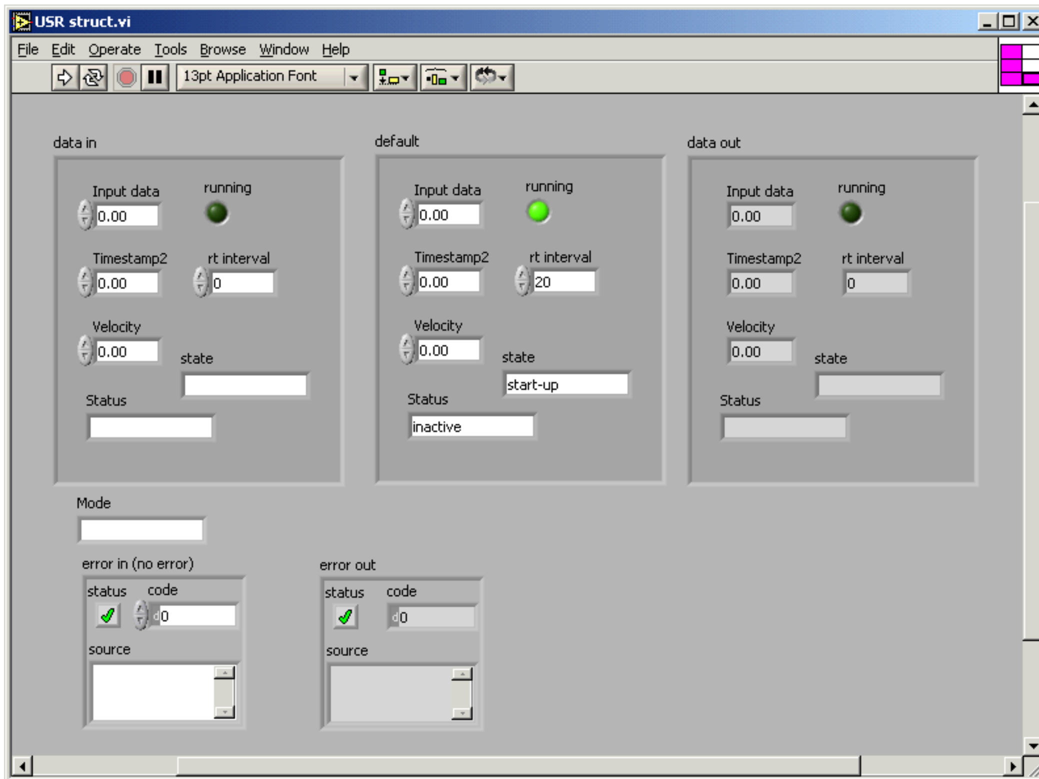
Now something strange has happened to the original *USR struct*. The simple double has been replaced by the new control.

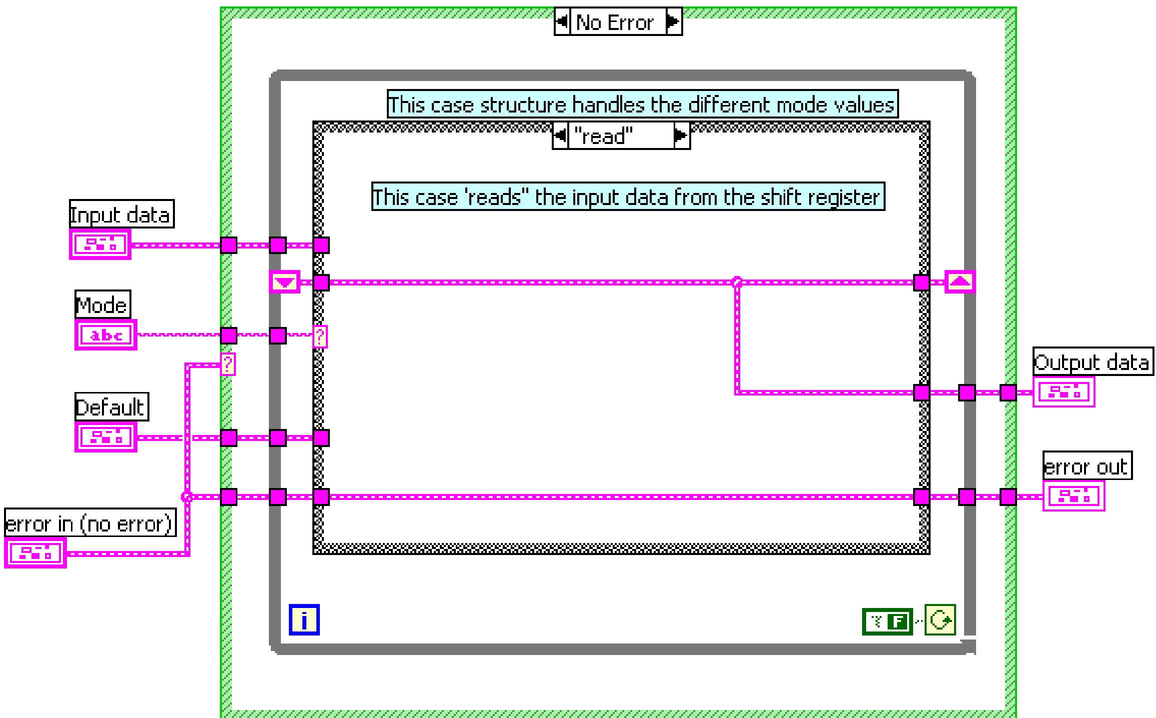
This is actually a powerful feature of LabView.

The next page shows how the front panel looks after tidying it up and replacing the *data out* indicator with a copy of the system data cluster changed to be an indicator. Also the default value is replaced with another instance of the same cluster.

The block diagram of the VI is identical except that the orange wires have become purple. Here the VI is storing many data values, but the wiring and structure is exactly the same!

The power of using a strict type-def is that you can change the original control (using the same procedure as you used to create it – select an instance of it and the in the edit menu select 'Customize Control'. However, all modules using the data will automatically be updated to include the additional data without any extra wiring or modifications!

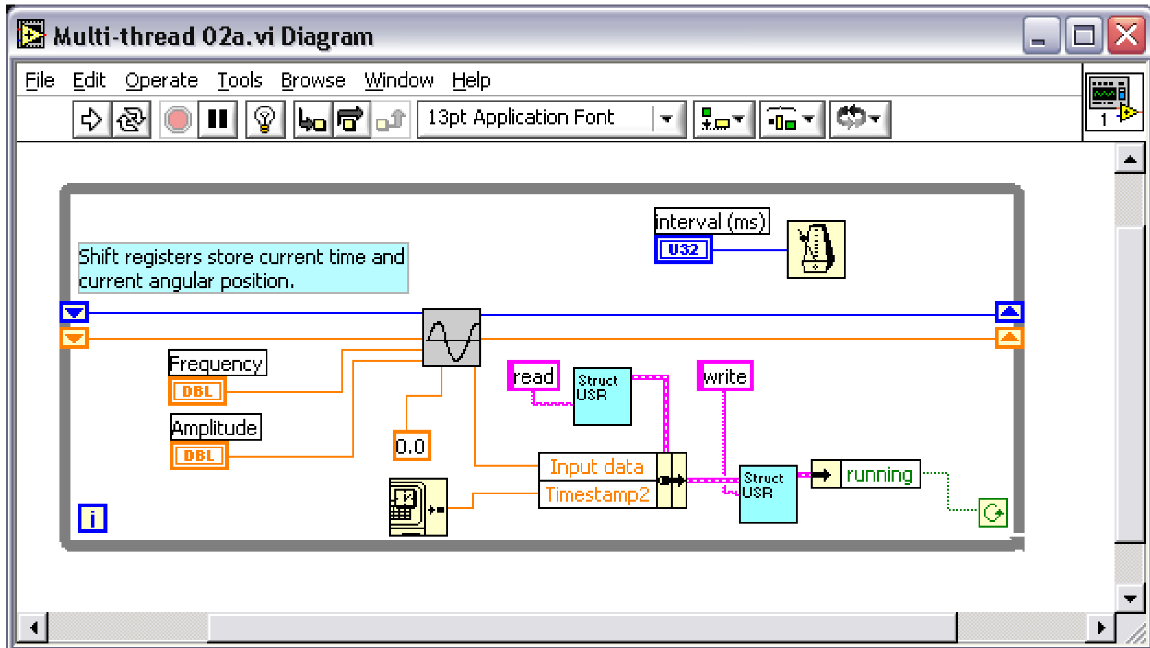




We are now near the end of this first set of exercises.

You can now modify (once again) the *Multi-thread 01 Master* and *Multi-thread 02* VI's to use the new *USR Struct* device to exchange data Now you can accommodate the timestamp data.

Writing the data requires that you first read the data cluster, then modify the elements you need to, and then write it. Here is a snippet of code needed to do that:



Note that the errors have been ignored. This is not a good idea: always wire the error inputs and outputs, but it saves a little complexity here.

Now you have enough background to read the application note on multi-threaded applications. Don't read it all just yet. A superficial reading is sufficient.