created 06 August 2013

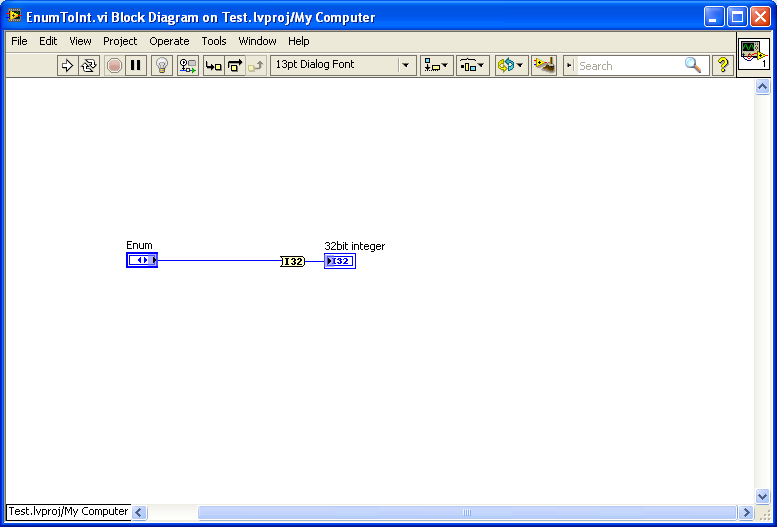
revised 14 September 2013

**Team 3061 LabVIEW Guide**

Right now, LabVIEW makes no sense. After reading this, you’ll be right where you should be (i.e. LabVIEW still makes no sense).

**Basic LabVIEW**

LabVIEW is... unique, to say the least. When you hear “programming,” you probably think of nice, neat lines of code, executing one after another. LabVIEW? Too good for that. Instead, LabVIEW uses a concept called **dataflow** programming, where pieces of code send data to each other through wires. A section of code only executes once it has received all of its inputs (i.e. wires)—no matter where that section of code is physically located. More on this later.



This wire carries data from the function.

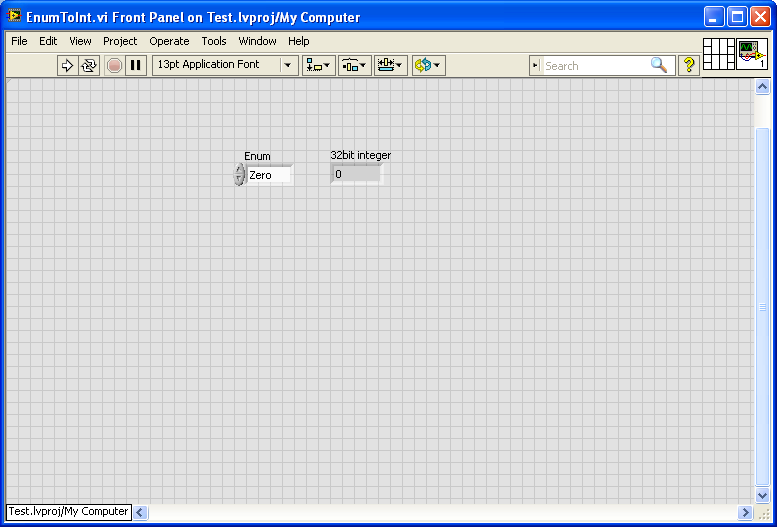
This wire carries data to the function.

This function only executes once it gets the data from the input wire. Whether that’s now, in 5 minutes, or in 5 years.

**Interface**

Here’s a rundown of what you need to know to make LabVIEW work. Well kind of, anyway.

Every program (from here on out called a “VI”) has two windows—a front panel (user interface, for buttons and flashy things) and a block diagram (actual code). Each window has buttons/menus to help you write your VI:



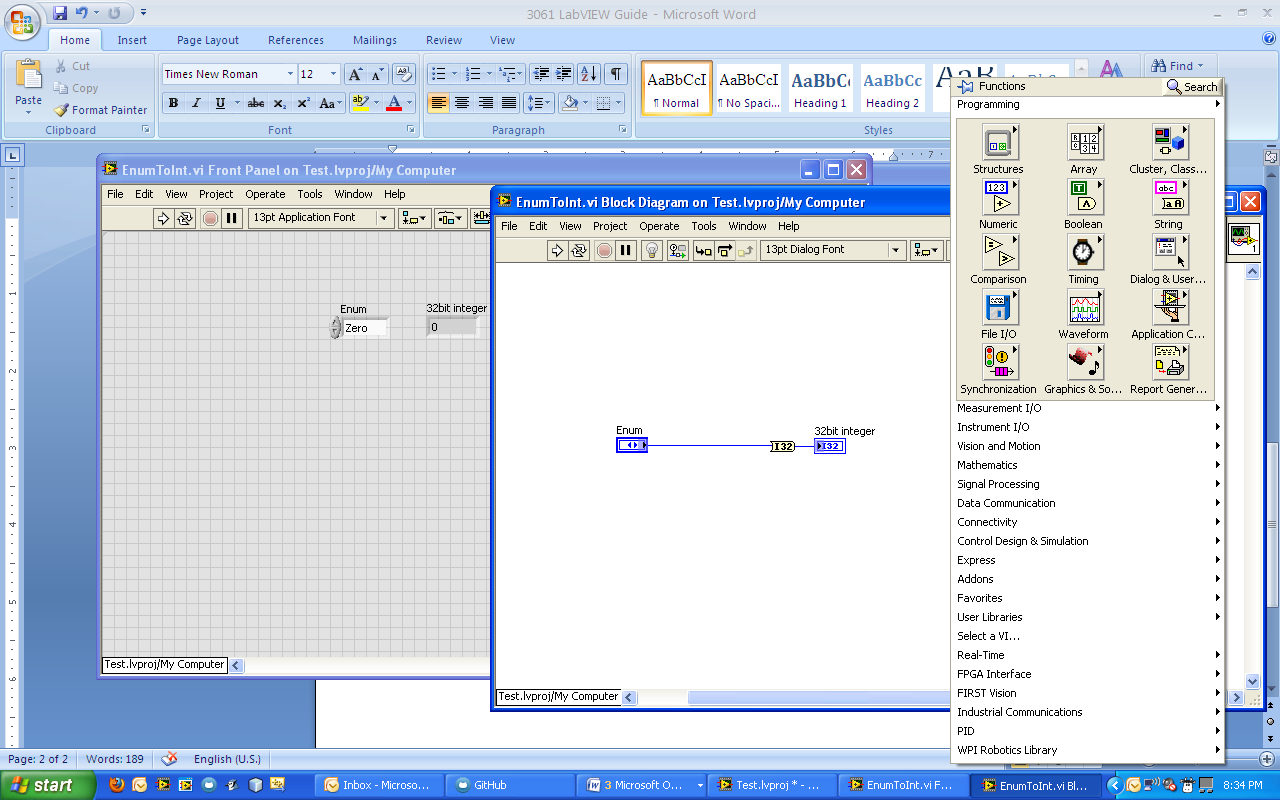
FRONT PANEL

4

2

3

1



BLOCK DIAGRAM

5

6

1. Run. Please don’t ask me to explain.

2. Stop. Get used to grumpily pressing this button. A lot.

3. Connector pane. This one shows the inputs and outputs for the VI (see “Making a subVI” for details).

4. Icon. Displayed when this VI is used inside another VI (see “Making a subVI”).

5. Execution highlighting. If this is pressed while the VI is running, LabVIEW will slow down execution and show, wire by wire, what data is being passed through the VI. Useful for debugging.

6. Menu. Right-clicking anywhere in the block diagram or front panel (I was just too lazy to show it in the front panel) will bring up a list of objects and functions that can be used in the VI.

Here are some must-have keyboard shortcuts:

* Ctrl-H: brings up the context help. Just hover over any object/function and the help will tell you what it does. I use this somewhere between all the time and constantly.
* Ctrl-E: switches between the block diagram and the front panel.
* Ctrl-B: deletes all broken wires (i.e. when you draw a wire and mess up).
* Ctrl-Shift-S: saves EVERYTHING. Spam this. Please.
* Ctrl-Space: lets you search for a specific function/VI. So if you know a VI exists but you just can’t find it, use this.

Also, going to Help >> Find Examples is a really useful resource if you want to see an example of how to use a particular feature of LabVIEW.

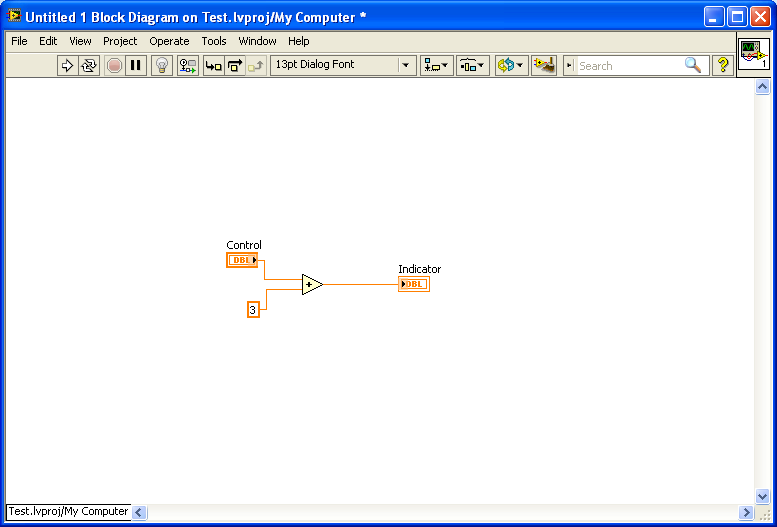
Each VI (.vi extension) is usually contained in a project (.lvproj file), which is just a collection of related VIs.

**Data**

**Controls, Indicators, and Constants**

In LabVIEW, pretty much every kind of data (numbers, strings, etc.) comes in three varieties: controls, indicators, and constants. Controls work as inputs (i.e. they send data to other code), while indicators work as outputs (they receive data from other code). Constants have a value that never changes, and like controls they send data to other code.

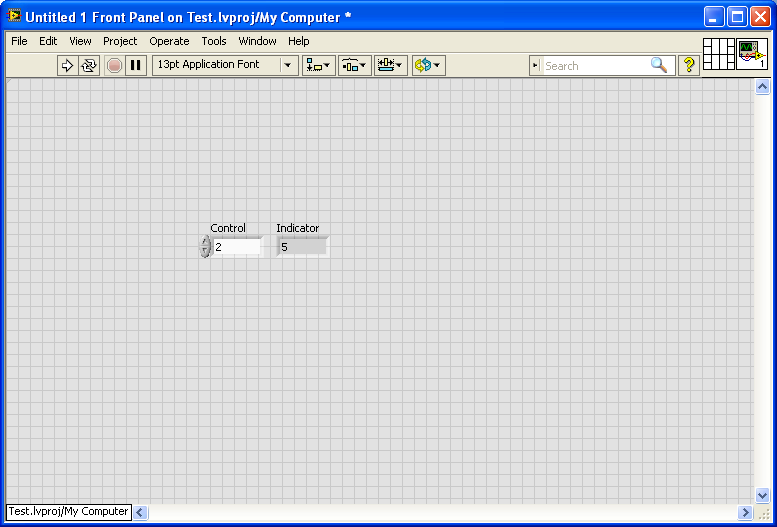
Both controls and indicators can be created from the front panel (through the menu). Constants can be created from the block diagram (again through the menu). On the block diagram, any of these objects (constant/control/indicator) can be right-clicked and converted to a different “variety” (i.e. an indicator can be changed to a control or a constant). Also—this is a good time-saver—if you right click on a function’s input or output terminal and mouse over “Create”, LabVIEW will give you the option of creating a constant, control, or indicator of the correct type (i.e. number, string, cluster etc.).



Both the control and the constant send data to the sum function.

The indicator receives the sum.

For blind people like me… that’s a “+”



Apparently, 2+3=5.

Thanks, LabVIEW.

**Data types**

LabVIEW uses a lot of different data types, and every function expects a specific one.

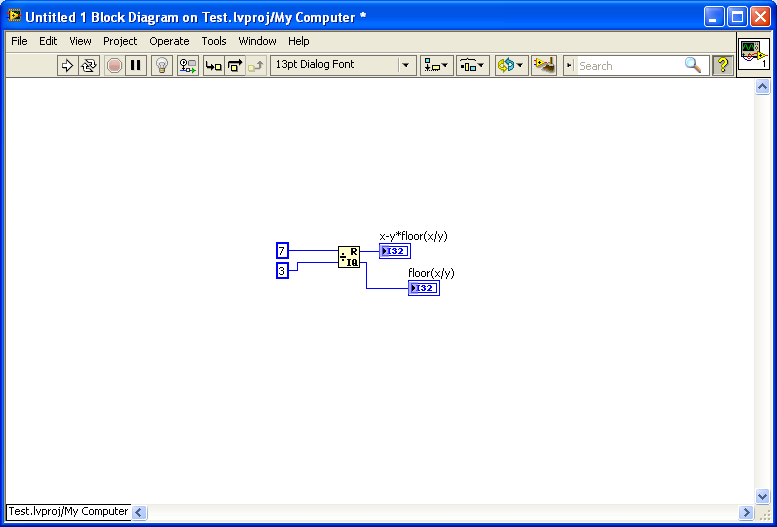
* Numbers: just to make life easy, LabVIEW has somewhere over a dozen different types of numbers—unsigned (i.e. only positive) integers, signed integers, floating point numbers (like 4.3453), and imaginary numbers. But for most purposes (and sanity), you only need the (blue) “numeric constant” (which is an integer) and the (orange) “DBL numeric constant” (which is a floating point). These can be found in the block diagram’s menu under “Numeric”. (Also, the numeric control/indicator found in the front panel menu defaults to floating point.)
* Boolean: true or false. These are often used with case structures and while loops (see “Structures”).
* String: text. Honestly, we don’t use this too much. For reasons to be explained very soon.
* Enum: a list of numbers paired with strings—basically, it’s easy to read for both computers and humans. So, if a particular action as several different states (like “go,” “wait,” or “stop”), it’s a lot easier for humans to choose from a list of strings than trying to remember what number goes with what state. Also, if some function requires a string, it’s convenient to use a enum to ensure that someone doesn’t accidentally mistype the string. (Believe me, I have firsthand experience with that.) Like all of the other data types, enums can be found in the menu on either the front panel or the block diagram (in the block diagram it’s under “numeric”). To edit the enum, you right-click, go to “properties”, tab over to “edit items”, then start typing in whatever you want.
* Array: a collection of different values of the same type (i.e. one array could have 5 integers, while another might have 3 strings). To create an array, grab an empty control/indicator/constant from the appropriate menu. Then drag an object of the type you want (number, string, etc.) into the array. The array should then switch to that type.
* Cluster: a collection of… whatever you want. You can create a cluster the same way you do an array; just drag any object you like into it. It could be a combination of numbers, booleans, arrays… seriously, whatever you want.

**Functions**

Here is some of the more useful LabVIEW magic (found on the block diagram menu). Use the context help (ctrl-H) for *lots* more detail. Keep in mind that there are way more functions than there are listed here.

Numeric:

* Add, subtract, multiply, divide, reciprocal… you get the idea.
* Quotient & Remainder: basically a “mod” (remainder) operator. This takes one number divided by another and gives you two results: the “quotient” is the result of the division with the decimal part chopped off, and the “remainder” is the mod part.



The quotient. In this case, 2. (7/3=2.33333)

The remainder. In this case, 1.

Boolean:

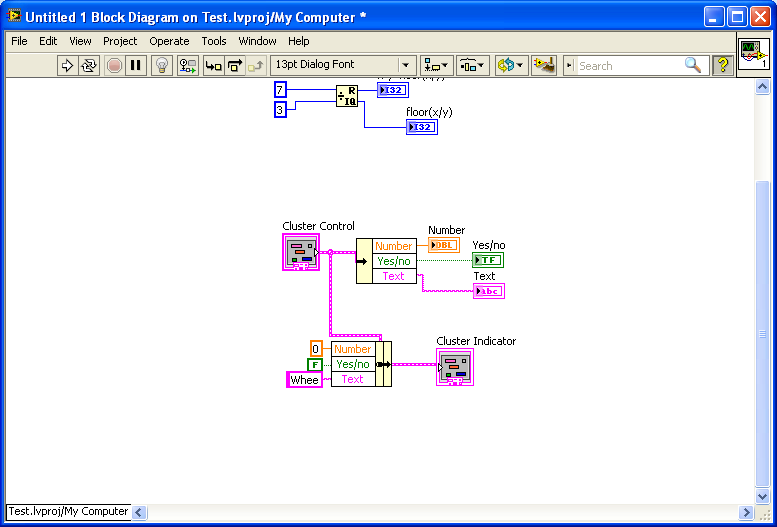
* AND, OR, NOT: these evaluate expressions involving booleans. AND returns true if both input booleans are true. OR returns true if at least one of the input booleans is true. Not returns the opposite value (i.e. NOT true is false).
* Comparisons: greater than, less than, equal to, etc.

Array:

* Array size: returns the number of elements in the array.
* Index array: returns the element at the index you specify. (Array indexes always start at 0, so the first element is at index 0, the second element is at index 1, and the nth element is at index n-1).
* Insert into array: insert the element you specify into an array at the index you specify.

Cluster:

* Unbundle by name/bundle by name: these are used to access or modify elements of a cluster (unbundle accesses the elements, bundle modifies them). When you create an array, you can give each element a unique name, and that name appears when you use this function. Really helps with readability.



Bundle

Totally meaningful labels

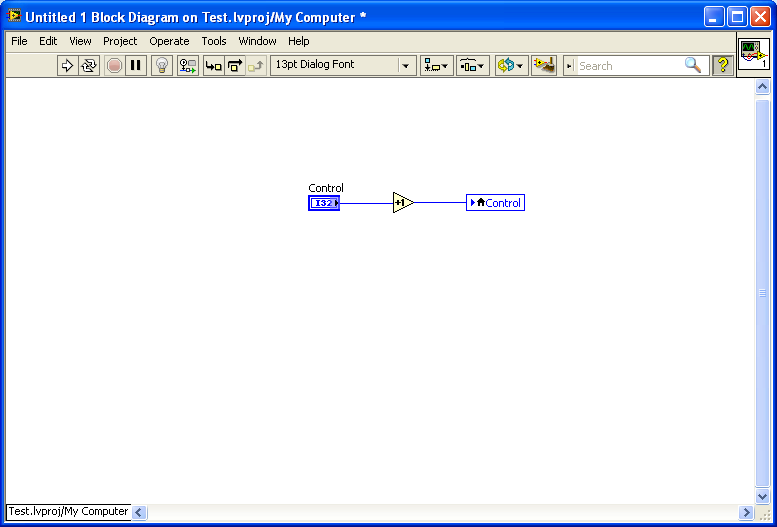
Unbundle

**Variables**

Variables let you make a copy of a value contained in a control or indicator. **Local** variables are used within the same VI as the original control/indicator, while **global** variables store the data for use in multiple VIs (so if I really wanted to access the same number from 5 different VIs, I’d use a global variable). Also, each variable is one of two types: write (i.e. you can give the variable a new value) or read (you can get the current value from the variable).

You can make a local variable by right-clicking on a control or indicator and selecting “create >> local variable”. To convert a variable to write or read, right-click on the variable and select “change to write” or “change to read”.

Then 9 will be stored in the local variable here…



So if you run this repeatedly, the value in the original “Control” will go up by 1each time. This is one way of writing to controls (which usually can only be read).

And then the value in the original “Control” gets updated to 9 as well…

Increment by 1

If “Control” has a value of 8…

To make a global variable, go to the project window (ctrl+shift+E). Go to “File >> New…”, expand “Other Files” in the dialog box, and select “global variable”. This will create a special VI that has only a front panel, and any objects added to that front panel become global variables. Each global variable can be accessed from any VI in this project.

**Structures**

**Case Structure**

A case structure is a section of code that has several different bits (“cases”) of code in the same place; however, at any given time, only one of those cases executes. So basically, the case structure allows you to choose between a couple of different possibilities, depending on current conditions. It’s pretty much equivalent to an if/else statement in a ~~sane~~ text-based language.

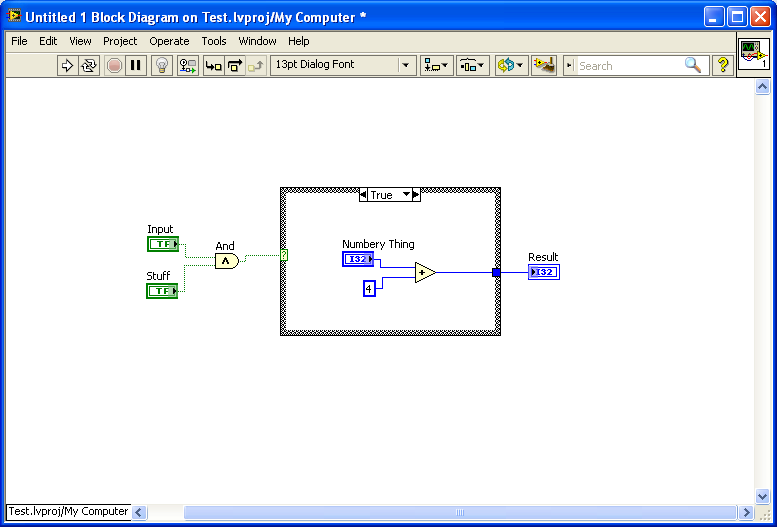
The case that executes is determined by what you pass to the case selector (the terminal on the left side of the case structure). Often, you pass a boolean—if the boolean is true, one case executes, and if it’s false, a different case executes. It’s also common to use an enum—every integer/string pair in the enum becomes a case (we practically lived off this last year).

Like most structures, the case structure can be found in the right-click menu on the block diagram.

If the boolean wired to the case selector is true, the case structure outputs “numbery thing” + 4.

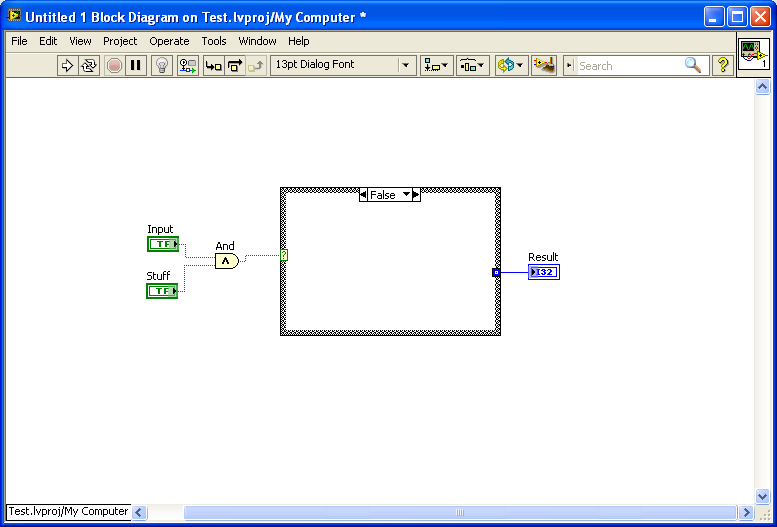
Also… if one case has an output like this, then all cases must have one. For once, laziness isn’t rewarded.

The case that executes is determined by the result of this AND.



If the boolean is false… this happens.

You can right click this “tunnel” and set it to “use default if unwired” if you don’t actually want to wire anything. So laziness works, after all. (The default is 0).



**Shift registers**

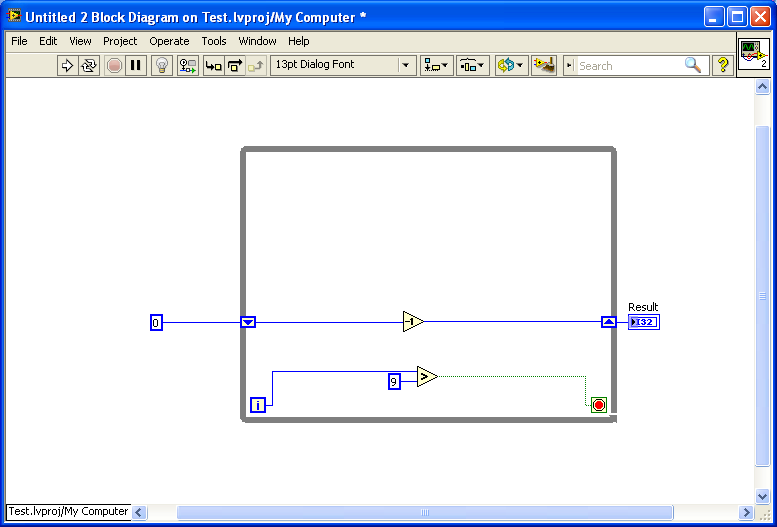
OK, these aren’t a structure, but the next two (for loops and while loops) use shift registers a lot. In a loop, you repeat a section of code over and over, and often it helps to have some data from the last repetition (i.e. iteration) in the current one. A shift register does exactly that; it takes some data from the n-1 iteration and gives it to the nth iteration.

To make a shift register, wire some data to the edge of the loop (it will create a tunnel). Right-click and select “replace with shift register”.

The result is stored in this shift register. Then, before the next iteration starts, the result is sent back to the beginning of the while loop.

Every iteration, the number is decremented by 1.

The shift register is initialized to 0.



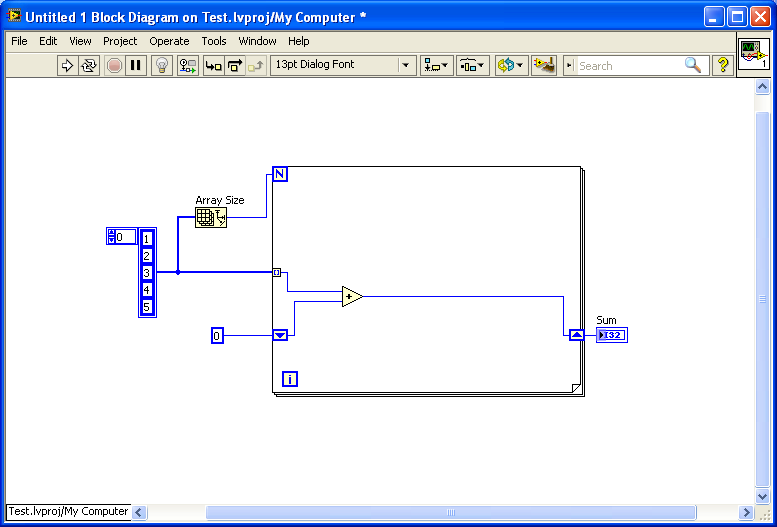
So for the second iteration, the number starts at

-1.

Total side note: this is the iteration counter (counts how many times the while loop has run).

**For loop**

A for loop repeats a section of code however many times you specify… that could be once, 10 times, or <some variable> \* 2 times. So, a for loop is best when you know how many times you want to repeat a bit of code.



…Meaning that this loop adds up all the numbers in the array. Sum = 15.

So each element is added to the value in the shift register and then stored back in the shift register…

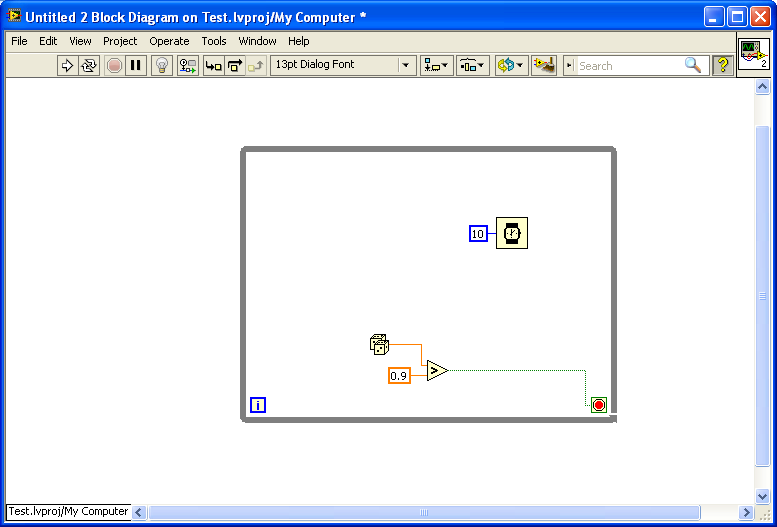
If you feed an array into a for loop, it will index it automatically (i.e. the first iteration it will grab the element at index 0, the second iteration it will get the element at index 1, etc.)…

This is where you tell the for loop how many times to run. In this case, it will run 5 times (the size of the array).

An array containing the numbers 1-5.

**While loop**

Unlike a for loop, a while loop continues as long as a condition is met… which means it could run once, or it could run forever. In LabVIEW, a boolean is connected to the conditional terminal, and the loop runs as long as the boolean is false (i.e. it will stop once the boolean becomes true).



It’s usually a good idea to include a small wait (found under “timing” in the block diagram menu) in while loops so the loop doesn’t hog your entire processor.

Everything continues like normal until the random number is greater than .9. At that point, the boolean will be true, and the while loop will stop.

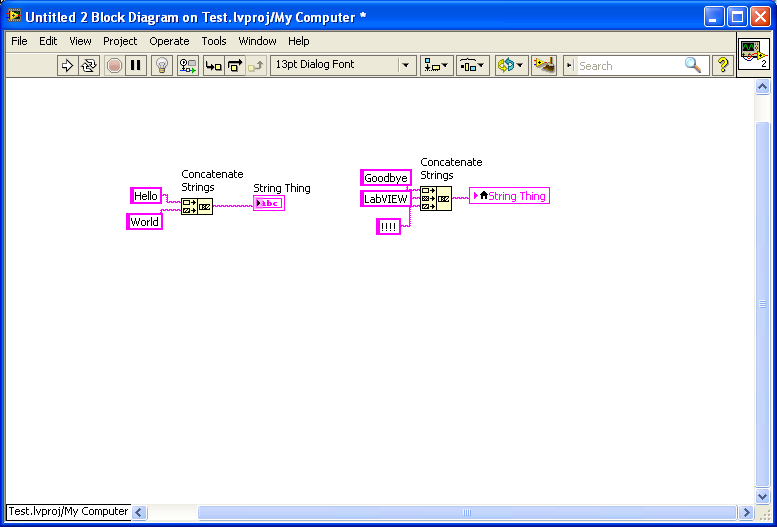
The random number function returns a number between 0 and 1 (found under “numeric” in the block diagram menu).

**Flat sequence structure**

A flat sequence structure is used when you want to force two or more pieces of code to execute in order. Each “frame” contains a section of code, and the first frame is guaranteed to execute before the second frame, the second frame is guaranteed to execute before the third frame, etc.

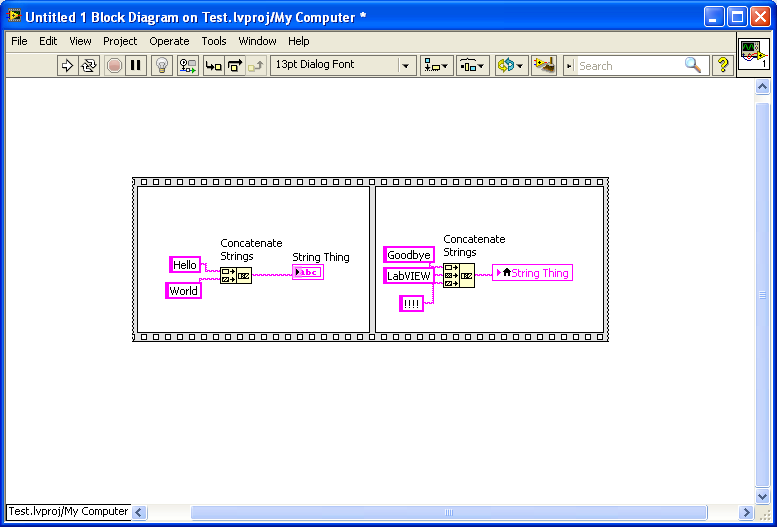
To add a frame to a flat sequence structure, right-click the structure and select “add frame after”. (Note: a stacked sequence is basically the same thing; the difference is that it doesn’t display all the frames at once like a flat sequence does.)

If you had two pieces of code like this, there is no way to know which will execute first. It might be the one on the left, or it might be the one on the right. (See “dataflow programming”.)



So “String Thing” might get the value “Hello World”, or it might get “Goodbye LabVIEW!!!!”. It’s unpredictable. And unpredictable code is… not good news.

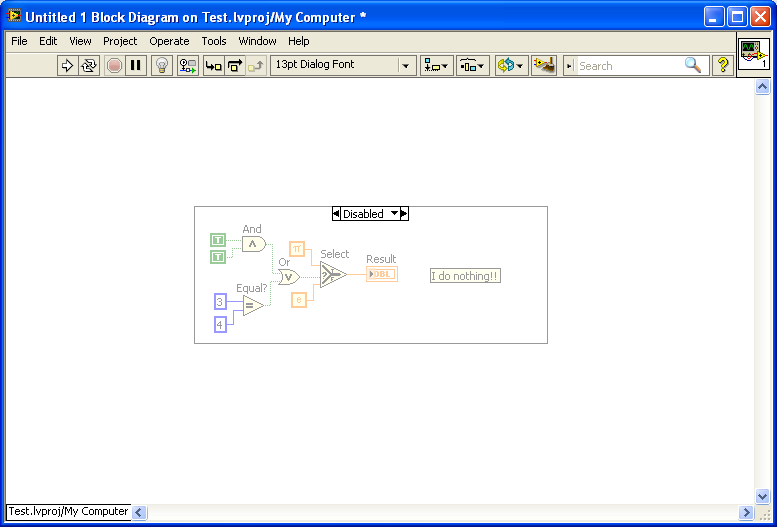
BUT if you used a flat sequence, the chunk on the left would be guaranteed to execute first. Which means that “String Thing” would get the value “Goodbye LabVIEW!!!!” every time, because that piece executes last.



Which also makes the chunk on the left totally useless. But oh well.

**Diagram disable structure**

A disable structure is LabVIEW’s version of “commenting out” code. If you throw a disable structure around a section of code, that code will not execute—it’ll be completely ignored by LabVIEW.



It was fun making it, though.

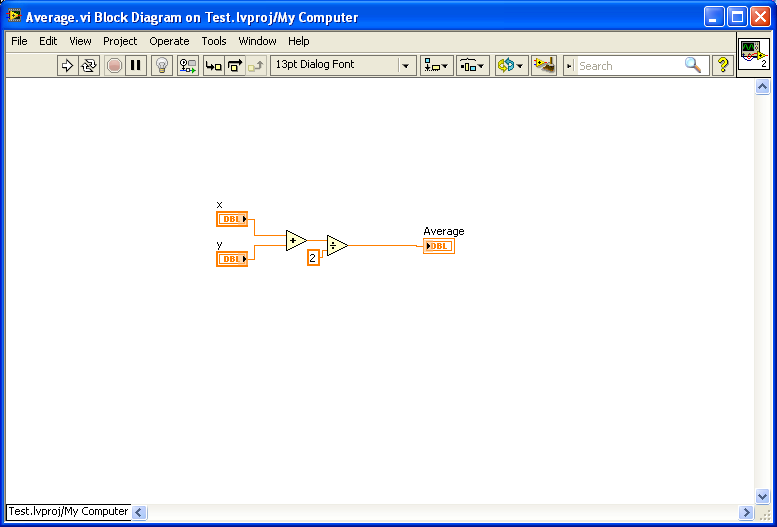
**Making a subVI**

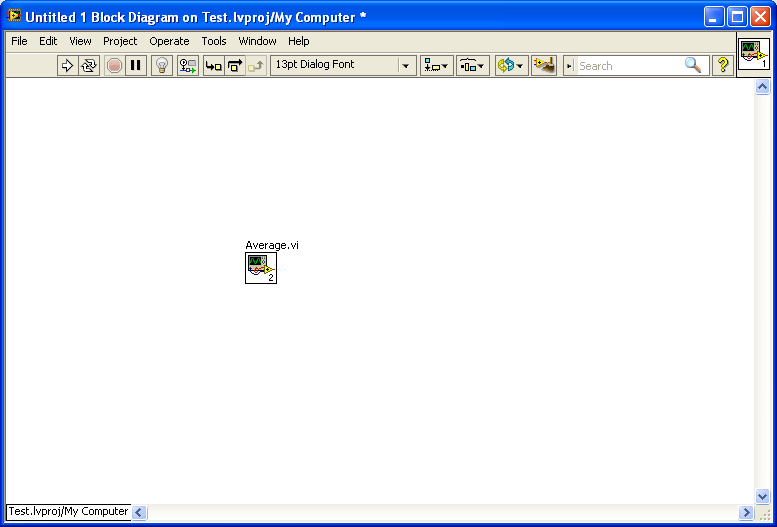
**Using subVIs**

So say you wrote a useful piece of code that you want to be able to use over and over again. Or maybe you want to break a large program into several smaller parts. This is where subVIs come in handy—they’re basically VIs used within other VIs.

To make a subVI, just save it as a normal VI. Then go to another VI, right-click in the block diagram, and click “select a VI”, which will bring up a dialog box where you can choose the VI you want to use as a subVI.

This VI…





As a subVI !

**Inputs and outputs**

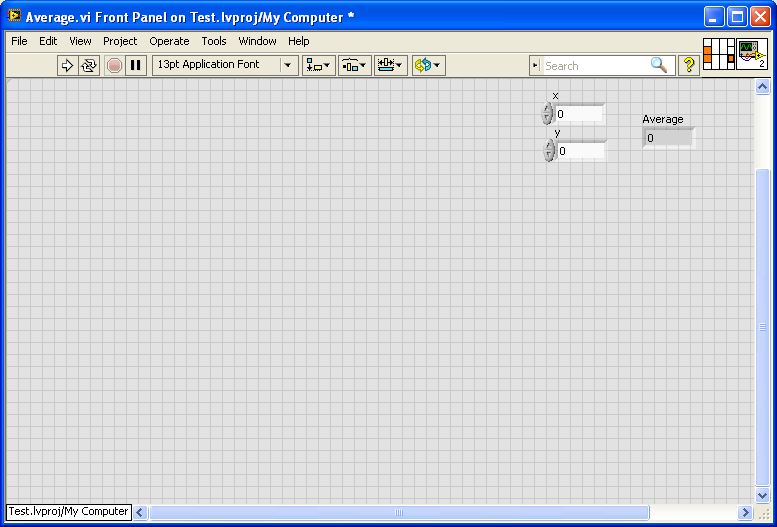
Most of the time, a subVI will have inputs and outputs—i.e. it will take in some data, process it, then return something useful (hopefully). To do this, open up your VI and go to the front panel. The connector pane is in the top right; click on one of the spaces, and then click on the control/indicator you would like to link that terminal to. Controls (i.e. inputs) should go on the left, while indicator s (i.e. outputs) should be on the right side.

It’s also a good idea to make some inputs required—this way you won’t accidentally forget to wire data to your subVI (because LabVIEW will complain very noisily). Just right-click one of the terminals you’re using, go to “this connection is”, and select “required”.

This terminal is “x”.

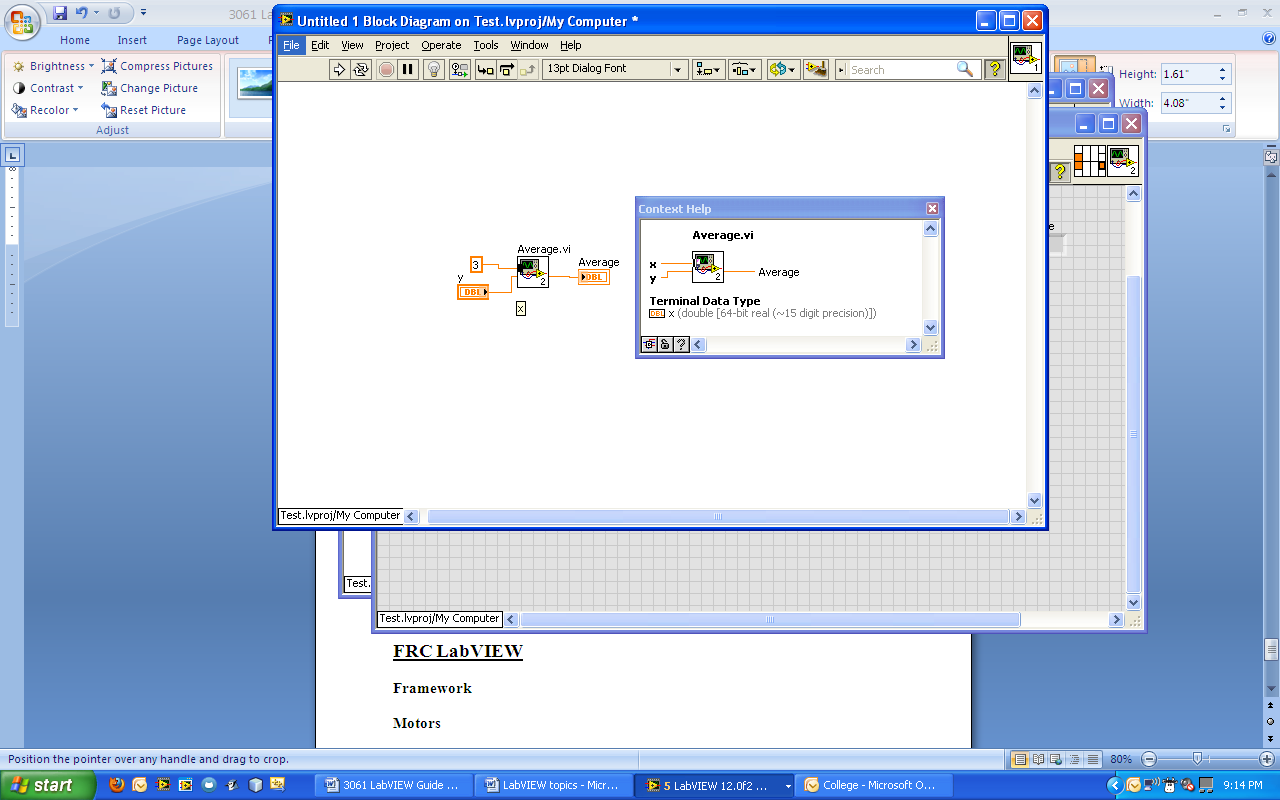
This is the output terminal (“average”).

This terminal is “y”.



Average, an indicator, is an output.

Since x and y are controls, they work as inputs.

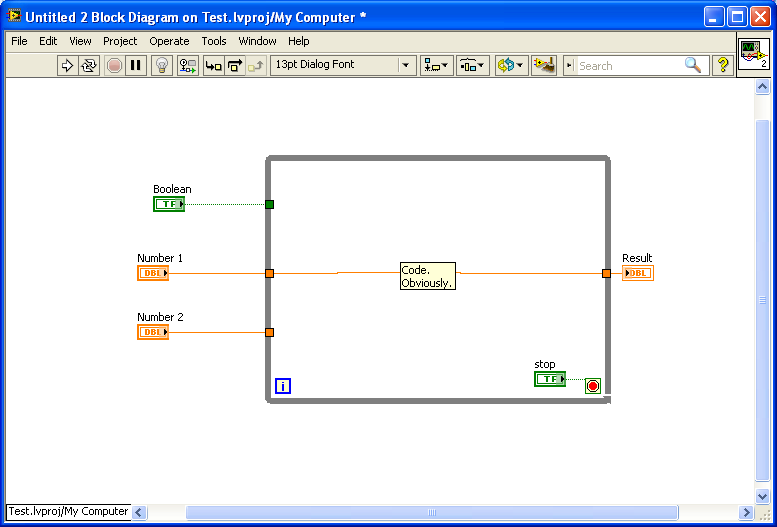


Shortcut: right-click a terminal and select “create control/ constant/ indicator” to make using a subVI easier.

The context help, Ctrl-H, will identify and describe the terminals.

**Dataflow programming**

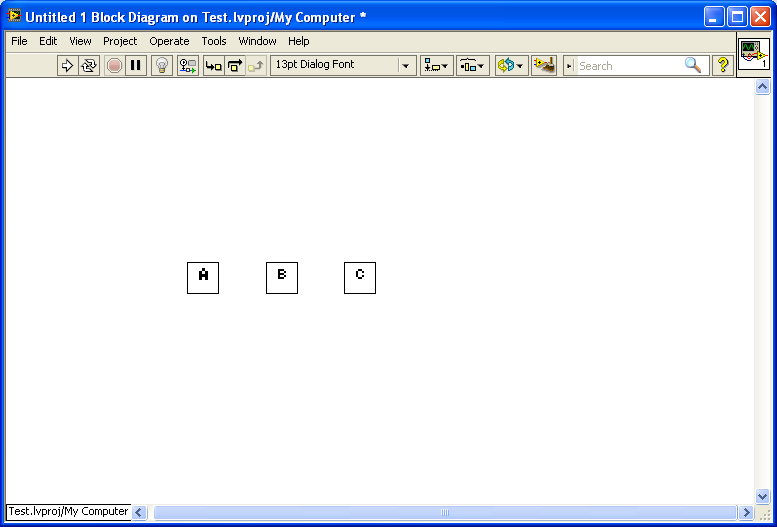
In LabVIEW, a piece of code only executes once it has received data from all the wires leading into it. So if you have a while loop and feed it two numbers and a boolean, the while loop will only start executing once it actually receives the two numbers and the boolean, and it would produce outputs only when the loop stops.



This stuff waits for the boolean and the two numbers to arrive, and returns a number when the entire loop finishes.

This can be really useful for, say, initializing values—specify a few inputs, and the loop or other piece of code will wait for those inputs.

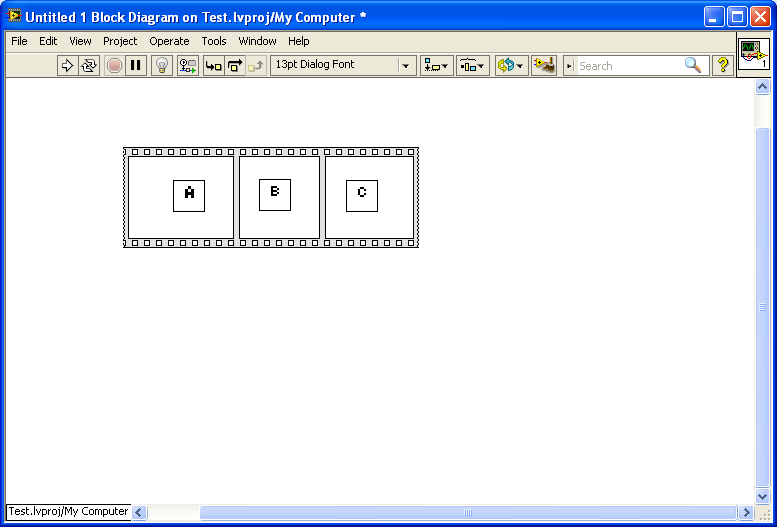
But it can also be frustrating if you want code to execute in a very specific order. Let’s say you have functions A, B, and C (so creative, I know), and want them to execute in alphabetical order. If you threw this down…



...they’ll execute in parallel with no guarantee of which one starts first—B has no reason to wait for A, and C has no reason to wait for B. So—sigh, LabVIEW—you have to give them a reason.

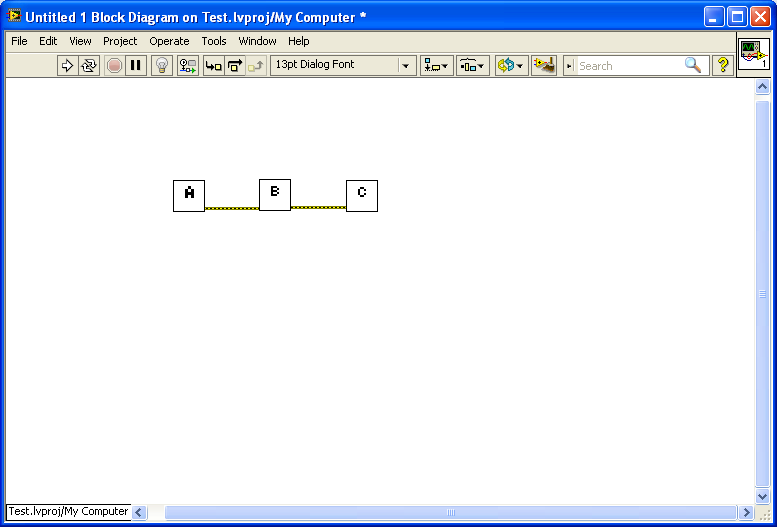
**Technique 1: flat sequence/stacked sequence**

You can use a flat sequence/stacked sequence to force the subVIs to execute in a certain order. See the entry under “structures” for details.



**Technique 2: error clusters**

LabVIEW provides a special “error cluster” that can provide useful information when (not if) something goes horribly wrong… and we can also hijack it to help with order of execution. It’s pretty common practice to write subVIs that take an error cluster as an input and return an error cluster as an output. So if you do this with A, B, and C, you can connect them with error clusters and force them to run in order.



This yellowish wire is an error cluster.

A only returns an error cluster when it finishes, so B has to wait for A. Then for the same reason, C has to wait for B. (Note: an error cluster constant can be found by right clicking in the block diagram and then going into the “dialog and user interface” submenu.)

**Debugging**

LabVIEW, thankfully, provides several tools to help you figure out what the problem is ~~if something goes wrong~~ whenever something blows up in your face.

* Probes: a probe will tell you, in real time, what data a wire is carrying. So if you probe a numeric wire and start running code, the probe will tell you what number that wire is carrying at all times during execution. To set a probe, right-click on a wire and (wait for it) select “probe”.
* Execution highlighting: clicking the light bulb button in the top-right of the window will, once you start running code, slow down execution to a human-comprehensible speed. LabVIEW will display what values each wire is carrying as the code runs, all at a very slow pace (unlike probes, which are real time).
* Breakpoints: inserting a breakpoint will cause execution to pause as soon as the VI reaches the breakpoint. This will give you a chance to step through code/look at the front panel/desperately try to figure out what’s going on. You can set a breakpoint by right-clicking on almost anything, selecting “breakpoint >> set breakpoint”.
* Front panel: you can set up the front panel to display pretty much anything related to the VI—from fancy charts to a simple LED boolean.

**Warning:** **if using a cRIO, you can only use the above techniques when you are running code deployed by clicking “run” in Robot Main.vi (i.e. NOT startup code).**

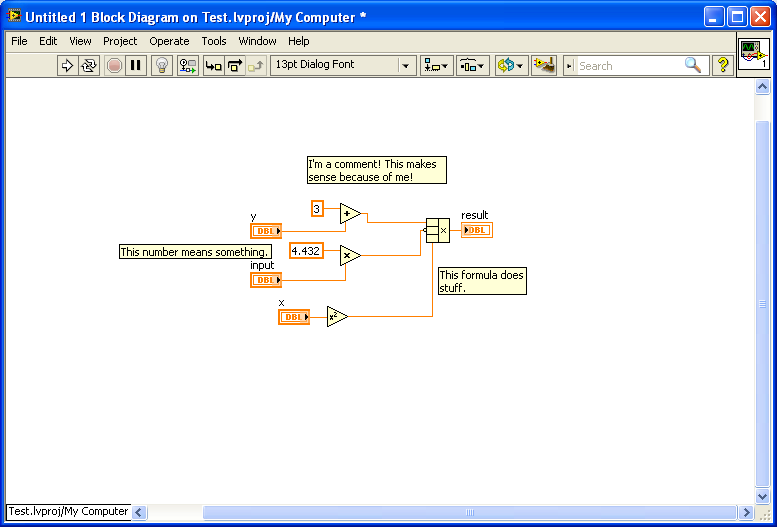
Here are some other things to consider if you’re tracking down a problem:

* Global variables: make sure you know every place a global variable is used. If two pieces of code trying to write to the same global variable simultaneously, that is probably bad. And probably is just a polite way of saying definitely.
* Lag: if a VI seems to be running slower than it should, make sure that all while loops have some kind of wait (like 20 ms). On the robot, read sensors as little as possible.
* Refnames (i.e. used to open hardware on the robot): check the spelling and make sure every piece of hardware has a unique refname. Or use an enum to select a name from a standard list.
* Default values: if a VI uses controls or indicators that have a pretty much constant or default value (like you want the control “input A” to always be 4.3), make sure you go to Edit >> Make Current Values Default. Otherwise, each time you use a VI, those values will revert to LabVIEW defaults (i.e. 0 for numbers, false for booleans, “” for strings, etc.)—i.e. the 4.3 would change to 0 without your ever knowing it.

**Documentation**

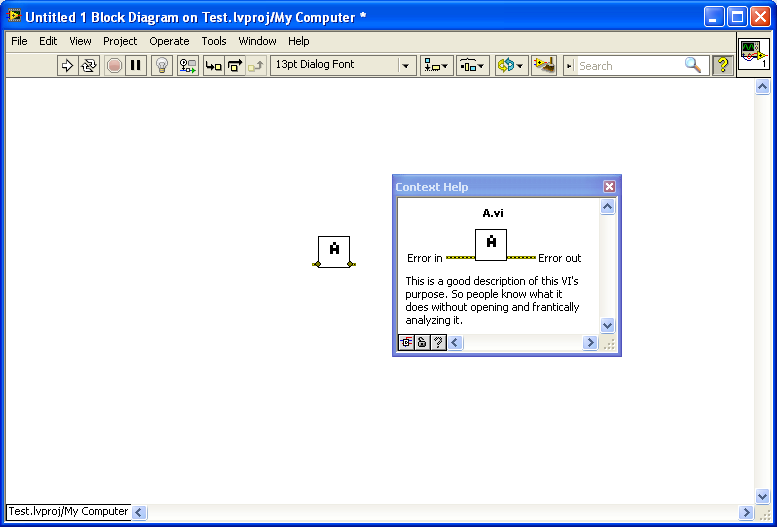
**Comments**

Say you’ve written a long, complex VI that is, in the best LabVIEW tradition, utterly confusing. Without comments, others will have no idea what it does, and you won’t either in a couple of weeks. So, the best way to clarify and maintain code is by making comments—short descriptions of what a small, potentially confusing piece of code does. To make a comment, double-click anywhere in the front panel or block diagram and start typing.



**Describing a subVI**

You can also write a description of a subVI as a whole—its overall purpose, what inputs it takes, and what outputs it returns. To do this, write click the icon in the top left, choose “VI Properties”, and select “Documentation” from the dialog that pops up.



**Code should always include these two things—comments and a description of the VI as a whole. Please. We might actually stay sane this way. Ya never know.**

**Advanced LabVIEW**

**Event-driven programming**

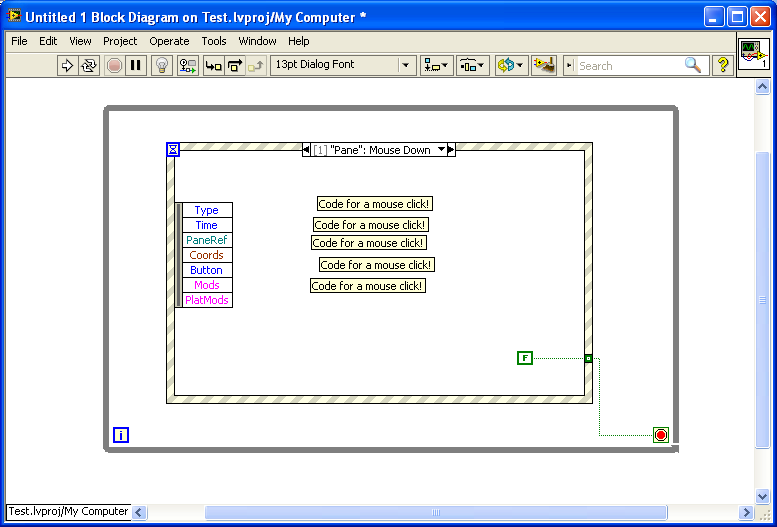
So, we want the robot to respond when we press a button, move a joystick, etc… and we could use while loops to keep checking those buttons over and over again, responding when something finally happens. Or we could use the event-driven features built into LabVIEW and make LabVIEW do all the hard work.

**Event Structure**

An event structure is used… to handle events (who woulda thunk?). An event structure is similar to a case structure in that it has multiple subdiagrams that could be executed; for an event structure, the subdiagram to execute is selected based on what event occurs (rather than a boolean, enum, etc.). So if “mouse click” and “button press” were two events, the “mouse clicked” subdiagram would run whenever someone clicked a mouse. If no events occur, the event structure just waits… and waits (which can be nice, because unlike a while loop it doesn’t need to run continuously).

Event structure

This code runs after a certain event occurs (in this case, mouse down)



This while loop determines when the event structure should end. If the “mouse down“ case passes true as an output, the while loop would end, and the event structure would no longer be able to respond to events. If it outputs false, then the while loop continues, and the event structure begins waiting for the next event.

Optional timeout. If you don’t wire a time (in ms) to this, the event structure will wait forever for an event to occur. Which may or may not be what you want.

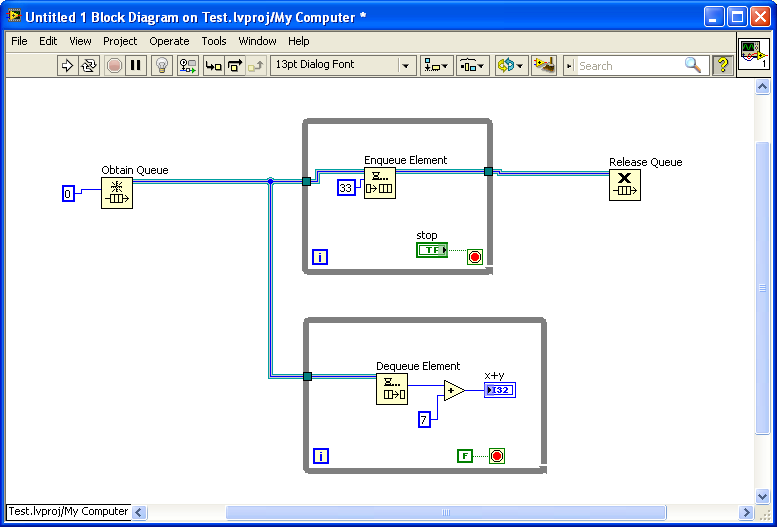
**Queues**

Small problem. User interface events (mouse click, pressing a control on the front panel, etc.) aren’t supported on the cRIO, since a robot has no use for them. So instead, LabVIEW gives us queues to work with. (Queues are found on the menu under Synchronization >> Queue Operations.)

Basically, to use a queue, you put elements onto the queue (i.e. enqueue elements), and elsewhere in the code you dequeue elements and respond to them. The nice thing about dequeuing is that if there are no elements in the queue, the dequeue VI will just wait—meaning it acts exactly like an event structure. The elements are like events, and the dequeue VI only runs when there are actually events to respond to.

So queues can follow a sequence like this:

* Create the queue.
* Enqueue elements when something happens (like someone presses a button on the controller).
* Dequeue elements and respond to them.
* Destroy the queue when done.



Destroys the queue.

Puts an element on the queue.

Creates the queue.

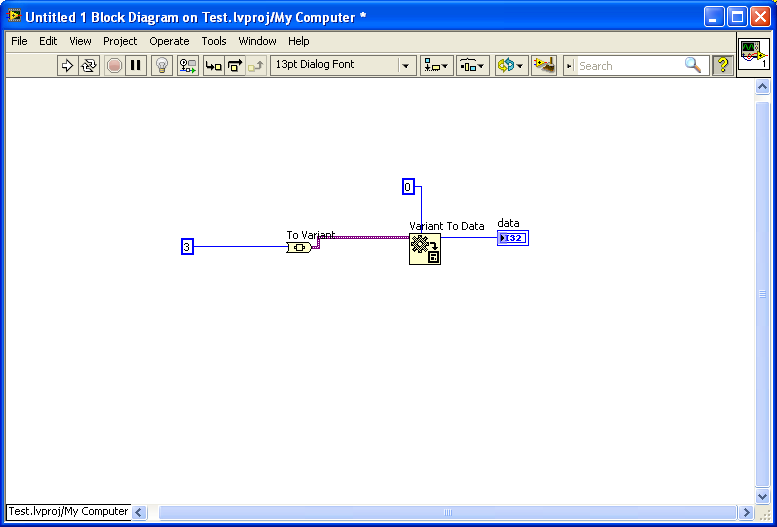
Takes an element off the queue. It will wait for an element to actually be put on the queue, so the while loop will pause if nothing is on the queue.

Elements in a queue also carry data that can be retrieved once the element is dequeued. In the example above, each element enqueued contains a number, and that number can be manipulated once the element is dequeued. (All elements in a queue must have the same kind of data, which is determined by the data type passed to the obtain queue VI.). So, we could control how the code responds to an element that is dequeued based on the data that is contained by that element—a boolean could tell the robot to do one thing, an enum could do another, etc.

**Variant data**

If all elements in a queue have to contain the same data type… that limits flexibility. Sure, we could define a hopelessly convoluted cluster to cover every possibility, but that uses a lot of memory and is not conducive to sanity. LabVIEW provides a variant data type to get around this—the variant data type can literally be anything. (Found on the menu under Cluster, Class, and Variant >> Variant).

To use the variant data type, you first convert any data (number, boolean, cluster etc.) to the variant data type. Then, when you want to use it, you convert it back by supplying the correct data type.



Tells LabVIEW what kind of data to expect.

Could be anything.

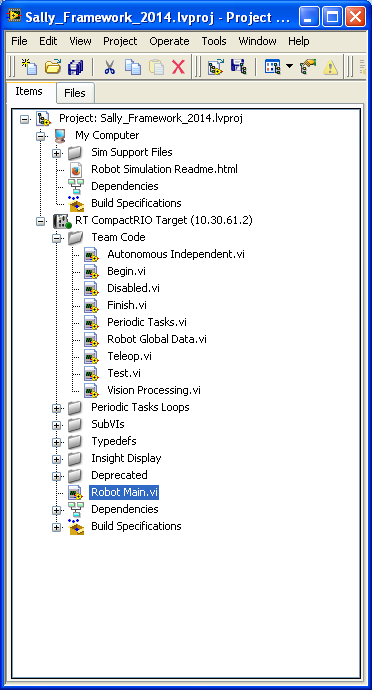
Convert the number to the variant data type.

Magic

**FRC LabVIEW**

**Framework**

During competition, a match is broken up into two periods: a 15-second autonomous period (no manual control allowed) and a two-minute-or-so teleoperated period (manual control), and the default framework we get from FRC is structure to reflect this. We can access the framework by opening the .lvproj file.



**Important VIs**

* **Main.vi:** This is the VI that is run first on startup. It is responsible for beginning/running all other parts of the game (autonomous, teleop, etc.). We should never need to modify this (unless you’re in a really masochistic mood…).
* **Begin.vi:** This VI is called by Main.vi immediately after the robot is powered on. It can be used to initialize anything the robot might need later, so we should do things like open references to motors and define initial values for global variables in Begin.
* **Robot Global Data.vi:** This VI doesn’t actually run any code—it’s a special VI that LabVIEW uses to store all of the robot’s global variables (e.g. current RPM of the launcher wheel, desired motor input for the drive train, etc.). So, whenever you need to create a global variable, it should be added as a control/indicator to Robot Global Data.
* **Periodic Tasks.vi:** This VI is called once after Begin finishes executing, and continues to execute regardless of the mode (autonomous, teleop, or disabled) until the robot is turned off. We use this to contain any loops that we want to run over and over (e.g. driving, pneumatics, etc.)—this way, we can control the drive train during both autonomous and teleop periods, for instance. This VI does in fact run when the robot is disabled (which makes perfect sense… not…), but it can’t really do anything since all hardware is locked out (i.e. motors cannot turn) while the robot is disabled.
* **Autonomous Independent.vi:** This is called ONCE at the beginning of the autonomous period. We should put our entire autonomous process—drive forward, shoot stuff, etc.—in this VI.
* **Teleop.vi:** This VI is called PERIODICALLY (about every 20 ms) during teleop period. We typically use this to gather user input from the controllers.

**Adding VIs**

The easiest way to add a completely new VI to the project is to right-click on a folder and select New >> VI. Make sure the VI is added under “RT CompactRIO Target (10.30.61.2)”, not “My Computer”.

**Hardware**

At some point, there comes every programmer’s worst nightmare… interacting with the real world. Here are a few ideas for using robot hardware (motors, solenoids, etc.) that will help you avoid (maybe) destroying everything in sight in a fit of senseless frustration:

**Initialization**

Before LabVIEW can use any hardware, it needs to know that the hardware exists and how to use the hardware. So, we set up all hardware during Begin. Basically, this involves two steps: calling the “open” VI for whatever piece of hardware you’re using (e.g. “WPI\_EncoderOpen.VI”), and assigning a refname to that piece of hardware (e.g. with “WPI\_EncoderRefNumRegistry Set.vi”. (The open VI creates a reference to that piece of hardware, and the refname just specifies a human-readable label for that hardware that can be used to retrieve the reference.)

**Usage**

When you’re ready to manipulate the hardware (e.g. in Periodic Tasks), you access the reference you created in Begin by using a refnum get VI (e.g. “WPI\_EncoderRefNum Registry Get.vi”). Then you get to play with the hardware/hopefully not destroy it too badly/stay up till midnight wondering why it doesn’t work. Cheers.

Note: all of the robot-specific VIs that FRC provides can be found in the menu under WPI Robotics Library. Below is a quick description of how to use some hardware… but for a more complete and techy description, use the context help, consult examples from our framework (or go to help >> find examples), or ask experts.

**Motors**

Spinny things. ~~Pretty simple~~ ~~kinda simple~~ well ok not simple. They make life more interesting though.

**Initialization**

To set up a motor, you need three things: the digital module it’s attached to, the PWM channel it uses, and the type of motor controller (i.e. Jaguar, Victor, or Talon); pass these to the open motor VI. Check with electrical peoples to find out what these are for each specific motor. (Also, if a motor is not working, it’s a good bet that one of these three is wrong).

**Usage**

To run a motor, you give it an input between -1 (full reverse) and 1 (full forward). It’s also a good idea not to change the motor input too quickly (i.e. don’t go from -1 to 1 instantaneously).

**PIDs**

Consult an expert. Like an actual, real, live expert.

**Encoders**

Basically, encoders can measure how fast something rotates. Don’t worry; it gets a lot more interesting than that. We use two types of encoders: magnetic encoders and optical encoders. Magnetic encoders output a voltage that changes (from, say, -1 V to 1 V) as the encoder rotates, while an optical encoder outputs a predefined number of counts per revolution (e.g. a 2400 CPR encoder outputs 2400 **c**ounts **p**er **r**evolution).

**Initialization—magnetic encoder**

To set up a magnetic encoder, you here’s what you need to do:

* Open an analog channel—this will let you read the voltage from the encoder. You’ll need to specify the analog module the encoder is wired to, as well as the physical analog channel (1-8) that the encoder is using. (WPI\_AnalogChannelOpen.vi)
* Then, you have to read and manipulate the voltage… or make something else do it for you. LabVIEW provides an analog trigger that will count every time the voltage passes a certain value. You’ll need to provide a reference to the analog channel you just opened, as well as a maximum and minimum voltage that will activate the trigger. (WPI\_AnalogTriggerOpen.vi)
* LabVIEW also provides a counter that will keep track of the count, along with some other useful data. To open a counter, provide a reference to the analog trigger you just opened. (WPI\_CounterOpen.vi)

**Usage—magnetic encoder**

WPI\_CounterGet.vi gives you two pieces of data: the period (i.e. time between counts), and the total number of counts. If you know the period between counts, and how much distance a count corresponds to in the real world (consult electrical peoples), you can calculate speed.

**Initialization—optical encoder**

An optical encoder should be a little simpler:

* Open two digital input channels. For each one, specify the digital module and physical DIO channel that each encoder uses (consult electrical peoples). (WPI\_DigitalInputOpen.vi)
* Then, with these two digital inputs and the decoding type (consult electrical peoples), open a reference to the encoder (WPI\_EncoderOpen.vi). While opening the encoder, you should also specify the “DistancePerCount” (i.e. the distance that each count corresponds to in the real world—1 inch, 1 degree, etc.).

**Usage—optical encoder**

WPI\_EncoderGet.vi gives you two useful pieces of data:

* Distance: how far the object in question has traveled. Basically the number of counts multiplied by the DistancePerCount you specified while opening the encoder.
* Rate: how fast the encoder is rotating, in counts per second.

**Limit switches**

A limit switch is basically a button—it can be on or off.

**Initialization**

To open a limit switch, you just need to specify the digital module and the DIO channel it uses (as always… annoy electrical peoples)—actually, you’re just opening a digital input. Yay. Done. (WPI\_DigitalInputOpen.vi)

**Usage**

Use the WPI\_DigitalInputGetValue.vi to return the state of the limit switch as a boolean. **Warning: false is returned when the limit switch is pressed, while true is returned when the limit switch while it is released.** Totally intuitive.

**Compressor**

A compressor just maintains pressurized air for use by any pneumatics.

**Initialization**

You need four things (use WPI\_CompressorOpen.vi):

* The relay channel used to power the compressor.
* The digital module that the relay is attached to.
* The DIO channel used for the pressure switch (a safety in case the pressure becomes too high).
* The digital module that the pressure switch is attached to.

**Usage**

After opening the compressor, you should run two VIs:

* WPI\_CompressorStart.vi: just give it the reference you opened. This will automatically turn the compressor on and off as needed.
* WPI\_CompressorControlLoop.vi: again, just give it the reference to the compressor you opened. This will monitor the safety switch.

At this point, everything is… automagical. Tears of joy are appropriate.

**Solenoid**

A solenoid controls a pneumatic cylinder. There are two types (single and double), but we’ve only ever used a single solenoid.

**Initialization—single solenoid**

To set up a solenoid, you just need to specify the solenoid channel it uses (yeah, the electrical peoples probably hate us by now… but bug them anyway). (WPI\_SolenoidOpen.vi)

**Usage—single solenoid**

Use the WPI\_SolenoidSet.vi to change the state of the cylinder. You can pass it one of two states (on or off), which, depending on how the cylinder was set up, could correspond to in or out.

**Gamepads**

We use two video game controllers to manipulate the robot. No, driving the robot is not a video game. Waaaaay better than that.

**Initialization**

You just need to specify which USB port on the laptop the gamepad uses. Even this doesn’t really matter—you can just physically switch the gamepads if need be. (WPI\_JoystickOpen.vi)

**Usage**

Use the WPI\_JoystickGet.vi to access all of the gamepads axes (which are analog, going from -1 to 1) and buttons (which are true or false). Consult examples from the framework to determine exactly which axes/buttons on the controller correspond to which axes/buttons in the code. You’d think that button 1 on the controller would be button 1in code, but ya never know.

**Dashboard**

The dashboard is our other form of user interface with the robot. We can use it to display sensor data on the laptop, as well as to transmit data to the robot. The dashboard stores data in “network tables” that can be accessed from both the robot and the laptop. **Warning: the dashboard is, at times, buggy, glitchy, or downright temperamental. On happy days, it works; on others, it flat-out refuses. Watch out for bad/incorrect data.**

The dashboard uses a completely separate project (e.g. Sally\_Dashboard\_2014.lvproj). In this project, open the front panel for Dashboard Main.vi to see how the dashboard will look on the laptop.

**Reading/writing from dashboard**

LabVIEW provides a set of read/write functions to communicate with the network tables.

* Write VIs (like “SD Write Number.vi”) will take a value from the control and write it to the network tables. You need to use a refname (a string) to indentify the value.
* Read VIs (like “SD Read Number.vi”) will read values from the network tables and output them to an indicator. You just need to supply a CORRECT refname to the read VI (i.e. it matches **exactly** a refname you passed to a write VI)—otherwise LabVIEW will have no idea what value you want.
* You can use read/write VIs in both the robot code and the dashboard code. For instance, if you wanted to specify how far the robot should drive in autonomous, you could write that value in the dashboard code and then read it in the robot code. Or, if you wanted to display sensor data on the laptop, you could write that in the robot code and read it in the dashboard code.

**Adding controls/indicators to the dashboard**

Just add a control/indicator to the block diagram (next to all the other controls/indicators), and then position it in the front panel so that vaguely resembles something that kinda looks organized.

**References**

[www.ni.com/pdf/manuals/320999e.pdf](http://www.ni.com/pdf/manuals/320999e.pdf) (a far more complete LabVIEW guide)

<http://www.ni.com/pdf/manuals/321393d.pdf#labview_style_guide> (style guide)

<https://decibel.ni.com/content/docs/DOC-8923> (links to FRC tutorials)