THE SECRET BOOK OF FRC LABVIEW

Geoff Nunes
Team 1391
Westtown School

Version 0.4
4 December 2013

Contents

Introduction 1
Chapter 1 — Elements of LabVIEW Programming 2
The Elements 2
LabVIEW Programs 3
Back to the List 5
Variables 5
Expressions and Assignments 7
Conditionals 10
Flow Control 12
Subroutines 18
Clusters 20
Graphing More 21
Problems 22
Chapter 2 – State Machines 25
Planning 25
Programming 27
Adding a Timeout 33
Delays 35
Problems 38
Chapter 3 — Introduction to the cRIO 40
LabVIEW Projects 41
Running Programs on the cRIO 44
Chapter 4 — The Robot Framework 49
Robot Main 49
Begin 51
Periodic Tasks 52
Robot Global Data 54
Teleop 55
(Very) Basic Teleop Code 55
Waiting Without Hanging 56
Avoiding a Death Spiral 59
Teleop Strategy 60

Autonomous 61
The Most Important Part 61
How Autonomous Works 61
The Autonomous Code 62
Disabled, Finish, and Test 63
Building, Running, and Debugging 64
The Driver Station 67
Developing Code Without a Robot 71
Chapter 5 — Input and Output 74
The Interface Sheet 74
Joysticks 75
Motors 75
Robot Drives 76
Drive Delays 76
Digital Input and Output 77
Input 77
Output 78
Relays 78
Encoders 78
Analog Input 81
Pneumatics 82
The Compressor 82
Valves 83
Doubles 84
Singles 84
Servo Motors 85
Chapter 6 — PID Control 86
The PID Concept 86
PID Math 87
Motor Math 89
Driving Simulator 91
Turret Simulator 95
Final PID Thoughts 96
Problems 96

Chapter 7 — Image Processing 98
Doing it on the laptop 98
Loading an image 98
How LabVIEW stores images 99
Image types 101
Getting images 102
Getting good images 103
Choosing your colors 105
Finding the BLOBS 107
Camera Optics 109
Finding the target distance 110
Advanced Stuff 114
Chapter 8 — The Dashboard 116
Basic Dashboard Communication 116
The Dashboard Project 118
Under the Hood (or Behind the Dashboard) 118
Loop 1: Simple Data Communication 119
Loop 2: Image Acquisition 120
Loop 3: Autobound Variables 121
Loop 4: The Network Table Tree 122
Loop 5: Kinect Data and Display 122
Loop 7: Window Size and Position 123
Loop 6: A Whole Lotta Stuff 123
Dashboard Strategies 129
Building Your Dashboard 129

Introduction

The motivation for this book comes from my own experience as an FRC mentor. At work I am (or at least pretend to be) a generally competent LabVIEW programmer. I've done some pretty cool stuff, like programming a robotic microscope that can automatically inspect a big sheet of glass: big enough to hold eight 32" flat panel TV screens. My first encounter with FRC LabVIEW, however, transformed me into a gibbering idiot. There is an elaborate but mysterious framework that runs the robot, and the main source of documentation seems to be the forum archives at Chief Delphi. The experts who regularly contribute there are great, but as I was madly trying to learn, and at the same time teach our team members, I found myself constantly wishing for a book that explained everything in one place. Hopefully, this is that book.

The intention here is that you can read the entire book, or only the parts that interest you. You can use it as a classroom text, or as a reference manual. Chapter 1 covers the basic principles of programming in LabVIEW. It is written under the assumption that you have already done at least some programming in a "traditional", text-based language, although if you haven't you should still be fine. Chapter 2 covers state machines, which are essential for programming robots. Chapter 3 introduces LabVIEW projects, which is how you manage all the code for your robot, and Chapter 4 covers the specific project that is the FRC Robot Framework. This chapter is where most of the secrets are revealed. Chapter 5 covers the various input and output devices, such as motors and sensors, you can attach to and control with your robot. Chapters 6 and 7 cover PID control and image processing. They contain much that can be done without a robot, and so can be covered in the pre-season, in spite of appearing later in the book. Finally Chapter 8 covers the Dashboard, which is simple to use, but complicated to understand.

This book includes, as a companion volume, a zip file of images for use with Chapter 7.

Although this book contains many example programs, you won't find a companion zip file for them. It takes practice to learn how to write compact, readable LabVIEW code, practice you will get by diagramming the examples yourself.

As you read this book and compare it to the actual software, please be aware that we are aiming at a moving target. A new version of the FRC robot framework is issued every year, and every year there are improvements and enhancements. You can pretty much be guaranteed that if you are building a robot in year n, this book was updated against a version of the framework from the year $n-1$. (Or $n-2$. Or worse.) Be flexible.

Finally, a note of thanks to both Worcester Polytechnic Institute and National Instruments. The FRC robot framework is a product of the robotics program at WPI, and is a large, complex, and amazingly robust piece of software. Without it, FRC would be way more work and way less fun. Equally amazing is the support that NI provides to FRC, as is the amount of time an effort their staff contribute on Chief Delphi and at competitions. I owe a particular debt to Greg McKaskle, whose white paper on vision targeting for the 2012 competition inspired much of Chapter 7.

Chapter 1 — Elements of LabVIEW Programming

This chapter is designed to get you up and running with LabVIEW as quickly as possible. It is probably a terrible way to actually learn LabVIEW. For that, I would recommend "LabVIEW for Everyone" by Jeffrey Travis and Jim Kring (Prentiss Hall, New York, 2006). It is a very complete book. It also thick enough to stun an ox.

You should read this chapter with your computer on and LabVIEW open. And you should actually program all the examples for yourself. LabVIEW is an "experimental" programming language in the sense that you will learn by doing experiments to see what happens. If you take a "reading only" approach, you will never really learn, and that will be very frustrating.

The Elements

All programming languages share a set of common elements. If you have written programs in a text-based language (*e.g.,* C and friends, Java, FORTRAN, BASIC, Python…), then you will be at least vaguely familiar with these elements. I'll give them a quick listing, and then revisit each one to show how it appears in LabVIEW. If you know all this already, then just jump ahead to the LabVIEW part.

1. *Variables* These are objects in which you can store data. They might hold a single number, or an array of numbers, or a string of text. Boolean variables can be either True or False.

2. *Expressions* These are the operations that manipulate the values stored in variables, as in A + B.

3. *Assignments* This is how you get values into variables. For example, you might assign a variable to have constant value, as in PI = 3.1415. Or you might use an expression to give a variable a value based on other variables, as in C = A + B.

4. *Conditionals* Also called branching statements. The simplest conditional allows one piece of code to execute if an expression evaluates to True, and a different piece to execute if the expression evaluates to False.

5. *Flow Control* This is the term generally applied to sections of code that repeat for a set number of times, or repeat until some condition is true. Strictly speaking, conditionals also control the flow of a program, so the distinction between them and flow control elements is a bit arbitrary.

6. *Subroutines* These are any pieces of code that can be packaged and used in some other piece of code. They go by lots of other names: function, procedure, or subprogram.

We'll stop the list here, and point out that you are not even a single page into this book and have already encountered its first lie. (Note the clever implication that there will be more lies to come.) I said at the start that *all* programming languages share a common set of elements, but that is not true. In a text-based program, an assignment, or a print command, or a command to write to a file, all go by the generic term *statement.* LabVIEW doesn't have statements. It is an entirely different kind of beast.

LabVIEW Programs

Before we revisit the list of elements, it is necessary to spend a moment on the structure of LabVIEW (LV for short) programs. In a text-based language, a (simple) program is just a single object: a text file. In LV, a program is two objects: a *front panel* which handles inputs and shows you the results, and a *block diagram* which contains the actual code. If you want, you can think of the front panel as the body and controls of a car: it's got the fancy paint job, the headlights, turn signals, steering wheel, and pedals. The block diagram is the engine that makes it go.

When you start LabVIEW (do it now) you get a window that looks something like Fig. 1.1. If it doesn't say "FRC" somewhere on this screen, you haven't installed everything that you need, and won't be able to follow much of this book.

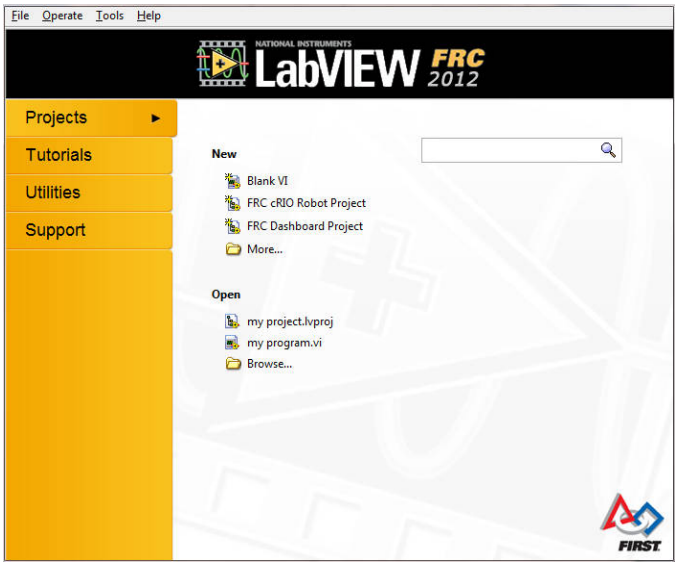


FIGURE 1.1 The LabVIEW Getting Started window.

Click on "Blank VI", and you will get a pair of windows, like Fig 1.2. The windows are conveniently labeled so you can see right away which one is the front panel and which one is the diagram. You will need to toggle back and forth between these two windows a lot. The easiest/fastest way to make this swap is with the *ctrl-E* key combination.

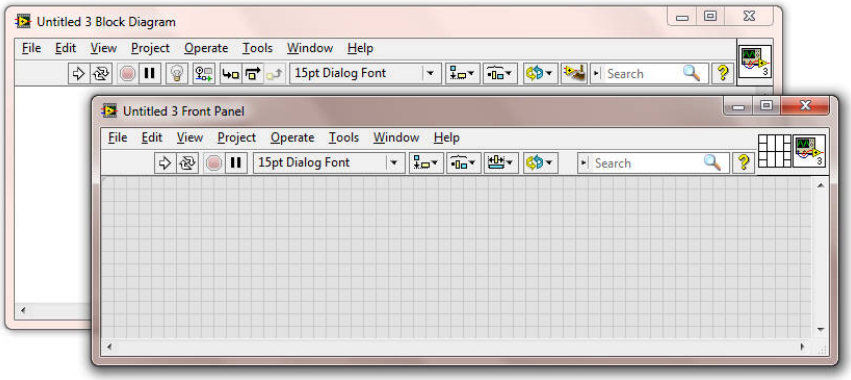


FIGURE 1.2 A new VI (LabVIEW program) showing the front panel (foreground) and block diagram (background).

To make this more concrete, Fig. 1.3 shows the front panel of a simple program (which has been run once to draw the data on the graph), and Fig. 1.4 shows the diagram that makes it work.

In LV, a program is officially called a VI, which is short for Virtual Instrument. I will use the terms "VI" and "program" interchangeably. Although the program in Figs. 1.3 and 1.4 doesn't do much, there is a lot of new "LabVIEW-ey" stuff going on. I promise that you will soon understand it all. But in order to do that, we need to get back to the list.

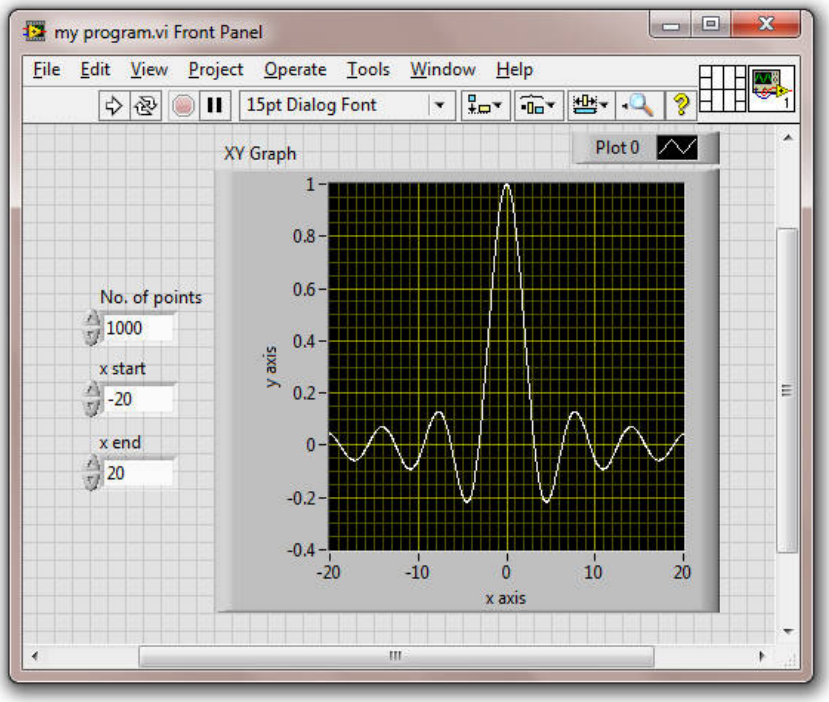


FIGURE 1.3 The front panel of a LabVIEW program. This program has been run. Otherwise, the graph would be blank.

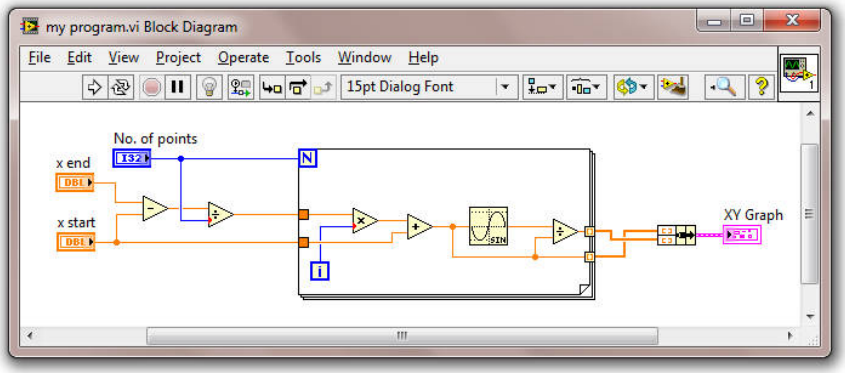


FIGURE 1.4 The block diagram of the program in Fig. 1.3.

Back to the List

Variables

Just as the programs have two parts, variables in LV have two parts: the thing that appears on the front panel, and the corresponding terminal on the diagram. They also come in two "flavors", depending on whether you read from them, or write to them. In Figs. 1.3 and 1.4, the objects labeled "No. of points", "x start", and "x end" are all *controls*. The program reads from them. The object labeled "XY Graph" is an *indicator*. The program writes to it. This division is very strict: variables are either controls or indicators. The program either reads from them, or writes to them, but not both. (There is a way around this using *local variables*. We'll get to them later.)

One of the cool features of LabVIEW is that variables can have names that would be illegal in any other language. They can have spaces and symbols and punctuation if you want. You can even give two different variables the same name! LabVIEW will always know which one is which, although you will not. Don't do this.

The most common method for adding a variable to your program is from the front panel. Open a blank VI, and right-click on the front panel. You should get a pop-up menu that looks something like the left-hand side of Fig. 1.5. To get the right-hand side of the figure, I slid the mouse over the upper left square, and the Numeric sub-menu popped open. Move the mouse over each item, and you get its name shown at the top. Everything here is either a control or an indicator. In the

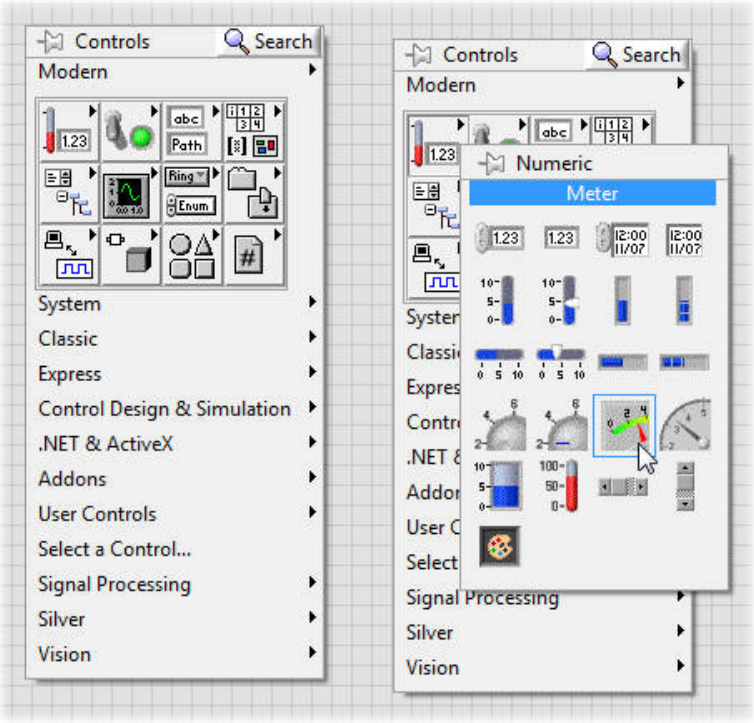


FIGURE 1.5 The Controls Palette for placing objects on the front panel.

upper left is a numeric control. To its right is a numeric indicator. They are not really different, except values are read from one and written to the other. In the second row, the leftmost object is a vertical fill slide (an indicator) and next to it is a vertical pointer slide (a control). There is no significant difference between the slider control and the text box control! In one case, you type the value you want the variable to have. In the other you move the slider until it has the value you want. The slider defaults to a zero to ten range when you drop it on the front panel, but you can double click on the starting and ending values and change them to anything you like. Similarly, there is no real difference between the indicators except how the value is displayed. In fact, both can have a digital display that you type into or read from, making the slider part pure eye candy.

One of the best designed and most powerful tools in LV is the right click, which brings up the *context menu*. As the name implies, the contents of this menu depends on the context in which you right-clicked. You should get in the habit of—as an experiment—right-clicking on things, and examining the list of options that is presented. If I tell you to do something in this book, but I don't tell you how, you can bet that it involves a right-click. I'll highlight a few key right-clicks as we go along, but there are far too many options to cover in a skinny volume like this one. Explore, explore!

If you haven't fixed it yet, the controls panel you get by right-clicking on the front panel looks nothing like Fig. 1.5. Probably the Express controls are the default. They are very limited, and you should fix yours to look like mine. Note the little thing that looks like a push-pin in the upper left of the palette. It *is* a push pin. Click on it and the palette is "pinned" to the screen. When you do that, a Customize button will appear, which will allow you to put the Modern palette at the top, and also control other aspects of how the palette looks. (I am really particular about my work environment. Time is short, and there is a lot to do. I insist on customizing the LV environment to make my work as efficient as possible. You should too.)

In addition to controls and indicators, there is a third variable form, the *constant.* Constants are controls that have a fixed value. They appear only on the diagram, as their value cannot be changed by the operation of the program.

In all cases (control, indicator, and constant), these objects also have a specific *representation.* You are very unlikely to use anything other than the big three: double precision real, 32-bit integer, and unsigned 32-bit integer. Once you have created a control, for example, you can right-click on it to change its representation. If you don't know the difference between a real and an integer in the context of computer programming, this book is *not* the place to learn it!

Variables can hold just a single value, or they can be arrays of values. These arrays can be one-dimensional, two-dimensional, or *n*-dimensional. There are also more complicated objects that are the equivalent of structures in C. In LV, these are called *clusters* in which several different kinds of objects (e.g. a real number, a Boolean, and an array of integers) can all be *bundled* together in a single object. You will find that clusters are common in FRC programming.

Expressions and Assignments

Time to learn by doing! We'll write a program that executes the statement "C = A + B", except that it won't be something you recognize as a statement (*i.e.* as a line of code). Create a new, blank VI. On the front panel, place two Numeric Controls named A and B, and a Numeric Indicator named C. Your front panel should look something like Fig. 1.6.

Flip over to viewing the diagram, you will have something that looks nothing like Fig. 1.7. To begin with, unless you have already fixed this, your terminals are not sleek and svelte like mine, but big, blocky, and awkward. Also, your terminals are probably randomly scattered about the diagram, and not neatly arranged in anticipation of the connections to be made. You should fix both of these things. To change how terminals appear, go to the Tools menu and select "Options…". Select the Block Diagram category and un-check "Place front panel terminals as icons". The new setting will apply to any new objects you create, but won't change the objects you've already put in your program. You get only one guess as to how to bring up a menu that will let you change how the already placed terminals appear. You will have to do that for each one.

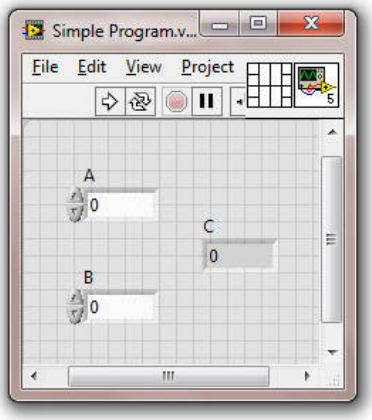


FIGURE 1.6 The front panel of a very simple program.

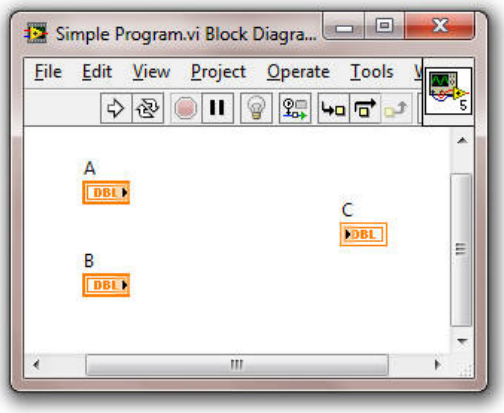


FIGURE 1.7 The block diagram for Fig. 1.6.

Right-click on the diagram to bring up the Functions Palette. If it doesn't look like mine (Fig. 1.8), or like the way *you* want it to, then pin it and customize it, just as you did with the Controls Palette. Find the Numeric sub-palette, select the Add function, and place it on your diagram. I've laid out the terminals in a suggestive fashion, and it should be obvious where it goes.

Now we need to connect things up. Hold down the shift key and right-click on the diagram. This brings up the Tools Palette (Fig. 1.9). You need to use the little spool symbol on the left, so you could select it. But it is much better to select the Multipurpose Tool at the top, which will automatically give you what you need, depending on where you move the mouse. In Fig. 1.9, the Multipurpose Tool has been selected.

It takes some getting used to the spool tool. When drawing a wire that has to turn corners (as in Fig. 1.10 below), you can click to set the location of the corner. You will notice that the tool is always trying to draw a vertical line segment, or a horizontal one. You can force it to switch from one to the other

(while drawing a wire) by hitting the space bar. Also, go to the Tools menu and select "Options…". Select the Block Diagram category and, under the Wiring sub-category, un-check "Enable automatic wire routing". But leave "Enable auto routing" checked. (Trust me. They *are* different!)

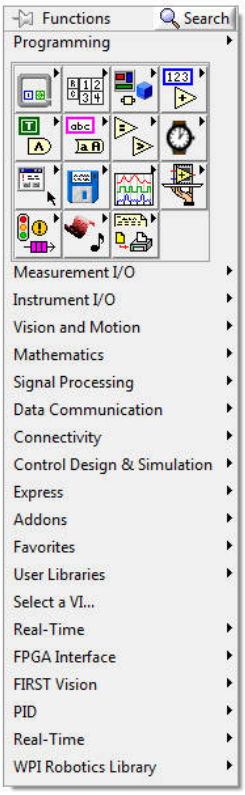


FIGURE 1.8 The Functions Palette.

FIGURE 1.9 The Tools Palette.

Now, with the Multipurpose Tool selected, if you hover the mouse over the little black triangle on the A terminal, the cursor should change to a spool. Click to start the wire, and connect it to the B terminal. This is wrong, and LabVIEW will tell you in two ways. The wire is drawn as a dashed line with a big red X, indicating that it is a broken wire. If you look in the upper left of your diagram (or front panel), you will see that you have "broken arrow" syndrome. A LabVIEW program with a broken arrow will not run. You can clear broken wires by selecting and deleting them, or you can use the *ctrl-B* key combination to delete all the broken wires in your program at once. (Use ctrl-B with caution. Sometimes a simple problem will cause all of a complex diagram to appear broken. Fixing that simple problem will be much less work than re-drawing the entire diagram.)

So, clear the broken wire, and wire the diagram to match Fig. 1.10. From the front panel you can type values into A and B, and hit the (un-broken) white arrow in the upper left to run the program and have the result of the addition assigned to C.

What did you not do? In languages like Java and C, you need to compile and link your program before you can run it. LabVIEW does all that automatically and more or less instantly. If you have the white arrow, your program is ready to run. (To be fair, that is only true for programs that run on your computer. You will need to compile your robot programs, but that doesn't happen until the next chapter.)

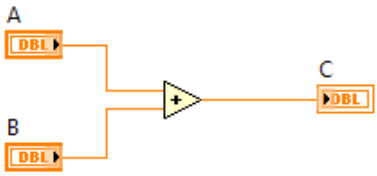


FIGURE 1.10 Our simple program, all "wired up.".

The multipurpose tool can be a bit tricky if you are using it to drag wires around on your diagram. I like a neat, easy to read diagram, so I select and drag the wires to where I want them. It is really, really easy to click on a wire, only to find that the move tool (a pointer arrow) changed to a wiring spool just as you clicked, so now you are adding a new wire to your wire instead of moving it. Hit the Escape key or right-click to get out of situations like this.

LabVIEW contains several automatic tools for neatening up your diagram. I am never quite happy with the results, so never use them. You may have a different opinion. Look for the button with the broom along the top of your block diagram, or select just the part of your diagram you want to neaten and type ctrl-u.

As I promised, there is no explicit assignment statement. The wires in Fig. 1.10 are LabVIEW's equivalent of assignment statements. But don't think of them like that. Think of them as wires. Data *flows* along the wires, from one object to another. LabVIEW is culturally insensitive. It is designed so that the data flow is, generally speaking, from left to right. My apologies if you learned to read in a non-Western culture and expect things to flow from right to left or top to bottom. National Instruments is headquartered in Texas, and you just don't mess with Texas. Please respect the left-to-right thing, and make all your programs clean, neat, and well organized. You will be really grateful for this when you are in the pit, you need to make a software change in under three minutes, and your whole team is screaming at you to hurry up. Programs that are easy to read are easy to debug. Programs that are a snarly mess are, well, a snarly mess.

The lack of an assignment statement brings up an interesting question: How would you program a simple assignment like I = I + 1? This is trivial an any text based language, but it requires reading from and writing to the same variable. In LV, we can read from OR write to a variable but not both. Except…for what are known as *local variables.*

Do this: in LabVIEW, create a new blank VI (ctrl-n for people in a hurry), and drop a numeric control on the front panel. Be creative and name it "I". On the block diagram, right-click on it and select **Create ▸ Local Variable**. The resulting object *is* the variable I, but in a form you can write to. If that is not what you need, you can right-click on the local variable and select **Change to Read**. Fig. 1.11 shows a program with three different solutions to the "I = I + 1" problem, all equally valid. Every time you run this program, the values will increase by one. The third example in the program shows that you can have several local variables that all refer to the same variable. They still have to follow the general LabVIEW "read or write" model in that each local can only be a Read type or a Write type, but you can have both types in the same program.

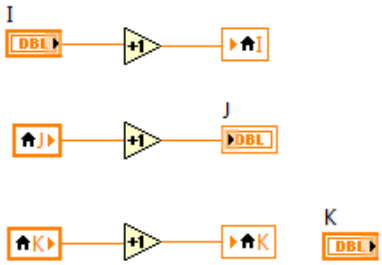


FIGURE 1.11 Three different implementations of I = I + 1.

The easiest way to make a local variable—assuming you already have one—is by the "control-click-drag" method. Hold down the control key, click on an existing local variable, and drag it to make a copy. Then you can *left*-click on the new copy to see a list of all your variables and select which one you want this local variable to be. (You can copy pretty much anything by the "control-click-drag" technique.)

Now that you know about local variables, be aware that you have been handed a very sharp implement with which it is all too easy to harm yourself. Do not fall into the Beginners Trap and start using local variables everywhere for everything. They consume a lot of memory and computer resources, and you can easily use them to make your program unpredictable and difficult to debug, neither of which is a desirable trait. They are largely unnecessary, and you should use them only when there is no other solution. Consider them Dangerous. They have an evil cousin, *global variables*, that are even worse. Globals are widely considered an Absolute No-No That Only An Idiot Would Use. They are also essential to programming your robot. We will encounter them in Chapter 4.

Conditionals

Changing the operation of a LabVIEW program based on whether a result is True or False is done with the Case Structure element, which is found on the Structures palette (see Fig. 1.12). Fig. 1.13 shows the front panel and block diagram of a simple program that changes its output depending on the value set by a front panel switch. (You should draw that example for yourself, because I'm going to ask to you to modify it in a moment.) The case structure in this example has two panels, one for the case that the input is True, and the other for the case that it is False. You can only view one at a time, but you can switch between them with the little control bar at the top. (You can also right-click on this control panel to get a list of other things you can do to modify this structure.)

The easiest way to switch between panels in any of LV's multi-panel structures is to make sure the cursor is inside the structure, hold down the control key, and scroll the wheel on your mouse. Don't have a wheel mouse? Don't whine. Don't tell me that you really like the touchpad and you're really good with it. Don't make excuses. Just get the mouse.

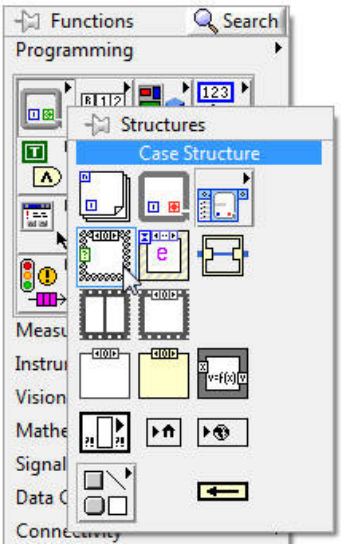


FIGURE 1.12 The Case Structure selected from the Structures palette.

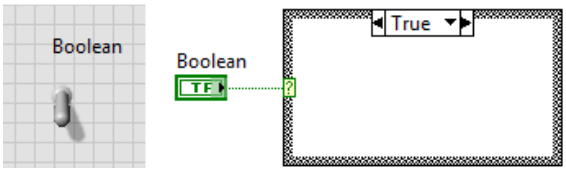


FIGURE 1.13 A simple case structure example program.

In Fig. 1.13 we have a simple If...Then…Else structure. If you need something more complicated, like an If…Then…Else If… kind of thing, then you just need to nest one case structure inside another.

LabVIEW case structures also play the role of constructs like switch-case in C and C++. Delete the Boolean control in the example of Fig. 1.13, and replace it with an *integer* Numeric. Note how the case structure changes. You can now add additional cases to correspond to other numbers. An important rule for case structures is that there *must* be a case for every possible value of the control. The control is a 32-bit integer and can therefore take on an infinite number of values (well, 2^{32} different values, which is a large number, if not quite infinite). For that reason, one of the cases must be labeled as the Default case. LabVIEW made it the 0 case, but you can make any of the other cases the default one. If the control value does not match one of the cases, then the default case will be executed. We will make extensive use of this kind of case structure in robot programming.

The control could also be a text string. It works the same way: if the control value (for example, "Do task A") exactly matches what you have typed in the control box for a case, then that case will be executed. If the control matches none of the cases, the default case is executed. If you don't have a default case (for a text or numeric control), then you will have a broken arrow. For my own projects (not FRC robots) I like to use case structures controlled by text strings, in which case I make sure there is a unique case for each possible string value. Then I make the default case match the string "Typo", and if that case ever executes, it pops up an error message telling me that I've mistyped something and showing me what it is. Of course, I could name the default case anything, since all mistakes will match it, but naming it "Typo" reminds me what the code does.

To see why you have broken arrow syndrome, click on the broken arrow.

Flow Control

If you have any programming experience, then you are already familiar with For Loops (called Do Loops in some languages) and While Loops. Each of them will repeat the same block of code over and over. A For Loop repeats a specified number of times. A While Loop repeats until some logical condition is satisfied. To give you flavor of how these two loops are similar and how they are different, we'll work through a couple of examples. I'm going to throw a bunch of other useful stuff at you too, so pay attention.

For Loops

We'll start with a very simple example: filling an array with 100 random numbers, as illustrated in Fig. 1.14. To make this program, I carried out the following steps. You should do this yourself now.

1. Make a new blank VI.
2. From the Structures palette, draw a For Loop on the block diagram
3. Right-click on the loop count (the blue "N" in the upper left) and create a constant. Set the value to 100.
4. From the Numeric palette, drop the random number generator inside the loop.
5. Using the Wiring Tool (the spool), wire the ouput of the random number generator to the right-hand side of the loop. Be careful to click exactly on the edge of the loop, and not outside the loop. If you ignored my advice and are using a track-pad instead of a mouse, expect this to be difficult. I have no sympathy.
6. If you did step 5 correctly, the random number generator is wired to what is known as an *output tunnel* on the right-hand edge of the loop. The tunnel should be a square box with a pair of square brackets nestled inside it. This particular tunnel is an *indexing tunnel*. On the inside of the loop, the wire carries a single real number. Wires connecting to the tunnel outside on the loop are arrays, as you will discover in Step 7.
7. Right-click on the output tunnel and create an indicator. It will have the default name "Array" which you can keep or change.

That's it. Run the program to fill the array with 100 random numbers between 0 and 1. On the front panel, discover that you can stretch the array indicators to show more than one element in the array. You can stretch down or to the right. Also discover that if you do it wrong, you stretch the size of the individual elements instead of showing more elements. Figure out how to use the index window to show the 100^{th} element (without stretching the array to show all 100 elements).

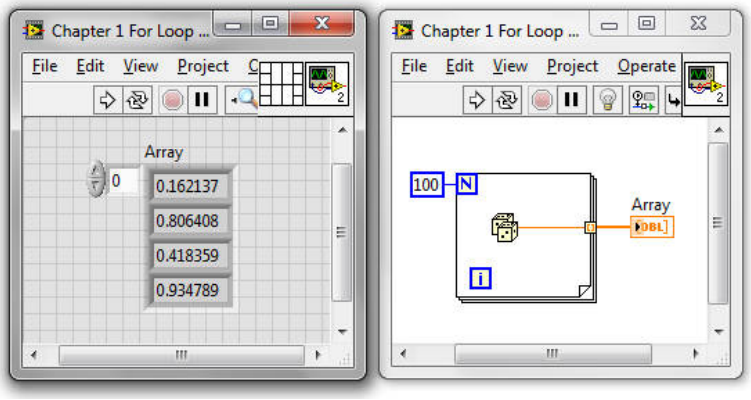


FIGURE 1.14 A simple program that fills an array with 100 random numbers.

Looking back at the block diagram, notice how the wire connecting the tunnel to the indicator is thick. This type of wire indicates a one-dimensional array. You will notice this in other cases: not only the color, but the pattern and the thickness of wires tell you what type of data they carry.

Now, before we go on to While Loops, we're going to introduce something that has nothing to do with loops, but is very useful in robotics: graphing data. When you are trying to sort out sensors, controls, and whatnot, it can be incredibly useful to throw up a graph of what is going on in real time. This will give you an idea how to do it. (Less useful in competitions. This stuff is really for debugging and understanding...)

Go to the front panel of your random number generator, and, from the graph palette, drop in both a Waveform Chart and a Waveform Graph. You can stretch these to resize them, change the axis labeling, go from automatically scaled axes (the default) to fixed axes of your choice, plus all kinds of other stuff, which I am going to leave you to figure out on your own. You just need to know that like everything else, these follow the Zen of LabVIEW, so things that work on other objects probably work in the same way on these.

On the block diagram, wire up the graphs exactly as shown in Fig. 1.15. If you put them in the wrong place, you will get broken wires. (Actually, Waveform Chart is smart enough to figure out how to behave if you wire it up on the "wrong" side of the loop. Waveform Graph is not.)

Finally, put a 100 ms delay inside your loop using the Wait(ms) function from the Timing palette (you can find it!). This delay serves no purpose except to slow things down enough to show you how the two different graphs behave. Run the program and observe the two graphs. We are illustrating the difference between a Waveform Chart, which accepts one point at a time, using the point number as the *x*-axis, and a Waveform Graph, which wants the entire array of *y* values at once. The *x*-axis is still the point number (index) of each y value.

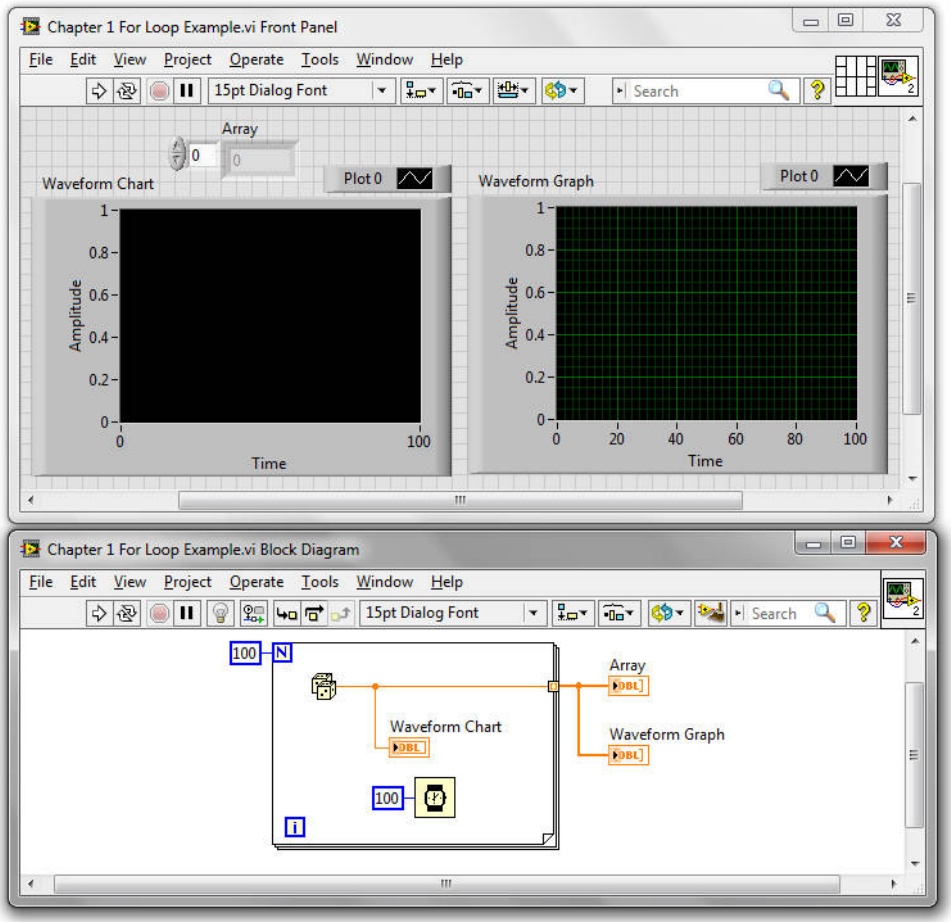


FIGURE 1.15 The random number program with two different kinds of graphs added.

There is a third kind of graph available, the XY Graph. This wants *two* arrays (bundled together in a cluster), one containing the *x* value of each point, and one containing the *y* values. We used one in Figs. 1.3 and 4.

The second important point being illustrated by this little program is the behavior of output tunnels. The data in the second graph does not appear until the loop is finished. That is because the data values in output tunnels are not available until all the code inside the structure has finished executing. I called it a "structure" instead of a "loop" on purpose. In addition to loops, all of the similar looking objects on the Structures palette (the ones that get drawn as rectangles on the block diagram) can have both input and output tunnels. No code inside a structure will execute until all the input tunnels have valid data present. No output tunnel will have valid data until all the code inside the structure has finished executing. Pay attention to these rules. Otherwise, your robot may hang waiting for some data input to become valid when you would much rather it be picking up the whatsamajig and placing it inside the doohickey.

Exercise E1: If you right-click on the output tunnel in the random number program, one of the available options is a sub-menu called "Tunnel Mode". Explore the difference between

"Indexing" and "Last Value". Hint: for more clarity, wire the loop index (the blue "I" in a box) to the tunnel instead of the random number generator.

While Loops

Draw the program illustrated in Fig. 1.16. There is a lot of new stuff here introduced all at once, so we'll take it one step at a time. First, notice the red stop sign like object in the lower right corner of the loop. This is the Conditional Terminal. It is currently configured for the condition Stop If True. On the front panel you have a slider control that can have any value between 0 and 10. On the diagram, this control is inside the loop, which means that the slider value will be read each iteration of the loop. There is also a single real value, the Threshold, which is read once when the program starts up. Once you've got it drawn, run it until you have a good sense of how it works.

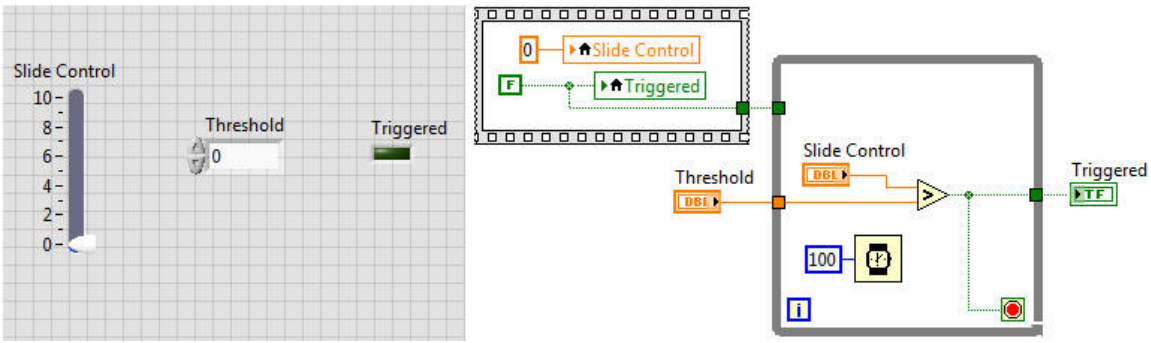


FIGURE 1.16 A simple example of a while loop. Note the structure on the left used to initialize variables.

Also in the block diagram is a "trick" to initialize the slider value and the Trigger Boolean. As long as a VI remains in memory, its internal variables maintain their values. If you did not include this initialization code, you would have to manually reset the value of the Trigger Boolean and drag the slider below the threshold value before restarting the program. The two assignments (to local variables) that initialize the Slider and Trigger are inside a Sequence structure. This kind of structure allows you to enforce the order in which events take place. Usually such a structure has multiple panels. Here we have only used a single panel, because all we are doing is using the rules about when code in structures can execute to enforce an order of operations. The output tunnel on the Sequence won't have valid data until both assignments have been made. The While Loop can't start until that output data from the Sequence is made available at its input tunnel. Note that we don't even do anything with that tunnel! The connection is only made as a way to force the order of events. Now, as with many things in LabVIEW, there is more than one way to get the job done. You could ask "why not just use a sequence, since it is designed to control the order of events?" That would work very well: draw a sequence structure with two panels. Put the variable initializations in the first panel and the rest of the code in the second. I'm showing you the wire trick because it is very useful, and often allows a more compact and cleaner way to accomplish the same thing.

You may be asking yourself "why the 100 ms delay inside the While Loop?" Well…perhaps you were not asking yourself that, but you *should* have been, so just take a moment to feel embarrassed. OK. That's enough embarrassment. But the delay is very important. This loop spends 99.99% of its time waiting for an input. In this case, it is waiting for you to

change the slider value. On a robot, it could be waiting for a switch to close, or some voltage to reach a particular value. But people and mechanical systems are slow, while computers are fast. Without the delay, the loop will run as fast as it possibly can, consuming all the available processor time and bringing your computer to its knees. Or it would, if you were not running some modern, robust, multi-threaded operating system on a state of the art, high-powered, quad-core processor. That loop would completely crush a computer from, say, the ancient year 2000. It would also completely crush the cRIO computer on your FRC robot, which is a relatively puny little thing, and which has an operating system that is not as well defended against that kind of basic error.

Exercise E2: Modify the While Loop example so that it operates correctly using a Continue If True termination condition.

Exercise E3: Modify the While Loop example so that you can change the threshold while the program is running.

Exercise E4: Modify the program to use a Sequence structure instead of the wire trick to force the initialization to happen before the loop starts.

Exercise E5: Remove the initialization code and explore how the program behaves without it.

Exercise E6: Re-write the For Loop example of Fig. 15 so that it uses a While Loop instead.

Shift Registers and Feedback Nodes

Before we leave the subject of loops entirely, let's take a moment to re-visit the I = I + 1 problem. Start by drawing the program illustrated in Fig. 1.17. The big, clunky tunnels with the up and down arrows are actually a single object, called a Shift Register. Whatever value is written to the right-hand side in one iteration of the loop is available to be read from the left-hand terminal on the next iteration. You create this object by right-clicking on either the left or right side of the loop and selecting **Add Shift Register**.

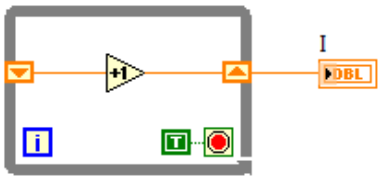


FIGURE 1.17 Another implementation of I = I + 1.

Note that a True constant has been wired to the Conditional Terminal of the loop. That will cause the loop to execute just one iteration each time the program is run (a While Loop always executes at least once). Run the program and look what happens to the value of I. Run it multiple times to verify that the value of I keeps incrementing. (Remember, variables and their values remain as long as the program remains in the computer's RAM memory.)

Exercise E7: Right-click on the left-hand terminal of the Shift Register. Select **Create ▸ Constant**, and set the value of that constant to something other than zero. Now run the program multiple times. Explain the behavior.

Finally, draw the program shown in Fig. 1.18. The arrow thing is a Feedback Node, and it is found on the Structures palette. Verify that it behaves just like the Shift Register-based program of Fig. 1.17.

These examples are kind of trivial, but Shift Registers and Feedback Nodes are essential in robot programming. We will see a lot of them later.

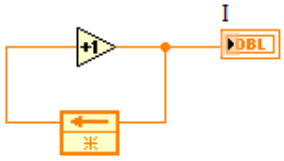


FIGURE 1.18 And yet another implementation of I = I + 1.

Sequences

Consider the simple program in of Fig. 1.19. When you run it, what is the value of C at the end? If you are thinking "left to right", you might guess -1. If you think about program execution time, you might guess 1. Both guesses are wrong. The correct answer is "this is Very Bad. You should *NEVER* do this!" This program has what is known as a Race Condition. LabVIEW looks at these two independent bits of code floating free on the block diagram, and feels perfectly free to execute them in whatever order it feels like. Actually, this is one of the most powerful features of LabVIEW. What it will really try to do is execute both pieces of code at once, or "in parallel" if you want to sound cooler. It will not just run the two pieces of code in parallel, but will try to optimize the compiled code so both bits complete as quickly and efficiently as possible. That sounds a bit ridiculous if all we are talking about is something as simple as Fig. 19, but you will see later in Chapter 4 that your robot will rely on multiple while loops, all doing complicated stuff, running in parallel. This is complicated to do in other programming languages, but trivially easy in LabVIEW.

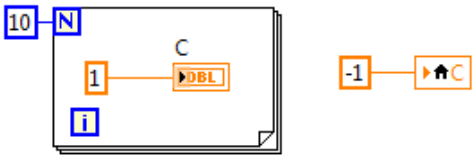


FIGURE 1.19 A race condition: C could be assigned either 1 or -1.

If you care about the final value of C in programs like this one, then you have to control the order in which things happen. There are lots of ways to do this (like the dummy wire trick of Fig. 16), including the Sequence structure (found on the Structure Palette with While and For loops). There is no equivalent in text-based languages because the structure of the code already is a sequence structure. If you put one line of code above another in your text file, you can be sure that the first line executes first, and the second line executes second.

Fig. 20 shows two different versions of the same idea. Both use a Sequence structure to ensure that the For loop completes before the value -1 is assigned to C. On the left is a flat Sequence in which you can see everything at once. On the right is a stacked Sequence that shows only one step at a time. You change the view with the little control at the top, just as with a Case structure.

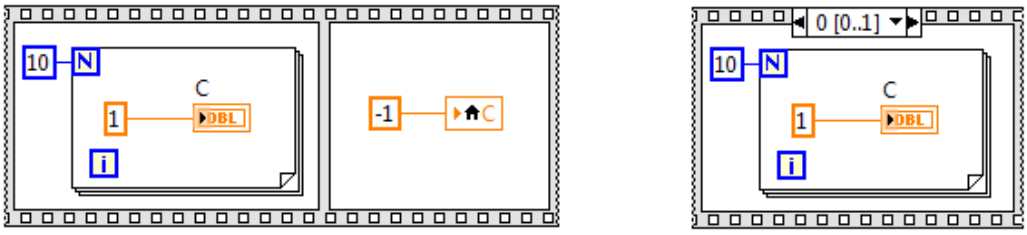


FIGURE 1.20 Preventing a race with a flat or stacked sequence.

Now that I've introduced them, I should tell you that you really want to avoid Sequence structures like the plague in your robot. They can be useful, but your primary means of controlling the order in which things happen will be a state machine, the subject of the next chapter.

Subroutines

We have come to the last item on our list of programming language "must haves". Subroutines serve two important functions. If there is some piece of code you use repeatedly, and at different places in your code, you can put that code inside a subroutine, and both make your code easier to read, and reduce the size of the compiled object. The code is compiled at one location in memory and re-used, although how this reduces the size of the program is a topic for a different book. A second important function is simply to make your code easier to read and easier to debug. If you have some complex piece of code that is a logically thought of as a single operation, it makes sense to package it as a subroutine or function (there is no distinction in LV), even if you only use that code at one location in the program.

As an example, we'll use an utterly simple piece of code that is useful in one of the homework problems at the end of the chapter. Fig. 1.21 shows a VI that takes a real number as in input, divides it by 100 (converting cents to dollars), and formats it into a string. It should take you no time at all to make your own version of this program, which you should do now.

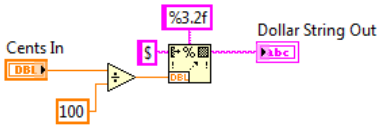


FIGURE 1.21 Block diagram of the Cents2DollarString VI.

Once you've saved this little program (using a sensible name like "Cents2DollarString.vi" that will tell you its function a year from now), it is just that, a program. To make it usable as a sub-VI (LabVIEW's name for a subroutine), you have to do some additional work. Look at the front panel. In the upper right corner are two squares. One is a generic LabVIEW icon for the program. The other is a square composed of a bunch of smaller squares and rectangles. This is the *connector pane* for the program. (In older versions of LabVIEW, you have to right click on the icon to get to the connector pane, but you should have the latest and greatest version from your team's Kit of Parts.)

If you place the mouse cursor over the connector pane, it will change to a wiring spool. Click on a square to use as an input terminal, and then click on the control that you want connected to that terminal. In this little example, there is only one control to choose, so choose it. Repeat the process for the output: click on a terminal in the connector pane, then click on the string indicator. The panes will acquire a color that shows the type of the data

that moves in or out through them. Just as variables in LV are either read or write, but never both, terminals are inputs or outputs, but never bidirectional.

When you pick a terminal to serve as an input or an output, remember the left-to-right flow of LabVIEW. The terminals on the left edge should *always* be wired as inputs, and the terminals on the right edge should *always* be wired as outputs. LabVIEW will *not* enforce this, so it is not a real rule, and you are free to violate it. You should however, think of it like spilling your lunch all down the front of your clothes. No one will stop you from doing it, but it's kind of embarrassing, and people will laugh at you. The (unwritten) rules for terminals in the middle, along the top and bottom edges of the connector pane are more relaxed. In some cases, I've seen terminals to the left of the center line wired as inputs, and terminals to the right wired as outputs. In other cases, all terminals along the top edge were wired as inputs, and all the ones along the bottom were outputs. You may encounter situations where, for example, you need all the middle terminals to be inputs. That's OK. Just don't break the rules along the left and right edges.

In the current situation, we have a program with only one input and only one output. The default connector pane, however, has a total of 12 terminals. If that strikes you as silly, right-click on the pane and select Patterns to choose a different arrangement of terminals (like the pattern that has only two terminals).

Once we've defined the inputs and outputs, we need to change the icon to something useful. When you use this subroutine, all you will see in your diagram is this icon, so it should immediately tell you which subroutine it is and give a clear idea of its function. Right-click on the icon to bring up a dead-simple bitmap editor. It is possible to spend a lot of time producing very fancy looking icons. But that takes time, and build season is very short! I recommend a very simple text icon, as shown in Fig. 1.22. I also recommend changing the font in the icon editor to 11 points from the default of 10. It makes the text actually readable. You only need to do this once and LabVIEW will remember your preference.

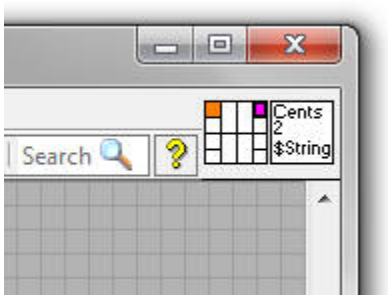


FIGURE 1.22 Icon and terminal pane of Cents2DollarString.vi.

To use the subroutine, create a new blank VI, right-click on the diagram and choose "Select a VI…" Point the dialog box at your saved subroutine, and drop a copy in the diagram. Use the right-click creation method to create a control and an indicator, and you are ready to go.

Exercise E8: Use the Cents-to-Dollar-String VI you just created as a subroutine in another VI. (Because I know you didn't do it yet.)

Clusters

We have covered everything on my list of "must haves" for a programming language, but there is one thing we should cover before we move on: clusters. The equivalent construct in C is a structure. It a way to build up a single object that contains a random assortment of other objects, like variables, arrays, and even other clusters. I've mentioned these already, but you really do need to understand them in order to code your robot, and they are full of LabVIEW Zen.

Fig. 1.23 shows a random collection of objects connected to a Bundle node, found on the Cluster, Class, & Variant palette. You can see the cluster contains a few Boolean controls, a real control, a text constant, an array of integer controls, and a real constant. The pink spotted wire that emerges from the node *is* the cluster, which I have wired directly to an Unbundle node. The Bundle node is one of these objects that only has two connection terminals when you first drop it on the diagram, and you drag vertically to re-size. The Unbundle node, on the other hand, will automatically resize itself to give you an output terminal for each item in the cluster. Note that in order to correctly wire to the outputs on the Unbundle node, you have to know the top-to-bottom order in which things were wired to the Bundle node. Otherwise, you have no idea which of the three Boolean outputs corresponds to which input.

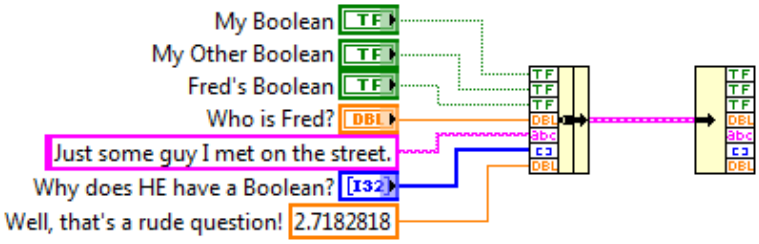


FIGURE 1.23 The Bundle and Unbundle nodes for manipulating clusters.

Well, there is an Easier Way. Also on the same tool palette is the Unbundle By Name node. If you were to wire that up to the cluster instead of the plain old Unbundle, you would get six (after you did a vertical drag, because you only get one showing when you first wire it up) output terminals, each colored and named to match the type and name of an input, so you would have no trouble knowing how to wire it up. Count carefully. There are seven inputs in Fig. 1.21. Why would there only be six outputs? Look again at the seven items. Five of them are controls, which automatically have a name, and the real constant apparently had a name you didn't know about. The text constant does not automatically. But you can give it one, and also change the name of the real constant. Right-click (what else?) on one of them, and select **Visible Items ▸ Label**. Type a name, and it can appear in the outputs of an Unbundle By Name node. No name, no output.

If you go back to the Cluster, etc. palette, you will see that there is also a Bundle By Name node, and in Fig. 1.24 I've used it so that the un-named text constant can be passed through the cluster and unbundled by name.

The naming is no longer controlled by the inputs. It is controlled by the cluster constant. This input is required, your program will not run unless you have wired this input of Bundle By Name. Each of the constants inside that cluster has a name (label), which I hid again after entering, just to make the diagram tidier. You can see from the list of names in the cluster

that they don't have to match the names of the inputs. They can be anything you want. You can see from the Unbundle By Name node that you are free to reorder the outputs in any way you want. You can do the same with the inputs on the bundling node.

Finally, note that the input with the names does not have to be a constant. It can be a cluster full of valid data. If you need to change the value of just one or two (or more) of the elements in the cluster, wire it to the input of a Bundle By Name, make the node show only the values that need to be changed, and wire in the new values. They will replace the original values in the output cluster.

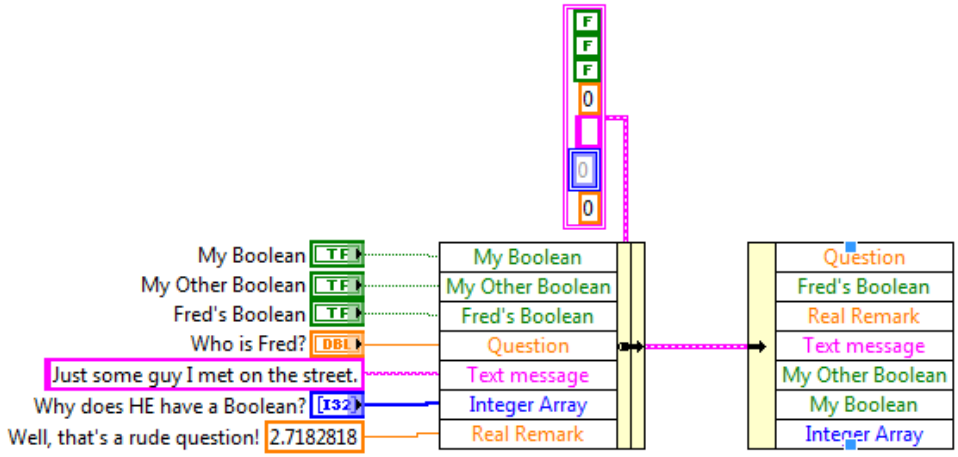
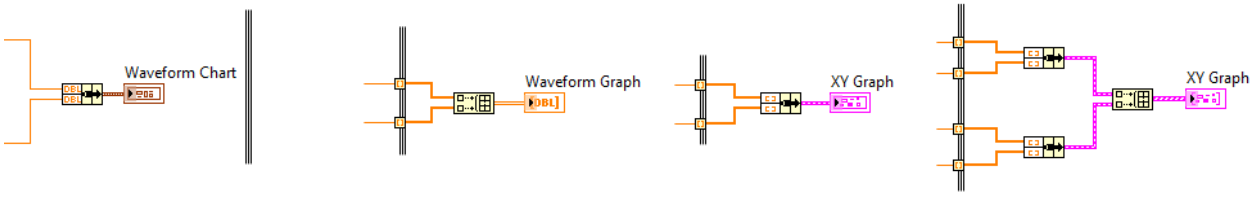


FIGURE 1.24 Bundling and unbundling by name.

Graphing More

At some point you may find it useful to plot more than one thing on a chart or graph. Many things in LabVIEW are intuitive. Many others become intuitive once you understand the Zen of LabVIEW. Not graphing multiple lines on the same plot. It's random.

A few pages back I explained that there are three kinds of graphs: Waveform Charts, Waveform Graphs, and XY Graphs. The two "waveforms" have no *x*-axis data. They are arrays, and the index of the array serves as the *x*-axis. For an XY Graph, you need two arrays, one for the *x*-axis, and one for the *y*-axis. They need to be the same size (obviously). Fig. 1.25 shows how to wire each of these cases to put two curves of data onto the same chart or graph. Extending to more than two is straightforward.



(a) (b) (c) (d)

FIGURE 1.25 Wiring more than one curve of data to charts and graphs. In every case there is a snippet of a For Loop showing. To the left of the snippet is inside the loop, and to the right is outside.

Part (a) of the figure shows how to put multiple curves on a Waveform Chart: You cluster them together with a Bundle node. Remember that the terminal for this chart goes *inside* a

loop, because it expects the data one point at a time. Part (b) of the figure shows how to do this for a Waveform Graph, which wants all of the data at once in an array. For multiple curves, use a Build Array node (from the Array palette). Make sure that Concatenate Inputs is not selected for this node. (What's that? Right-click on the node…) Based on the rule for Waveform Chart, you might expect that the arrays should be clustered together, but you would be wrong…

Part (c) shows how to wire up a *single* curve for display in an XY Graph. As in the waveform case, "graph" means that you need to have all the data on hand before you can plot, so the graphing terminal is outside your loop. The *x*-axis data gets wired to the *upper* input of the Bundle node. Finally, part (d) shows how to plot multiple curves on an XY graph. Each curve is bundled into a cluster, and then the clusters are gathered together in an array. Each curve has its own, independent *x*-axis (which could be the same for all the curves, or not: they're independent). The curves do not have to have the same number of points (that independent thing again).

Problems

Problem 1.1 – There are three "essential" logic operators on the Boolean palette: AND, OR, and XOR (exclusive or). Write a VI that reads two Boolean controls (A and B) and simultaneously displays the output from all three operations. It should be clear on the front panel which indicator is showing the result of which operation.

Problem 1.2 – Modify the program you wrote for Problem 1.1 to include the result of one more logical operation: Take the output of (A AND B) and XOR it with the output of (A XOR B). Arrange the front panel so the output of this new operation is easy to compare with the output of (A OR B).

Problem 1.3 – Create a VI with four Boolean controls and one *integer* indicator. Name the controls Bit 0, Bit 1, Bit 2, and Bit 3. Write a program that reads the four controls, treats them as a four bit binary number, converts the result into an integer, and writes it to the indicator. For example, if only Bit 0 is set, the indicator should get a 1. If only Bit 3 is set, the indicator should get a 4.

Problem 1.4 – Write a program in which you use Bundle By Name to change the value of one element in a cluster with several elements.

Problem 1.5 – The built-in random number generator in LabVIEW returns numbers uniformly distributed between 0 and 1.

a) Write a subroutine that, when called, returns a *uniform deviate,* which is to say, a random number uniformly distributed between -1 and 1. (Hint: it uses the built in generator and a bit of math.)

b) Write a program that generates an array filled with 10,000 uniform deviates, and then generates a histogram of the data in the array. Plot the histogram. Your histogram should have 50 intervals, and the *y* axis of your plot should go from 0 to 250. (Hint: There is a VI on the Probability & Statistics palette that will do the work of generating the histogram from the array. The output of this VI wants to be connected to an XY Graph.)

Problem 1.6 – Write a program that fills a 100 by 100 array with uniform deviates.

Problem 1.7 – Combine your solutions to Problems 1.5 and 1.6 in the following way: write a program that fills an array with 10,000 random numbers, but the random numbers are calculated by finding the average of 100 uniform deviates. Make a histogram of these 10,000 random numbers, using 50 intervals as in Problem 1.4. (Hint: There is a VI on the Probability & Statistics palette that will find the average of the values in an array.) Are these 10,000 random numbers uniformly distributed? (If you are curious about what is going on, mathematically speaking, look up the Central Limit Theorem.)

Problem 1.8 – Figure out how to use the timing functions to tell you the execution time of your solutions to problems 1.6 and 1.7.

Problem 1.9 – Write a subroutine that implements the Box-Mueller algorithm, which is as follows:

1. Choose two uniform deviates, u_1 and u_2.
2. Perform the following test: calculate $s = (u_1)^2 + (u_2)^2$. If t is either zero, or ≥ 1, reject these deviates and return to step 1. Otherwise, proceed to step 3.
3. Calculate
$$g_1 = u_1 \sqrt{-2 \frac{\ln(s)}{s}}$$
4. Return g_1 as the output of your subroutine.

(The clever student will notice that we are neglecting an important efficiency: we can actually obtain *two* numbers from a single call to the Box-Mueller routine, g_1 as above, and g_2 calculated using u_2 instead of u_1 in step 3. Of course, I haven't told you what this routine does, because you will find out experimentally in Prob. 1.10, but you might guess if you've done Prob. 1.7.)

Problem 1.10 – Copy your solution to Problem 1.7, and instead of averaging 100 uniform deviates, make a single call to your Box-Mueller subroutine. Compare the output and execution time of the two programs.

Problem 1.11 – Write a program that continuously reads a slider, knob, or other front panel control, and sets a Boolean indicator True if the control goes above some threshold, and False if the control goes back down through the threshold. The threshold value should be settable on the front panel while the program is running. There should be a button on the front panel that will stop execution of the program when it is pressed.

Problem 1.12 – Write a program to find all the prime numbers less than *n* (settable from the front panel) using the algorithm known as Eratosthenes' Sieve:

1. Create a list of consecutive integers from 2 to n: (2, 3, 4, ..., *n*).
2. Initially, let *p* equal 2, the first prime number.
3. Starting from *p*, count up in increments of *p* and mark each of these numbers greater than *p* itself in the list. These numbers will be 2*p*, 3*p*, 4*p*, etc.; note that some of them may have already been marked.
4. Find the first number greater than *p* in the list that is not marked; let *p* now equal this number (which is the next prime).
5. If there were no more unmarked numbers in the list, stop. Otherwise, repeat from step 3.

Problem 1.13 – Write a subroutine named Timeout.vi. It has two inputs, which I will call Limit In and Start Time In. They should be un-signed 32 bit integers. There are two outputs: a Boolean (called Timeout) and an un-signed 32 bit integer called Start Time Out. This subroutine should read the Tick Count (one of the LabVIEW timer functions), and subtract the value of Start Time In. If the result of this subtraction is greater than Limit, then Timeout should be set to True. Start Time Out should just be the value of Start Time In. Use the default connector pane, wiring Limit to the upper left corner, Start Time In to the lower left corner, Timeout to the upper right corner, and Start Time Out to the lower right corner. (This VI will come in handy programming your robot. Trust me.)

Chapter 2 – State Machines

Buckle your seatbelt. State machines are probably the most important concept and programming technique for robotics, so it is really important that you understand them. But they are a little bit complicated, and they require lots of LabVIEW Zen. So…buckle your seatbelt.

What is a "state machine"? The "machine" is your program. When it is controlling your robot, then the machine is the hardware/software combination, but for the purposes of understanding, you can focus only on the software aspect. This program spends all of its time in "states", which are bits of executable code. Each state does a minimum of two things. It (1) carries out some useful operation, like running a motor, and (2) it checks to see if some event has occurred that will cause it to transition to another state. In many cases, the "useful operation" will be to do nothing: the state's only function is to wait for an event to occur.

To make this abstract idea concrete, we're going to build a state machine that pretends to launch a rocket. The program will have five states:

1. **Ready** The program waits for the launch command.
2. **Countdown** The program counts down the seconds to launch.
3. **Fire** The program fires the rocket engine for five seconds.
4. **Engine Stop** The program turns off the rocket engine.
5. **Abort** Code that is executed if the Abort button is pressed.

Planning

The first step in creating this program is *not* to start programming! The first step is to create a *state diagram.* This is a diagram that shows, at a very high level, what it is your program is supposed to do, and it contains a rather important idea: you should know what your program needs to do *before* you start writing it! The more thought and clarity that you can put into the state diagram, the easier it will be to write the code. Of course, I can hear you now: "Yeah, yeah. He's just some old fuddy-duddy, and I'm a hip, roboteer, and I can *write code.*" So you will be tempted to skip this first step. Don't. Robots are complicated, and winning robots will have well thought out, cleanly designed software, the kind that only comes out of a process that begins with state diagrams and *flow charts.*

Fig. 2.1 shows the rather simple state diagram for our rather simple state machine. (The planning documents for a competition robot might have 10 or 20 pages of state diagrams.) I like to make these in PowerPoint, because it has nice tools for making the drawings, and you can project it, which makes working as a team much easier. You can see that each of the five states is represented by a little oval. The arrows represent events which take the machine from one state to another, or return it to the same state. These arrows that do nothing but return the machine to the state that it is already in may seem silly to you, but when we get to the way the code works, you will see that those arrows have real meaning.

The Ready state just waits for the Launch button to be pressed. The default action is to return to the same state. If the Launch button is pressed, then the program does not return to the ready state, but moves to the Countdown state. This is an example of a state transition

being triggered by an event that is external to the program. In this case the user presses a button, but we could equally well be waiting for a sensor to reach a certain value.

The Countdown state decrements a counter and compares the result with zero. If the counter is greater than zero, then the default action is to return to the same state. The "event" in this case is internal. The program itself defines the event that triggers a state change.

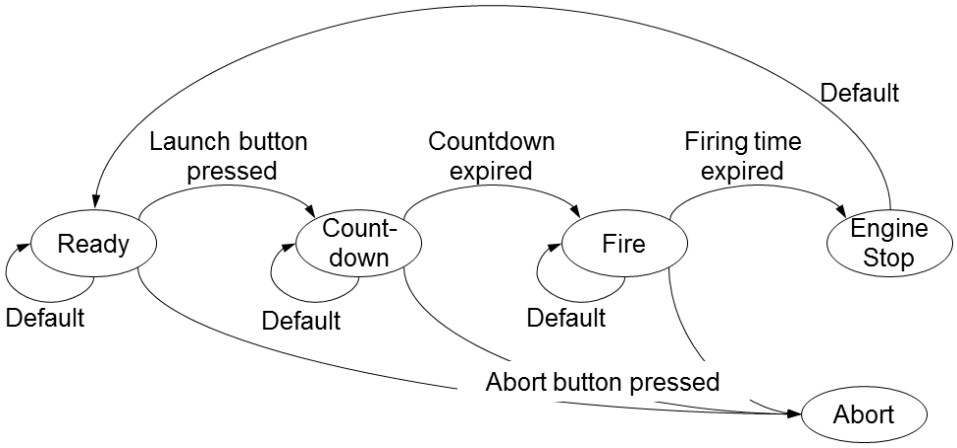


FIGURE 2.1 The state diagram for our rocket launcher example.

The Fire state is very similar to the Countdown state. We're decrementing a counter, and when it reaches zero, the program moves to the Engine Stop state.

The Engine Stop state is trivial. It automatically transitions back to the Ready state, so technically, it is not even needed. The Fire state could transition directly to the Ready state, and that would not be a wrong thing to do. It was a judgment call on my part to decide that a separate state to turn off the rocket engine made for a cleaner, easier to understand program.

Finally, you can see that the first three states also monitor the Abort button. If the button is pressed, the program transitions to the Abort state and stops execution.

Having drawn our state diagram, we can proceed to making flow diagrams. You need one for every state, so Fig. 2.2 has five separate ones. If the actions within a state are really complex, you might not be able to fit everything on one page (PowerPoint slide), but if that is really the case, maybe you should divide the action of that state into two or more states. The simpler a given state is, the easier it will be to debug.

The standard convention in flow charts is to put Yes/No (True/False) decisions in a diamond, with flow coming in one vertex and out one of two others. Assignment statements and other actions the code performs are enclosed in rectangles. There are many more conventions than this, but I wouldn't get too caught up in those details. The important thing is to be clear, so that you and your teammates can read them and tell what your program is supposed to do.

As an example of how to read these diagrams, consider the Ready State in Fig. 2.2. When the code that makes up the state starts executing, the first thing that happens is a check to see if the Abort button has been pressed. If it has, then we immediately transition to the Abort state. If not, then we check to see if the Launch button has been pressed. If has not, then we return to the Ready state (the code will actually exit the Ready state, and then re-enter it, as you will see in a bit.) If the Launch button has been pressed, then we load the Count varia-

ble with the number 10 (because we will count down for 10 seconds) and transition to the Countdown state.

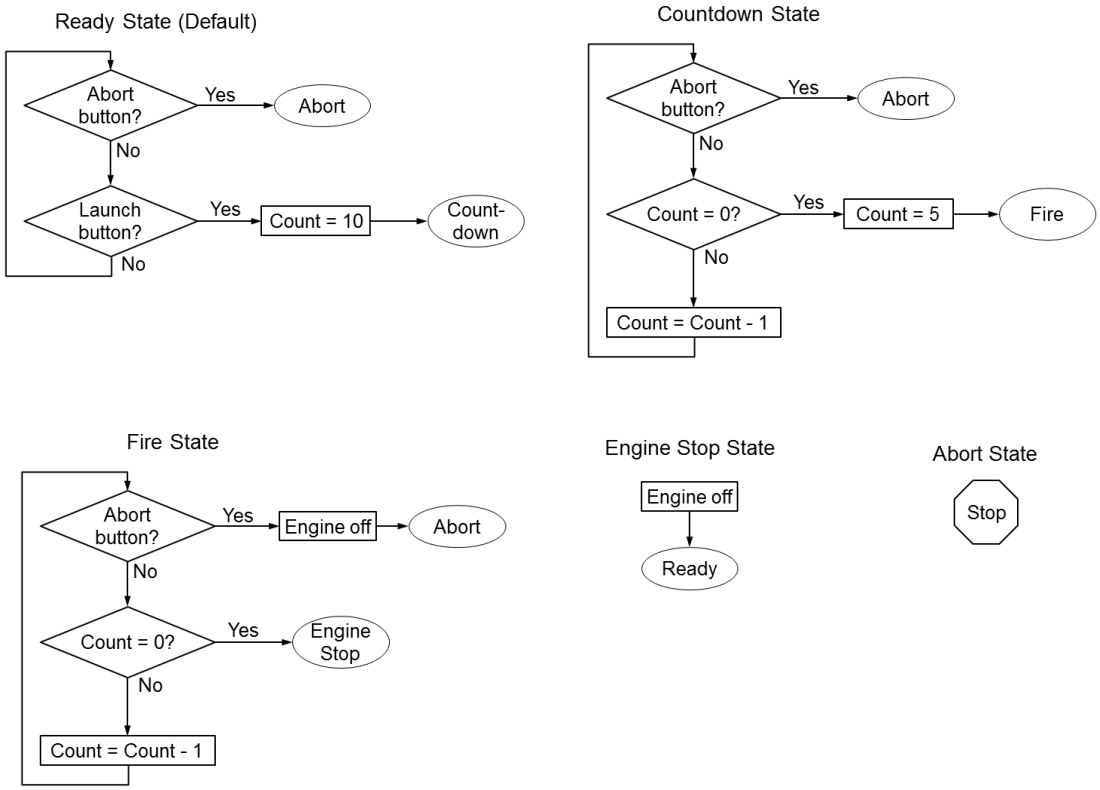


FIGURE 2.2 Flow diagrams for each of the five states in our rocket launching state machine.

Programming

Before we get down to actual programming details, let's get a high-altitude look at a state machine. The heart of the state machine is a Case Structure inside a While Loop that has a Shift Register, as shown in Fig. 2.3. Each case of the Case Structure is a state in the state machine. On each iteration of the While Loop, the value stored in the Shift Register is used to select which case

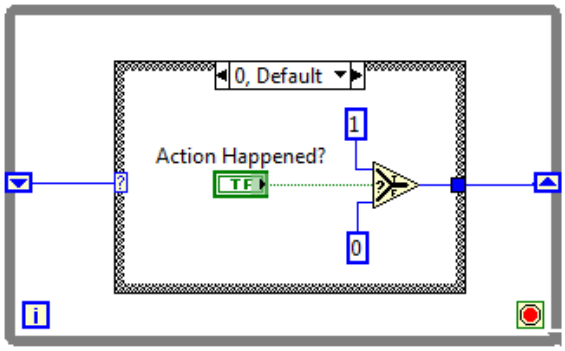


FIGURE 2.3 The basic structure of a state machine.

(state) executes. As that code runs, one of its tasks is to write a value to the Shift Register, a value which will be stored and read on the next iteration of the loop. The program shown in Fig. 2. 3 doesn't do anything useful (in fact, it won't run because I haven't wired the Conditional Terminal of the loop), but it shows the basic structure.

If the Boolean variable named Action Happened? remains false, then a zero is written out to the right-hand terminal of the shift register. On the next iteration of the loop, this zero will be applied to the case selector and cause this same state 0 to execute again. If the Boolean is true, then the value 1 is written to the shift register, and state 1 will execute next.

In practice, it is extremely difficult to keep track of your states by number. It is much easier to give them names, and as we get into the details, you will see how we do that (although the fundamental control is still by integer values, they are just hidden behind the names).

As you can see, the essential programming structure for a state machine is the case structure. A state machine that does something elegant may have a lot of cases. The quickest way to move through this structure to find a particular case is to put the mouse cursor anywhere inside that case structure, hold down the control key, and scroll your mouse wheel.

Now we are ready to write the program that implements Fig. 2. 1 and its dependent flow diagrams. We'll start with the fundamental control element, the one that defines our states (and allows us to call them by name instead of remembering numbers). Start by creating a new VI, and then drop an Enum control onto the front panel (from the Ring & Enum palette). "Enum" is short for "enumerated". It is an integer data type, with the value of the integer associated with a readable text string. The Text Ring and Menu Ring controls are very similar, but for controlling a state machine, it has to be an Enum.

Rename your Enum to something informative like "Launcher State". Then, right-click on the control and select **Advanced ▸ Customize…** This will open up a copy of the Enum in a new window that looks like the front panel of a VI, but you will see that it doesn't have a block diagram. Before you do anything else, look along the tool ribbon at the top of the VI panel and find where it says "Control". Press the little arrow to the right and select "Typedef" from the drop-down list (see Fig. 2.4). This is a very important step, and things won't work right if you don't do it.

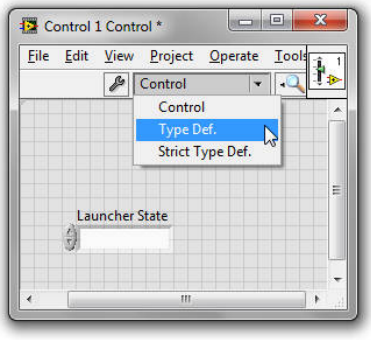


FIGURE 2.4 Making a Control into a Type Definition.

Next, right-click on this new copy of the Enum, and select **Edit Items…** This will pop up a window in which you can type your five states, as shown in Fig. 2.5. Click OK to close this window, and then save this new control. You should save it in the same location as you will

be saving the Launcher program, and you should probably give it the same name you gave to the control when you created it (Launcher State?). After you have saved the control, close it. When you do that, you will get a pop-up message asking if you want to replace the original control with the new one. You can answer yes or no, because the next thing you are going to do is discard that original control. Close the VI that you first dropped the Enum control onto, *without saving it.*

I'm not going to tell you how to be organized and disciplined in the way you save your files. I'm only going to point out that you will eventually be sorry if you are not.

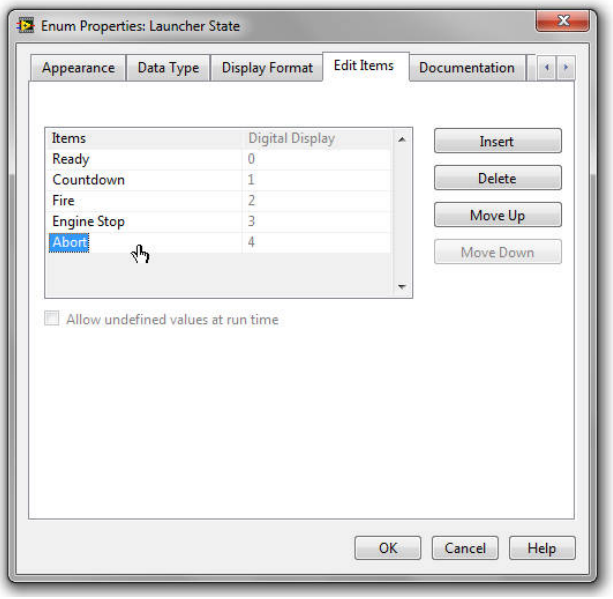


FIGURE 2.5 Assigning names to the integers in the Enum.

Open a new VI and make the front panel look something like Fig. 2.6. There are two push-buttons (Boolean controls) and a real indicator. "Engines" is a Boolean indicator. I've jiggered the colors so that the False state is black and the True state is red, and made a whole bunch of other font and sizing changes. You can figure out how to do this kind of thing for yourself.

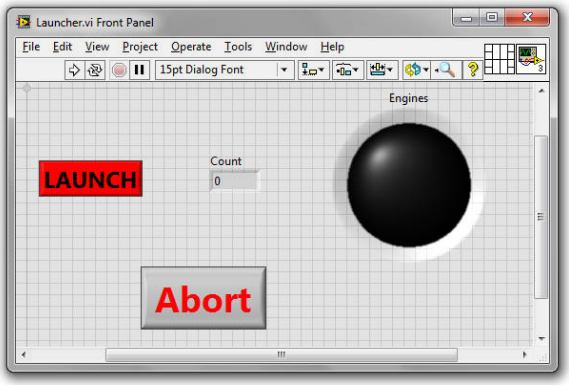


FIGURE 2.6 My front panel for the Launcher. Yours may look different.

Now switch to the block diagram, where we will (finally) begin to build the actual state machine. Here are the steps:

1. Draw a While Loop. Make it plenty big: approximately square, and close to the full height of your screen. This will make for a very "roomy" VI. Later on, your programs will be more complex, and you will find value in creating a very compact layout. (For now, just group the four terminals of the front panel objects together an drag them to one side. We'll position them later.)
2. Right-click on the left-hand edge of the loop, and select **Add Shift Register…**
3. Right-click on a blank space in the diagram and choose **Select a VI…** From the dialog that pops up, select the new control you just created (it will have the extension .ctl), and wire it as in Fig. 2.7, so that the first state the machine enters is the Ready state.

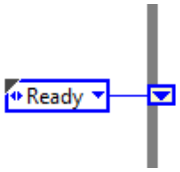


FIGURE 2.7 Wiring the state control to the shift register.

4. Grab the terminal for the Abort button and put it inside the While Loop, close to the left side. You are doing this now to help you size things right in the next step. (As you go through the next few steps, you may find it useful to look ahead to Fig. 2.8.)
5. Draw a Case Structure *inside* the While Loop, but *do not include the Abort terminal* inside the structure. The structure should mostly fill the While Loop. Don't make it so it jams right up to the edge of the loop, but leave a comfortable margin all around. The Abort terminal should give you an idea of the right margin size.
6. Wire the left-hand terminal of the Shift Register to the select terminal of the Case Structure. Watch in awe as it automatically changes from a True/False structure, to one with two cases with names that come from your state control variable. The cases of this Case Structure *are* the states in the state machine!
7. Make sure the case that is showing is *not* the default state. Right-click on the name of the case (probably "Countdown"), and select Delete This Case. This may seem crazy, but it is a trick to save you time. For now, your case structure should only have one case, the Ready case.
8. Refer back to Fig. 2.2. We need to convert the flow diagram for the Ready state into actual code. The first thing that happens is a True/False decision based on the Abort button. So draw a case structure *inside the Ready case*, and wire the Abort terminal to its selector. If you look at the flow diagram, you will see that everything that happens in this state is "under" this case, so it should pretty much fill the Ready case.
9. Make sure the True case is showing in the structure you just wired to the Abort terminal. Put a copy of your state control inside this case. The easiest way to do this is to "control-drag-copy" the one you added to the diagram back in Step 3. Hold down the Control key, click on the existing control, and drag it to make a copy. Then click on the little down arrow on the control and select the Abort case.
10. Wire the control to the shift register as shown in Fig. 2.8. Note the tunnel that appears as a white square. This is an error (which we will fix in a moment). It is telling

you that you have not wired to this output in every case. You must provide a valid output for every case in order for the program to run. Note the 100 ms delay, which every program that waits for human input should have!

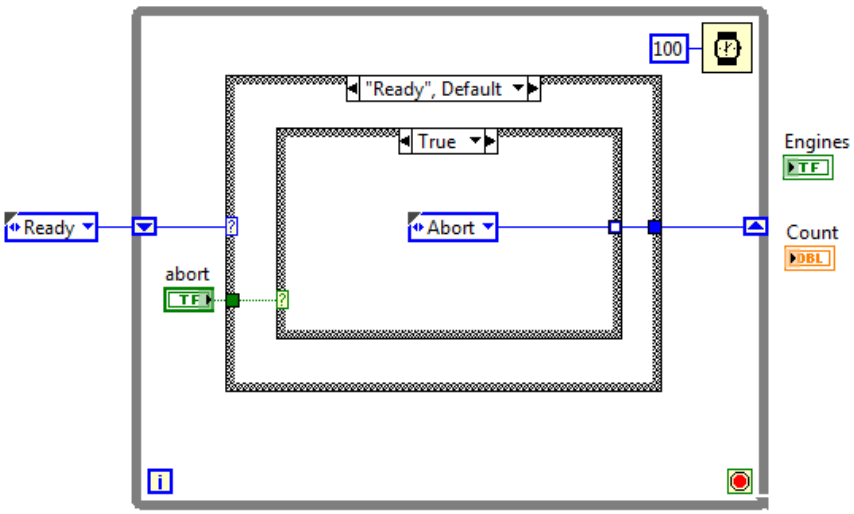


FIGURE 2.8 Wiring the Ready state to handle the Abort button being pressed.

11. Switch the inner case to the False value (Abort button not pressed), and wire it as shown in Fig. 2.9. The constant loads the Count variable with the number of seconds we will count down for. Note that some of the output tunnels that the orange wire passes through have a little white dot in the center. These have been set to **Use Default If Unwired** (via a right-click, of course). *NEVER* do that with the tunnels connected to the state control shift register! You always want to be controlling what state the program goes to next.

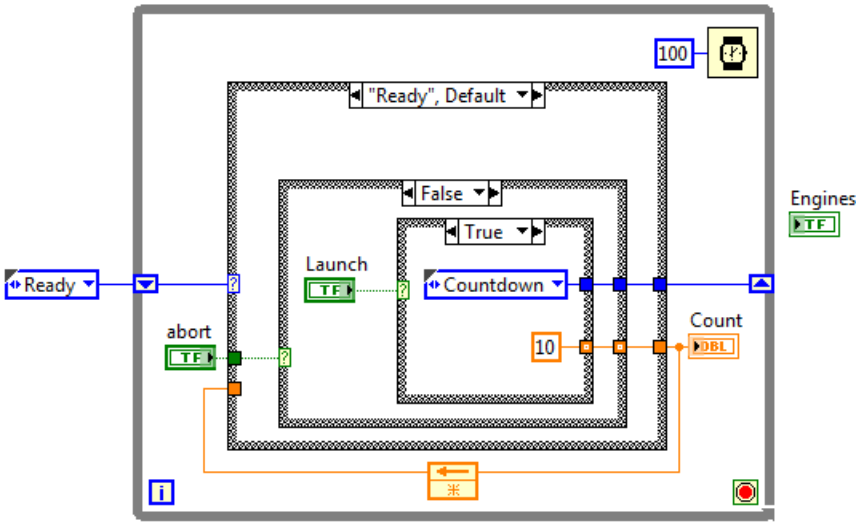


FIGURE 2.9 Wiring the Ready state to handle the Launch button being pressed.

12. In this step we will do no programming, but pause to reflect on what the program is going to do. The While Loop will run continuously (until the Abort button is pressed, but we haven't written that code yet). On each iteration, the loop will take the value of the state control (stored in the shift register), and apply it to the main Case Structure. The Case Structure will then switch to that case, and execute the code in that case. That code will *always* include a copy of the state control variable that is wired out to the right-hand terminal of the shift register. In other word, the executing code in the present state determines what case/state will be executed on the next iteration of the While Loop. If you look back over the last few steps, you can see that we have written code in which the Ready state will force either the Abort state or the Countdown state to be executed on the next iteration of the While Loop.
13. If you refer to flow diagram, you will see that we have not yet coded the case where neither button is pressed and the state machine just returns to the Ready state. Do that now. (Note the lack of instructions. You have to figure this one out yourself.)
14. Right-click on the word "Ready" on the main case structure and select **Duplicate Case**. This will create the Countdown state for you, mostly drawn.
15. Duplicating the case created a copy of the Launch button (called Launch 2). This is a typical penalty you have to pay for using this shortcut. Simply delete Launch 2 now.
16. Set the innermost Case Structure to show the False case and wire it to look like Fig. 2.10. Now, instead of reading the value of the Launch button, we are checking to see if the count has reached zero. If it has not, then we decrement the counter by 1 and return to the Countdown state. Note the new timer which makes the state last for a full second (so that we are counting down by seconds, as is traditional when launching rockets).

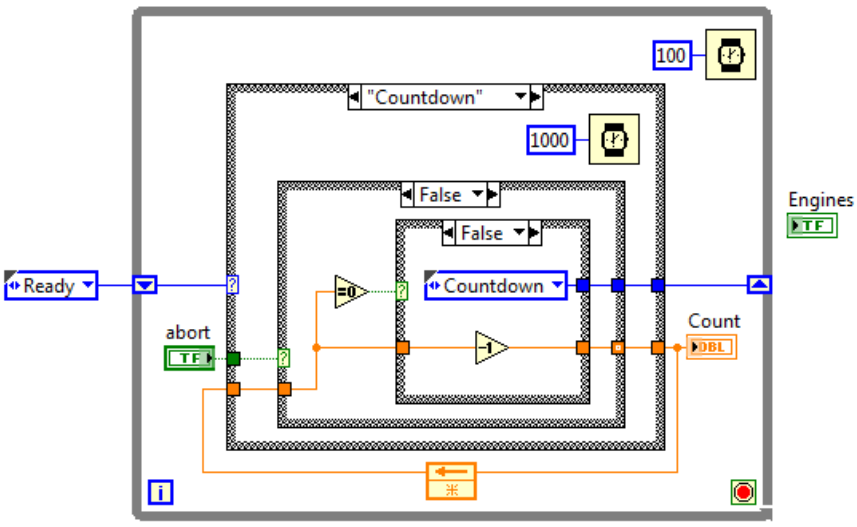


FIGURE 2.10 Wiring the Countodown state to check if we have counted all the way down.

17. If the count *is* equal to zero (True on the innermost case), then we should load the Count wire with 5 (because we will fire the engines only for five seconds), and transition to the Fire state. I will leave you to figure that code out for yourself.

18. Choose Duplicate Case again to create the Fire state as copy of the Countdown state. This time we don't make pesky copies of controls that need to be deleted.
19. Here we want to stay in the Fire state if the count is not yet zero. If it is zero, we want to go to the Engine Stop case. And when we do, we don't need to load any new value into the Count. Make it so.
20. Of course, you need to actually fire the engines. The terminal for that indicator has been left sitting around outside the loop. Now you need to bring it inside and wire it up in such a way that it gets the value True when the engines are firing, and the value false otherwise. Figure out how. (Hint: Default If Unwired.)
21. Duplicate Case again to create the Engine Stop case. Now we no longer look at the Abort button, so select the case structure that reads it, and delete it. This wipes out the entire contents of the case.
22. This case only needs to set Engines to False, and transition to the Ready state. Go ahead and write that code yourself.
23. Duplicate Case one last time to create the Abort case. All this case needs to do is send a True value to the While Loop conditional terminal (without catching Broken Arrow Syndrome!). Since we are stopping the loop, there will not be a next state. But we still have to wire *something* to the state control tunnel or the program won't run. Wire a copy of the state control with the value set to Abort. Once you make that last connection, you should no longer have a broken arrow and the program should be ready for testing.

Test the program. Does it work? Do the Engines come on when you think they should? (Hint: the 1 second delay in the Countdown and Fire states probably should not be placed identically.) Fix everything until you are satisfied that you have done a professional job and would not be embarrassed to show your program in public.

Exercise E9: Replace the feedback node in the Launcher example with a shift register.

Adding a Timeout

Unfortunately, our little rocket launcher is a somewhat artificial example, but that is the nature of the beast. In fact, in the next bit, we're going to make it even more artificial, so that we can discuss an essential element of robot design. I have mentioned several times that often a state's main, or even only function is to wait for something to happen. If you look at the Ready state in our rocket launcher, it is a state that waits only for a button to be pressed. But what if the thing you are waiting for never happens?

To make the question more concrete, consider a robot that shoots things (a fairly popular FRC robot task). You might have a belt system that delivers the thing to the shooter mechanism, and a sensor that detects the passage of the thing from the belt to the shooter. The program might have a state called Run The Belt. The robot stays in this state until the sensor detects the thing has been shot, and then goes to another state, allowing whatever needs to happen next to happen. Now suppose the thing jams, or the belt comes loose, and the detector never trips. Then the robot is stuck in this state until the end of the match. You need an out: some kind of mechanism that ends the state anyway, and allows you to do something else with the robot.

In our example, if you start the program, and then go eat lunch, the program will simply sit there patiently, waiting for you to come back and press a button. Let's make the program impatient. If the user hasn't hit either the Launch button or the Abort button within 10 seconds, the program will "time out." Since we are going to modify our code, the first thing we should do is update our documentation. Fig. 11 shows an updated flow chart for the Ready state. Now this state checks to see if a timeout has occurred before it checks the buttons.

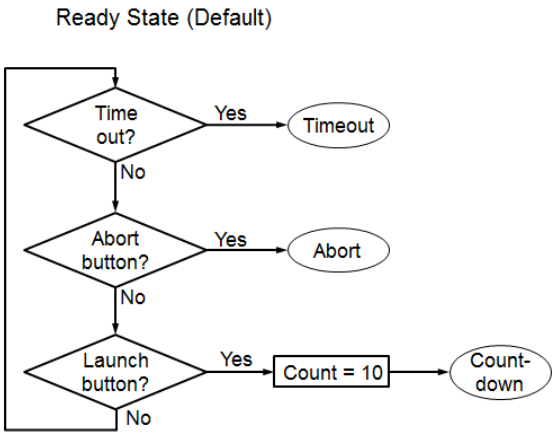


FIGURE 2.11 Flow chart for the Ready state with a time out.

The rest of this example will proceed assuming you have completed Problem 1.13. If you haven't, go back and do that now. Then you can come back here and finish.

1. On your block diagram, right-click on any of the copies of your state control and select **Open Type Def.** When that opens up, right-click on the control itself and select **Edit Items…**
2. Add a new state named Timeout to the list. You can add it at the end of the list, or you can insert it somewhere in the middle of the list. When you are done, save the control and close the window that popped up. If you check, you will see that the new state is present in every copy of the state control (because we made it a Type Definition, way back when we created it).
3. Modify your Ready state to match Fig. 2.12. There is a new case structure that encloses the one that reads the Abort button. I'm showing the other case (which you should make sure is the True case!). The False case contains the code you wrote before. Note the use of the **Use Default If Unwired** tunnel on the way out of the case structure.
4. What is going on? When the program starts, it reads the millisecond clock, and stores that value in the shift register. Your Timeout VI reads the millisecond clock every iteration of the loop. If the difference between the current reading and the stored reading exceeds 10,000 ms, the state machine transitions to the Timeout state. This state is used to tidy up whatever loose ends need to be tidied up before returning to normal operations. In our example, there's nothing to do.
5. We're sending the state machine to the Timeout state, so that state better exist. Switch the case structure so the Engine Stop case is showing, and duplicate this case. It will automagically be named Timeout. Change the state control so that the machine is sent to the Abort state next.

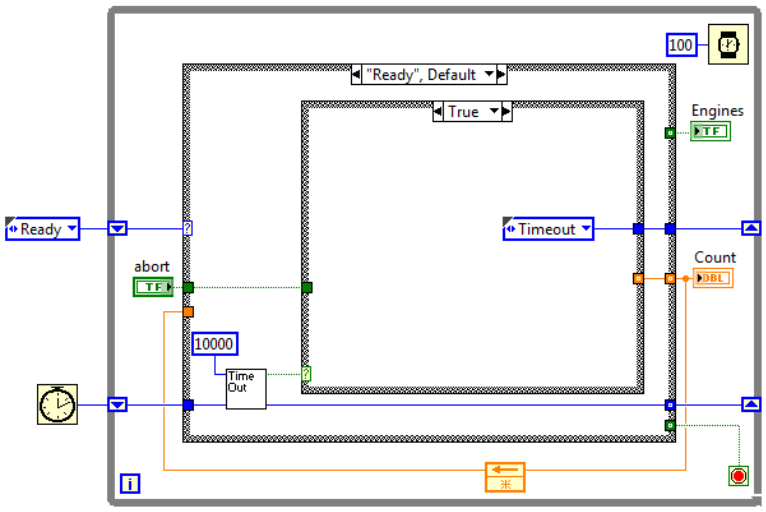


FIGURE 2.12 The Ready state with a timeout test added.

6. We have one last step. The Engine Stop state transitions to the Ready state. When we do that, we want to re-start the 10 second timeout. So read the millisecond clock and feed that new value to the shift register, as shown in Fig. 2.13.

If you think about it, I've now shown you two methods to get a state machine out of trouble: use of a timeout, and the use of a manually pressed abort button. Both are available to you as you design your robot. The abort button has immediate effect (or can have, see Prob. 2.1), but the operator has to remember to press it. The timeout is automatic, but not immediate. Each approach will have its place. A third approach is the use of a "dead man switch" where you stay in a state only as long as a button is held down, which has some of the advantages of both.

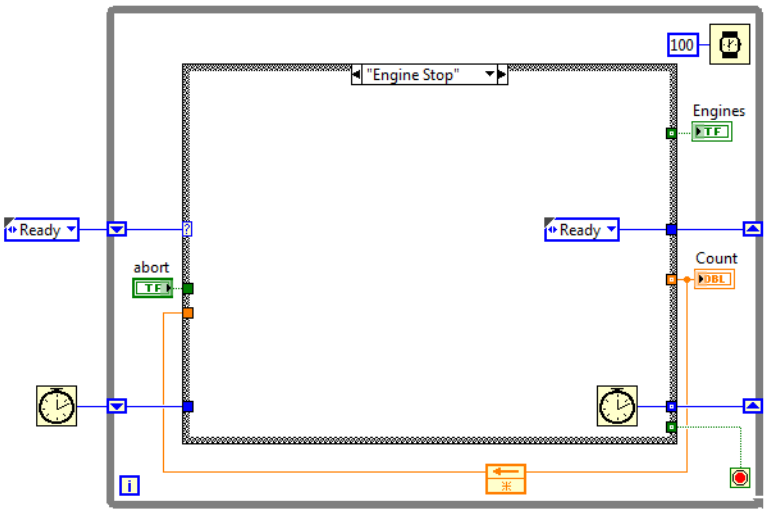


FIGURE 2.13 The Engine Stop state with a reset of the 10 second timeout.

Delays

If you play with the Rocket Launcher example, you will notice that the Abort button is unsatisfactory. There is a delay of up to a second before the countdown stops or the engines shut

off. This is an important problem to be solved. It's not important for our little example, because our little example is not important. But you may have a situation where some aspect of your competition robot code needs to delay for, say, a second. If you put in that delay like we have done here (as in Fig. 2.14), you will have a problem. All of your code will hang, including the code that lets the driver control the robot, until this *blocking call* completes. If the driver happens to have the robot going at full speed when the delay starts, it will keep going at that full speed (with no steering possible) for the full second of the delay. Exactly why this will happen won't be clear to you until we've discussed the guts of the FRC Robot Framework, so just trust me that it will. (OK. So that was a lie. Your robot won't keep going for a full second. There is a safety system that will disable your robot. The end effect is the same for you, though. You won't gain control of your robot until your delay has elapsed.)

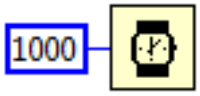


FIGURE 2.14 The millisecond delay timer is a blocking call. No structure containing it can complete until the timer expires.

So, let's modify the Rocket Launcher so our one-second delays are implemented in a "nice" way. This may strike you as a non-trivial modification, and indeed, it can get a little complex. But it is important. So we start with new flow charts, as in Fig. 15. To implement these diagrams, open up your Rocket Launcher and follow these steps:

1. Add a Shift Register, and load it with the millisecond clock as the state machine transitions from the Ready state to the Countdown state, as in Fig. 2.16. Note the use of Use Default if Unwired tunnels.
2. Switch to the Countdown state. You will need more space inside the innermost case structure. A quick way to get this is to hold down the control key, and drag a rectangle where you want the extra space to appear. LabVIEW will expand your diagram to clear that rectangle. Be aware that it is doing this to all layers of your program, so you may have some cleaning up to do afterwards. (I usually stretch things by hand when I know the control-drag method will affect a lot of things I can't see.)

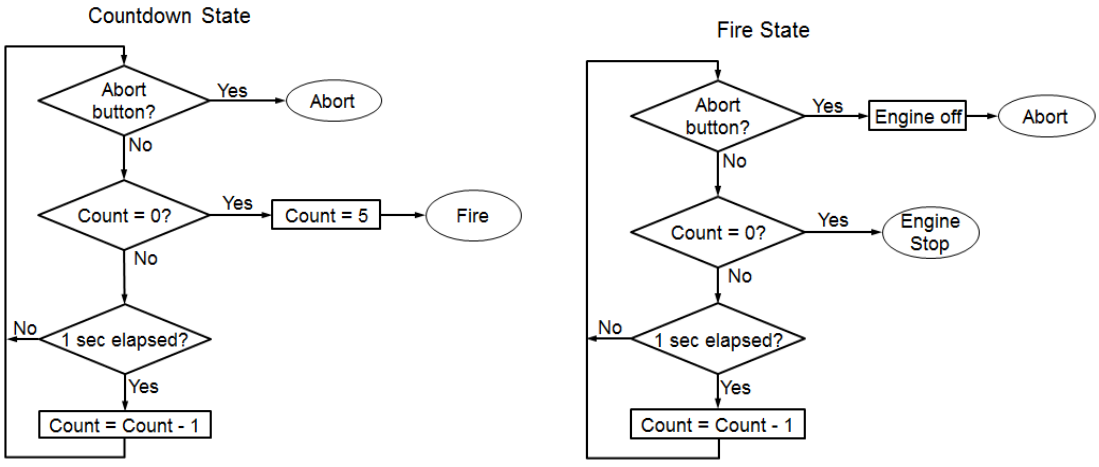


FIGURE 2.15 New flow charts for the Countdown and Fire states.

3. Use the Timeout VI you made earlier, and a new, innermost case structure to give the code shown in Fig. 2.17. The case showing is what we do when the 1 second timeout has expired: we decrement Count by 1, and stay in the Countdown state.

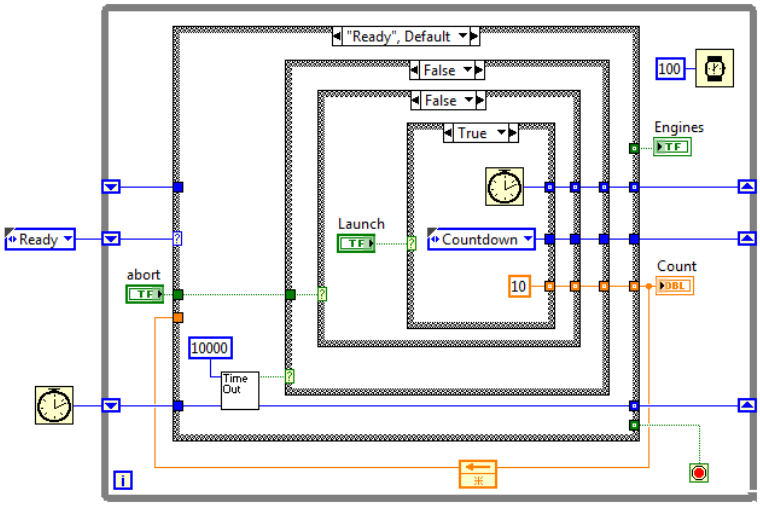


FIGURE 2.16 The Ready state showing the new Shift Register being loaded with the current time.

4. Fig. 2.18 shows the other case: what we do when the timer has not yet expired. We remain in the Countdown state, but we don't decrement Count. Instead we just pass the input through to the output.

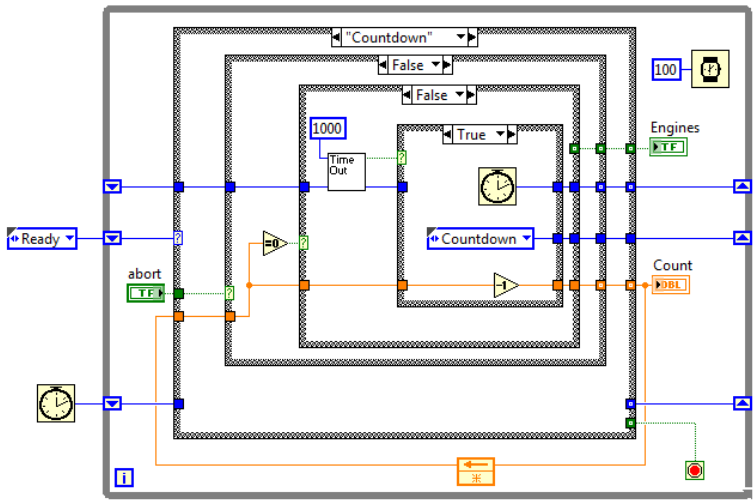


FIGURE 2.17 The Countdown state showing the code to handle expiration of the 1 second timeout.

5. Now consider the case that executes when Count reaches zero. Fig. 2.19 shows just that case structure with the True case visible. Note that we need the timer one more time, because we have to mark off the "zero" second in the countdown. So, to go with the Timeout VI, there is again a new, innermost case structure. The True case

showing corresponds to the 1 second timer expiring. We set the Engines Boolean to True, reload our timing shift register with a new start time (because we will be using the same delay in Fire), load the value 5 into Count, and transition to the Fire state. The False case (not shown) does nothing but keep us in the Countdown state. We don't need to keep track of the value of Count, because we are just leaving it at zero until the 1 second timer expires.

6. Modify the Fire state in the same way we modified Countdown in Steps 3 and 4. You don't need the extra second (the one added in Step 5).

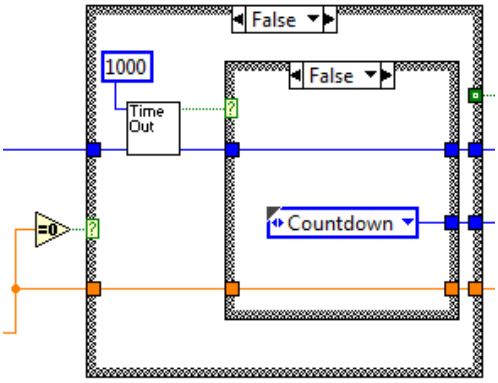


FIGURE 2.18

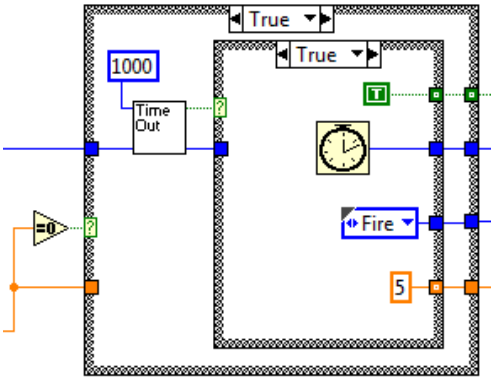


FIGURE 2.19

That's it! If you've done it right, the Abort button should work instantly (or within 100 ms, which is near enough to instant).

Problems

Problem 2.1 – I have claimed that a While Loop that waits for human input should have a 100 ms delay so as to free up the CPU for other tasks. Write a program to test your reaction time. It should, of course, use a state machine. There should be a "Go" button, a random delay of some length (with a minimum delay of a second or two) and then a visual signal. The program should report how long (in milliseconds) it took the user to press a button in response to the signal. The machine should automatically reset to the state where it waits for the "Go" button.

Problem 2.2 – Modify your solution to Problem 2.1 so that you accumulate and display a histogram of reaction times.

Problem 2.3 – Build a state machine that mimics the performance of a robot action for 3 seconds. The action could be anything: running a motor, or closing a gripper. For this program, the "action" will be turning a front panel Boolean indicator True for 3 seconds. The action should be initiated by a front panel button press (modeling the pressing of a button on a robot joystick controller. The action should complete only once for a single button press, no matter how long the button is held. In order to trigger the action again, the operator needs to release the button and then re-press it.

Problem 2.4 – Build a state machine the mimics the performance of a robot action completely under operator control. As in the last problem, the "action" is setting a front panel Boolean indicator True. The indicator should remain True for as long as a front panel button is pressed, and should go False within 100 ms of that button being released.

The next three problems deal with the simulation of a machine that dispenses Drool Cola. A can of Drool costs $1.25. The machine accepts nickels, dimes, quarters and dollars. It has a display that shows how much money has been input, and can also display other messages. For example, it displays a message indicating that it has enough money for a purchase, or the amount of change due, or the amount being returned if the user hits the cancel button. If the user inputs some money, but then does nothing else, after 30 seconds the machine gives up and returns the money. When there is no action, the display might have a message that tries to entice passerby into purchasing a Drool.

Problem 2.5 – Design the state diagram for a soda machine to dispense Drool Cola. Hint: this machine is cyclical: it starts out in a waiting state, and returns to that state. If your state diagram has a generally circular layout, it will be clean and easy to read.

Problem 2.6 – Draw flow diagrams for each state of your Drool Cola state machine.

Problem 2.7 – Code your Drool Cola machine.

Chapter 3 — Introduction to the cRIO

It is difficult to discuss programming an FRC robot without some discussion of the hardware. At the same time, this book is about software, not hardware. So I am going to do the minimum here, and assume that I can talk about, for example, encoders without going into the details of how you hook one up. (In Chapter 5, when we discuss programming for individual devices, I'll point you to where you can find examples that include connection instructions.)

The overall hardware scheme for an FRC robot is shown in Fig. 3.1. Your code runs on the cRIO (shown in Fig. 3.2), which is a very small computer built into a chassis. (cRIO is not an invitation to visit Brazil, but stands for "compact real-time input/output", which says a lot, or very little, depending on your point of view.) The slots in the chassis accept a wide range of plug-in modules from National Instruments. In FRC, we use only one for digital input and output, one for analog inputs, and one to drive relays.

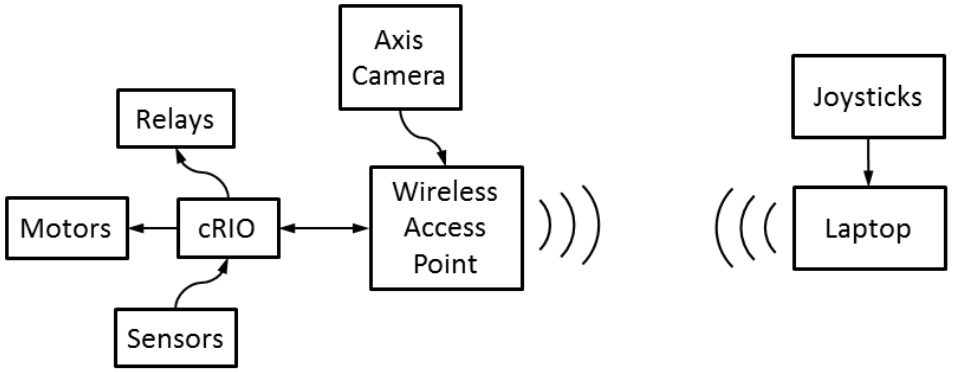


FIGURE 3.1 Schematic diagram of an FRC robot control system.

The cRIO is connected by a cable to a wireless access point (commonly called a WiFi router, or just "the radio"). You will both program and control the robot through this communication link, which can be made either by WiFi or a direct cable connection. (At a competition, only actively competing robots get to use WiFi. All your programming changes and runs on the practice field have to be done via a cable, so be sure to own a long one.)

(a) (b)

FIGURE 3.2 (a) The cRIO. An earlier version has eight slots, two Ethernet ports, and some other extras. In (b) you can see Rio.

In addition to the code you write for the cRIO, there are several other software components, some of which you will need to write, while other parts are already written for you. Fig. 3.3 is the counterpart to Fig. 3.1, but for the software scheme. Notice that your code is not some independent unit, but resides inside something called the FRC Robot Framework.

As it turns out, actually programming a robot from zero is an enormous task—too big for an FRC build season, or even a full year of effort. Fortunately, the heavy lifting has been done for you by the FIRST Robotics Resource Center at Worcester Polytechnic Institute, in collaboration with National Instruments. Their Framework consists of both a high-level architecture, and a large number of low-level subroutines (Vis). Your code will be written as an intermediate layer: you will write VIs that use the low-level routines, but are in turn called by the high-level architecture. I have never met any of the people behind the Framework, but I'm extremely grateful for all their effort, and you should be too. Because of the Framework, you get to focus on solving the problem posed by the competition, rather than worrying about how to read a USB port, interpret the data as a joystick position, and somehow get that data over the WiFi and into your program.

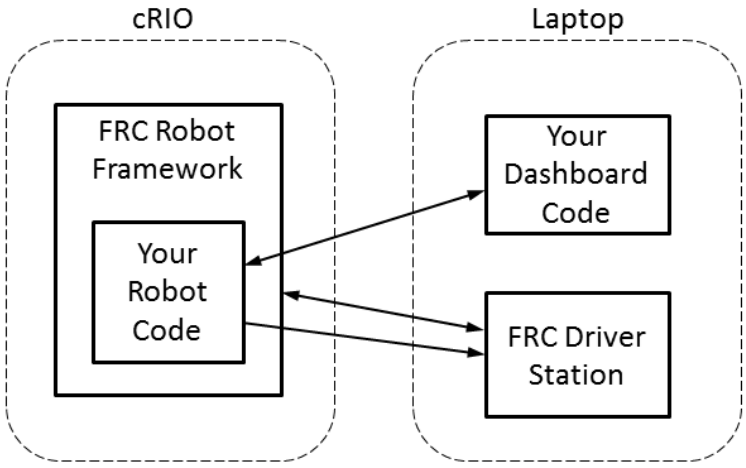


FIGURE 3.3 The FRC Robotics software scheme.

Note that the FRC Driver Station communicates bi-directionally with the Robot Framework, but only uni-directionally and with your code. This allows you to display some very basic, pre-determined diagnostic data without writing a custom Dashboard. As that remark implies, FRC LabVIEW includes a default Dashboard that you can use if you don't need to do anything fancy. If, on the other hand, you need to do image processing or other serious computation, you can use that default code as a starting point and build your own Dashboard. (Dashboard use and programming are covered in Chapter 8.) In Fig. 3.1, you will notice that the camera is connected to the radio. Images from the camera are beamed directly to your laptop so you can process and/or display them on the Dashboard without involving the cRIO, which does not have the horsepower to both run the robot and do serious number crunching.

LabVIEW Projects

Your robot's program will be big. You will write at least several VIs, and the Framework itself includes hundreds of subroutines. In order to assemble all this stuff into a working program and get it loaded onto your robot requires some serious organization. LabVIEW's

mechanism for this is the "project", and the associated project file, which would be named, *e.g.* MyBigProject.lvproj.

If your version of Windows is not displaying file extensions, then you will not be able to tell the difference between MyRobot.vi and MyRobot.lvproj. You should fix that now. If you don't know how, find someone who can show you.

As usual, the best way to learn is by doing. If your copy of LabVIEW is not currently running, start it now. If it is running, close all your open VIs so that you go back to the start screen (Fig. 1.1). Under **New**, click on **FRC cRIO Robot Project**, which should bring up a window as shown in Fig. 3.4.

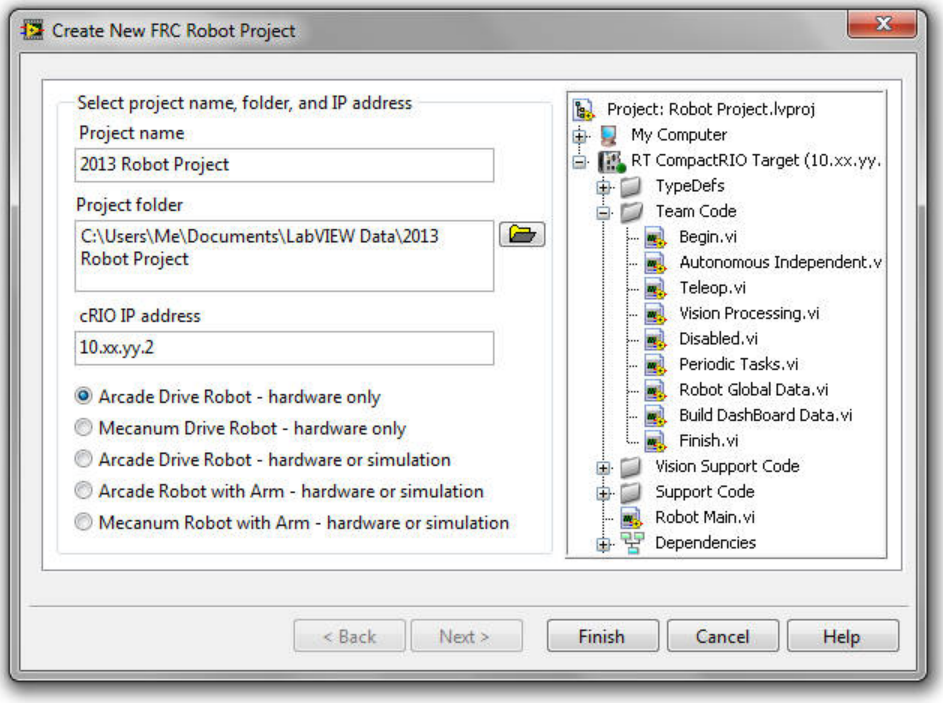


FIGURE 3.4 The Create New FRC Robot Project window.

Now that the window has popped up, you have three things to do.

1. Give your project a sensible name. This will be the file name on the hard drive, so pick something meaningful and comprehensible. For now, "Practice Project" might be a good choice. Later on, you might have a name that makes it clear that this is *the* project for your competition robot.
2. Choose a location for the project. This one is non-trivial if you have plans for more than one person to work on the code, and they won't all be doing it on the same computer. I've developed a strategy that works for me and the team I work with, but it might not be right for you. For a practice project, there is no problem accepting the default, but put some thought into how to share code across multiple users before build season starts.
3. Enter your cRIO IP address. If your team is FRC number 1234, then replace the xx with 12 and the yy with 34. I am going to assume that you have somehow learned—

although not from me—how to set up all your IP addresses, image your cRIO, etc. (This sort of information can be found on the FIRST web site.)

4. (There is a Monty Python joke lurking here.) Choose the type of drive train you want. Your choice doesn't matter as you can change it later. For the purposes of this book, choose a "hardware only" option as we will not be discussing the LabVIEW simulation feature at all.

When you've finished these tasks, LabVIEW will grind away for a while and eventually produce a "project tree" like the one shown in Fig. 3.5. Your window won't look exactly like mine because I've opened several of the folders so we can see and discuss what's inside.

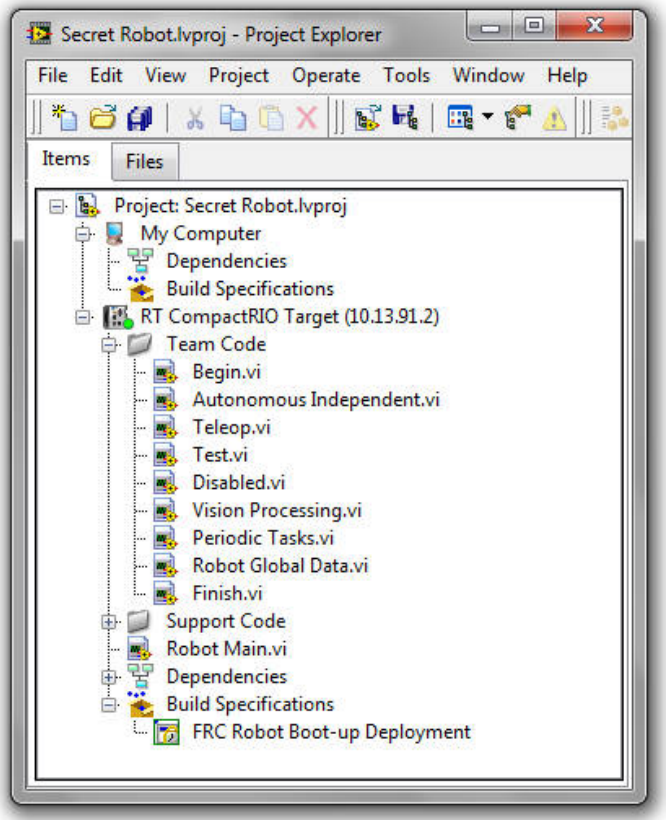


FIGURE 3.5 The project tree for the Robot Framework.

The first thing to note is that the directory (folder) structure you see is entirely virtual. If you go to the directory you specified for the project, you will see that there are no sub-folders. All the files are just dumped together in the main project folder. You might be tempted to impose some organization on this mess, but you should resist that thought. There are advantages to having all the files in one folder, and you can impose all the organization you need through the tree.

Not only is the organization virtual, but the "files" in the tree are virtual too. They are shortcuts. If you right-click on one of the VIs, you will see that **Remove from Project** is an option, but not **Delete**. Removing a file from the project leaves it on the hard drive, which is nice if you decide you needed it after all.

You can create new folders, delete them, drag VIs from one folder to another, rearrange their order, and stuff like that. You can add existing VIs to the project, and it doesn't matter what folder those VIs are actually sitting in (although I highly recommend that you place any VIs you create for your robot in the project folder with the Robot Framework). It is also the case that any sub-VIs that you create do not actually have to appear in this tree. If the main program that calls these routines is in the tree, they will automatically be included when the code is compiled. On the other hand, any code that will serve as a "main" program running on the cRIO *must* appear in the tree.

If you look carefully at Fig. 3.5, you will see that the tree is divided into two sections: "My Computer" and "RT CompactRIO Target". This is telling you where the code will run. Ordinarily, you will not have any VIs under My Computer. You will likely want to have some code that runs on your laptop during a competition, but it will be part of the Dashboard project, not the robot project. On the other hand, if you are working stuff out (doing simulations of your robot, or calculating ball trajectories or something), you may find it convenient to include these files in the project and run them from the tree. Just be aware that the robot code cannot call any of these VIs. To start development of a new program in either category, right-click on *e.g.*, "RT Compact RIO Target" and select **New ▸ VI**.

If you forgot to set the IP address that matches your team number, or want to change it for any reason, you can right click on "RT Compact RIO Target" and select **Properties**.

Running Programs on the cRIO

There are two ways to run programs on the cRIO, corresponding to two types of memory available, EPROM and DRAM. The EPROM (electrically programmable read-only memory; the type in use today is known as "flash" memory) is used for competition code. A program written there is "permanent" in the sense that when you cycle the power on the cRIO, the program remains in memory, and can be run again without downloading. We'll defer discussion of how to do that to the next chapter.

The other way to run a program is in DRAM (dynamic random access memory), where the code remains only as long as you don't turn off the cRIO, reboot it, or lose communication between the cRIO and your laptop/driver station (which happens a lot more often than you'd like). The advantage here is that the front panel of the program remains active on your laptop even as the code itself runs on the cRIO. This can be a tremendous help in getting your robot to work, tuned up, and your software debugged. It also allows you to write small test programs that just do one or two things, which can be a very nice way to solve a particular problem without having to work with the entire Robot Framework. In particular, I find it very useful to look at graphs of sensor readings. Code that has been "Deployed" to the cRIO (is running from EPROM) does not show you a front panel. If there are things that you need to see, you can design their display into your Dashboard, but that's not a five minute job…

As an example, we'll build and run a little program that I find useful to have on hand. It will quickly show you what names the FRC system is assigning to the buttons and other controls on your joystick (or game controller, or whatever you are using). So…start a new VI. (You might consider creating a new folder in your project tree called "Test Programs" to store it in.) Draw the program shown in Fig. 3.6. In order to do this, you will need to use VIs from WPI. But first, turn so that you are facing in the direction of Worcester, Massachusetts (prounounced *wuss*-ter, blame the British), and make a mental note of thanks.

If you look back at Fig. 1.8, you will see that the last item, way down at the bottom of the block diagram context menu, is **WPI Robotics Library**. Select that and pin the palette that you get. You really want to spend some quality time exploring here. Everything your robot actually does, be it run a motor, read an encoder, or operate a pneumatic cylinder, you will do through subroutines here.

For now, though, we just want the Joystick palette. It is a sub-palette under either **RobotDrive** or **DriverStation**. There you will find both the Open and Get routines shown in Fig. 3.6. Create the controls (the constant that selects USB 1) and indicators (Buttons and Axes) by right-click-creating on the VI terminals. Your program will not work if you do not establish communication with the Driver Station, so you need the little Start COM routine (found on the DriverStation palette). This routine runs in parallel with the While Loop, so make sure you place it outside that loop. Be sure to save your program.

Once you start messing with the WPI library, you will start to be asked if you want to save changes to VIs you have never heard of and whose function you have no idea about. This will happen when you close VIs or when you close down LabVIEW. Just say "Yes" to all of these requests. Unless a VI in the library has been re-worked recently, LabVIEW is likely to understand it as having been written in an "old" version, and wants to know if it's OK to save it in the "new" version. Over time, you will get fewer and fewer of these requests, but they will never go away.

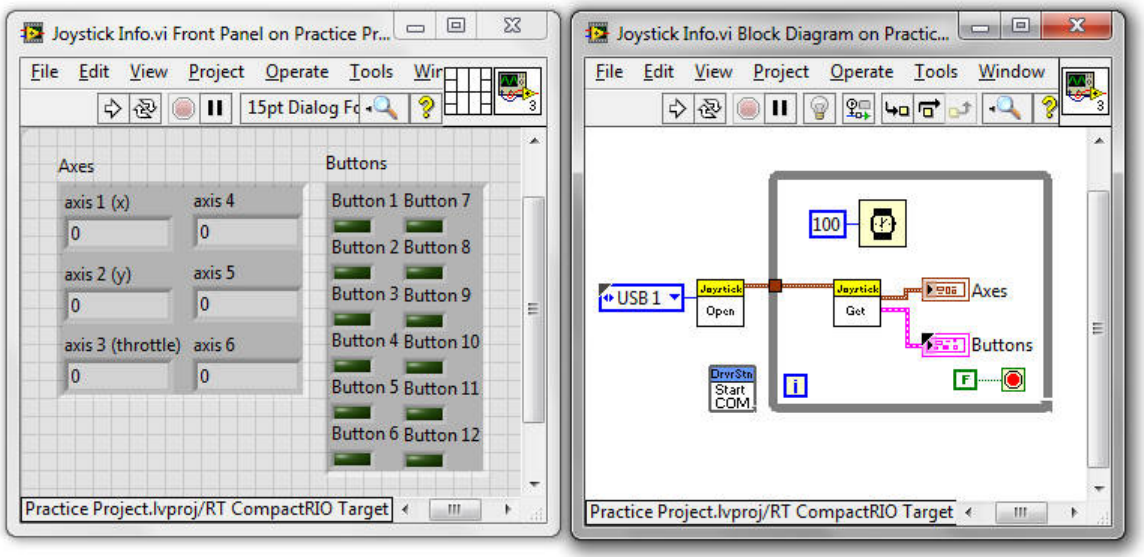


FIGURE 3.6 A simple test program to display joystick control values.

Now we come to the unavoidable. To go any further, you need to be connected to a cRIO. I'm going to assume that you have one. If you need help on configuring your setup, look on your hard drive for something like **C:\Program Files\National Instruments\LabVIEW 20xx\manuals\FRC Programming Guide\index.html**. Of course, "xx" will be the year of your LabVIEW version, not actually "xx". There's other good stuff here, so it's worth a look. You also need to have a joystick or other gaming controller plugged into your laptop. If you are running Windows 7, you need to have administrator privileges.

1. Start the FRC Driver Station, if you haven't already. There should be a shortcut on your Desktop, but if not, you can find it in **C:\Program Files\FRC Driver Station**. Launching the Driver Station will also launch the default Dashboard. You can close that, as it won't actually be getting any data. (If your Driver Station is locked in the lower left corner of your screen, use the Window Mode control—in the middle of the Driver Station panel—to show both the Driver Station and the Dashboard as floating windows. Then you can close the Dashboard.) If you are connected to the cRIO, the Communications indicator should be green (see Fig. 3.7).
2. Click on the "Diagnostics" tab. Look at the list of USB Devices for a green LED indicator. If you have more than one device plugged in, there should be more than one indicator lit. Press any button on the joystick you will be using for this test. The indicator for that joystick should turn blue.
3. If the joystick you are testing was not Joystick 1, click on the "Setup" tab. At the right you should see a list of the USB controllers plugged into your laptop. You can re-arrange this list by clicking on a controller name and dragging it to the position you want. Drag your test controller to the top of the list so that it is Joystick 1. (Alternatively, you could change the code to make the USB port selection on Open Joystick match the position of the controller you want to test.)

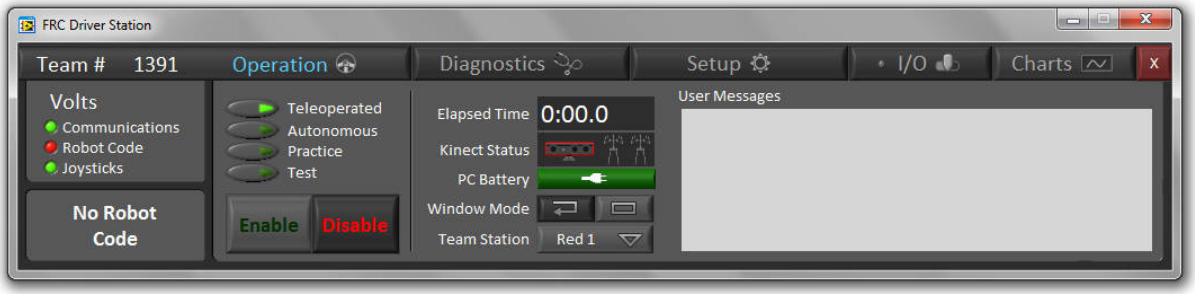


FIGURE 3.7 The FRC driver station showing the Operation tab. At the left you can see the green LED indicating communication with the robot, and the red one indicating that we haven't loaded any code yet.

4. Save your code.
5. Click the white "Run" arrow on your Joystick Info VI. This will cause it to be compiled and downloaded to the cRIO. You will get a pop-up window that starts out looking like Fig. 3.8. You may be asked to save a lot of VIs you have never heard of. Just agree to save them. Sometimes this process will fail, but then work fine when you try it again. Sometimes you will have to cycle the power on the cRIO and radio before it will work. An alternative strategy to isolate where a communication problem lies is to use a cable direct from your laptop to the radio/router, or to the cRIO itself. If you have an older cRIO with two Ethernet ports, you must plug your laptop into Port 1.
6. When the download completes, the Robot Code LED should be green and your VI should be running. Press buttons and move joysticks to see the response on the front panel.

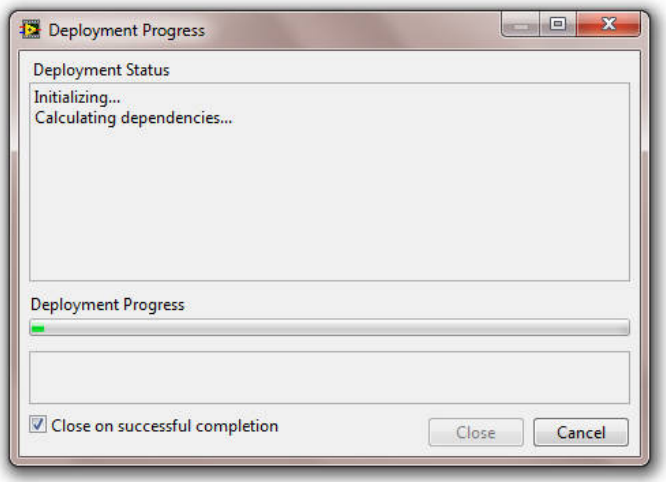


FIGURE 3.8 The deployment pop-up window.

If your cRIO is on an actual robot, with motors and controllers all wired up, you could make yourself a little test program for driving. Fig. 3.9 shows a very simple program to drive a robot using a PS2-style USB game controller. You can see that not very much code is required if all you need to do is make something that drives.

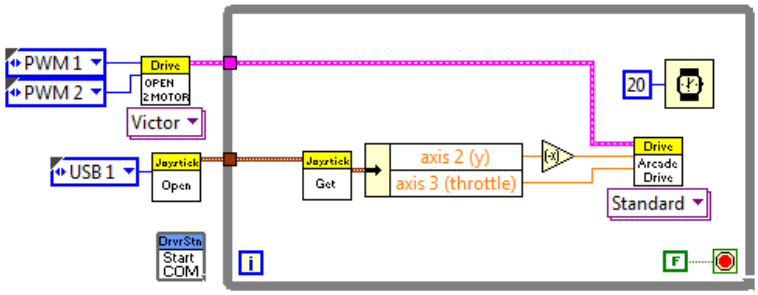


FIGURE 3.9 An example of simple driving code.

Before we move on to the Robot Framework, it is worth emphasizing again how useful this mode of running a program is for debugging and testing. Fig. 3.10 shows the front panel of a test program used to tune the control loop for a motor (that was part of a ball shooting mechanism). It's too small to actually read anything, but the point of the figure is the graphs. You can show things in a visual manner that really lets you figure out what is going on.

Bear in mind that your actual competition VIs can have diagnostic front panels too. You can't run them with the panel showing at a competition, but you can run them this way at home, and get a good handle on what is and is not working.

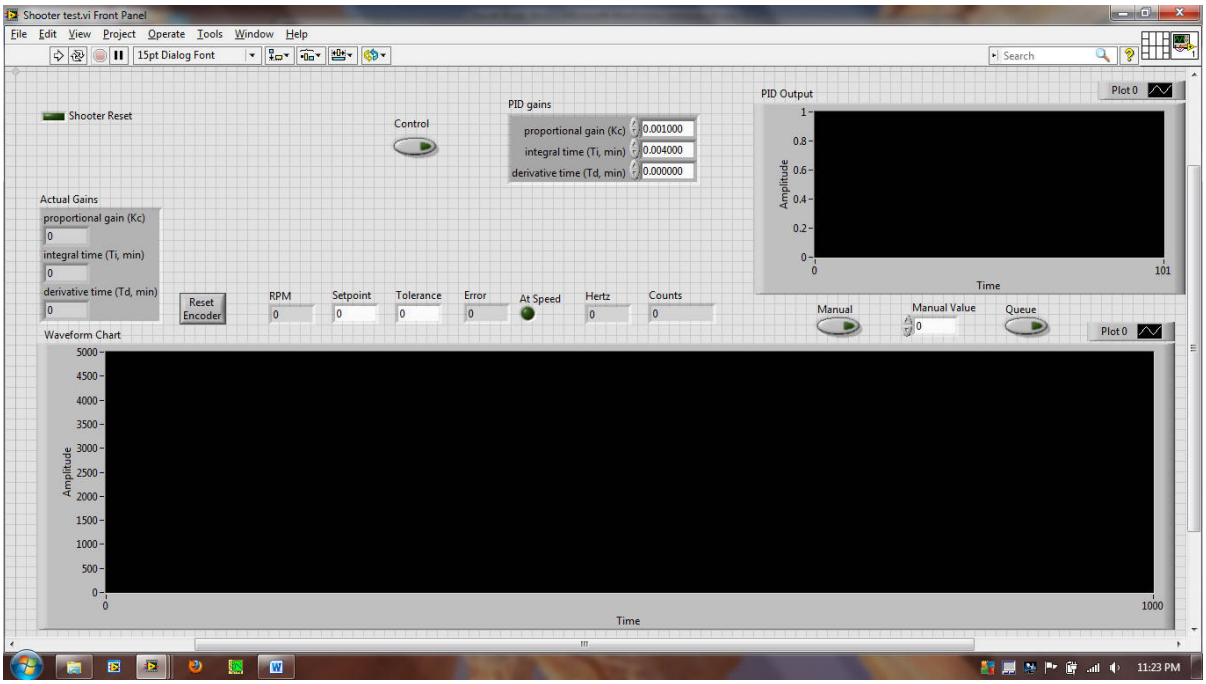


FIGURE 3.10 An example of a test program with charts. Big charts. This is a real program used to tune the turret aiming on our 2012 Rebound Rumble robot.

Chapter 4 — The Robot Framework

Have a quick glance back at Fig. 3.5, which shows the project tree for the Robot Framework. In this chapter we will explore in detail the key elements of this framework. They are Robot Main, the "top" VI that calls everything else, and four others: Begin, Periodic Tasks, Autonomous Independent, and Teleop. There are a few minor players to discuss as well, and also some new concepts, such as Global Variables. Finally, we'll spend a bit more time with the Driver Station. The VIs in this tree have been "written" for you, which is to say, they have some code to give you examples to look at, but you will discard most of this "boiler plate" in favor of your own programs. The key thing is that here you don't start new VIs from scratch, but edit existing ones (except, of course, that being a wise and experienced programmer, you will organize your code using sub-VIs, and those you will start from scratch).

Before we begin, one word of warning: this book was written using a particular version of the Framework, but a new version is released, with modifications and tweaks, every year. So what you see in the latest version might not exactly match the descriptions here. A certain amount of intellectual flexibility will serve you well in that case.

Robot Main

In case you can't tell by its special status in the Framework tree, Robot Main is *the* VI. It controls everything and calls (directly or indirectly) all the other VIs. If you want to run your program in RAM (which you will do often while first developing it), you click the white arrow on *this* VI and no other. Paradoxically, you will do only a minimal amount of coding here. Robot Main controls everything, but does almost nothing. All the heavy lifting is done in the VIs that it calls.

The first thing to notice is that what you are looking at in Fig. 4.1 is mainly a state machine. The state is determined by the Driver Station Get Mode VI (which we will not mess with!) which gets its marching orders from the Driver Station. In a competition, the Driver Station gets the mode from the FRC field controls. At home, you can set up the Driver Station to put the robot in whatever mode you prefer. There are six states: Autonomous Enabled, Teleop Enabled, Test (both Enabled and Disabled), Disabled (for Auton and Teleop), Timeout, and Finish. We won't talk much about the last two. Finish never runs. Well, not true. If you press the Finish button on the front panel of Robot Main while it is running, then the Finish VI will run. At a competition, the robot goes directly to the Disabled state at the end of the match. The Timeout state only runs if there is a communication problem with the Driver Station. If you look, you will see there is no code in the state. If you were having ongoing communication problems, you could put some diagnostic code in here to *maybe* figure out what's going on.

The operation and timing of this state machine is controlled by the Driver Station. Every 20 milliseconds a fresh data packet arrives from the Driver Station, and is put into a cache by the Robot Start Communication VI (which we first encountered in Chapter 3). Robot Start Communication signals Get Mode (via the thin green wire connected to the funny circle within a square symbol) that there is fresh data in the cache. Get Mode has been paused, waiting for that signal. When it arrives, Get Mode reads the cache, and selects which case in Robot Main will run next. Get Mode also has two other outputs, which can be useful, but are not essential. We'll discuss those when we talk about Teleop in a bit. You should also note

that although the state machine is designed to loop at the Driver Station data rate (every 20 ms), you can slow it way down with your Teleop code, and potentially make problems for yourself. More on this topic when we discuss Teleop.

Outside the While Loop of the state machine, there's a bunch of other stuff. The obviously named Begin is the first VI to run. In here you will set up all your motors and sensors. This VI contains no loops. All its code executes once and is done. Note the use of the Error output from Begin to ensure that the other VI's and the state machine only start once Begin has completed. We used a trick like this back in Fig. 1.16. There are three VIs running parallel that are called here. NT Server is required to sending data to and from the Dashboard. Leave it alone. We will discuss Periodic Tasks (whose icon says Timed Tasks) later. If you are not planning on doing any image processing on the cRIO (and you most likely shouldn't), then you should delete the call to the Vision VI from Robot Main. Even if Enable Vision is set to False, this VI continues to read images from the camera, consuming bandwidth and CPU cycles. But is if you do have some reason to image process on the cRIO instead of on your laptop, then you need to modify Vision Processing.vi to include your routines.

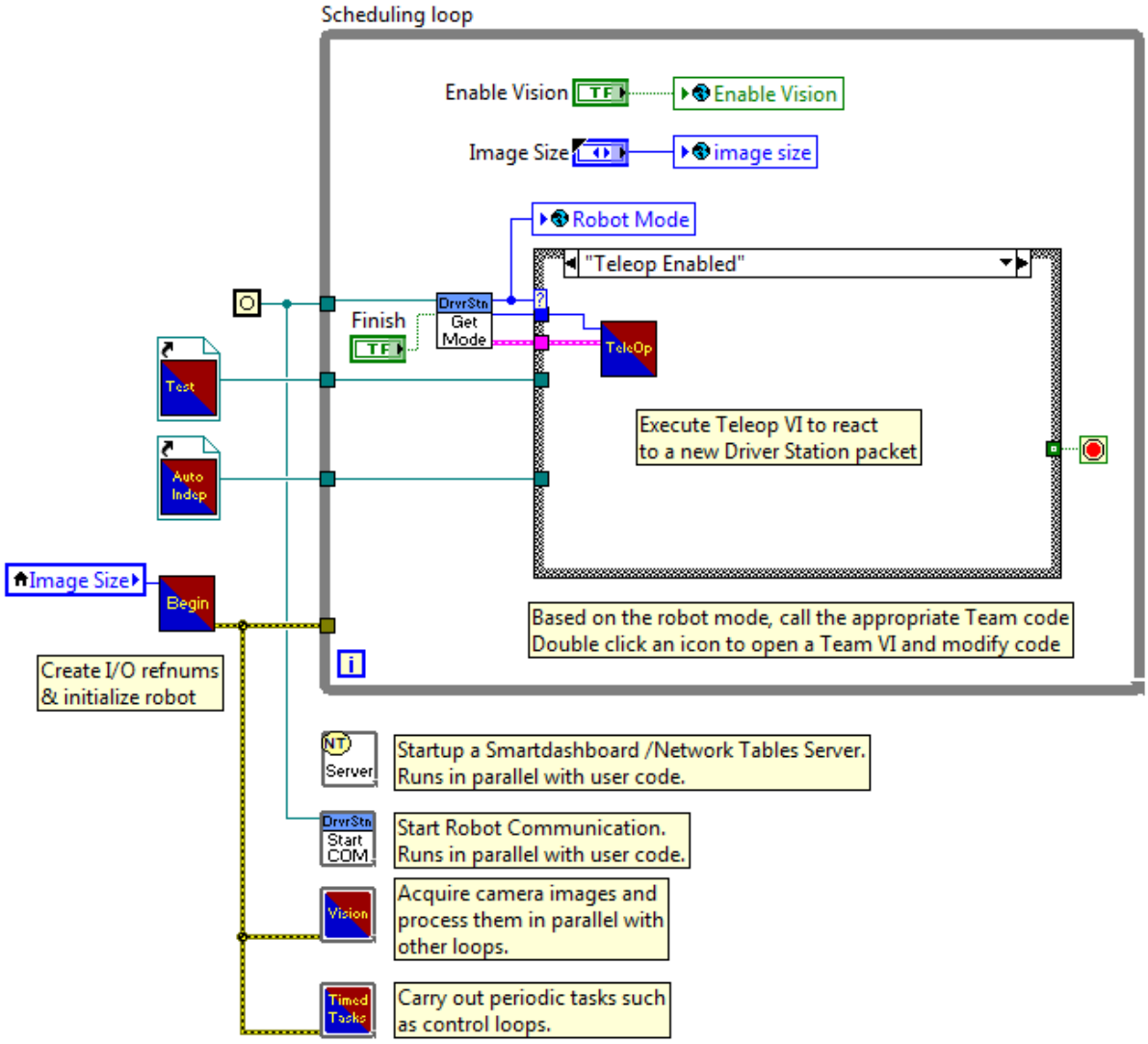


FIGURE 4.1 The "as provided" Robot Main VI.

Finally, there are two remaining items: objects of a type we have not seen before labeled "Test" and "Auto Indep". The first is part of the robot Test Mode, which we will not discuss until Chapter 8. The second has to do with how your Autonomous code gets started and stopped, which we will discuss a bit later in this chapter.

Before we examine the individual sub-VIs in detail, let me make a few recommendations. I like to clean up the default robot code to get rid of stuff I know I'll never use. Unless you have some very special reason to do so, you will not be using the cRIO for image processing. Delete the Vision Processing VI from the diagram for Robot Main. Then you can delete the Enable Vision and Image Size controls and the objects they are wired to. That will break the local variable wired as an input to Begin.vi, so delete it as well.

Begin

It is always best to begin at the beginning, or so they say. Fig. 4.2 shows the default code provided with the Robot Framework. I've drawn a box around the code for setting up the camera. You should just delete this code. You can accomplish the same thing from the Dashboard, where you can, in fact, change the camera settings "on the fly" to compensate for the actual field conditions. Don't get carried away and delete the "error out" indicator, even though it will no longer be connected to anything. It's connected *outside* the VI (in Robot Main), and used to make sure no other VIs start executing before Begin finishes.

The rest of the code shows the setting up of a pair of motors, and of a joystick. This is almost certainly not exactly what you want for your robot, but it shows the idea. You "open" everything in here: motors, joysticks, encoders, gyros, and what have you, and you wire the reference output of that Open VI to the reference *input* of an object with the name "WPI_*Something*RefNum Registry Set.vi". The "something" will tell you that it is a reference to, for example, an encoder. What is happening here is that setting up the motor (or

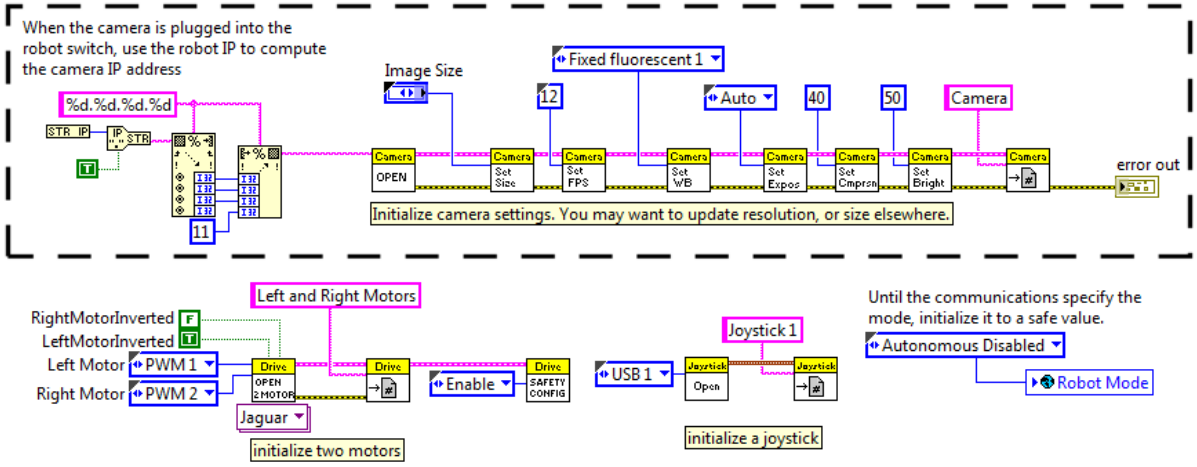


FIGURE 4.2 The default Begin code. Delete the code inside the dashed lines unless you have good reason not to.

whatever) creates a cluster of information about how this device is configured. This information is stored in an object called a Functional Global, together with a unique name that you provide. This information is then available in other VIs, using this name that you provided, and without having to draw a wire from the Begin VI to all the other VIs that actually

use this device. I'm not going to talk any more about Functional Globals and how they work, but they are quite generally useful, and now that you know what they are called, you can learn more about them from The Intertubes.

Fig. 4.3 shows a fully configured Begin.vi from Team 1391's 2012 competition robot. Don't worry about the details. We'll cover those in Chapter 5. This figure is just here to give you an idea of what your own code will look like when you are done. But there are a couple of things to notice. First, pay attention to how neat it is. That neatness really helps when you need to make a change in a hurry. Second, if it seems a bit cramped, that is because an effort has been made to arrange it so all the code fits on one screen. That makes it harder to miss important things because they happen to be off screen when the VI opens.

This figure also shows another use for Begin.vi. This is a good place to set variables that need to have a particular value when your Autonomous code starts. It's not the only place, but it is a handy one. In this particular example, the variables needed to be set so that the code in Periodic Tasks behaved properly when our Autonomous code started.

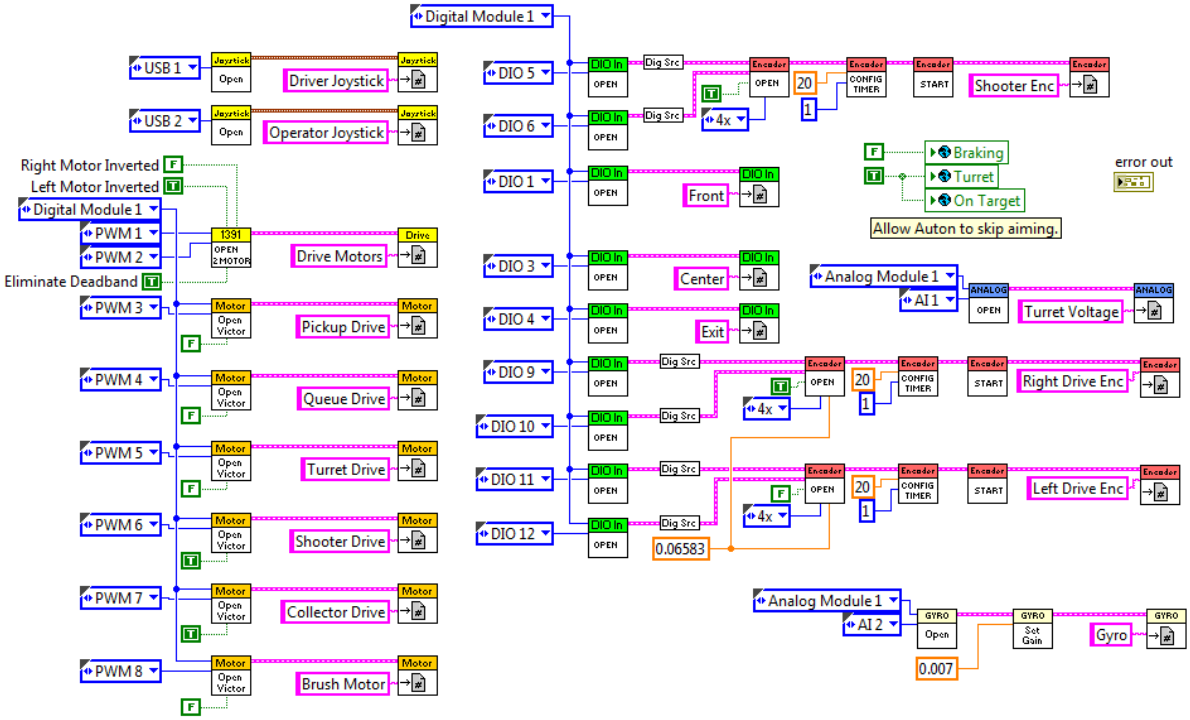


FIGURE 4.3 An example of the Begin code from a completion robot.

Periodic Tasks

There are many things that you may want your robot to do that are not directly linked to the action on the field. For example, you might want to continuously read the encoders on your drive train so that you always have the current speed of the robot available. You might want to continuously gather data and send it to your custom dashboard. You will probably also want to receive data back from your custom Dashboard, with the results of processing the images from your camera, for example. None of these activities depend on the actions of the team operating the robot, and it makes a great deal of sense to let them happen quietly in the

background. Periodic Tasks (whose icon in Robot Main is labeled Timed Tasks) is the place to do it.

Another good use of Periodic Tasks is PID control of things other than driving. (PID what? See Chapter 6.) If you need to precisely control the speed of a wheel that is part of a shooter mechanism, you would like the loop that does this controlling to run at a smooth and steady rate. Loop timing is slow and can be a bit uncertain in Teleop, so it is better to put the PID here. You can then control its operation from Teleop (or Autonomous). In fact, since this shooter mechanism is likely to be used in *both* Teleop and Autonomous, putting the PID in Periodic Tasks avoids unnecessary duplication of code.

Fig. 4.4 shows the default code for Periodic Tasks that comes with the Framework, although I've radically squished it to make a compact figure. You could discard it all (except for the error in control!) and start from scratch, if that makes your life easier. Or, you could stretch the loops here to fit your stuff in. But the code here does illustrate a number of important points. The first of these is that not everything needs to go at the same speed, and you should reduce the load on the cRIO by putting things that don't need to go quickly into a separate, slower loop. Note also that the loops are infinite: the False value wired to the condition terminal keeps them running until Robot Main is stopped.

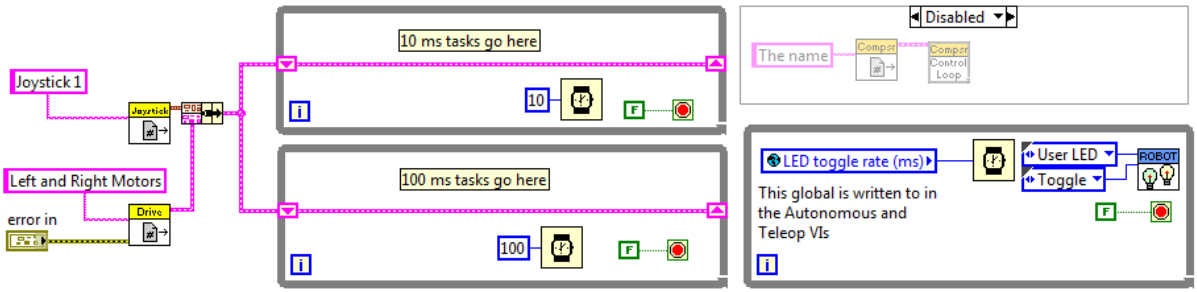


FIGURE 4.4 The default code in Periodic Tasks.vi

A second thing to note is the use of the Reference Get VIs to allow reading of devices set up in Begin.vi. Be sure to put those calls outside your loops. You don't need to waste CPU cycles retrieving the device info on every iteration of the loop. I don't know why the references are wired to the loops via shift registers. Unless you are planning on modifying the information in the references, a simple tunnel is sufficient.

A third thing shown in the code is the use of the Diagram Disable structure (found on the Structures Palette with Loops and Case Structures). This is how you comment out code in LabVIEW. Inside that is the code to run the compressor, which not everyone uses, so it comes disabled by default. Be careful with Disable structures. If they cross data wires, LabVIEW will automatically do things that result in your code *not* getting a broken arrow. That doesn't mean your code will actually work as you expect! Make sure you haven't broken some other thing whenever you comment out bits of your code.

The last important thing Fig. 4.4 shows you is how to communicate between Periodic Tasks and the rest of your code. The example here is the toggling of the User LED on the cRIO. Every time the loop executes, the LED changes state (turns off it was on, etc.). The rate of the loop is controlled by a Global Variable, which is the subject of the next section.

One thing to be clear on: Periodic Tasks is called exactly once by Robot Main. It is up to you to make the tasks periodic by putting them inside infinite loops. You will note that in Fig. 4.4, the compressor VI is not inside a loop. That is because that VI *is* a loop, and runs continuously once it is called. (If you examine the icon graphics carefully, you will see that the outside border tries to look like a While loop.)

Robot Global Data

If you look back at Fig. 3.5, you will see that the project tree for the Framework contains something called Robot Global Data.vi. Open it an you will find a VI with a front panel, but no block diagram, as in Fig. 4.5. The three objects you see in the figure are global variables. They are accessible from any LabVIEW VI running on your cRIO. In fact, we already met two of them back in Fig. 4.1, and you should refer back to that diagram to get a clear under-

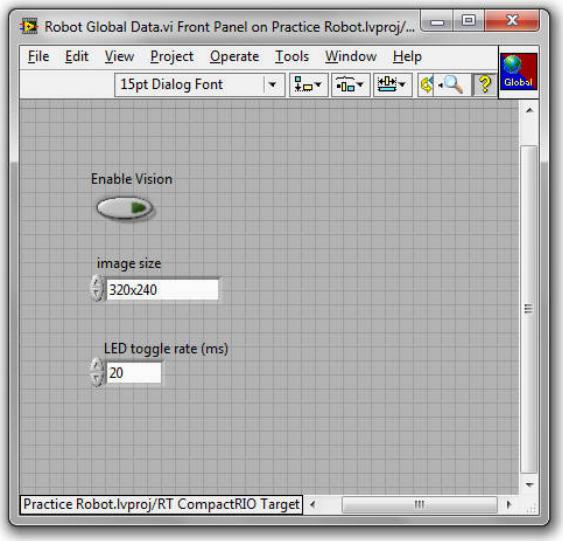


FIGURE 4.5 The default Robot Global Data.vi.

standing of what is going on. In that diagram, there are the terminals to two controls, Enable Vision and Image Size. The controls themselves are on the front panel of Robot Main, and are entirely local to that VI. The values of both controls are being written to objects that look like local variables, but have a globe instead of a house as a symbol. These are global variables, and are simply instances of the two variables with those names shown here in Fig. 4.5.

If you need more global variables (and you will), simply drop whatever controls and indicators you need onto the front panel of Robot Global Data.vi. To add a global variable to a VI, right-click on the block diagram, choose **Select a VI...**, and then select Robot Global Data.vi. Once you have the variable dropped, click on it to choose the actual variable you want from the pop-up list. As is the case with everything else, these globals need to be set for reading or writing, just like local variables. Once you have one global in the diagram, the easiest way to get more is by the control-drag method.

Since the globals are accessible from any VI, they provide a natural way to communicate between Periodic Tasks and Autonomous or Teleop. For example, you could use a loop in periodic tasks to continuously read an encoder and put the current value into a global variable. When your Teleop routine needs to know that encoder value, it has only to read the global.

Or, Teleop could change the setpoing in a PID control loop that is running in Periodic Tasks by writing to a global that Periodic Tasks reads.

Often, particularly in Autonomous, you need numbers that have to be determined by experiment on the practice field. For example, you may need your robot to move forward for a certain amount of time at a certain power. You could, of course, put this number as a constant inside your Autonomous code. But then, if you need to make changes, you need to hunt through your code for all the locations where that constant is used. An alternative is to put the value in a global variable, and make the specific number the default value for that variable. Just type the number into the variable, right click on the variable, and select **Data Operations ▸ Make Current Value Default**. Be sure to save Robot Global Data after doing this or the default value will be lost. This strategy puts all your key parameters in one easy to find spot.

Let me emphasize again that "global" means global. Any VI running on your cRIO can access the same variable. So, for example, you can set a value somewhere in *any* VI that is running on your robot, and read it in *any* other VI running on your robot, whether that VI was called directly as part of Robot Main (like Begin, Teleop, or Periodic Tasks), or is running in parallel to Robot Main (like Autonomous, as we shall see shortly).

Teleop

In order to understand how Teleop works, you need to understand how Robot Main works. Refer back to Fig. 4.1. Robot Main is a state machine, but unlike the examples we studied in Ch. 2, the individual states do not determine the program flow. Which state executes next is controlled exclusively by Get Mode. On each iteration of the loop, Get Mode is called, and waits for a control packet from the Driver Station. Included in this packet is the state the robot should be in, which Get Mode outputs to the selection terminal of the case structure. If the mode is Teleop, then the Teleop case is executed. That case calls your Teleop code, which should execute and *quickly* complete. "Quickly" here means in well under 100 milliseconds. As discussed above, the Driver Station sends a data packet every 20 ms. But Get Mode cannot read that packet and initiate a new call to your Teleop code until the previous call completes! So in Teleop, the state machine runs at the rate set by the execution time of your code, (unless your code completes in under 20 ms). As we will discuss in more detail below, because you are fundamentally in control of the timing here, it is possible to cause yourself some real trouble. The sober recommendation is that you not do that. (And a detail for worriers: because of the way in which communication is handled, when your code does complete and allow Get Mode to run, the data from the Driver Station, including such critical things as joystick positions and button presses, is the latest available from the most recently arrived Driver Station packet.)

(Very) Basic Teleop Code

Fig. 4.6 shows the default Teleop code that comes with the Framework. It's a bit confusing, but the first thing to do is mentally filter out all the VIs that have "SD" in the upper left corner of their icon. These VIs send and receive Dashboard data, and can be ignored for now. (We'll discuss them in Chapter 8). If you apply that mental filter, then there is mainly code that reads a joystick and sends the joystick values to a motor drive VI.

Notice what there is not: loops. This code will execute once, and the VI will terminate. That is how the Framework is designed to operate. All the looping is done by the While Loop in Robot Main. There is a control called Call Context, whose value is controlled by Get Mode. When Robot Main executes, the first time Teleop.vi is called, Call Context will have the value "Init". On subsequent calls, it will have the value "Execute". In principle, it will have the value "Stop" on the last call, but I am not sure that ever happens in real life. You can use this input to initialize things on the first call to Teleop if you need to.

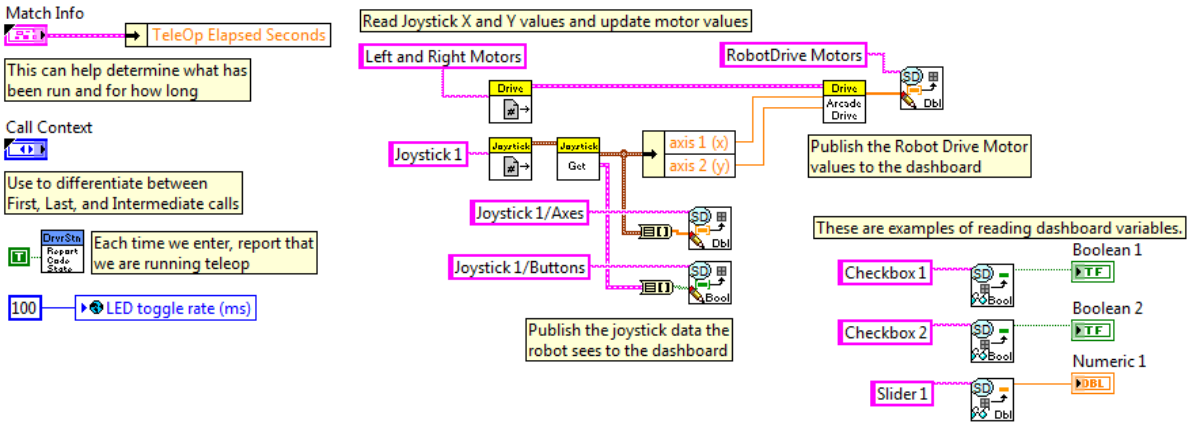


FIGURE 4.6 The default Teleop.vi.

There are a couple of other possibly useful bits in here. Get Mode passes a cluster containing information that can inform your software exactly where it is in the match. The number of seconds that have elapsed in Teleop are brought out by default, but there is other information in there as well, which you can discover by dragging the top and bottom edges of the existing Un-bundle By Name node. There is a call to a VI that informs the Driver Station that you are running Teleop, so you can see there that the code is actually running on your robot (which can be hard to know for sure when you are running in competition mode, as you have no access to either the front panel or the block diagram.) And finally, there is an example of using global variables to pass information to Periodic Tasks, in this case the delay which Periodic Tasks will use to blink the User LED as an additional way for you to know which piece of code is running.

Writing the Teleop code (together with Periodic Tasks) is the essential thing you will do to make your robot competitive. But don't do too much of it. The Driver Station has a Charts tab, and one of the things plotted is the percent usage of the CPU on the cRIO. If your usage in Teleop or Autonomous averages much above 60%, your code is doing too much. Try to simplify it. If CPU usage approaches 100%, your robot is in serious trouble, and you should expect it to not work at a competition, even if it seems to work at home.

Waiting Without Hanging

Back in Ch. 2 we discussed the importance of delays and their proper implementation. That subject is so important that we going to revisit it here. Suppose you need to your robot to do a specific thing for a set amount of time. In this example, let's imagine that it needs to squeeze a pneumatic gripper closed for 2 seconds when a button is pressed on a joystick. We don't want to have to hold the button down, we just want to press it and have the 2 second squeeze be automatic. ***Don't do it like Fig. 4.7!*** This snippet of code will squeeze the grip-

per, but your robot will be totally un-drivable for the entire 2 seconds! The millisecond timer is a blocking call. Teleop.vi can't complete until the timer completes. That means it can't be run again to get you new joystick readings, or do anything else. I can't emphasize enough: *don't do it this way.*

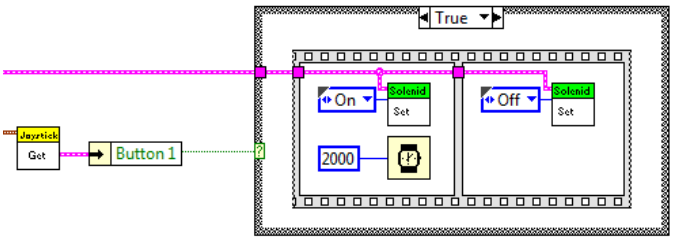


FIGURE 4.7 How **NOT** to implement delays! (The False case is completely blank.)

Doing it right is a bit more complicated. One possible way (there are others) to implement this delay is shown in Fig. 4.8. This is not as simple as it could be, because the code is written so that the Squeeze button can be released, and our two second squeeze will still operate. Also, if the button is re-pressed during the two seconds, the timing does not restart. (So I'm showing you three things at once. Such a bargain!)

If the variable Squeeze is False, and the joystick button is not pressed, then we execute the False case which is completely blank. You could remove this outer case structure (and make a few other changes), but it would increase the computational overhead (by a small amount). You would always be reading a timer and feeding data to the feedback node. As it is, we really do nothing until the button is pressed. When it is pressed, we execute the case shown in the figure. Because Squeeze is still False, we close our gripper, load the millisecond clock into the feedback node, and set Squeeze true. This last step will keep us in the outer True case even when the joystick button is no longer being pressed.

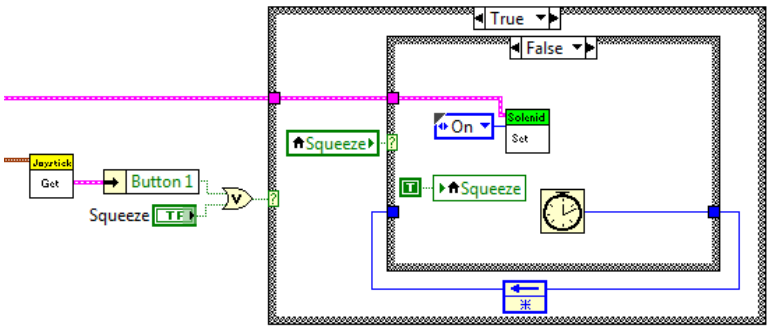


FIGURE 4.8 The initial state of a delay code for Teleop.

On the next call to Teleop, we will execute the code shown in Fig. 4.9. Now Squeeze is True, so we check the time (see Problem 1.13 if you haven't written your Timeout VI yet), and if it has not expired, we do nothing except keep passing our saved start time back into the feedback node.

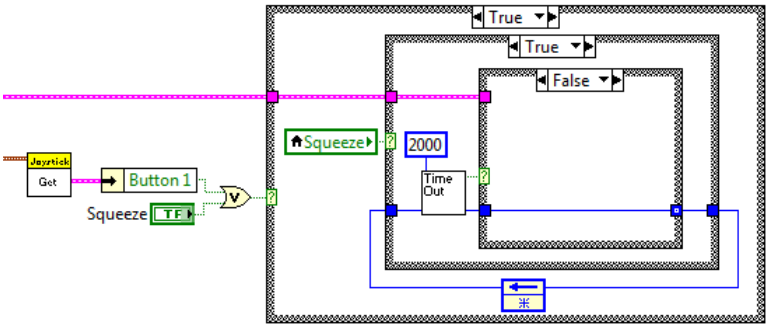


FIGURE 4.9 The code that runs after the button is pressed but before 2 seconds has elapsed.

Finally, when 2 seconds have passed, Timeout will be True, and we will execute the code shown in Fig. 4.10. This code releases the gripper and sets Squeeze to False so that we return to our original state of doing nothing.

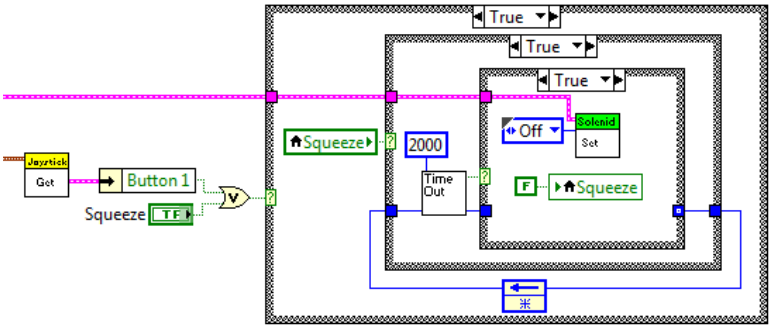


FIGURE 4.10 The end of the delay.

Of course, there are other approaches to button presses, and you should consider them in planning how your robot operates. For example, you could require the Squeeze button to be held down for the entire two seconds. The ultimate duration of the squeeze would still be controlled by a timer, but you could abort the squeeze by releasing the button before the two seconds are up. Or, you might want a squeeze that has no timing it at all. The squeeze persists only as long as the button is held. Each has its own advantages and disadvantages.

It may not be obvious, but the bit of code in Figs. 4.8 through 4.10 is a state machine. It's state diagram is shown in Fig. 4.11. What's missing from our usual picture of a state machine is the While Loop. That's because *the Robot Framework supplies the loop*. All we need to provide is the guts. So your Teleop code can contain state machines as well. They can be as complicated as your robot requires (but please, no more complicated than that). Just remember that you can't wrap them in While Loops, because that looping is already provided for you. Values that need to be kept from one iteration to the next can be stored in feedback loops, as I did in the example, or just written into variables. That's what I did with Squeeze, and if you think about it, I could have done the same thing with the start time. (There is never only one way to code something!)

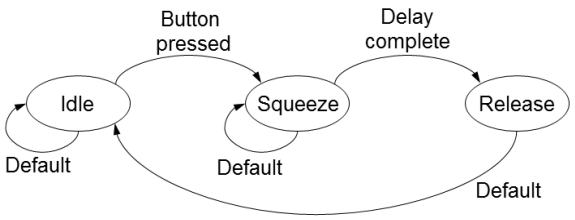


FIGURE 4.11 The state diagram for the gripper squeeze example.

Avoiding a Death Spiral

In order for you to write code that makes your robot behave as you expect, it turns out that you need a pretty good understanding of how the robot acquires and uses the control inputs. The ultimate pace setter is the Driver Station running on your laptop. That is, after all, where your joystick and other controls are plugged in. Initiated by the Driver Station, the following sequence of events occurs every 20 ms:

1. The Driver Station gathers up the current robot mode, joystick positions, button presses, etc., and sends a data packet to the robot.
2. The data packet is received by Start COM, which is running as a sub-VI under Robot Main.
3. Start COM caches (stores) the data from the packet and signals "OK" to Get Mode (which has been paused, waiting for this signal, if your Teleop code has completed).
4. Get Mode then reads the communication cache to find out which mode the robot is in, and then selects the appropriate state in the state machine for execution.

This appears to be all very straightforward, but it has consequences that you should be aware of. For example, when your code calls Joystick Get, the returned data is *not* the current position of the joystick, but the current contents of the joystick cache. You can make code that reads the joystick as fast as you like (by, e.g. putting the code in Periodic Tasks), but you won't actually get fresh readings of the joystick position faster than every 20 ms (which is pretty darn fast). If there are communication problems so that packets are dropped, your robot's responsiveness may be affected, because the looping of the state machine only happens when a fresh packet arrives.

At the start of our discussion of Teleop, I mentioned that your Teleop code needs to complete in under 100 ms. It might seem that the reason for this is that you want Teleop to be all done and waiting for fresh data when it arrives, and that is true, but there is actually a more urgent reason. There is a safety check system that requires you to write data to your drive motors every 100 ms (which you do by calling any one of the motor drive VIs). If you don't, then your motor outputs will be set to zero, stopping your robot, and in the Diagnostics tab on the Driver Station, you will get the following error message:

> Watchdog Expiration: System 2, User 1
> ERROR <Code> -44061 occurred at "Drive Motors" in the VI path: Robot Main.vi
> <time>07:37:13 03/03/2012
> FRC: The loop that contains RobotDrive is not running fast enough. This error can occur if the loop contains too much code, or if one or more other loops are starving the RobotDrive loop.

This disabling of the motors is not permanent. On the next execution of your code, fresh values are sent to the motors and operations resume. Because the communication system is

not perfect, there will occasionally be dropped packets and this safety check system will operate. Since 100 ms is not very long, you won't even notice it.

Well, you won't notice it unless your robot enters a death spiral. Every time a safety timeout occurs, the robot "throws an error", and you will get a cryptic message visible on the Diagnostics tab of the Driver Station. The problem is that error handling is very time consuming, and slows down the execution of everything else, like your Teleop code. So if you code is on the hairy edge, which is to say it takes close to 100 ms to complete a single iteration, then the error handling will cause it to take longer on the next execution, which will throw more errors, slowing things down, causing more errors…in other words, a death spiral.

So, what are the steps to avoid a death spiral?

1. Monitor your CPU usage on the Charts tab of the Driver Station. Make sure you are running at only about 60% of capacity during Teleop (and Autonomous).
2. Monitor the message window on the Diagnostics tab. You will inevitably get the occasional message here, but if you press the Clear Errors button and the window immediately refills with errors, then you have a problem.
3. Use the Elapsed Times VI to time how long it takes Teleop to complete. This VI is found in the Support Code section of the Framework project tree. You use this when running your code in RAM (so you have access to the VI front panels). Put this VI inside Teleop, open its front panel, and run the robot code. The VI will tell you the time (in milliseconds) between successive calls to Teleop. You can also use this VI inside loops in Periodic Tasks or Autonomous.
4. Build your own timers into your code, and send that information up to your custom Dashboard (see Chapter 8) so that it is always accessible.
5. Make sure that you write to your drive motors on every call to Teleop, even if only to send a value of zero.
6. Live dangerously. (Optional. Not Recommended). There is a Safety Config VI on the **RobotDrive ► Advanced** palette that you can place in Begin.vi and use to turn off the safety checking system. For use by Wizards only.

The general symptom of a death spiral is that your robot will twitch, often quite violently. What is happening is that the safety system is cutting the motors out, and normal operation of the Framework is turning them back on. The two systems fight each other, and your robot loses. If your robot comes down with a case of St. Vitus dance, then your number one suspect is that the execution time of your Teleop code is longer than 100 ms.

Teleop Strategy

Now that you understand how Telop works, what should you do about it? Well, some things need to run fast, and some things don't. Your driving code will most likely be quite simple, and there is no point in it being faster than a human anyway, so probably you can keep that all in Teleop. Similarly, motors that are simply turned on and off by the driving team can stay in Teleop. On the other hand, consider PID control to hold the position or speed of something constant at a specified value. (You will actually be able to consider such a thing after you have mastered Chapter 6.) That kind of code should run pretty fast, and the good news is that it can. The process variable will be a sensor reading, and that does not come from the Driver Station. So, you can read the sensor and run the PID with a fast loop in Periodic Tasks.vi. If the setpoint for the PID has to come from the Driver Station (from some

joystick or other control you set up), you can read it in Teleop and pass the value to Periodic Tasks through a global variable.

Autonomous

It turns out that Autonomous is just like Teleop—except… Except that there is no joystick control, of course. More importantly, there is no external looping. Your VI is called once, and runs for the entire autonomous period. So it has to have an internal While Loop (or loops) so that it keeps doing what it is supposed to be doing for the entire autonomous period.

The Most Important Part

The last part of the last sentence was oh so casually phrased, and yet so absolutely critical. "…for the entire autonomous period" is literally true! Your Autonomous VI ***MUST NOT END***. If the code completes and has nothing left to do, it *must* continue to loop. If your VI actually terminates, the Framework will throw an error when it tries to stop a program that is already stopped. In principle, that should not cause any actual problems, and at your home base, it will not cause any problems. And I have learned directly from NI staff who worked on the Framework that it can't cause a problem. But I have also personally seen a robot that, when connected to the Field Management System (FMS) at a competition, would not function during Teleop. All it took to fix the problem and allow the robot to compete was the addition of an infinite loop to the end of the Autonomous code so that the VI never stopped running. So, let's not say that an Autonomous that never stops is a *requirement*, let's just say that it can't hurt, and might even be a good idea.

How Autonomous Works

This is not obvious. When you look at Robot Main, it is clear how it calls Teleop, but what it is doing with Autonomous is very different. For starters, the VI doesn't even appear inside the state machine, but only in this weird green box outside the main While Loop, as shown in Fig. 4.1. This green box is a Static VI Reference node. Usually, when you drop a VI into a program, the code is in some real sense right "there" where you dropped it. The Autonomous code is not "there" at that location in Robot Main. Instead, we are telling LabVIEW that this is a VI we will want to run later, so it should make a note of it. That note is called a "reference", which is to say, the green line that comes out contains a reference to our Autonomous code. We can run the code using an "invoke node" that looks up the reference and runs the VI that it finds there. In this case, "runs" means "launches so that it executes in parallel with Robot Main."

Fig. 4.12 shows how Robot Main invokes your Autonomous code. The reference is passed into VI, whose contents is—surprise!—another state machine. The guts of the Start/Stop VI are shown in Fig. 13. The first time the control output of Get Mode is set to Autonomous, the Call Context output is set to Init. The Init case contains a VI whose icon reads "Start BG VI". This VI "invokes" your autonomous code, which is to say, it starts your code running in parallel with Robot Main (you can open this VI to see how it's actually done). Robot Main itself continues to loop, but now Call Context has the value Execute, and you can see that the code does nothing but continue to keep track of the VI reference using a feedback node. Finally, at the end of Autonomous, Get Mode (under the influence of the Driver Station) sets Call Context to Stop, and a VI is called to invoke the termination of your Autonomous VI. Attempting to stop a VI that is not, in fact, running throws an error.

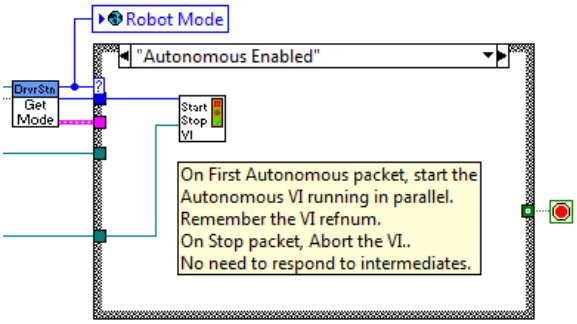


FIGURE 4.12 The Autonomous state in the Robot Main state machine.

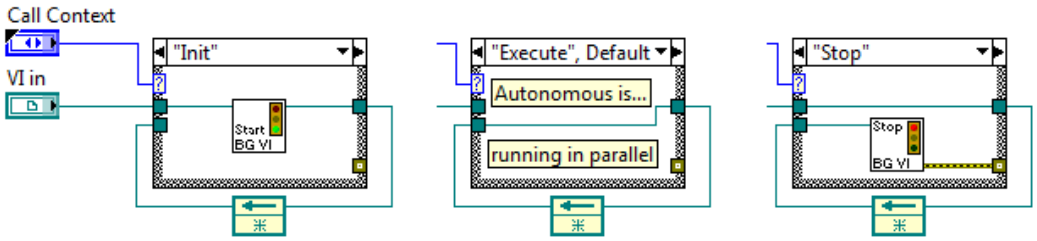


FIGURE 4.13 The cases in the Start/Stop VI that calls your Autonomous code.

The Autonomous Code

Fig. 4.14 shows the essential bits that come in the default Autonomous code. I say "essential," but I should really say "constant". From year to year, Autonomous is the code that varies the most. Sometimes you get code that gives you a big head start on developing a functioning Auton program for that year's game. Other years you get just a bit of sample code that gives you an idea of things you can do, but is not really game specific. So the figure shows none of that. It only shows the VIs that communicate with the Driver Station.

There is a VI that tells you whether you are in a Red or Blue alliance, and which of the three driver positions you are plugged into (which just reads a setting on the Driver Station, so if you forget to set it there, this data is useless). There's a fast loop that signals the Driver Station that Autonomous code is running, and there is an assignment of a slow blink speed to the User LED. Note that if you are careful to not delete this fast loop, it will automatically provide you with an Autonomous VI that never ends.

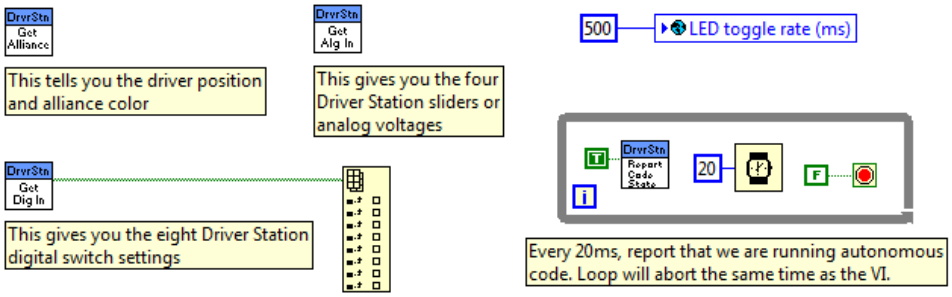


FIGURE 4.14 The "standard" bits of Autonomous Independent.vi.

Also shown in Fig. 4.14 are two VIs that let you pass data to your Autonomous code, labeled as Get Dig In and Get Alg In. They pass eight Boolean and four analog values, respectively. You set up these values on the Driver Station I/O tab (see Fig. 4.15). The Booleans are just set up as switches, while you can set the analog values by using the sliders, or by typing a value in the text window. These values are latched at the start of the Autonomous period so that you can't use them as some kind of illegal joystick control. Whatever you set up before the match is what you have for the whole Auton period, but it does provide a way of, for example, selecting among possible behaviors without having to change the code on your robot.

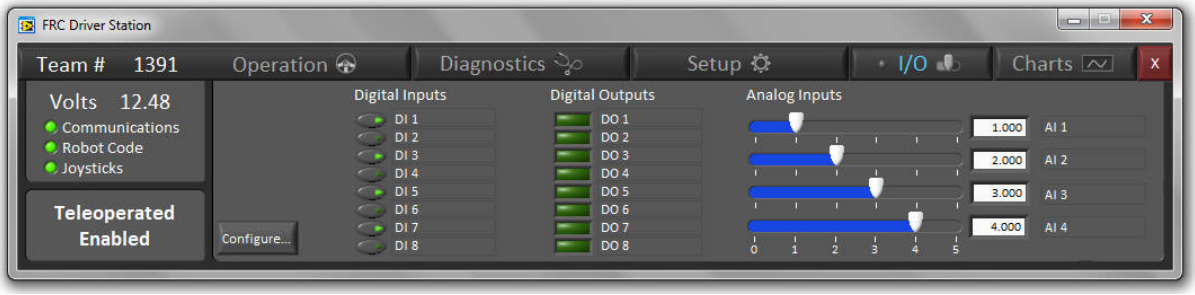


FIGURE 4.15 The I/O tab of the Driver Station showing digital and analog controls for sending information to your robot to be read at the start of Autonomous.

If you use the analog data channels, do not write code that depends on the exact value of the number passed. For example, if your code is designed to test whether x is equal to 3.14159, it will never find that it is. Your analog numbers are divided by 5, multiplied by 1024, and passed as integers. This conversion is undone on the robot, but the truncation leads to small inaccuracies. You can only set positive numbers using the analog sliders, but you can type negative numbers into the text windows. Since the integers used to pass the data down to the robot are signed, you will probably get a correctly converted negative number at the robot, but you should verify this works as you expect.

You can also see from Fig. 4.13 that there are also eight digital "outputs". There is an additional VI on the **DriverStation ▸ Compatiblity I/O** palette that will allow you to write values to these indicators. They can be used for diagnostics, but are probably too small to be useful during a competition. If your robot needs to send you a message, you should use the Dashboard, where you are free to design your own big, bright, flashing signals that could possibly be noticed amid all the excitement of a competition.

Disabled, Finish, and Test

As I mentioned before, the Finish routine is never called at a competition. In fact, there is no way for the Driver Station to call it. It can only be invoked by pressing the Finish button on the front panel of Robot Main. If you are not going to delete the call (by making the Finish case in Robot Main empty), then you might want to edit this VI so that reference names match your actual reference names, etc., but nothing bad will happen if you don't. In the worst case, you will get an error message when you press that Finish button. But you were already stopping the program, so it doesn't matter.

Disabled.vi is another matter. This VI is constantly being called when your robot is powered up, but not yet enabled. Fig. 4.16 shows the default code. Notice that this code is called during a very important time in a competition match: after your robot is powered up on the

field, but before the match has started. At this point, Begin.vi has already finished, and Periodic Tasks.vi is running. If you need to read a sensor or perform a calibration so that everything is ready when Autonomous starts, now's a good time to do it. You can't run any motors, of course, nor any pneumatics, but you can read all the analog and digital inputs from the cRIO, and you can write to global variables.

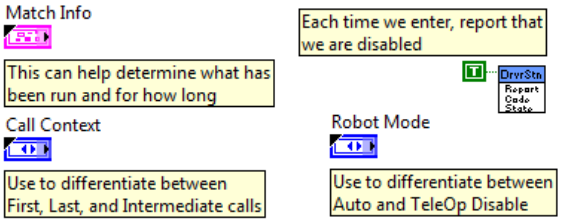


FIGURE 4.16 The default Disabled code that comes with the Framework.

One thing you must not have in Disabled is any loops, particularly no infinite loops. There is no external mechanism to shut this VI down, so if a loop is running when the match starts, your robot will not start. It will stay in the disabled state until that loop finishes. If the loop never finishes, then your robot is stuck for the duration of the match. I can promise that if you do this, your teammates will yell at you. Graciously, and with professionalism, of course, but they'll be yelling all the same.

Finally, there is a Test state in Robot Main that invokes the Test.vi (which you can find in the project tree). It's called by reference, just like your Autonomous code. In only makes sense to talk about Test Mode in the context of the Dashboard, so we will make no further mention of it here.

Building, Running, and Debugging

Building and Running

As we have discussed, you can run your code on your robot by clicking the white arrow on Robot Main.vi. That puts the code in RAM, which means it will be gone as soon as you turn off the power (actually, as soon as execution stops). For competition, you have to build and deploy your code. If you expand Build Specifications in the project tree, you will see FRC Robot Boot-up Deployment (as in Fig. 4.17). If you are deploying for the first time, right-click and select **Properties…** On the first page (Information), look for the "Local destination directory" field. You can change this to suit yourself, or leave it as the default.

When you are ready, right-click on FRC Robot Boot-up Deployment, and select **Build**. Once that process is complete, make sure your Driver Station is communicating with the robot, right-click again and select **Run as startup**. That's it. If there are no communication problems, your code will download and the robot will reboot. (If there are communication problems, just try again. If it doesn't work the next time, turn your robot on.) When cRIO comes back after the reboot, your code will be running in ROM, and will start running every time you power up the robot. If you need to make changes to your code (which you won't, because everyone's first try is always perfect…), just build it again and set it as startup. The new code will replace the old and run when the cRIO starts.

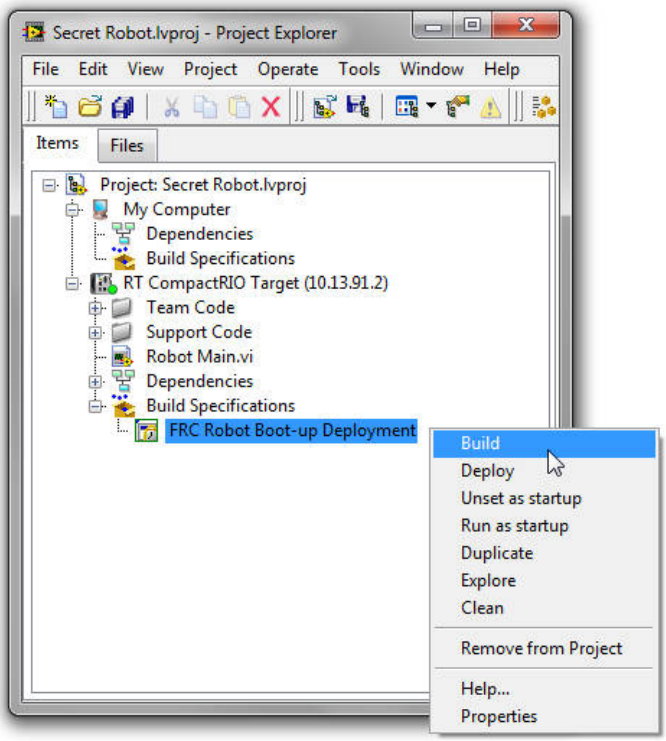


FIGURE 4.17 Building and deploying your code.

You can **Unset as startup** to leave your code in ROM, but have it *not* automatically start when the robot re-boots. This will let you load and run code in RAM for further development. If you want to remove the code entirely, you will have to use the Imaging Tool to re-image your cRIO. That leaves you with a "blank" robot. (I'm not going to talk about the Imaging Tool, because I am assuming you've already learned this kind of stuff from the official FRC documentation distributed at the start of every competition season.)

Debugging

As I have already mentioned, your code will work perfectly the first time, and if it doesn't, you should feel very, very bad. OK, that's a lie. *Nobody's* code works right the first time, and I really mean nobody. And when it doesn't work right, you need tools to figure out why. LabVIEW has a number of such tools built in. The simplest is the Light Bulb. On the block diagram, a few icons to the right of the Run arrow, is the Highlight Execution icon (a light bulb). Click that and your code will slow way down, little dots will show your data flow, and you will be able to see the values being passed into and out of structures and sub-VIs. This will work well for small programs running on a PC or laptop. It will not work for the Robot Framework.

More useful is the Probe, which lets you see the values passed in data wires as the program executes. If your program is running, switch to the diagram. Every time you bring the mouse near a wire, the cursor will switch to a little "P" symbol, as shown in the upper left of Fig. 4.18. Click on the wire, and you will insert a probe. The upper right of the figure shows a simple program with three probes inserted. Below is the window that pops up to show you the contents of each probe. This method will work on your robot program when it is running in RAM and you have access to both the front panel and the diagram.

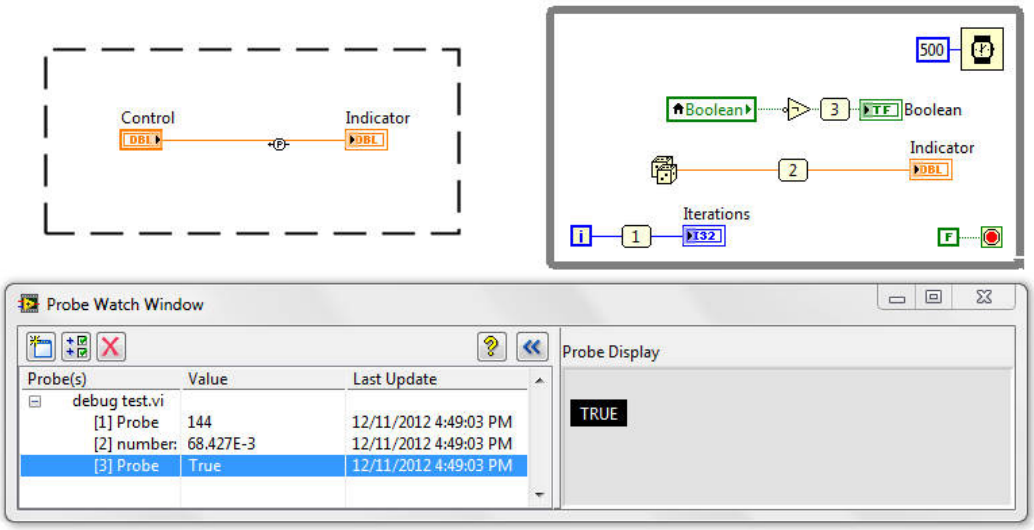


FIGURE 4.18 Probing your code for debugging.

If the program is not running, you can right-click on any wire and select **Probe** from the context menu. This will install a probe in the selected wire, which will then update once you start the program. You can also set breakpoints. When the execution of the program gets to a breakpoint, it will stop. You can then use the toolbar to resume execution, or step through it bit by bit. This can be very useful, but it expect it not to be much use in debugging your robot because halting your code messes with communications and timing.

Often, the most useful thing to know in order to understand what your robot is doing is to record the history of the states that the robot goes through. Fig. 4.19 shows a bit of the code for our Rocket Launcher example from Chapter 2, which I have modified to record the state history. The middle blue wire is the state variable, and holds the value of the next state the

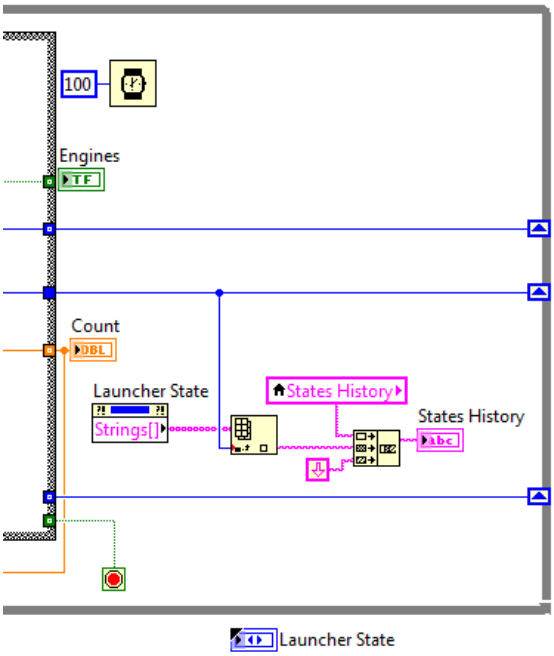


FIGURE 4.19 Recording the history of a state machine.

machine will go to. I started by right-clicking on this line to create an indicator. Since we don't need to read this indicator, I moved it outside the loop. You can see it at the bottom of the figure. Then I right-clicked on this (otherwise unused) indicator and selected **Create ▸ Property Node ▸ Strings[]**. You can see the property node labeled Launcher State in the diagram. The output of the node is an array containing the strings you used to label your states. Use the actual value of the state variable to index that array and concatenate it (with a line feed!) into a string indicator. You should make that string indicator a global variable in Robot Global Data. You can stretch the indicator vertically and also right-click on it and select **Visible Items ▸ Vertical Scrollbar**. When you run the program, you will get a history of the state machine. If it is a Teleop state machine, a new value gets added every 100 ms, so prepare for a lot of scrolling. Even so, this is a very useful trick.

Exercise E10: Modify your Launcher program to include the code shown here. Stretch the front panel indicator for States History so that at least 25 lines of text can be seen. Figure out how to use a property node to control the scrolling so that once the first 25 states have been appended to States History, is scrolls *upward*, so the most recently visited states are always visible.

Of course, if your code is deployed, you can't use the debugger. If you need to understand what is happening inside your code, you will need to send data to the Dashboard, or text messages to the Driver Station (see below). And you will need to know in advance which variables need to be watched. This is not nearly as useful as a debugger, but if you have a very specific issue that needs to be resolved, it can be better than nothing.

The Driver Station

The Driver Station (DS) is the interface between your robot and the rest of the world. Your joystick and other control signals to the robot pass through the Driver Station, as do commands from the field system at a competition. So, it's worth knowing a bit more about how this piece of software works.

As I mentioned back in Chapter 3, if you are running Windows 7, you need to be logged on as an administrator in order for the Driver Station to work properly. You launch the DS by double clicking on the Desktop icon that was installed when you installed LabVIEW and the FRC extensions. Doing so will also launch the Dashboard. If you have it set up to do so, the DS will also reach into the innards of your laptop and change the IP address settings on both your wireless and wired network connections. The number one reason for communication problems between your Driver Station and your robot is forgetting to set your IP address correctly. Having the the Driver Station do it for you will simplify your life. You can turn off this feature from the Setup tab, discussed below.

Switching your IP connection back to DHCP can be a pain using the Windows GUI. Instead, create a text file with the following contents:

```
@echo off
netsh interface ipv4 set address "Wireless Network Connection"
dhcp
exit
```

Name this file **wifi.bat** or something similar, and put it in a folder that is in your computer's path (*e.g.*, in C:\Windows). Now you can just type "wifi" in the Windows Start Menu to go

back to normal operation. For Windows XP, replace `ipv4` with `ip`. To reset your cable connection, replace `Wireless Network` with `Local Area`.

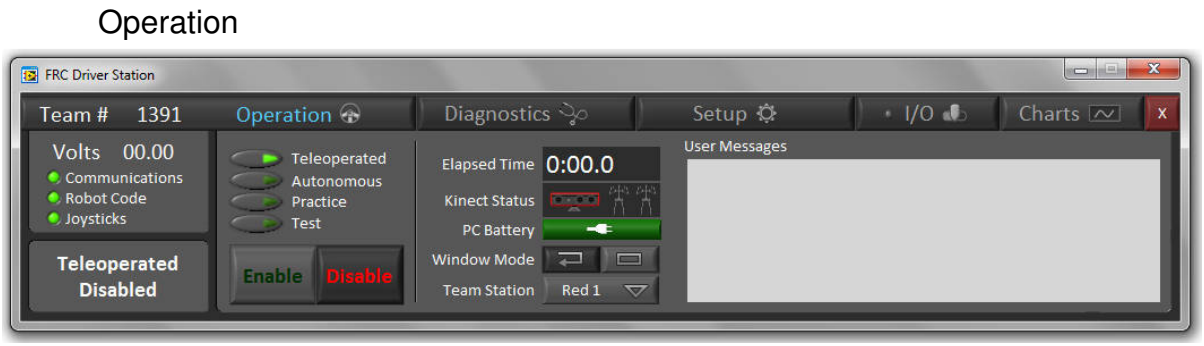


FIGURE 4.20 The Operation tab of the Driver Station.

Fig. 4.20 shows the Operation tab. At the left, you can see the battery voltage, the fact that the DS is communicating with robot, that the robot has code, and that joysticks have been detected. Below that is a status message telling us that (in this case) the robot is disabled, but that it will run in Teleop mode when it is enabled. To the right of these status indicators are four switches that control what part of your code will run. If you select Teleoperated or Autonomous, then only the corresponding code will run. In either of these modes, the code is allowed to run for as long as it wants, or until you disable it. If you select Practice mode, then the DS will simulate a competition match, first running your Autonomous code, then your Teleop code, and then disabling your robot. (You set up the timing for this mode on the Setup tab, discussed in a bit.) If you select Test, the robot will go into Test Mode. Read Chapter 8 before doing this

To start things running, you hit the Enable button, or F1 on your laptop. To disable the robot (sending Robot Main to the Disabled state), hit the Disable button or the Enter key on your laptop. To emergency stop the robot, hit the space bar on your laptop. That's a nice big target, but you must re-boot the robot following an emergency stop (or re-load your code if you are running in RAM).

Moving over one column to the right, you can see the elapsed time indicator. In practice mode, this will start over at zero when Teleop starts. Below the elapsed time display is the Kinect status indicator. Below the Kinect status is a pair of buttons which control whether the DS is a floating window (as shown in Fig. 4.20) or locked down in the lower left corner of the screen ("competition mode"). Finally, at the bottom is a control that lets you tell the robot your alliance color and driver station (read out by a VI shown in Fig. 4.14).

Finally, on the right-hand side is a User Message window where your robot can send text messages. The VI for sending these messages (Write User Message) is found on the Driver Station palette. This facility can be useful for debugging, but should be considered totally unreadable during a competition. The text is just way too small to be noticed and read in the heat of battle.

Fig. 4.21 shows the Diagnostics tab. You can see the LEDs indicating that we have two joysticks, but no Kinect, and that there is communication to the wireless bridge, and to the robot. Note that one of the joystick LEDs is blue, indicating that a button on that joystick is being pressed. This allows you to tell which joystick is which. If the joystick designations are

wrong, you can change them on the Setup page. At a competition, when you are properly connected to the field, the FMS (Field Management System) LED will be lit. Occasionally the DS will not find your joysticks and all the LEDs will be gray. Usually un-plugging and re-plugging one of the USB connectors will trigger some magic and the LEDs will go green.

Diagnostics

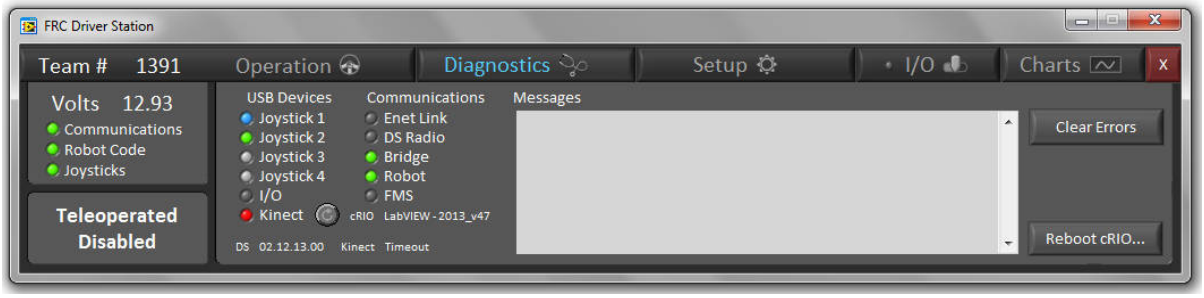


FIGURE 4.21 The Diagnostics tab of the Driver Station.

Next to the status LEDs is the error message window. When your code throws errors, they can be read here. About ninety percent of the time, the only kind of error you need to worry about is a Death Spiral error. Fig. 4.22 shows that kind of error from a robot that was purposely put into a Death Spiral. This particular one is from RobotDrive, but you can get very similarly worded messages from other subsystems, *e.g.*, Solenoid Set or MotorControl SetOutput. In each case, your code is taking longer than 100 ms to complete an iteration.

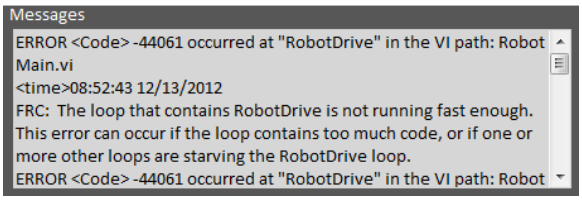


FIGURE 4.22 Error messages from a Robot in a Death Spiral.

Be aware: you *will* occasionally get message like these from a robot that is running perfectly fine and has no problems. The communication system between the DS and your robot occasionally drops packets, which can cause errors of the type shown in Fig. 4.22. If you are uncertain, hit the Clear Errors button to the right of the message window. If things are fine, the window will stay empty for a while. If things are not fine, the window will immediately refill with "robot too slow" error messages.

Below the Clear Errors button in Fig. 4.21 is a button to reboot the cRIO.

Setup

In Fig. 4.23 you can see the Setup tab of the DS. There's a spot to type in your team number, and below that a button choose your network interface card (NIC). Hitting this will pop up a window that will let you choose whether to have the DS change your IP settings, or do it yourself. The controls are separate for the cable and wireless network connections. Below that you can switch between using a local Dashboard and a remote one. I've never done that, so I can offer no guidance on the how or the why.

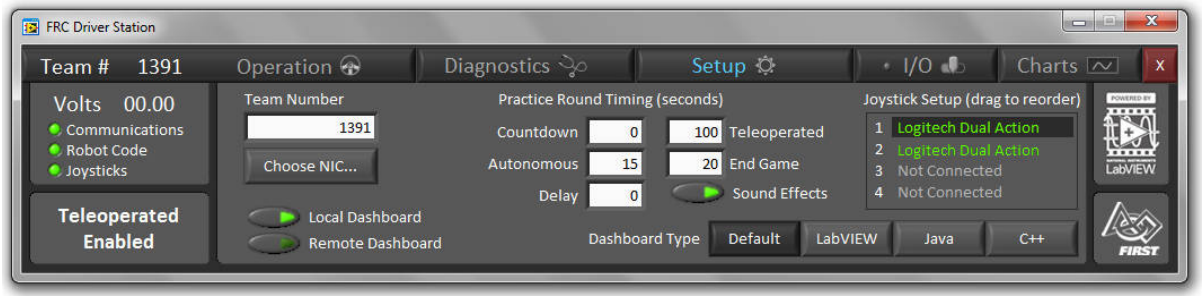


FIGURE 4.23 The Setup tab of the Driver Station.

In the center of the tab are the fields for setting up your Practice mode timing. You can also turn on the standard FRC competition match sound effects, if you want to step up the realism by a notch. Endgame is just a continuation of Teleop. It makes no difference to the operation of the robot. It only controls when the Endgame sound effect gets played.

Further to the right is the joystick list. If your joysticks are out of order (as determined using the Diagnostics tab), then you can click and drag them to have the order you want. When they are identical devices, as in Fig. 4.23, you want to double and triple check that you've got it right, especially when getting ready for a match.

Finally, below the joystick list is a row of buttons that allow you to select which Dashboard you will be using. Since this is a LabVIEW book, you'd think I would recommend pressing the LabVIEW button. Alas, it is not documented, so there's no way to tell how to point it at your own custom Dashboard. Leave the Default button pressed and replace the default Dashboard with yours (see Chapter 8).

I/O

This tab, shown in Fig. 4.15, was discussed in the Autonomous section. There's nothing to add here.

Charts

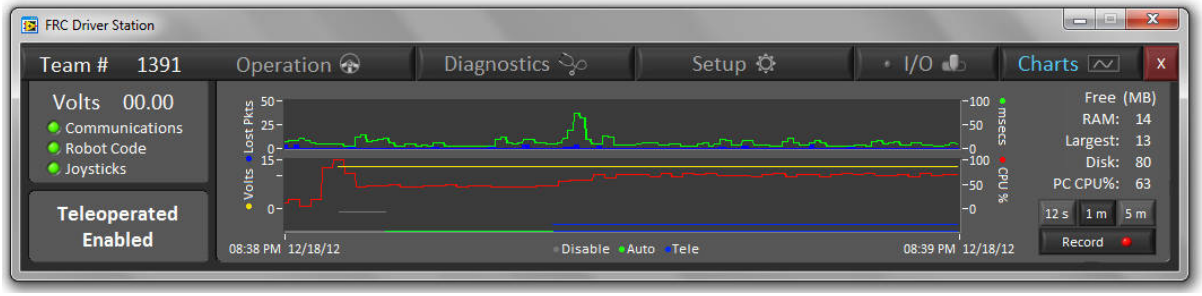


FIGURE 4.24 The Charts tab of the Driver Station.

The charts tab, shown in Fig. 4.24 is quite helpful for diagnosing problems with your robot. In the upper graph, you can see the number of lost packets (in blue) and the round-trip communication time between the robot and the DS (in green). This does not depend directly on your code, because communication is handled by the Robot Start Communication routine. It depends indirectly on your code, however, because if your code hogs the CPU, the communication routine won't be able to run as fast, and that round-trip time will start to climb.

The red line in the lower graph is the most important one here. It's the CPU usage line. In the figure, you can see it spike (at the left end) as code is downloaded to the cRIO. During Disabled and Autonomous, the code is not doing much of anything, and you can see that the CPU usage is below 50%. (You can tell the robot mode by looking at the gray, green, and blue lines along the bottom of the lower graph. They tell you the mode of the robot.

When Autonomous ends, you can see the CPU usage rise to an unhealthy 80 or 90 per cent. Teleop contained a While Loop crammed with floating point math and no delay. The loop timer was set at 200 ms and thus jammed up the error handling pipe line. Don't do this. Make sure your CPU percentage is around 60% or less.

Also on the same graph is the battery voltage (in yellow). This code, while killing the CPU, ran no motors, so the graph is flat. If you run motors, you will see this dip into the range of 8 to 9 V. If it drops too low, you need to recharge the battery. If it dips too low when the battery is freshly charged, you need a new battery. Also in this graph are some fainter lines that tell you what state Robot Main is in: gray for Disabled, green for Autonomous, and blue for Teleop.

To the right of the graphs are some buttons to select the length of time covered by the graph, and a button to control whether the data displayed in the charts is recorded to disk. The default is to record the data, and you should leave it on. The viewer is in C:\Program Files\FRC Driver Station. The data is stored elsewhere, but the viewer knows how to find it. The files have not been cleverly named, so the resulting alphabetic sorting can make it a pain to find the file you want. Unfortunately, all that is logged is what is shown in the charts. It would be very nice if the error messages that show up in the Diagnostics tab were also logged, but they are not.

Above the logging and graphing controls are some indicators that show you how much free RAM the cRIO has, the largest single chunk of free RAM, and how much free space is left on the cRIO's "disk". Since the cRIO doesn't have a disk, I am assuming this refers to the amount of free EPROM memory.

Developing Code Without a Robot

Face it. Build season is just barely 6½ weeks long. If you are very lucky, you will have a robot to test your code on in the last week of the season. What are you going to do in the mean time? You are going to develop your control logic, plan and code your state machines, and test them on your laptop in "simulators" that you will design yourself.

To be specific, consider Fig. 4.25, which shows a snippet of the Teleop code from our 2012 Rebound Rumble robot. This robot had horizontal belts that brought a ball in to the middle of the robot (called the Pickup), and vertical belts that carried the balls upwards to our shooter (called the Queue). The robot had three infrared sensors, only two of which concern us here. The Pickup sensor (the upper digital input in Fig. 4. 25) was at the front of the robot. If a ball blocked it, the Pickup state machine (the VI named Pickup) went from the Idle state to the Running state, where the motor for the horizontal belts was turned on. When the ball got to the middle of the robot, it would trigger the Center sensor, which caused the Queue state machine to go into action. This state machine would use the state of the Center sensor to figure out when it had picked up the ball, and would (via a global variable) turn off the

Pickup. The actual logic was a bit more complicated than I have described, so it was very important to be able to test it thoroughly.

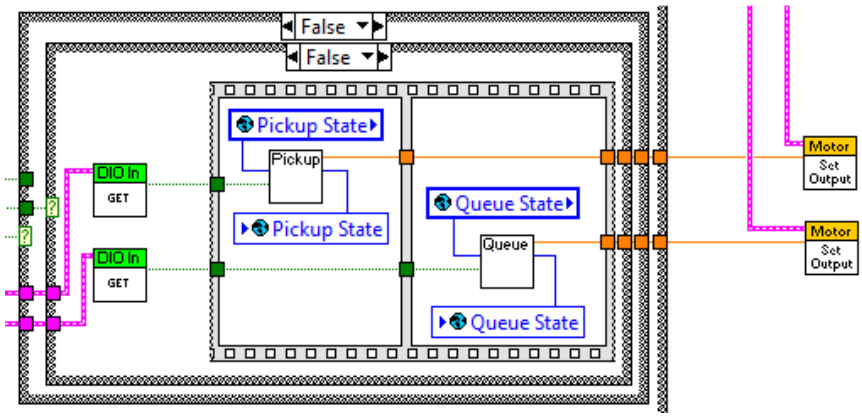


FIGURE 4.25 A piece of Teleop code from Team 1391's 2012 robot.

To test it, we went to the Project Tree, and dragged the Pickup and Queue VIs up to the part of the Tree for code that runs on "My Computer" rather than on the cRIO. We also dragged up Robot Global Data and our Check Timeout VI, which both Pickup and Queue call. We used all of these in a simulator, as shown in Figs. 4.26 and 4.27. Fig. 4. 26 shows the front panel. You can see that there are two slider-type controls that are used as indicators to show the position of the ball. The front sensor is a switch, so the user can simulate the arrival of a ball. The center sensor is controlled by the logic of the simulator, shown in Fig. 4. 27. I won't go through the details here, because they are very specific to our code, and yours will be different.

But there are two important take-aways: first, you can simulate large parts of the operational logic for your robot without the actual robot, and second, it will be a fair amount of work, and possibly quite tricky to get the simulator to work correctly. Unfortunately, you need to have the simulator working properly to get your robot code working. Probably they will both circle in to the correct solution at the same time. When you are confident that your VIs are ready, you can go back to the Project Tree and drag them back to the lower, cRIO section.

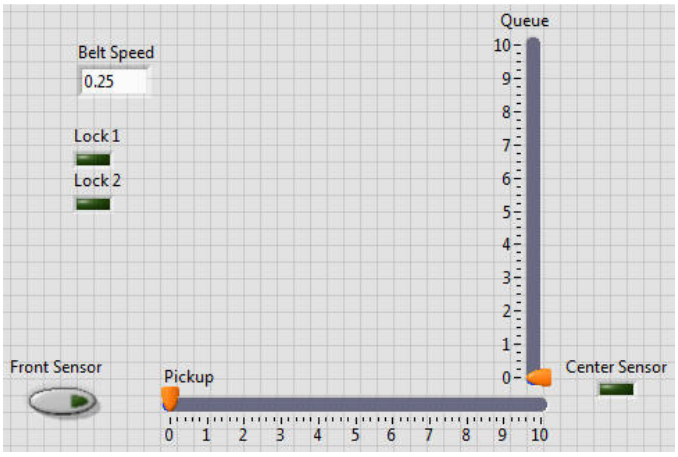


FIGURE 4.26 Front panel of a code tester/robot simulator. Booleans take the place of the actual robot sensors, and sliders are *indicators* representing the balls

moving in the belt system. The stuff in the upper left is related to making the simulator work, and not part of the code being tested.

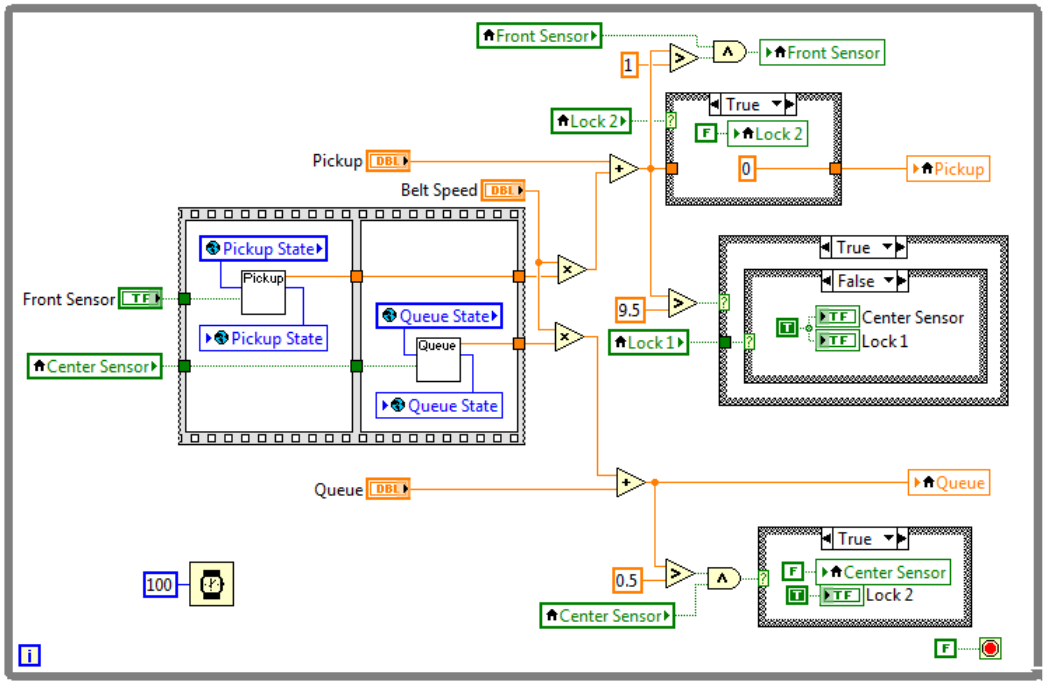


FIGURE 4.27 Block diagram of the simulator. Only the flat sequence contains the code being tested. The rest of the code is just there to allow the testing.

Chapter 5 — Input and Output

In a way, you could argue that this is the most important chapter, because it covers the point at which your software actually meets the robot. It describes how to connect things that are "outputs", like motors and solenoids, and things that are "inputs", like limit switches, encoders, and accelerometers. Alas, this book is not the best place to learn about this stuff. The best place is contained within your copy of LabVIEW itself. Go to the LabVIEW **Help** menu and select **Find Examples…** This will open up the NI Example Finder. If you look down the list of folders, you will see one called FRC Robotics. If you open that folder, you will find more folders, and within those, extremely well documented examples of both how to write code for these objects, and how to physically wire them. So, rather than duplicate all of that careful effort, this chapter will serve mostly as a kind of quick reference, with a few extra insights and observations tossed in here and there.

The Interface Sheet

OK, this section is *not* about something you will find in the LabVIEW examples, so you see the lies continue unabated.

If you want to avoid panic in the pits and other desperate measures to get things working at the last minute, you will need an interface sheet. This sheet tells you what things are plugged in where. It should be posted on the wall of your shop and—most critically—it should be posted up in your pit at a competition. If a PWM or sensor cable gets pulled out, this sheet will help you get it put right quickly. Especially if you have carefully labeled every cable…

This can be a very simple document. As an example, Fig. 5.1 shows our interface sheet from 2012. You will notice that we appear to have "skipped" using some inputs. Actually, we were using them, but then had to take some stuff off to make weight. Re-assigning all the inputs would have taken too much time, would have created new opportunities for mistakes, and is totally unnecessary. Better to just leave things where they are and have blanks.

Channel	Name	Inverted?		Channel	Name
DIO 1	Front Ball Sensor			USB 1	Driver Joystick
DIO 2				USB 2	Operator Joystick
DIO 3	Center Sensor				
DIO 4	Exit Ball Sensor			AI 1	Turret Voltage
DIO 5	Shooter A	Yes			
DIO 6	Shooter B			Solenoid 1	
DIO 7				Solenoid 2	
DIO 8				Solenoid 3	
DIO 9	Right Drive A	Yes		Solenoid 4	
DIO 10	Right Drive B			Solenoid 5	
DIO 11	Left Drive A			Solenoid 6	
DIO 12	Left Drive B			Solenoid 7	
DIO 13					
DIO 14					
PWM 1	Right Drive				
PWM 2	Left Drive	Yes			
PWM 3	Pickup Drive				
PWM 4	Queue Drive				
PWM 5	Turret Drive				
PWM 6	Shooter Drive	Yes			
PWM 7	Collector Lower				
PWM 8	Brush Drive				
PWM 9					
PWM 10					

FIGURE 5.1 A sample interface sheet.

Joysticks

Fig. 5.2 shows how to wire up a joystick. The left diagram goes in Begin.vi, and the right diagram goes in Teleop. The Scaler value is an optional input to make your joysticks more sensitive. Here's how it works: the raw joystick value is an 8-bit signed integer, so it can have (integer) values from -128 to 127. The raw value is divided by the value of Scaler to give a real number output between -1 and 1. Scaler is also an 8-bit signed integer, so it can't be larger than 127 (which is the default value if you don't wire anything to this input). If Scaler is, say, 64, then the joystick (real number) output will reach 1.0 when the joystick is only at half range. As you move the joystick further, the output remains "clipped" at 1.

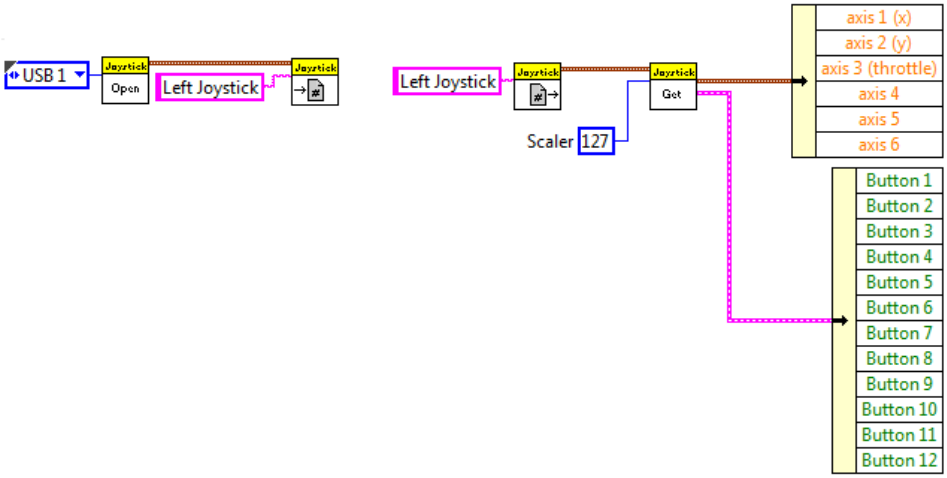


FIGURE 5.2 Joystick wiring. The left hand bit of code goes in Begin.vi. The right-hand bit goes in Teleop.vi.

Motors

Motor control VIs come in two types: "simple" controls that make programing your drive motors easy, and "advanced" controls that allow you to wire up individual motors yourself. In certain respects, the "advanced" controls are simpler. Fig. 5.3 shows the wiring for a single motor. The Open VI is "polymorphic", which means that it can be one of several versions. You have to select from the drop-down list so that the type you are using matches your actual type of motor controller.

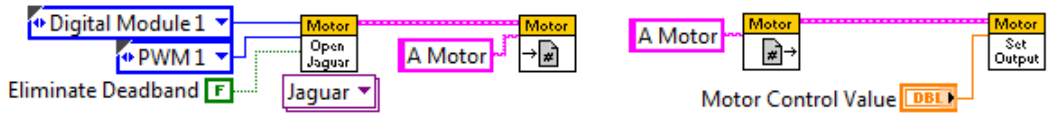


FIGURE 5.3 Single motor wiring. The left hand bit of code goes in Begin.vi. The right-hand bit goes in your drive code (which could be Teleop.vi, Autonomous Independent.vi, or Periodic Tasks.vi). The polymorphic selector on the Open VI is set for a Jaguar controller.

The output from a typical joystick controller is rarely zero, even when the joystick is centered so that you expect there to be no signal to make your motor turn. To protect you from unwanted motion from a "zeroed" joystick, the motor control system has a built-in "deadband". Small signals (near zero) that are within the deadband are ignored. You may not always

want this (for example, if you are controlling a motor with a PID), and the Eliminate Deadband input on the Open routine allows you to turn it off by wiring a True value to this input.

Robot Drives

Depending on your robot design, the required code for driving it can be quite complicated. The FRC code package included pre-made VIs for the most common drive schemes. Fig. 5.4 shows the code for an "arcade" style drive, in which a forward-back joystick motion controls the robot speed, and a left-right joystick motion steers the robot. You can use a single joystick for this, or separate ones (Fig. 5. 4 shows a single joystick arrangement).

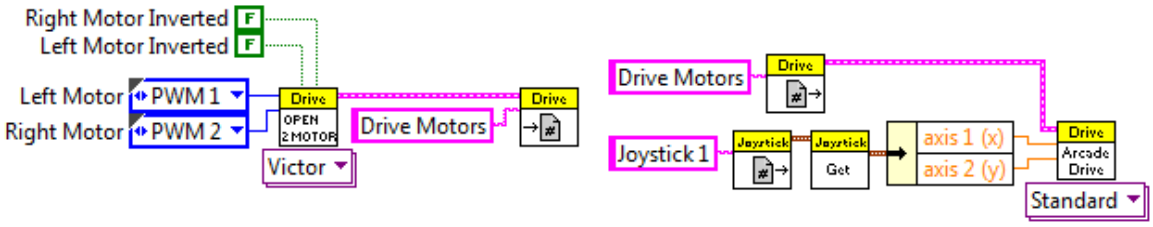


FIGURE 5.4 A simple arcade drive. The left hand bit of code goes in Begin.vi. The right-hand bit goes in your drive code (which could be Teleop.vi or Periodic Tasks.vi, but not Autonomous because it reads a joystick). The polymorphic selector on the Open VI is set for a Victor controller (we don't take sides here!).

In addition to Arcade Drive, there are VIs for tank and holonomic (or mecanum) drives. Be sure to use the Open 4 Motors VI with a mechanum drive.

Drive Delays

Back in Chapter 2, we discussed how to implement delays as part of a state machine so that you would not send your robot into a Death Spiral. Well, if you are absolutely determined to program your robot wrong (*i.e.,* not use state machines), then you might find the Drive Delay VI shown in Fig. 5.5 useful. The figure shows a robot set to move forward for 1 second. The VI in the middle is the delay. You tell it how long you would like to delay, and how often you should feed the Watchdog so that your robot is not stopped for safety reasons. In this case, the Watchdog is being fed every 50 ms. When the delay is over, the third VI executes, stopping the robot. The Drive Delay VI only operates with Robot Drive VIs. It won't work with the lower level motor control VIs. Bear in mind that functionally, this is still a blocking call. Whatever code contains this node can't complete or break away from what it is doing until the full delay time has passed.

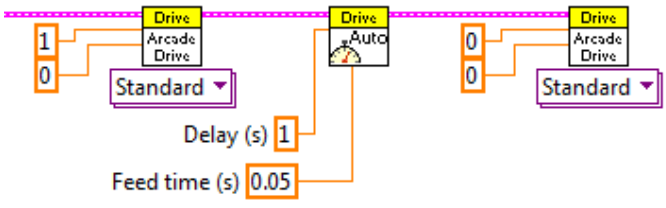


FIGURE 5.5 A delay done without a state machine, but also without annoying the Watchdog.

Digital Input and Output

Digital input and output (DIO) is done through the Digital Sidecar (connected to a 9403 module plugged into the cRIO). There are 14 available DIO channels which you can configure individually for either input or output (but not both at the same time). There are also 8 additional Relay channels. These are digital output only channels that can supply enough current to operate a relay, and also have some other features which we will discuss in a bit.

Input

Probably the most common use of digital input is to detect whether a switch is open or closed. Often this is a mechanical switch used to detect whether some part on your robot is at the limit of its allowed motion. The pressure switch that is part of the pneumatic system is another example of a switch that used with a digital input. Fig. 5.6 shows code configuring DIO channel 1 as an input, and also code to read the value.

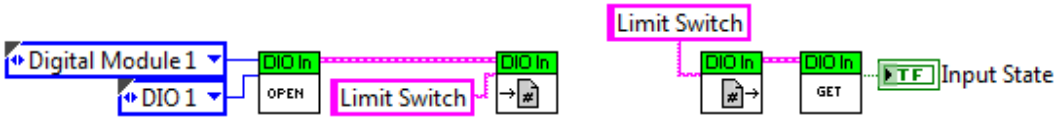


FIGURE 5.6 Code for a digital input. By now you can figure out for yourself which part belongs in Begin.vi and which part goes in your control code.

The DIO channel has three pins. One is a ground (marked "(-)"), one is +5 V (marked PWR), and the third is the signal, which in this case is an input. If the signal wire is connected to the ground pin, the input value will return False. If it is connected to the +5 V pin, then the value will return True. ***What if the signal wire is not connected to anything?*** There is what is known as a "pull up" resistor somewhere in the electronics, so if the signal wire is not connected to anything (or "left floating," as we like to say), the input will be "pulled up" to +5 V and the value will return True.

It is not great design to rely on this pulled-up True value. For example, consider the standard microswitch that you get every year in the Kit of Parts. It looks something like Fig. 5.7. Suppose you want a True input when the switch is closed (by pressing the lever). Hook up the signal wire to the common terminal, hook the ground wire to the NC ("normally closed") terminal, and the +5 V wire to the NO ("normally open") terminal. Now the signal wire is connected directly to ground and the DIO input will read False—until the lever is pressed. Then the NC contact opens and the NO contact closes. Now the signal wire is connected directly to +5 V, and the DIO input will read True.

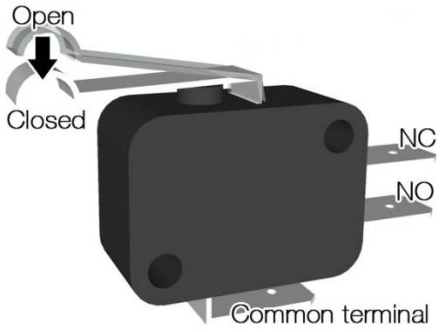


FIGURE 5.7 A standard microswitch

Output

Fig. 5.8 shows code to configure and use a digital output. Not surprisingly, it looks a lot like the digital input code. If the control Output State is True, then the signal wire will be pulled up to +5 V, and if it is False, then the signal wire will be pulled down to 0 V.

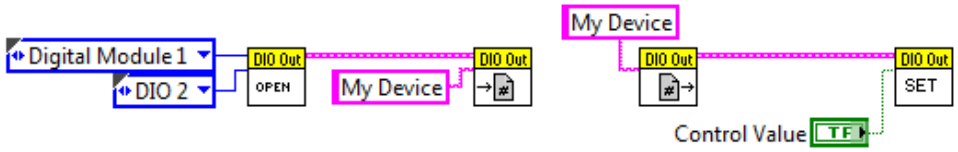


FIGURE 5.8 Code for a digital output

Be aware that if you do nothing to a DIO channel, but simply leave it un-configured, the signal wire will float up to about +4 V. It probably can't supply much current in this state, so it couldn't close a Spike relay, but it might be able to activate other devices that read a voltage without drawing much current (a device with an FET input, if you want to get technical). This situation can occur if you configure DIO channel 8 to control your device, but accidentally plug it into channel 9, which you didn't configure. So pay attention.

Relays

Like the DIO channels, the relay channels have three pins, but the function is very different. This is because, as a bonus feature, the relay channels have been configured to allow you to run a motor off a Spike relay. The motor runs only at one speed—full on—but you can run it in either direction.

Let's start with the code, shown in Fig. 5.9. When you open the relay channel, you have to specify "Forward", "Reverse", or "Both Directions". When you set the relay, your choices are "Off", "On", "Forward", and "Reverse". The relay channel actually has *two* outputs, imaginatively named "A" and "B" (squint hard at the Digital Sidecar and you will see the label telling you which pin is which). Table 5.1 shows you how each output pin depends on the inputs to the Open and Set VIs.

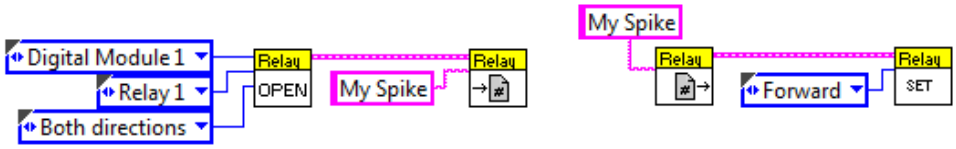


FIGURE 5.9 Code to set up a relay.

If you set up the Relay Direction to be "both" then you will be able to control the direction of rotation (but not the speed) of a motor powered through a Spike.

Encoders

Encoders are used to tell you how far and how fast a shaft has rotated. Typically, you have these attached to some rotating parts in your drive train, and you can use them to move the robot a specific distance (useful in Autonomous) or at a specific speed. As you can see from Fig. 5.10 below, a quadrature encoder (the type typically included in the FRC Kit Of Parts, and the most useful for a robot) uses up two DIO channels.

TABLE 5.1 In this table "1" represents the output pin at +5 V.

Relay Direction	Relay Value	A	B	Relay Direction	Relay Value	A	B
Both Directions	Off	0	0	Forward	Forward	1	0
	On	1	1		Reverse	0	0
	Forward	1	0	Reverse	Off	0	0
	Reverse	0	1		On	0	1
Forward	Off	0	0		Forward	0	0
	On	1	0		Reverse	0	1

Unlike the sensors we've discussed so far, an encoder needs quite a bit of configuration in order to be useful. If you have a tank or arcade drive robot, and are including encoders, a bit of thought will convince you that unless you do something very unusual, one of the encoders will need Invert Direction set to True. Whether it's the left or right encoder will depend on how they are installed, which direction you define as "forward", and how you feel about negative numbers. The reason for using a quadrature encoder is that it can tell which way the shaft is rotating. Starting from zero, one direction of rotation will give a positive distance travelled, and the other will give a negative distance.

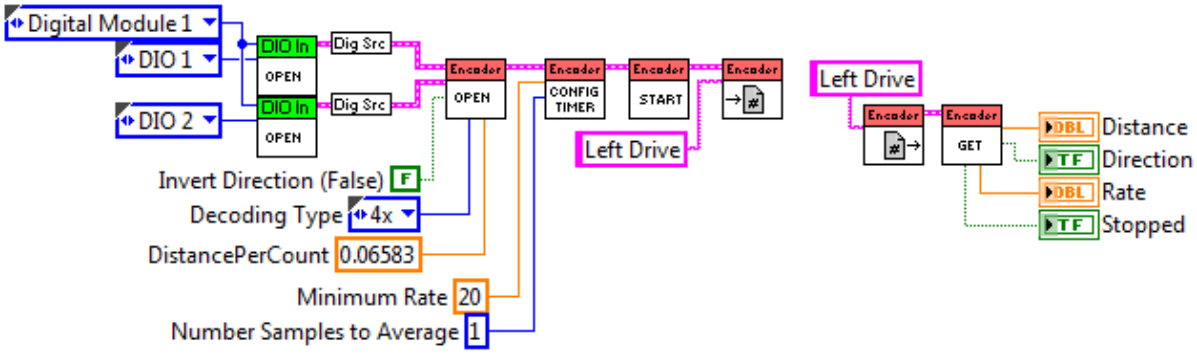


FIGURE 5.10 Code to open and read a quadrature encoder.

When you use the Get VI to obtain the distance, the number you get is the total accumulated distance since the encoder started. If it moves ten feet forward, and then ten feet back, the returned distance will be zero (or close to it). If the robot has been moving mostly forward for a whole competition match, the returned distance will be a very large number, which may not be convenient. For that reason, the Encoder VIs include a Reset VI. Whenever it is called, it resets the total distance travelled to zero. Fig. 5.11 offers some code showing how you might implement a reset under software control.

The Decoding Type configuration input selects the type of encoder you have, which will almost certainly be a quadrature one, so select "4x".

The DistancePerCount input is your calibration, so you want to take the time to get a good number here. Make a test program to set up and read your encoders (with the distance reported to the front panel so you can write it down). Be sure to include the ability to reset (zero) the encoder with a front panel control or joystick button. For now, set DistancePerCount to 1. Now zero the encoders and push or drag your robot a carefully measured distance over a surface similar to what it will see in competition. This distance should be large: 20 to 30 feet (6 to 9 meters in a unit system not based on the shoe size of a long dead English king).

Do this several times, and average your recorded distance (which is really just the number of actual encoder pulses, since the calibration factor is one). Now set DistancePerCount equal to the distance you dragged your robot divided by your average number of counts. You can use any distance unit you like. Just remember what it was!

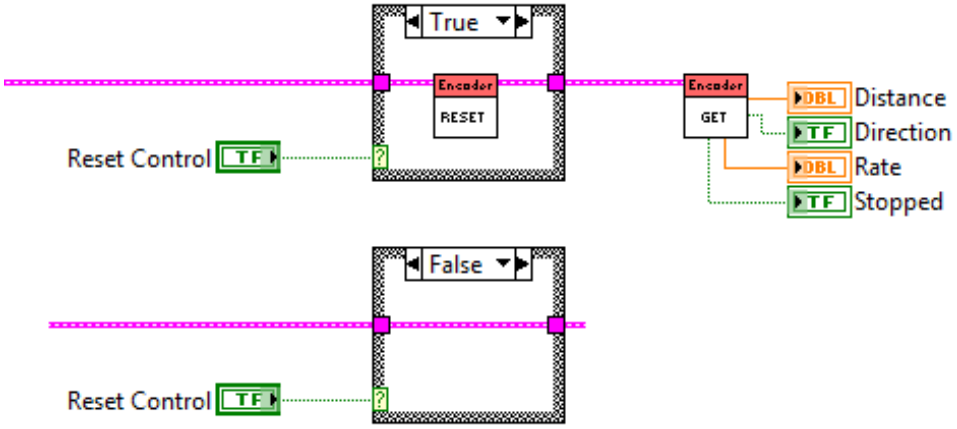


FIGURE 5.11 Sample code to reset and encoder.

As you can see from Fig. 5.10, the Get VI can also set a Boolean output True if it detects the robot is no longer moving (possibly useful in Autonomous). But under real conditions, the encoder on a fully stopped robot may still emit pulses at some rate. They will probably be alternating positive and negative pulses, but pulses nonetheless. If the number of these pulses per second is less than the value of Minimum Rate, the Get VI will consider the robot to be stopped and will set that output true. You are unlikely to need a value different from the default of 20.

According to the documentation, Number Samples to Average "…specifies the number of samples of the timer to average when calculating the pulse rate." At one point in the past, there was an acknowledged bug in the FPGA code for the encoder rate (in the 2011 season), and as a result, teams had to develop a "do it yourself" solution, such as is shown in Fig. 5.12. Of course, this solution doesn't do any averaging, and encoders can be a bit noisy. If needed, you can add a boxcar filter (which you will learn about in the very next section.)

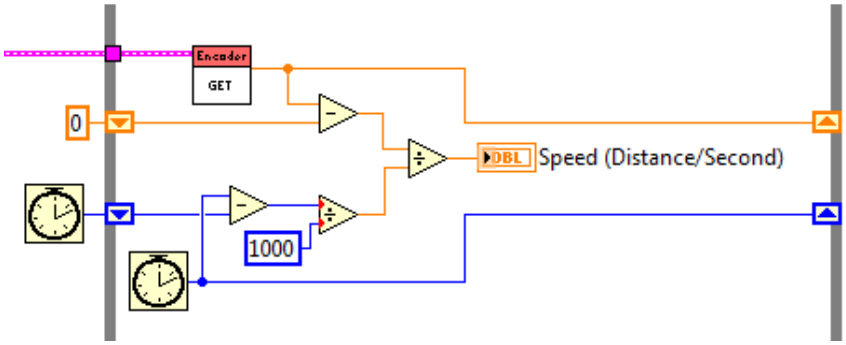


FIGURE 5.12 Code to calculate your robot's speed. This goes in a fast loop (10 or 20 ms loop time) in Periodic Tasks.vi. You should be able to figure out how to modify it to give some other rate (like the RPM of a shooter wheel) if that is what you need.

Analog Input

The analog input plug-in to the cRIO is used to read voltages directly. In truth, the digital inputs read voltages too, but then interprets them as either a zero or a one (True or False). If, on the other hand, you apply +5 V to an analog input, you will get a real number (e.g. 5.00) when you read it. There are many accelerometer and gyro sensors that are read using analog inputs (although the accelerometer distributed with the Kit of Parts lately has been a digital device). The FRC code supplies specialized drivers for some of these sensors. For others you need to do it yourself by reading a voltage (or voltages) and programing whatever math is required to convert that into a distance, angle, or whatever. I'm going to leave these specific sensor types to the provided LabVIEW examples, and only describe "vanilla" Analog Input, which will be enough to get you started. For planning purposes, keep in mind that while there are a total of eight analog inputs available on the cRIO plug-in, you only get to use seven of them, as channel 8 is used to read the battery voltage.

Fig. 5.13 shows the simplest possible to read an analog voltage. One problem with analog signals is that they can be noisy. (One of the things that made digital electronics such a brilliant invention was its noise immunity.) There are many possible solutions to noise. The best is to use large signals (a volt or more) and make meaningful changes in the sensor value large as well. For example, if you are using a potentiometer as an analog sensor to determine the rotational position of a device, try to set it up so that the smallest position change you care to know about results in a voltage change of 0.1 V or more. In that case, you most likely not be affected by the noise level.

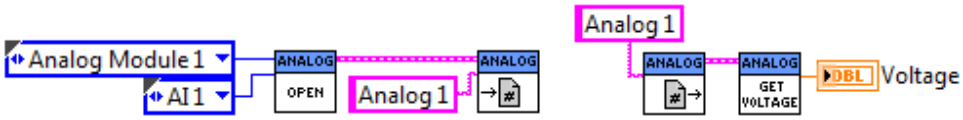


FIGURE 5.13 Simple code to set up and read an analog voltage.

Of course, it may not be possible to arrange for large voltage changes, and you may be stuck with a situation where your signal-to-noise-ratio (S/N) is not as large as you'd like. In that case you can improve the S/N by averaging. There is a Get Average Voltage VI that will give you a smoothed signal, but I'm going to recommend that you implement your own boxcar averaging scheme. The main reason for this recommendation is that the Get Average Voltage VI is a bit of a black box, and it is very clear what the boxcar average is doing. Plus, this way, you get to learn about boxcar filters, which I know you've always wanted!

Fig. 5.14 shows a simple boxcar implementation. An array of ten elements is held in a shift register. On each loop iteration, the array is rotated by one element, so the oldest data value (in position 9) is "rotated" into position 0, and then replaced by a new reading. You take the average of the array as your sensor value. Ten samples is kind of the optimum number of samples. For random noise, S/N improves as $\sqrt{n}$, where n is the size of the storage array. If n is ten, then you make things better by a factor of 3. If you make n equal to 100, then your averaged value will lag behind the input voltage, and your S/N will only be improved by a factor of 10. Fig. 5.15 shows the effect of ten and 100 sample boxcars on a simulated noisy voltage. Be aware that the lagging behavior is a characteristic of any filtering scheme, not just the boxcar, so if you try too hard to clean up your signal, your system response becomes "laggy."

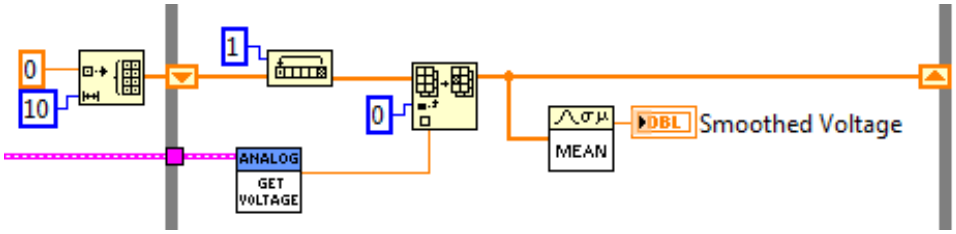


FIGURE 5.14 Code to implement a boxcar averager. This goes in a fast loop (10 or 20 ms loop time) in Periodic Tasks.vi.

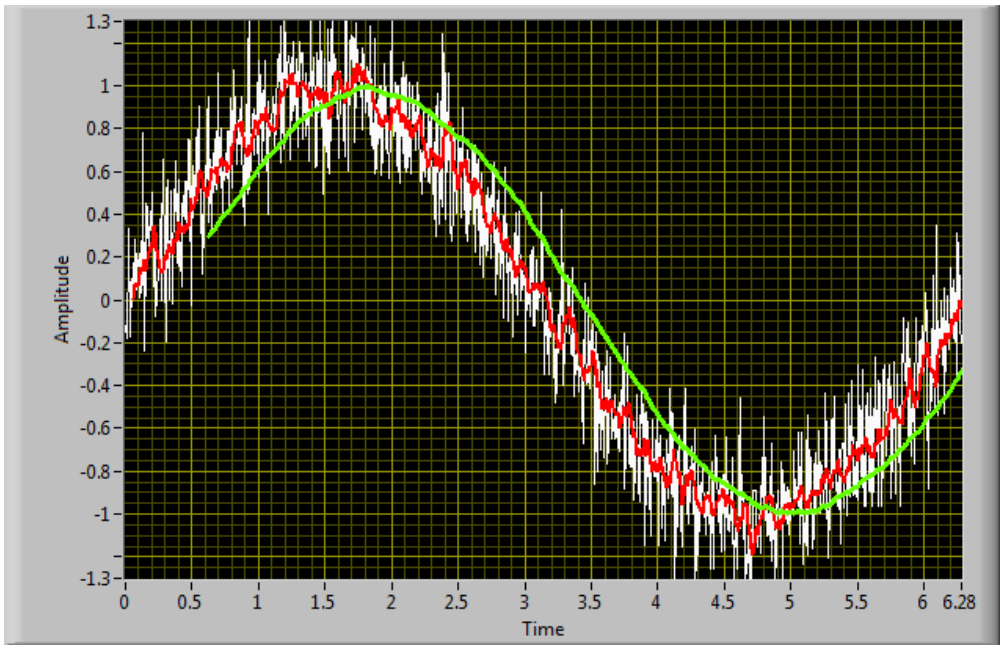


FIGURE 5.15 The original noisy signal is shown in white. The red curve is the result of a 10 sample boxcar. The green curve is the result of a 100 sample boxcar. You can see that it is much smoother, but also lags the original signal by quite a bit. The gap at the beginning occurs because the output of a boxcar is garbage until the array has been filled with measurements.

Pneumatics

The FRC pneumatic system is actually two systems: one to control the compressor that provides the necessary supply of high pressure air, and one to control the solenoid valves that operate your pneumatic cylinders.

The Compressor

The compressor system has—from a software point of view—two key components: a digital input that tells the system to turn the compressor on or off, and a relay output that actually turns the compressor on or off. Setting up these inputs and outputs is handled by the compressor VIs. You don't have to set them up yourself, but you do have to tell the compressor system which channels you are using. Fig. 5.16 shows the appropriate code, both the part for Begin.vi, and the part that goes in Periodic Tasks.

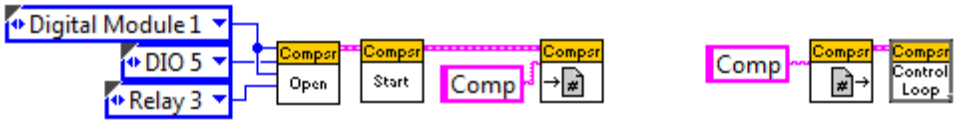


FIGURE 5.16 Left: code to set up the compressor. In this example, the DIO channel for the pressure switch and the relay channel for the Spike relay are both connected to the same Digital Sidecar, but they could be on separate ones. Right: code to run the compressor, which goes in Periodic Tasks. Note that the Control Loop is already a loop, and should *not* go inside a While Loop.

In principle, you do not need to know the following, but it may help you troubleshoot when things aren't going right: the pressure switch is normally closed (causing the compressor to run), and opens when the pressure reaches 120 psi (stopping the compressor). The switch will stay open until the pressure has fallen all the way to 100 psi, at which point it will close again. This hysteresis ensures that the compressor runs occasionally to top up the pressure, instead of running continuously.

Valves

We can't talk about the rest of your pneumatic system without talking about valves. Generally speaking, you will deal with 5 port valves. These have one port for the air supply, two ports to extend and retract a cylinder, and two vent ports. These valves come in two flavors: Single or Double. If your valves are from SMC, they will look something like what is shown in Fig. 5.17 (shamelessly grabbed from the Web in what I hope falls under the Fair Use portion of copyright law…). What is nice about this figure is that is shows two valves, a Single and a Double. They both sit on top of a manifold that allows for connection of a common air supply. Bear in mind that there are many configurations possible. For example, each valve could sit on its own base. Or, you could be using valves from Festo, which look completely different, and don't have a separate base. Don't be distracted by these details, but concentrate on the functional differences, which are important.

FIGURE 5.17 A Single solenoid valve (one orange dot) and a Double (2 dots). These valves are from SMC. They don't look *exactly* like what you are likely to find in your team's box of random pneumatic stuff, but are close enough. If you press on the dot, the valve will act as if a voltage has been applied to that coil.

First, a minor difference. Solenoid valves are part of the pneumatic (air) system, but they are electrical devices. They will either operate on 12 VDC or 24 VDC. You can supply either to the solenoid module in the cRIO, but all your valves have to operate at the same voltage. So

make a choice early in your planning process, and then make sure that you don't accidentally install the wrong type.

Doubles

Double valves have two coils, as they are called, and therefore use up two of the eight channels in a solenoid module. Fig. 5.18 gives schematic picture of how the valve operates. You energize one of the coils to drive the valve so that the internal connections are as shown in the left-hand figure. This supplies air behind the drive piston, and allows the space in front to vent. This will cause the cylinder rod to extend. If you de-energize the coil, the valve will stay in this position. To retract the cylinder, you energize the other coil. This drives the valve to the configuration shown on the right, which now vents the space behind the piston, and pressurizes the space in front.

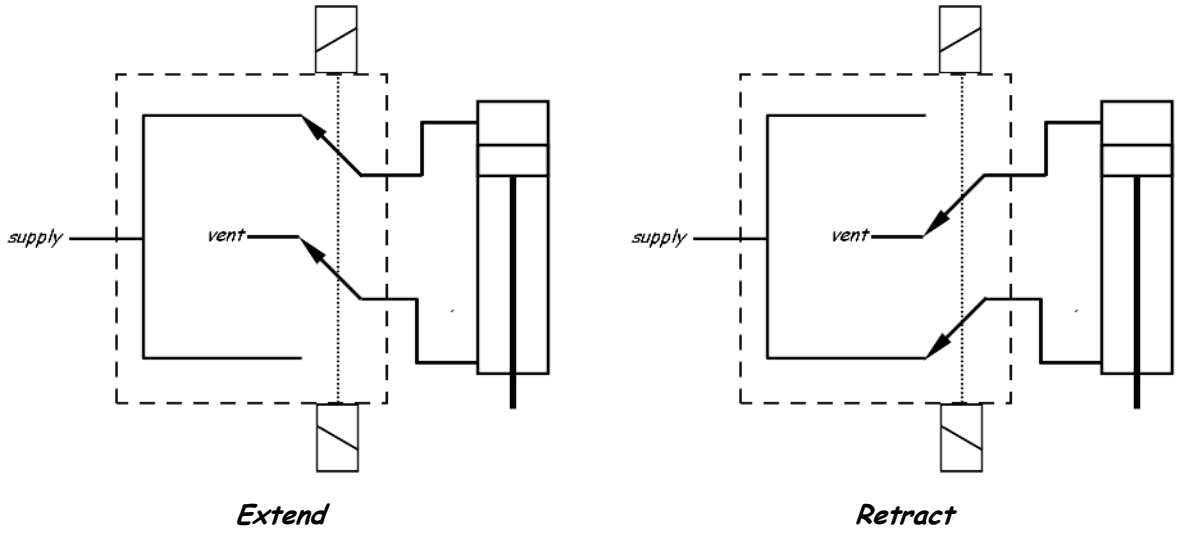


FIGURE 5.18 A schematic view of a 5-port Double solenoid valve (connected to a pneumatic cylinder) in operation. The valve body is the dotted box. The cylinder is the awkwardly drawn object on the right. Be aware that this is a *very* non-standard diagram. Professional drafting symbols for solenoid valves are even more confusing.

The code to drive a Double solenoid is shown in Fig. 5.19. You define forward and reverse channels when you open the solenoid, and select either Forward or Reverse from among the four options when you call the Set VI, depending on which coil you want to energize. The On option energizes the Forward coil. The Off option de-energizes both coils, but there is really no harm in leaving a coil energized for the duration of a match.

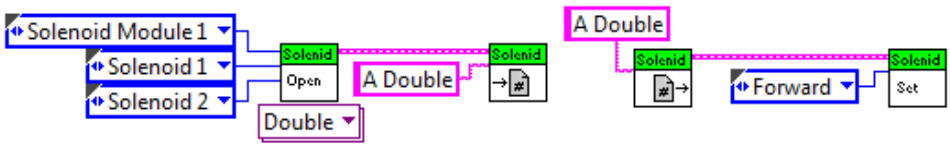


FIGURE 5.19 Simple code for the set up and operation of a Double solenoid.

Singles

A Single solenoid valve is pretty much like a Double, except that it just has one coil. (I know. You are astounded at this revelation.) The missing coil is replaced by a spring. There

are two consequences to this difference, one is important, and the other is really important. The important consequence is that the Single valve only uses up one channel on your solenoid module. If your robot calls for a lot of pneumatics, and you don't want to add a second solenoid module to your cRIO, then use Single valves.

The really important consequence is that when the single coil is de-energized, the spring automatically pushes the valve back to the other configuration. So, imagine the following scenario: the competition rules allow you to earn 10 points if your robot is hanging from a bar, not touching the ground, at the end of a match. You design a pneumatic hook that extends to grab the bar, and retracts to lift the robot. If you use a Single valve (and hook it up right), the robot can lift itself even after the robot power is disabled at the end of a match. So if you've managed to hook the bar, but haven't left enough time to hit the button and lift the robot, you will get the 10 points anyway. Use a Double solenoid for this application, and you can already hear the classic "waa, waa, waaaa" failure sound effect playing in your head.

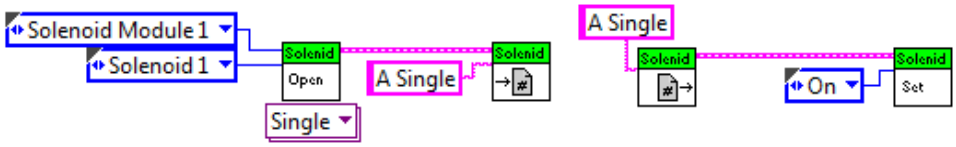


FIGURE 5.20 Simple code for the set up and operation of a Single solenoid.

Fig. 5.20 shows some simple code for the set up and operation of a Single solenoid. Of the four choices for the input to the Set VI, you can use On and Forward interchangeably, and Off and Reverse interchangeably.

Servo Motors

A servo motor is a rotary actuator. You can read and set it's angular position. Industrial robots use lots of servos, and control not just their position, but their speed and acceleration, often with amazing precision. In the FRC world, we have small plastic servos that don't have much torque, and only rotate about 180°. The software tools only allow us to control position. Servos are most commonly used to aim a camera, and can be used, in combination with image processing, to track an object. Fig. 5.21 shows some basic code to open and control a servo, but if you are interested in using a gimbal-mounted camera for object tracking, then you should look at the example (found by selecting **Find Examples…** on the **Help** menu and then scrolling down to the FRC Robotics folder and looking in Actuators). This is quite a sophisticated example, and you will learn a lot by studying it, even though it does the image processing on the cRIO (more about that in Chapter 7).

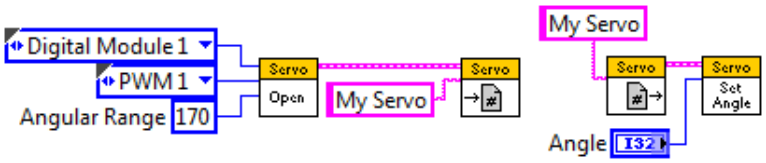


FIGURE 5.21 Simple code for the set up and operation of a servo.

Chapter 6 — PID Control

This is advanced topic, but a useful one. PID is shorthand for Proportional, Integral, and Differential, and it provides a way to command your robot to a speed or a position automatically, rather than by trying to get it just right by hand (and under the pressure of competition, etc.) It is essential in "real" robots, the kind you find on factory floors assembling cars. It is also a sophisticated subject, and I cannot claim to be an expert. So what you will get from this chapter will be pretty basic, but it should be enough to allow you to use PID control in your own robots.

The PID Concept

PID is all about control. Suppose there is something on your robot you would like to control. To make it specific, imagine that there is a motor that you would like to have spinning at a specific RPM. We'll call your desired RPM the "setpoint". In order to make this control work, you'll have to be reading the rotation rate of the motor with an encoder. The output of this encoder is known as the "process variable". The goal of PID control is to change the drive signal to the motor until the process variable equals the setpoint. If the motor is spinning too quickly, the drive signal needs to be reduced. If the motor is spinning too slowly, then the drive signal needs to be increased.

That all sounds very simple and straightforward, but it turns out that making such a control scheme work requires a bit of sophistication. Let us designate the desired RPM of this motor, the setpoint, as s. We'll designate the actual speed of the motor, the process variable, as p. Finally, the input to the motor control VI (a real number between -1 and +1) is designated c, for control value. The difference between what we want and what we have is the error, $e = s - p$.

For large errors, you might think that a good way to calculate c, the correct drive signal to the motor, would be to simply take e multiplied by a scale factor (called the proportional gain, and labeled K_P). This is proportional control: the farther your actual speed is from the desired speed, the bigger the control signal sent to the motor. If you think about it, however, this kind of control scheme is destined to failure. As your motor gets closer and closer to spinning at the desired speed, the smaller the error. That means that your control value gets smaller, causing the motor speed to drop, which then increases the error. If that sounds to you like a situation in which your motor speed would oscillate: speeding up and then slowing down, you would be correct. Oscillation is a well known symptom of a proportional-only control system in which the gain is too large. If the gain is too small, the symptom is steady behavior, but with a large, constant error.

The solution to the proportional only scheme is to add "an integral term." That is to say, develop a contribution to c that depends on the error signal accumulated over time. When the error is large, a big contribution is made to the control signal. When the error becomes small, the control signal remains large enough to hold the motor speed where you want it because the control signal depends not just on the current small value of the error, but also on the large past value.

There you have the main idea behind PID control: the control output is computed from the difference between the setpoint and the process variable, both in direct proportion, and

through integration over time. As the name PID implies, there is also a third contribution, the derivative term, that is large when the error signal is changing rapidly, and small when the error signal is changing slowly. Each term has its own gain, so you can control their relative importance. Setting these gains is a bit of an art, known as "tuning the PID", but before we can discuss how you might go about that, we need to get much more serious about how a PID really works.

PID Math

As we have just outline, let p be the value of something you would like to control, the process value. Let s be the setpoint, the value you would like p to have. And finally, let c be the control value, the knob you turn to change the value of p.

If $e = s - p$, then c is calculated by

$$c = K_P\left(e + \frac{1}{T_I}\int e\,dt - T_D\frac{de}{dt}\right).$$

Notice that there are three terms inside the large parentheses: the first term is proportional to e, the second term involves the integral of e over time, and the last term depends on the derivative of e with respect to time. Note that the so-called proportional gain, K_P, multiplies everything, and therefore serves as a kind of overall gain. This is not a requirement, but it is the way the PID algorithm is implemented in the standard LabVIEW routine.

Let's look at the effect of each of these terms. The proportional term is simplest. If e is large, then $K_P e$ will be large, resulting in a large c being applied to your system. But as the system gets close to your control point, that is as p gets close to s, this term becomes useless. The error is going to zero, causing your control signal to go to zero. As a result, a purely proportional controller has only two modes: the small K_P regime and the large K_P regime. In the small K_P regime, there is a steady state with a constant value for e. The output is constant, but can never quite be at the desired value. As you increase K_P, you can make that error smaller, but at some point the controller will flip over into the large K_P regime where it oscillates. This is not a good regime for a robot, at least a robot that wishes to remain in one piece.

To get the error to zero, you need the integral term. The effect of the integration is to build up a running total of the error. As p approaches s, the proportional term drops away, but this built up integral term does not, and it is this term that provides c in the steady state, while at the same time allowing e to go all the way to zero. As you think about the effect of this term, and the meaning of the integral, it may appear that the term will rise without limit, but don't forget that the error can be negative, which will subtract from the total.

The integral term also has another effect. If the setpoint changes, the integral term will still be trying to hold the output at the old value. It will only "forget" about the old value and integrate to the new value in a time of T_I. So if you set this value too high, the system will respond too slowly to changes in the setpoint.

Finally, we have the D term. D stands for Derivative, but also Dangerous. It can be used to control overshoot, but can make things oscillate, sometimes violently. If possible, I leave $T_D = 0$. If I need some, I use *very small* amounts. The default control for this value shows

only three digits past the decimal point. I change the display format to show four or five, because that's where I'm going to be setting values.

Since we're in a section called "PID Math", it is worth discussing how the PID is actually implemented in LabVIEW. Proportional control is easy. We just need to calculate $c = K_P(s - p)$. Fig. 6.1 shows the contents of a VI that would implement this. Note the True value wired to the loop terminal. This loop executes only once (and is in fact not necessary right now, but will be useful in a minute), so the VI would need to be embedded in a While Loop (like the While Loop of a state machine controlling your robot).

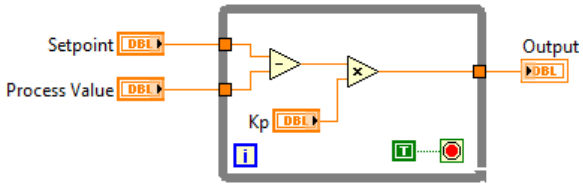


FIGURE 6.1 The block diagram for a proportional only control routine.

How about the I term? We need to numerically integrate the error with respect to time. There are many established ways of doing this, but we are going to use the simplest, known as *Euler's approximation*. (Cool people pronounce Euler, "oiler".) Remember, the PID control is inside a loop, and that loop is called every Δt seconds. On the n^{th} call to this routine, we calculate the new value of the integral term, I_{n+1}, according to

$$I_{n+1} = I_n + e\,\Delta t.$$

So, when the routine is called, it needs to have I_n stored somewhere handy, it has to use it to calculate I_{n+1}, and then it has to store that new value somewhere, like where I_n used to be. There are lots of ways to do this, but a shift register is a very convenient one. Fig. 6.2 shows the code to implement this integration equation together with the proportional code. As promised, the value of the integral is accumulated in a shift register. A second shift register measures the time between successive calls to provide Δt.

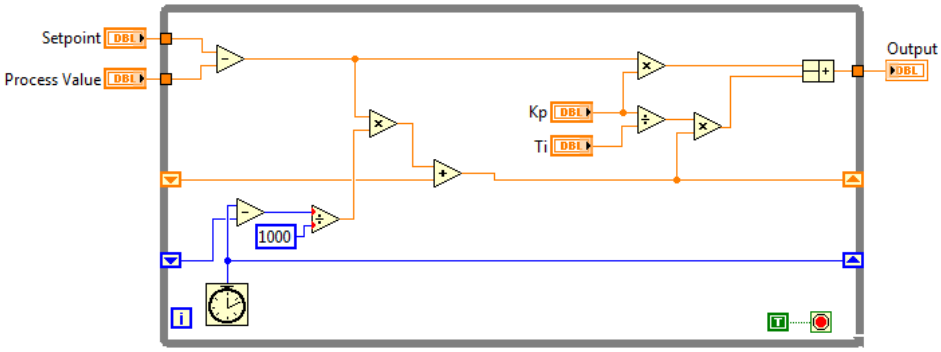


FIGURE 6.2 Code to add an integral control term to Fig. 6.1. Note the conversion of Δt to seconds from milliseconds.

Implementing a derivative term is very similar. Now we need to have the error from the last call, e_n, and the error from this call, e_{n+1} to calculate

$$\frac{de}{dt} \approx \frac{e_{n+1} - e_n}{\Delta t}.$$

Fig. 6.3 shows the code with the D term included. You could use the code in this figure as a PID controller, and it would do the job, but I don't recommend it. Use the PID routine provided in the PID palette, which does the integration using the trapezoidal rule (which adds a bit of accuracy), and has a lot of other nice features. For example, if you set Ti equal to zero, it turns off the integral control, giving you just a P (or PD) controller. Our little toy version doesn't handle that so nicely. If you're curious, you can inspect the diagram of the LabVIEW routine to see exactly what it does.

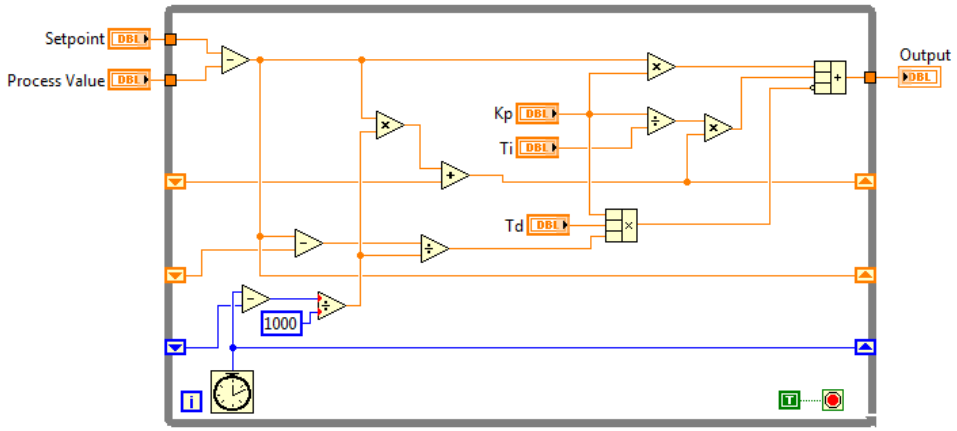


FIGURE 6.3 The full PID implementation. The middle shift register holds the error value between calls.

Motor Math

Here's the deal: there's no way to really understand how to use PID control without doing it. But maybe you don't have a robot handy and ready for experiments. Any maybe you don't want to start by using PID control to convert your competition robot into scrap metal. So we're going to build a simulator: a piece of software that behaves more or less like a real robot, and you can play with that. And we'll give you all the details, so that you will understand how to modify the simulator so that it represents not the particular robot I've chosen to model, but whatever it is you are building for your own robot.

So, to begin at the beginning: electric motors. Internally, DC motors have some permanent magnets and some coils of wire that act as electromagnets. There's some clever arrangement called a commutator to reverse the current at just the right time and keep the shaft (which carries the coils) turning in the right direction. But that is not our concern here. (You should teach yourself about DC motors using the Web. Here's a nice place to start: http://hyperphysics.phy-astr.gsu.edu/hbase/magnetic/motdc.html) Our concern is *back emf*. There's a physical phenomenon that goes by the name Lenz's Law, where a coil will resist any change in the magnetic field that loops through it. The coil does this by generating a voltage, called a back emf, that opposes the voltage from (in the case of your robot) the battery. As a result, if you were to connect the motor directly to the battery, the current through the motor would be given by

$$i = \frac{V_B - V_e}{R},$$

where V_B is the battery voltage and V_e is the back emf. R is the resistance of the coils inside the motor.

Of course, we are not connecting the motor directly to a battery, but a driving it with an H-bridge motor controller and setting the current with a pulse width modulated (PWM) control voltage. If your controller is a Victor 884, then[1]

$$i \approx \frac{V_B - k_e\omega}{R} D,$$

where D is the duty cycle of the PWM, and I have replaced the back emf voltage by $V_e = k_e\omega$. k_e is a property of the motor, and ω is its angular speed in radians per second. If your controller is a Jaguar, Victor 888, Talon, or other "modern", high-frequency controller, then

$$i \approx \frac{V_B D - k_e\omega}{R}.$$

This may not seem like much of a difference, but it actually matters a lot. (For a derivation and discussion of these results, see http://vamfun.wordpress.com/2012/07/21/derivation-of-formulas-to-estimate-h-bridge-controller-current-vex-jaguarvictor-draft/)

Notice that in both cases, the current is decreased as the motor spins faster. This decrease matters, because the amount of torque generated by the motor (that is available to you) is

$$\tau = k_T i - \tau_f,$$

where k_T is the torque constant for the motor and τ_f is the internal friction in the motor. So, as the motor speed increases, the net current through it decreases, and the available torque falls. Eventually, the available torque drops to zero, which is what happens if you let the motor run with no load. It reaches a maximum speed (called the free or no-load speed) and draws what is called the free current. When you get a motor, its specifications will include values for the free speed and free current, as well as for the stall torque and stall current, all of which are valid for a specified voltage V.

From these numbers you can *approximately* calculate

$$k_T = \frac{\tau_{\text{stall}}}{i_{\text{stall}}}, \qquad R = \frac{V}{i_{\text{stall}}}$$

and

$$\tau_f = \frac{k_T(V_B - k_e\omega_{\text{free}})}{R}.$$

If you work in a consistent set of units (i.e. MKS), then k_e and k_T will be numerically equal to each other, so having calculated k_T, you know k_e. Don't forget to convert ω_{free} to radians per second from RPM.

Exercise E11: Prove that k_T and k_e have the same units.

At this point, we finally have enough theory to build a motor model. We'll model a CIM motor driven by one of the old Victor 884's. The table below gives specifications for all the motors we will be working with in this chapter.

---

[1] Please. The symbol ω is the Greek letter omega. Cool people definitely do not call it a "double u."

TABLE 6.1

Motor	Stall Torque (oz-in)	Stall Current (A)	Free Speed (RPM)	Free Current (A)
CIM	344	133	5310	2.7
RS755	61.1	30	7300	1.1
9015	60.64	63.8	16000	1.2

Fig. 6.4 shows the code for modeling a CIM driven by a Victor 884 controller. The inputs are the duty cycle (varying from -1 to 1), and the angular frequency of the motor. The output is torque in Newton-meters. The False case just sends a zero value to the output (the motor can't spin if it is not generating enough torque to overcome the internal friction). You will need to build this VI.

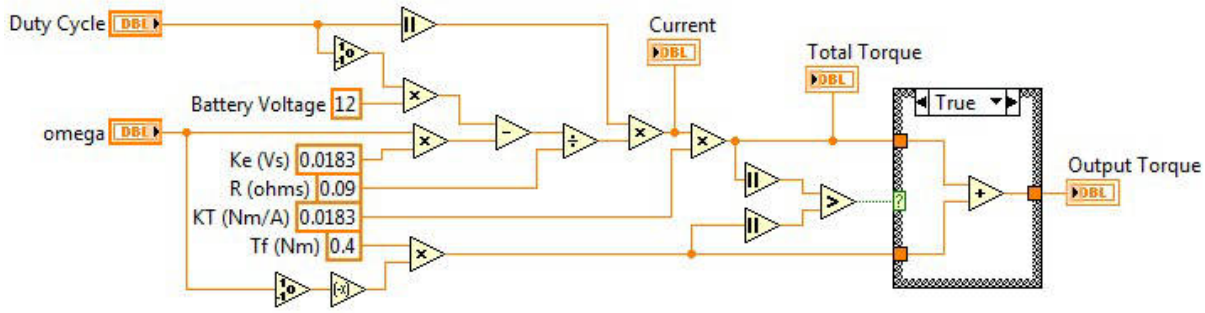


FIGURE 6.4 Code to model a CIM motor driven by a Victor 884 (old style) controller. I've made the Label visible on the constants so the text moves with them.

Driving Simulator

Now that we have a motor, we can build up the rest of our model robot. The motor provides torque, which is a function of the PWM duty cycle D and the motor speed: $\tau(D, \omega)$. In our robot we will have four of these motors. We will imagine that this is a "kit bot", with a CIMple gearbox which gears down the speed of the motors by a factor 4.67. Gearing down the speed gears *up* the torque by the same amount. The chain drive gears the speed down (torque up) by an additional factor of 2.17 for a total gear train ratio of 10.11. The kit wheels have a 6 inch diameter (r = 0.075 m), so if the motor torque is τ, the drive force propelling the robot is $F_D = 10.11\tau/r$.

To make our robot a bit more realistic, we should include some friction. Modeling friction turns out to be tricky. If the robot is moving, then $F_f = \mu mg$ just has a constant value and is oppositely directed to the velocity (m is the mass of the robot and μ is the coefficient of friction). But if the robot is not moving, then friction is opposite to the applied force from the wheels, but exactly equal in magnitude. Working that out, and letting the friction "break" when the applied force rises above some fixed value (I chose μmg, the moving friction value, just to keep it simple) makes for a surprisingly complex VI. My solution is shown in Figs. 6.5 and 6.6. This VI takes the drive force as an input, and returns the net force $F_{\text{net}} = F_D + F_f$. It also has additional inputs: velocity, mass, and coefficient of friction.

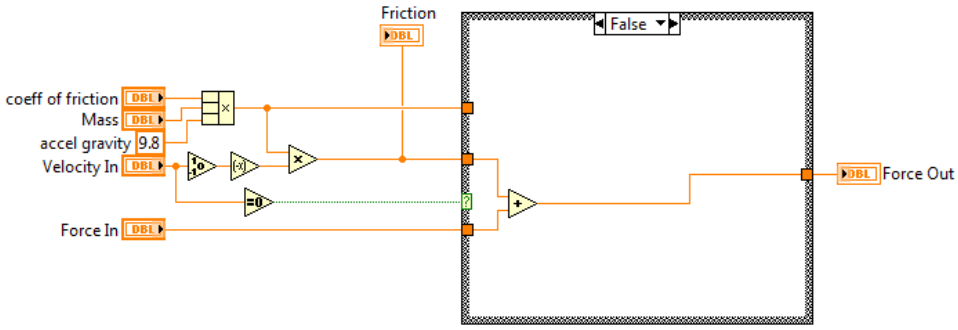


FIGURE 6.5 A friction model showing the case that the robot is moving.

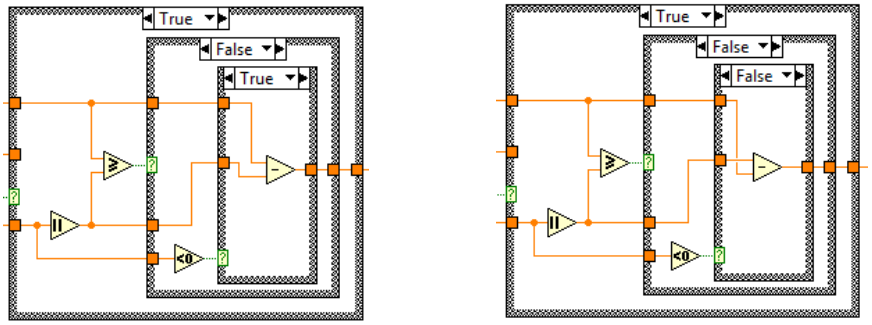


FIGURE 6.6 Contents of the True case in Fig. 6.5. The case that the middle structure is True is not shown. It corresponds to the case that the drive force is not large enough to break friction and just passes a zero to Force Out.

Now we can calculate the acceleration from $a = F_{\text{net}}/m$, but how do we get the robot velocity? If you've had physics with calculus, you know that

$$v = \int a\, dt.$$

And if you haven't, well, I just told you. And, if you are paying attention, you've noticed that we already encountered a very similar integration in our discussion of the PID algorithm. We tackled that integral with Euler's approximation, so we can apply the same trick here. Our simulator will run in a While Loop, and a shift register will allow us to calculate

$$v_{n+1} = v_n + a\,\Delta t.$$

So, it is time for you to build the simulator. Fig. 6.7 shows the front panel, and Fig. 6.8 shows the diagram. Don't worry about the contents of the cases not shown. We'll get to them.

Look over the diagram first and make sure you can identify the salient features. The upper shift register (orange line) contains the velocity as calculated by the above equation. The Duty Cycle is an input to the motor model, and the torque is the output. This torque is multiplied by four (because we are modeling a robot driven by four motors), and then gets multiplied by the drive train ratio and divided by the wheel radius. The resulting force is fed into the friction model, and the net force comes out the other side. This force is divided by the 50 kg mass of the robot to make an acceleration, which is integrated into the shift register. The motor model and the friction both need a velocity input, for which we use the "old" velocity, not the new one we are about to calculate. For the motor model, the linear velocity of the robot is multiplied by the same factor as the output torque to convert it to the angular fre-

quency of the motor. The friction model just takes the ordinary velocity of the robot. That covers the basics.

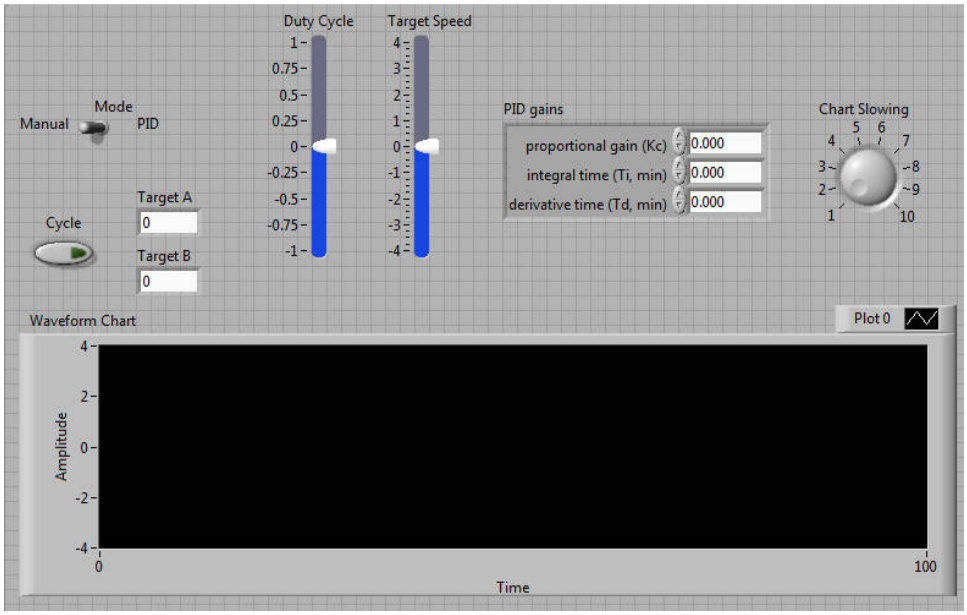


FIGURE 6.7 Front panel of the driving simulator. You should actually start with the block diagram in the next figure, and refer back to this later.

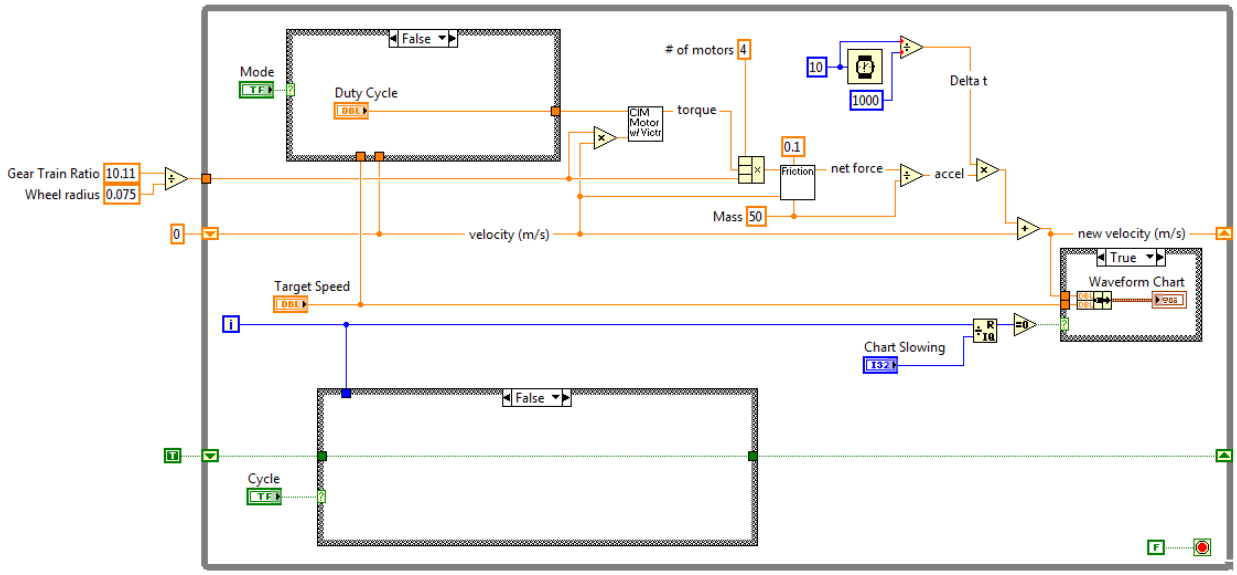


FIGURE 6.8 Block diagram of the driving simulator. See the text for a detailed explanation.

Now we come to some "bells and whistles". It is extremely useful to be able to watch both the velocity and the Target Speed (our setpoint when we get to the PID part) on a graph, so in the block diagram you can see them put together in a cluster and wired to the graph over on the right. The chart moves extremely fast, however, as it plots 100 points per second, so I have included "slowing" code that causes only every n^{th} point to be plotted, where n varies

from 1 to 10 and is controlled by a knob on the front panel. The other case in the structure that contains the graph is completely empty.

When trying to "tune" a PID controller, it can be extremely useful to have the setpoint switch between two values. Fig. 6.9 shows the True case for the Cycle case structure. Every 200 iterations of the simulator we are toggling a Boolean (stored in a shift register), and the value of that Boolean controls whether Target A or Target B are written into Target Speed. The programming here is a bit sloppy (but we are in a hurry). Since Target Speed is a front panel control, only your personal self-restraint keeps you from wantonly manipulating that control while the Cycle code is supposed to be setting it to a specific value.

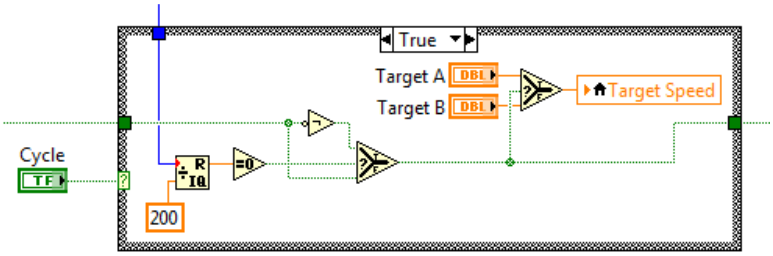


FIGURE 6.9 Code to cycle the Target Speed between two fixed values.

Now we finally come to the point of the entire exercise: PID control. Fig. 6.10 shows the True case for the Mode switch. You can see that the Target Speed is wired to the setpoint input, and the (old) velocity is wired to the process variable input. The output of the PID is the duty cycle input to the motor controller. Since the duty cycle can only vary from -1 to 1, we limit the PID output to the same range. We will need to manipulate the PID gains while the program is running, so those are brought to the front panel as a control. Again, there is some sloppy code. We pretend that the Duty Cycle control is an indicator during PID control, which allows you to use the front panel to fight the PID.

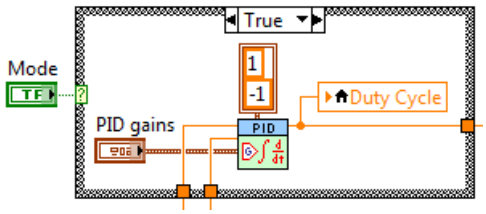


FIGURE 6.10 The True case for the Mode switch. This is the part we have been waiting for: actual PID control.

Once you put in the code in Fig. 6. 10, you are ready to drive. Start by running the VI with Mode set to False, so you are doing the driving. For amusement, turn on the Cycle switch and make the Target Speed toggle between 3.5 and -3.5. See if you can keep the robot speed matched to the target by hand. When you've had enough of that, switch to PID control. The challenge now is to "tune" the PID to give the performance you want. Start with P gain only. As a first guess, set the gain so that the maximum possible error gives an output of 1. Notice that as you increase the proportional gain, the speed of the robot gets closer and closer to the target, but never quite gets there. If you make the gain too large, the robot's speed will start to oscillate (most easily visible in the Duty Cycle slide control), but because of the heavy mass and the friction, there is a very large window between enough P gain and too much.

When you think you have a good amount of proportional gain, try adding some I or some D. It would be best to be in Cycle mode so that you can see the effect of the integral term. While we would like the robot to snap to the new target speed as fast as possible, we only have a finite force available, and that is what ultimately limits how quickly the robot can adjust to a new setpoint. But you should be able to tune it up so that the speed and accuracy if the control is pretty impressive.

Turret Simulator

The driving simulator was an example of speed control. The other common application is positional control. As an example, lets simulate an aiming turret, similar to what many FRC teams (including ours) had for the 2012 Rebound Rumble competition.

Here is what we'll simulate: An 18" diameter turret with a moment of inertia of 2 kg m^2. We'll drive it with a Banebots RS775 motor mated to a 71:1 planetary gear reduction. We'll use a 1.5" diameter sprocket to drive the 18" diameter turret, so the total gear reduction is 852. You can copy your CIM motor model and just change the parameters to make an RS775 model. For friction, copy your Friction VI, and modify it so that the input is not a coefficient of friction, but the actual frictional torque. The rest of the logic is already correct, so that is the only change you need to make. Fig. 6.11 is the block diagram you need to make. It is a straightforward modification of the driving simulator, so I recommend you copy that for a starting point. The front panel is so identical that I'm not going to bother with a figure. You will need to change the limits on what is now Target Angle. My turret swings between -45 and +45 degrees. Also, the Frictional Torque is a front panel control.

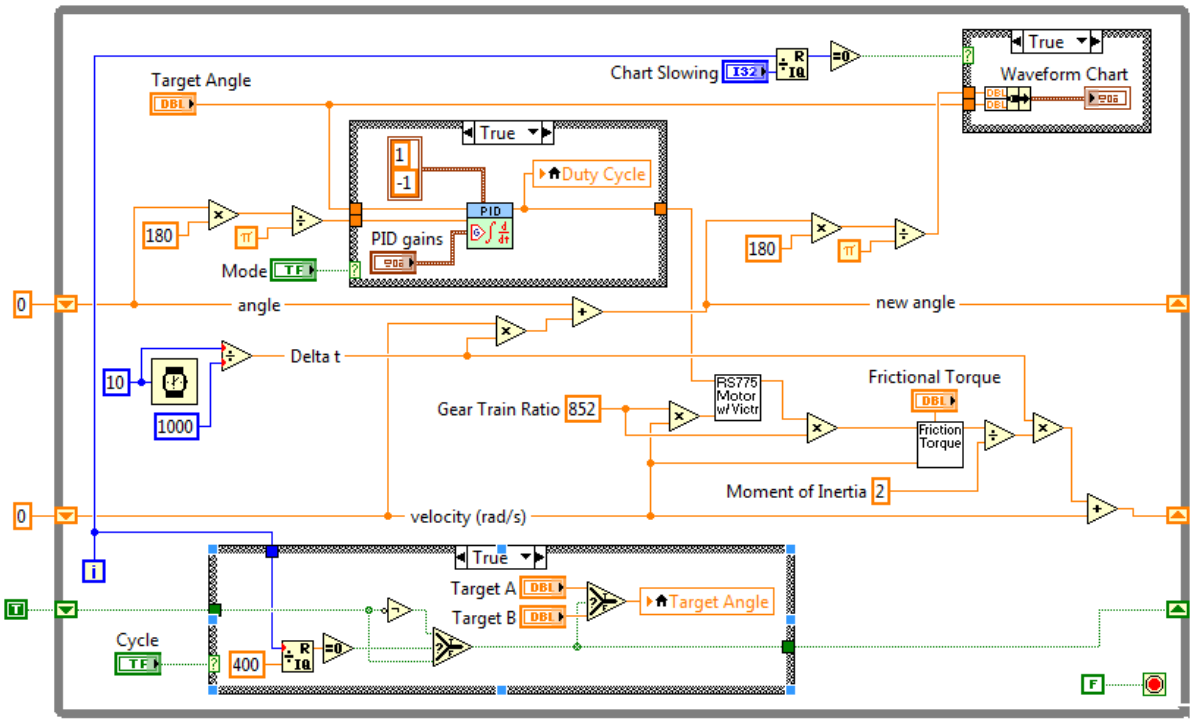


FIGURE 6.11 Block diagram for the turret simulator.

If you look carefully at the diagram, you can see the similarity to the driving simulator. We apply a Duty Cycle to the motor, obtain a (geared up) torque, reduce it by some friction, and

calculate an (angular) acceleration. We integrate that acceleration to obtain the angular velocity (in radians per second) of the turret. To get position, we simply integrate again. The integrated angle is in radians, but we convert it to degrees *before* applying the PID control. There's no requirement to make this conversion, but people are better at degrees than radians, and this allows us to set the target position in degrees. You will also notice that the chart is plotted in degrees.

Once you have completed the VI, go ahead and start tuning the PID. Start with a Frictional Torque of 1 N-m. Again, start with only the P term. You will find that this is a completely different kind of beast from the driving simulator. Now we are no longer limited by the torque of the motor, and it runs at maximum speed while the turret is rotating (provided the P gain is large enough). To rotate the turret faster, you need a different motor. If you are having trouble finding a good tuning, put the simulator into cycle mode and start with a very, very low P gain. Every time the Target Angle switches, the turret will swing to the new position, and then wiggle around that position, with the wiggles eventually dying out. Gradually increase the P gain until the wiggles don't die off any more, then back it off just a bit. Now you only need to add a *small* amount of either I gain or D gain to get rid of the wiggles (I'm not telling you which).

Final PID Thoughts

Before we move on to image processing, here are just a few musings on PIDs. Bear in mind that these simulators present a highly idealized version of PID control. In a real system, there will be noise on the sensor readings that give you velocity or position. There will be backlash in the motion system, and many other challenges. Tuning can be a nerve wracking experience when the consequence of getting it badly wrong is serious damage to your robot.

As you think about what kind of control system you want, bear in mind that just as you are not required to use both the I and D terms, you are also not required to use *either*. Suppose all you are trying to do is compensate for a non-linear motor controller so that your robot moves forward smoothly with the joystick, instead of doing nothing for the first bit, and then suddenly jumping. In that case, you don't really care if the speed gets exactly to the setting that matches the joystick position. You are generally moving around trying to catch something or push something. Close is close enough, and a P term is all you need.

Problems

Problem 6.1 – Change the motor in the Turret Simulator to the 9015 (see Table 6.1). Investigate how this changes the turret behavior.

Problem 6.2 – (Advanced) The PID simulators are unrealistic on a number of levels, one of which is their lack of noise. Fig. 6.12 shows the diagram of a "noise maker" VI. This will provide you with random noise (approximately between -1 and 1, before you multiply it by Scale), but with an adjustable characteristic frequency. You can set the Averaging input anywhere from 1 to 100. The larger the number, the lower the frequency of the noise.

Try making the velocity in the Driving Simulator noisy. Add the output of this VI into the velocity just after it is read out from the shift register. Start with a Scale of 0.1, which is unrealistically large, but makes things easy to see. Try an Averaging value of 1. Does the noise have a big effect? What if the Averaging is 10? 100?

You can use this VI to explore the effect of noise on the output of the motor, or on the input from any sensor. Just change the Scale input to give you noise of the appropriate size. If you want to have more than one noise source in your VI, launch the VI Properties window (ctrl-I), and select the Execution menu. Check the "Reentrant Execution" box so that each copy of the Noise Maker has its own set of random numbers.

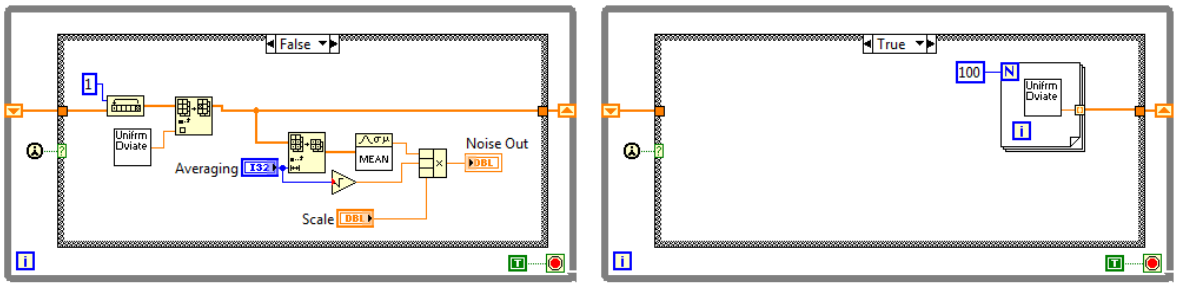


FIGURE 6.12 Two views of the Noise Maker VI. The right-hand diagram executes only the first time the VI is called, and fills a 100 element array with random numbers. All subsequent calls to the VI use the left-hand diagram. Averaging and Scale are inputs, Noise Out is an output.

Problem 6.3 – Make a new motor model for the Driving Simulator to use a Jaguar type controller instead of a Victor 884.

Chapter 7 — Image Processing

Image processing is also an advanced topic, but one that is a lot of fun. The classic idea of a robot is a fully autonomous machine: one that can sense the world around it and take action based on those inputs. Vision is arguably the most powerful sense for humans. With not too much effort, it can be a powerful sense for your robot too: it will be able to track targets, aim itself, and measure distances.

Doing it on the laptop

Once upon a time, back in the dark ages, if you wanted to do image processing, you had to do it on your cRIO. Since image processing requires a lot of computer power, a robot doing image processing had trouble doing anything else. But now we all have full-blown wireless routers on our robots, so images from the camera can go directly to your laptop, and you can do the image processing there. Then you can send the results of that processing, which should only be a few numbers, or maybe even a single number, back to the robot so that it can aim, or move, or whatever.

That being the case, in all the examples in this chapter, we'll just be reading images from disk and running the programs on a PC. Since you will learn by doing much more effectifly than you will learn by reading, be sure to obtain Secret Test Images.zip, which you should find where you found this book. As you program your robot for competition, you'll want to do something similar to what goes on in this chapter, as you can easily have useful images available for programming well before your robot is ready for code. Especially if you make the effort to set up the vision targets in some quick and easy way. (The images in Secret Test Images.zip were taken of dummy targets glued to poster board and taped to a classroom wall. The camera was on a tripod, carefully set so it was at the correct height in relation to the targets.)

Loading an image

To get started, open up a new, blank VI (on your laptop…no cRIO in this chapter!). Right click on the block diagram to bring up the Functions Palette (Fig. 1.8) and locate two subpalettes: the FIRST Vision Palette, which contains most of what you need, and the Vision and Motion Palette, which contains even more image processing functions, a few of which you will also need.

Fig. 7.1 shows a very simple program to load and display an image. The first VI is IMAQ Create. This allocates a block of computer memory which will hold the image. LabVIEW's memory management is sophisticated enough that it doesn't need to know how big the image is (and therefore how much memory to set aside). It just needs to be told that there is an image coming, and what type it is. In this case, the image is "RGB (U32)", which means a color image, where the color information for each pixel is encoded as three numbers: the intensity of the red, green, and blue color channels. You also need to name this chunk of memory. I've chosen the not very imaginative name "RGB Image".

The next VI reads the indicated image from disk and slots it into that chunk of memory. This VI needs the absolute path for the image, which in this case is rather long. If you want to read in a bunch of files from a folder (using a While Loop, for example), you use stuff from

the File I/O and String palettes to build up the path names automatically. (If you've gotten this far, you can probably figure out how on your own!)

Finally, the image wire goes to a front panel indicator, which displays the image. The front panel indicator has its own little tools palette. It is useful to know that if you have selected the magnifier tool, holding down the shift key turns it into a de-magnifier. Below the image is an information bar, which we will find extremely important. Reading left to right, it tells you the size of the image, the type of the image, the color of the pixel the cursor is currently over, and the location of the cursor. You should move the cursor around and make sure you understand where the point (0,0) is located, because that will be important.

In the figure, the cursor is the little cross up in the sky above the building. The color of that pixel given by the three numbers indicating the red, green, and blue intensities (in that order). These numbers are restricted (for this type of image) to the range 0 – 255. A black pixel will have the value 0, 0, 0, while a white one will be 255, 255, 255.

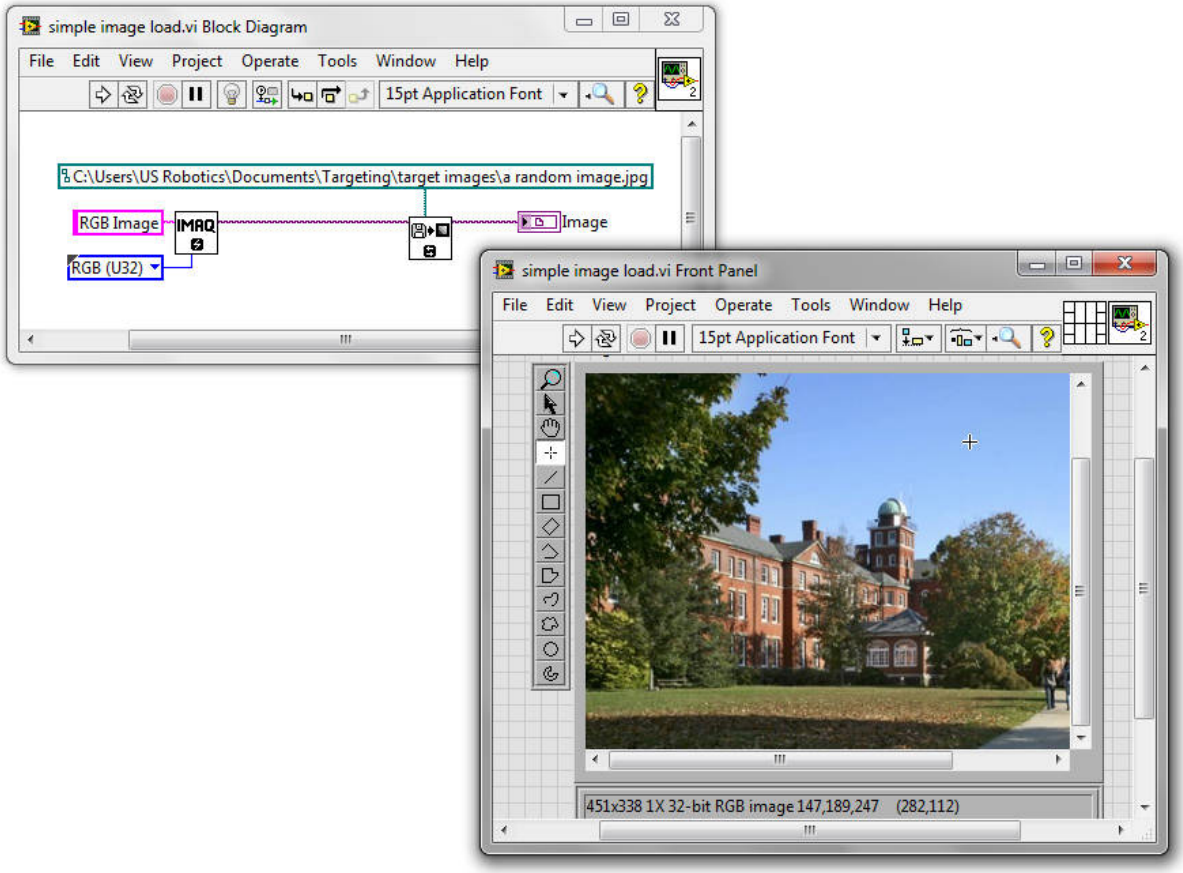


FIGURE 7.1 A simple program to load and display an image.

How LabVIEW stores images

The wire that represents an image variable is totally unlike all the other wires in LabVIEW. It should blink or something to remind you how different it is, but it doesn't. You just have to remember (which will turn out to be hard, because sometimes it will seem to you like the wires behave normally…).

Modify the simple image loading program so that it looks like the code in Fig. 7.2. Start by adding the code to show both a number and its square root on the front panel. (This is to remind you how "normal" wires behave.) Then add a second image display indicator to the front panel, and make the image that will display in this second indicator a grayscale image. To do that, you need the Cast Image VI. You find that by opening the Functions Palette and selecting **Vision and Motion ▸ Vision Utilities ▸ Image Management ▸ IMAQ Cast Image**. Set the Image Type to "Grayscale (U8)", as shown in the figure, and run the VI. Is this the result you expected? How is it different from the little square root example? What is going on?

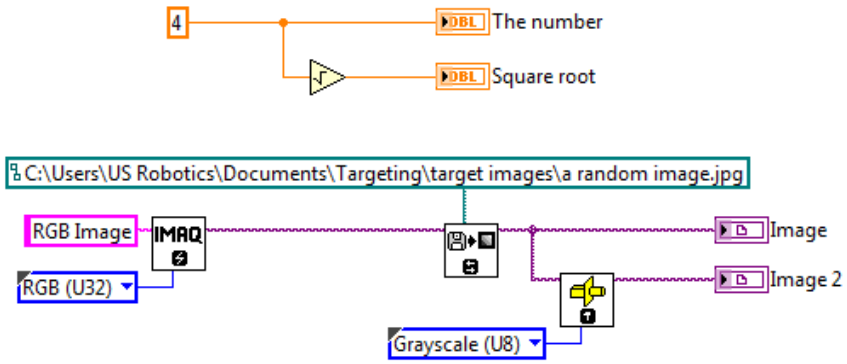


FIGURE 7.2 A program that is intended to display color and grayscale versions of the same image.

The difficulty is that the IMAQ Create VI creates only one memory block. Any VI that modifies an image modifies that one copy. Any VI that then grabs that image, whether to display it, as in our example, or to perform further operations, grabs the one and only copy. The standard LabVIEW programming model is that data values "flow" down the wires. At a node like the orange dot in the square root code, the flowing stream divides, and independent copies of the data flow down each wire. Images don't flow. When you connect a VI to a wire, you are not catching a copy of whatever is flowing by. You are instead reaching out and grabbing the one and only "master" copy of the image.

So, you would you display color and grayscale versions of an image side by side? After all, it seems like a pretty reasonable thing to do. If you examine the Cast Image VI carefully, you will see it has two image inputs: "Image Src" and "Image Dst" (source and destination). To get a second, independent image, you have to allocate a second block of memory and make that second block the "destination" for a grayscale copy of the image. Modify your program so that it looks like Fig. 7.3 and see whether gives you both color and grayscale.

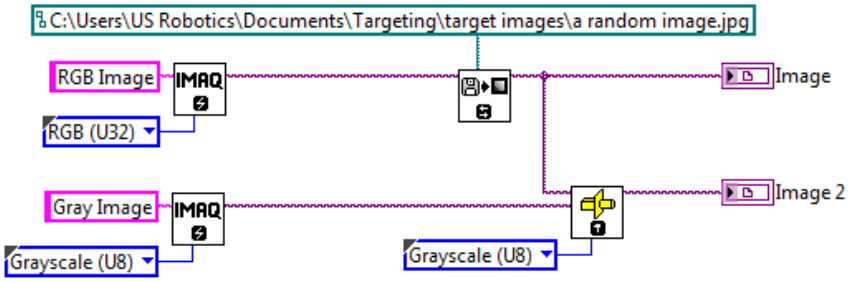


FIGURE 7.3 A working version of Fig. 7.2

You will find that many of the VIs you use in image processing work like Cast Image: they have both source and destination inputs so you can either modify the original image, or put the modified data in a second image.

Image types

At this point we should probably take a little side trip and discuss image types, since we've already encountered two. By "type" I do not mean "format". In FRC, you are most likely to encounter the JPEG format, since that is the default for the Axis camera. But you are probably also aware that there are other formats like GIF and TIFF and PNG. LabVIEW can handle all of these, and it does so when the image is read into the program. Once it is in the program, it is a LabVIEW image, and the original format no longer matters, only the type.

Think of an image as a two-dimensional array. Each element in the array is a pixel in the image. It is what is inside these elements that determines the image type.

If the image type is "Grayscale (U8), then the array element is a single unsigned integer that can have a single value from 0 to 255. With only one "channel" available, the image can carry no color information, only intensity information. As a result, the image is "black and white", or, as we sophisticates say, grayscale. (Unless we are subjects, or former subjects, of Queen Elizabeth, in which case we say "greyscale". It has a completely different spelling.)

If the image is a color image, then each element contains three independent numbers. If the color mapping is RGB, then the three numbers give the red, green, and blue values for the pixel. These numbers are all 8-bit unsigned integers, and so can have values from 0 to 255. For reasons of convenience and memory management, LabVIEW packs these three numbers together into a single 32-bit integer, which is why this type is called "RGB (U32)". If you are still awake, you will notice that three color channels times 8 bits per channel makes 24 bits, not 32. From a data handling and storage point of view, a 24-bit integer is a very ugly duckling, so it is worth carrying eight unused bits to make things nice.

While you do need three numbers to specify the color of a pixel in a color image, there is no law of nature that says they need to be red, green, and blue. They could be Hue, Saturation, and Luminance, in which case the image type would be "HSL (U32)". Once again, each pixel is described by three 8-bit numbers, but the information is encoded completely differently. A good analogy is specifying a coordinate in three dimensions. (OK. This is only a good analogy if you've seen this before in your math or physics class. Otherwise it is completely useless.) You can locate that point by specifying positions along three axes as in (x, y, z). You can also specify that same point in spherical coordinates, in which case you specify a distance from the axis and two angles: (r, θ, ϕ). Possibly this image type will be useful to you. We will get to that in a bit.

If you look at the options on IMAQ Create, you will see that your other options for encoding the pixel are 48-bit color, 16-bit integer (both signed and unsigned), and both real and complex numbers. These all have their place, but that place is probably not FRC.

Before we move on, there is one last topic related to image type. Your computer's display is a color display. If the image is color, there is really no flexibility about how to show it on your screen. But if the image is grayscale, then we are using a single number to specify the value of three numbers (the red, green, and blue pixel intensities on your screen). This is

done with a look-up table, usually called a palette. So, if the image pixel value is 37, the display software looks in the palette for entry number 37. There is will find three separate numbers, telling it what red, green and blue intensities to use to display this pixel. To get the grayscale image when you ran our little example, LabVIEW used—no surprise—the Grayscale palette. In this palette, all three numbers are set to the same value (which would be 37 in the case that the pixel value is 37). But other choices are possible! If you go back to the front panel of our example, and right click on the grayscale image, you can find the palette menu, as shown in Fig. 7.4. Try the other palettes. Believe it or not, we will find the Binary palette to be extremely useful, in spite of the fact that it makes the sample image look like some kind of hallucination. Bear in mind that when you change the palette like this, you are *not* changing the image, only how it is displayed on your screen.

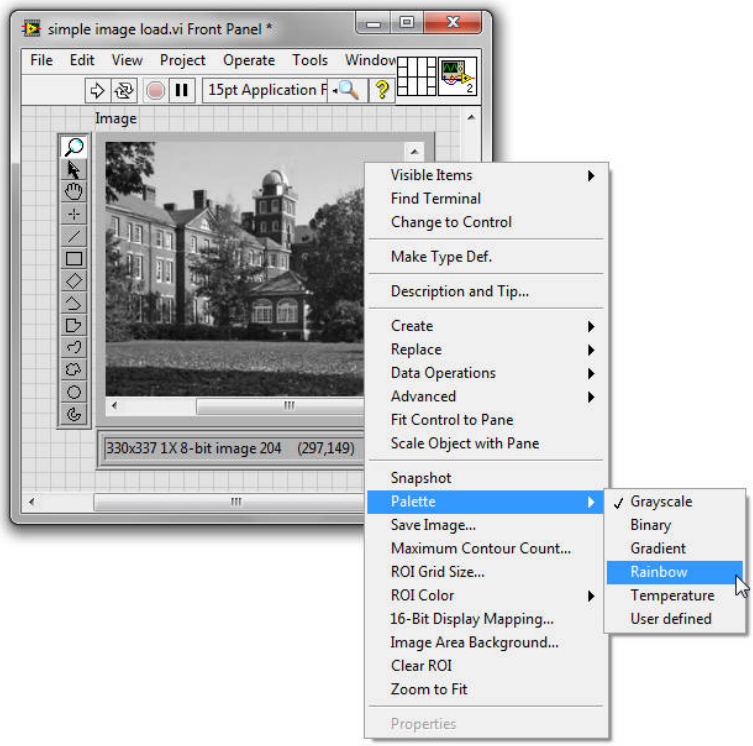


FIGURE 7.4 The display palette, which affects how grayscale images are displayed, but not color images.

Getting images

Our goal is to process a color image like the one below in Fig. 7.7 and return information about the targets: where in the image they are located, and how big they are. For learning purposes, you can work from the images distributed with this book. But once the new competition is announced, you should work as quickly as possible to generate test images that come as close to real competition images as possible. To that end, you need to be able to use your robot camera to take test images, as was done to create the test images included with this book. The program shown in Fig. 7.5 will do that for you. The front panel has a vision display and a button. The program runs on your laptop (not your cRIO) and continuously acquires and displays images on your laptop. (Use an Ethernet cable to directly connect the

camera to your computer.) When you press the Snap button, it will take whatever image it currently has and write it to disk, after prompting you for a file name.

You will notice that the camera IP address is set to the FRC default (10.te.am.11). That means your computer needs to be on that network. The easiest way to make that happen is to launch the Driver Station. You will also probably have to disable your wireless card so that Windows is not confused about which network to use. (There is documentation on the FIRST web site telling you how to set the IP address on your camera. The FRC LabVIEW distribution includes a utility program that will do it for you.)

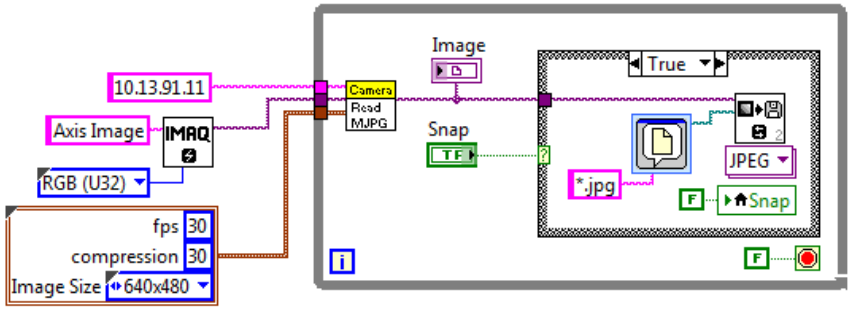


FIGURE 7.5 Program to acquire images using an Axis camera.

Getting good images

Now that you know how to get images, you need to get good images, because that's what good image processing starts with. How much control you have over the images coming in from the camera depends to a certain extent on the game. In recent years (2011 – 2013), the things you want to find have been marked with retroreflective tape, which gives you a fair amount of control. In the 2010 Breakaway competition, the vision target was a high contrast black on white target, while in the 2009 Lunacy competition the objects to look for were marked with bright fluorescent colors. The apparent brightness of those markers, and to a certain extent their apparent color, were a function of the lighting on whatever competition field you happened to be on. That kind of variability can be a challenge.

FIGURE 7.6 An "angel eye" ring light mounted on a camera. Note the piece of duct tape used to keep the adjustable focus from being accidentally adjusted.

If the competition uses retroreflective tape, then your task is much easier. Put a ring of LED lights around the lens of the camera. The kind you want go by the term “angel eyes”, and are used as decorative accents on automobiles. They are quite bright, and are designed to run directly off 12V DC power, which is quite convenient. Fig. 7.6 shows a small one of these LED rings mounted on an older model Axis camera, while Fig. 7.7 shows a set of targets illuminated by those LEDs. Note that the lighting conditions are otherwise crummy. In particular, the ceiling lights could be expected to be trouble, but the combination of the reflective tape and the light source tightly positioned around the camera lens makes the targets “pop”. You can buy much larger angel eyes with many more LEDs, and you may want to put them on your robot because you just want to, but for lighting up a retroreflective target, a small one is all you need.

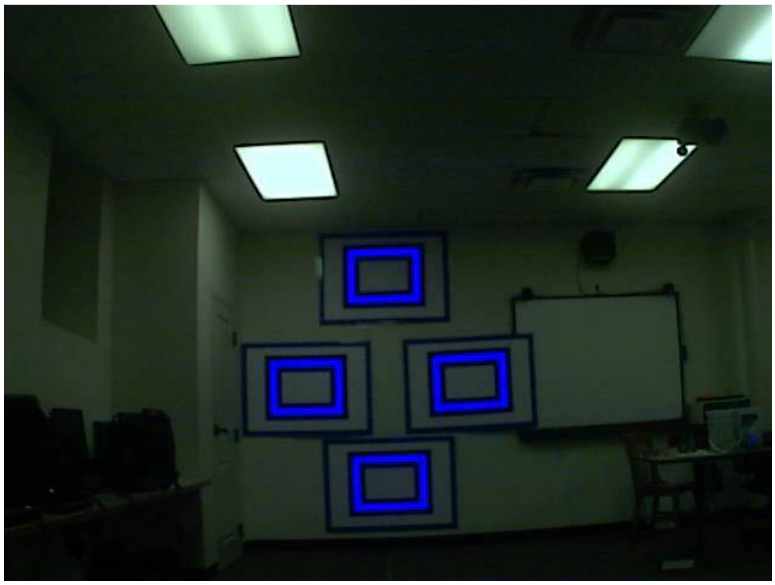


FIGURE 7.7 A test image with four retroreflective targets illuminated by a single, small “angel eye”.

Actually, there is one other trick that can really help here. The camera was prepared by aiming it at a clean white wall that was illuminated by a 500 W halogen work light. It was painful to look at, it was so bright. The camera was allowed to automatically determine the exposure for this really bright scene, but then that exposure setting was locked and made permanent.

You can use a web browser to log directly into the camera and make these adjustments. If you don’t have a manual that tells you how to do this, you can download one directly from the Axis web site.

If the camera is allowed to adjust its exposure, then the relative brightness of the targets will be changing as your robot moves around, and you would like to avoid that. With the exposure locked, and the light source for the targets in a fixed position relative to your camera, the appearance of the targets should not vary much from image to image. Because of how the retroflectors work, if another robot is lighting up the target at the same time as yours, your camera will not “see” any of their light, only your own, which is a pretty neat trick.

Choosing your colors

Your first task is to find out what color your targets are. To do that, you should code the program shown in Fig. 7.8. This program reads in an image and displays it twice, once as an RGB image, and once as an HSL image. They don't look any different, but the way in which the color is encoded is very different. The easiest way to see that is to look down at the bottom of the figure. You can see that for both images, the cursor (not shown) was on the pixel at (506,306). To the left of that are the three color values. The two sets of numbers are very different. In the left image, since the cursor was sitting on one of the targets, the blue channel as a very high value, while the red and green channels are at or near zero.

The HSL image is more complicated. The color space is best thought of using cylindrical coordinates, r, z, and θ. The angle, θ, is the color coordinate, hue. Red is at $0/0°/0$ (I'm giving you three different names for the same angle here). As θ (hue) increases, the color shifts through orange and then yellow until at $85/120°/2\pi/3$ it is pure green. Increase the hue value more, and the color shifts through cyan to pure blue at $171/240°/4\pi/3$. Increase θ (hue) and the color shifts through magenta and back to red at $255/360°/2\pi$. Saturation is the radial coordinate. If r is small, then the pixel has a nearly equal amount of all colors and is, as a result, gray rather than colored. As r (the saturation) value increases, the pixel becomes less and less gray, and more purely the color that corresponds to the hue value. When that saturation value is 255, then the color of the pixel is said to be "saturated". If it happens to be a blue pixel, it just can't get any bluer than that. Finally, the z coordinate is luminance. If $z = 0$, the pixel is black, and the hue and saturation values don't matter. As z/luminance increases, the pixel becomes brighter. At a luminance of 255, the pixel is completely white and again the other two values don't matter. The pixel is most colored at luminance values around 128.

OK. What about the confusing graphs in Fig. 7. 8? These are histograms. We'll start on the left, because it's easier. Look only at the red curve to start. The horizontal axis is the channel value (the red channel in this case, because I set the graph up to plot the red data using a red curve. Pretty subtle, don't you think?). The vertical axis tells you how many pixels have that particular channel value. For example, about 12,000 pixels in the image have a red value of 40. So, the meaning of the green and blue curves is pretty obvious. There is a hump in all three curves in the range 25 to 75. This is most of the pixels in the image. These are low values because this is overall a pretty dark image. The exception is the blue curve up above a value of 240. These pixels are the targets. (Note that unlike the example in Fig. 7.7, this image does not include any ceiling lights, or other bright white light sources. If you have some of those in your image, figuring out how to find the target gets much harder.)

The histogram for the HSL image is much harder to interpret. Again we start with the red curve, which is the graph of hue values. Whatever you might call the overall tone of our test image, you would not call it red. As a consequence, the hue value is zero at zero and again at the high values up near 255. In fact, almost all the hue values seem to be in the range from about 80 to 170: the green/blue part of the color spectrum. The curve is very spiky in this region. Which spike represents our targets? The narrow peak at about 170, although that is not immediately obvious. Whatever color our targets are, they are very much that color. They are highly saturated; there's nothing gray about them at all. So the spike in the green curve (saturation) at 255 contains our target pixels. But the image is overall pretty dark, even our targets. So the blue curve (luminance) has no high values. Our target pixels are

somewhere in that spiky hump, evenly mixed in with all the rest, so we can't specify a range of luminance values that will pick them out.

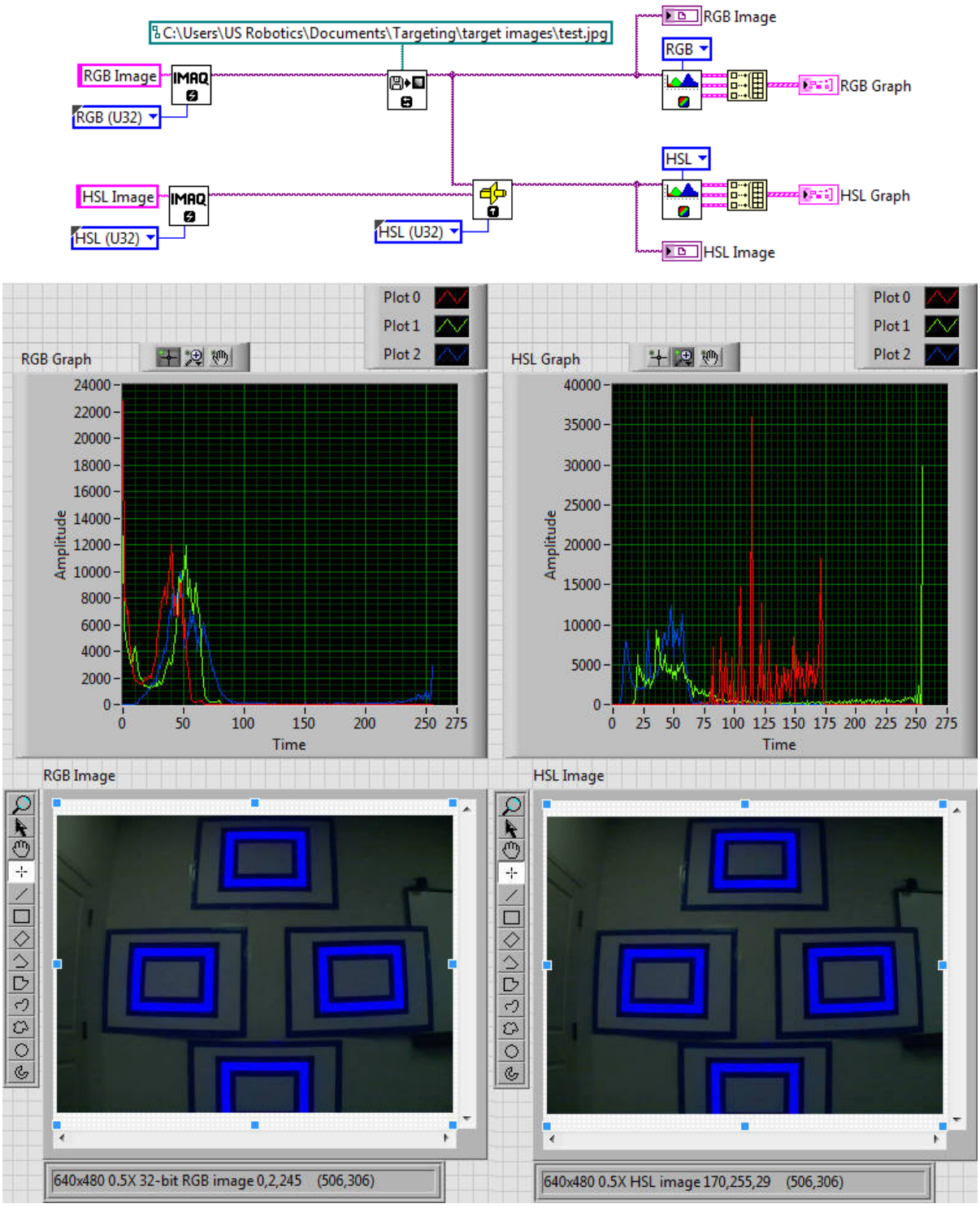


FIGURE 7.8 Code and front panel for a program that takes an image and displays two histograms: one for an RGB version of the image, and one for an HSL image. Also shown is the image itself. Although the cursor is not shown on the images, you can see that the info panel at the bottom is showing the RGB and HSL values for the same pixel.

So, what was the point of this exercise? To specify three color values that together represent a target pixel, and definitely do not represent a non-target pixel. Actually, we don't want to specify, for example, a single blue value, or single hue value, but a narrow range. Looking at the histograms can help us decide whether the job will be easier for an RGB encoded image, or an HSL encoded image. For this particular case, I'd go with RGB. If we scan the image for pixels that have a blue value in the range from something like 128 up to 255, we will find our target pixels. In addition, if we demand that the red and blue values are small, we won't get confused by white lights, and will find *only* our target pixels.

Finding the BLOBS

This is where the actual image processing starts. "BLOB" is image processing short hand for "binary large object". All the cool kids are saying it. So start by modifying the code from Fig. 7.8 so that it matches Fig. 7.9, which shows a pretty cool piece of magic. The right hand image display uses the Binary palette (and will just appear black if you forget and leave the palette as Grayscale. Figure out why!) The VI that is doing all the work is called IMAQ ColorThreshold. You need to tell it what kind of image you are giving it (RGB in this example), the ranges for the pixels it should "find", and the replacement value, which is 1. Pixels whose RGB values match the range will be set to the replacement value. All the others will be set to zero. Because you want to compare the "found" image with the original, you need to allocate two memory blocks (in your actual competition code, you may want to skip displaying the RGB image—or even any image at all!).

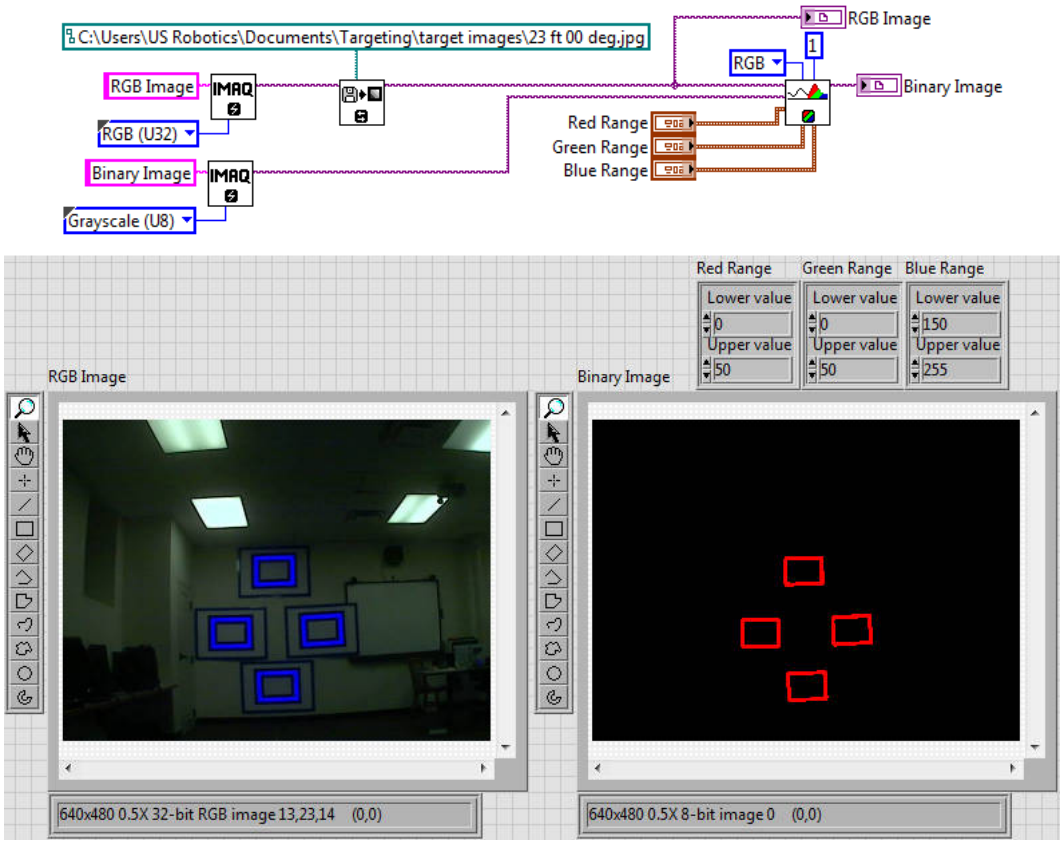


FIGURE 7.9 The first stage in BLOB finding: identifying the target pixels.

Don't just run the code with the RGB ranges shown. Experiment so you get a feel for how things behave. How high can you have the red and green values and still exclude the ceiling lights? Your goal here is to find a set of ranges where the actual range values don't matter. That is, if you raise or lower a range value by 10, the same pixels are still identified as "target". If you are comfortably in the middle of this "it doesn't matter" zone, then your target finding will be robust: small changes in the target color due to changes in field lighting or robot position will not affect your ability to find what you are looking for.

Now that we have found our BLOBs, we need to analyze them. Fig. 7.10 shows a modification to our sample program. I've cut off the boring stuff showing how the image is read from disk, and I no longer show the original image. There's a lot of new stuff here, so we'll take it one VI at a time.

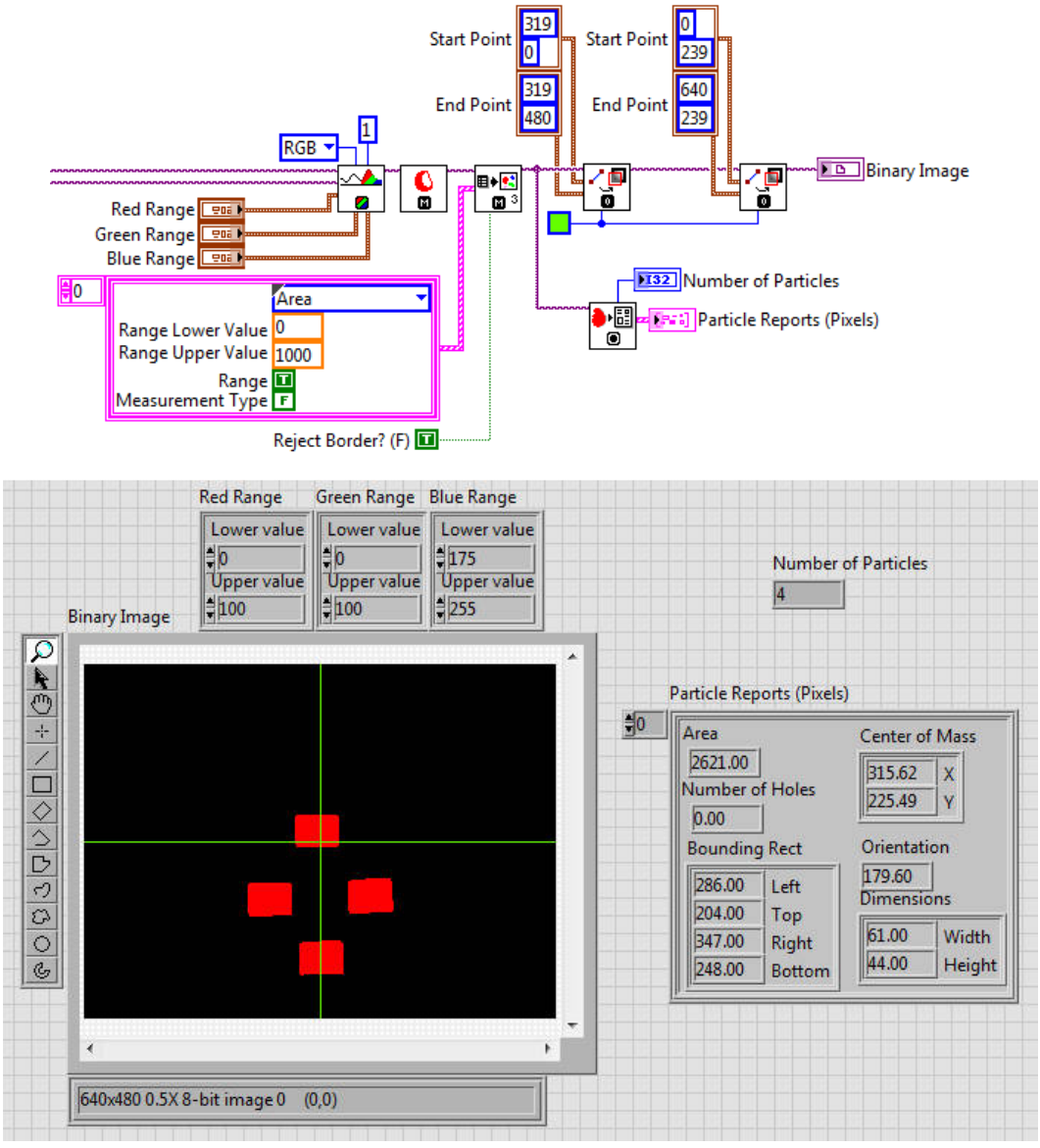


FIGURE 7.10 A complete BLOB finding program.

You can see that the first VI is the thresholding VI from Fig. 7.9. The next VI is IMAQ Convex Hull. This VI takes each blob in the image and fills it in. Obviously the large regions in the center of the target have been filled. But it also smooths the edges a bit, because it wraps an imaginary string around the object and pulls it taught. All the pixels inside that string are then filled in.

Next is IMAQ Particle Filter 3 (there have been several versions of the particle filter over the years. The number lets us know which we are using). This VI is doing two important things at once. First, it is eliminating any particles that are too small to be a target. There are none in this perfectly lit test image, but in many circumstances you can get reflections from shiny bits, or lights on the field that are detected as particles. We don't want to aim our robot based on these, so we eliminate them with this filter. Anything with an area less than 1000 pixels won't show up as red in our binary image. (You will need to figure what is the right area setting for this filter for the competition you are coding for!).

The other thing this filter is doing for us is eliminating any blobs that touch the edges of the image. We are going to use geometrical measurements on these rectangles. If one of them is cut off by the edge of the camera frame, then we have no way of telling how big it is, or even where it is precisely located in the field of view. If we eliminate these targets, then we don't have to worry about accidentally using bad information. You may not want to do this in all designs, but for this set of targets, it's a good strategy.

The next two VIs have nothing to do with actual blob finding, but if you want to draw lines on an image, here's how to do it.

Finally, there is IMAQ Particle Analysis Reports, which returns a set of standard measurements on each blob. Create the indicator, and then look at the front panel to see what you get. The output is an array of clusters. The size of the array is equal to the number of particles found. You actually have two choices here. There is a VI, found under the Motion and Vision sub-palette, called IMAQ Particle Analysis. This second VI allows you to choose which properties of the particles are measured and reported. There are a huge number of them to choose among. For this example, we are using the "Reports" version because it is simpler and has everything we need: location and apparent dimensions of each target. Be aware that both versions of the particle analysis routine give you the results in both camera pixels and "real world" units. We need to use the pixel outputs because we don't know the calibration factor needed to convert pixels to real world units (*e.g.*, centimeters). In fact, determining that conversion factor is, in a way, what we will be using the camera for.

Camera Optics

Before we can figure out what to do with the information we've obtained from processing our images, we need to make a small digression into optics. Fig. 7.11 shows a top view of the camera in front of the targets, which we assume to be all together in one plane. This plane is imaged onto the CCD chip in the camera. If this was an actual optics course, we'd call the plane of the targets the *image plane*, and the plane of the CCD the *image plane*. We'd also discuss how the image is inverted, but since the camera software takes care of that for us, we'll just skip the whole thing. The key thing here is the angle θ_H, which is the *horizontal field of view* of the camera. If you have the older model Axis camera (the model 206), this angle is specified to be 54°. If you have the newer camera (Axis model M1011), it is

specified to be 47°. As you will see in a bit, this number is not quite right, at least for the older camera. We'll see how to use test images to determine exactly what it should be.

There are three ways to talk about the field of view, and for what we're doing here, you want to keep all three of them in your head at once. The first way is as an angle, as we've just mentioned. The second way is as the number of camera pixels. This depends on how you have the camera resolution set. In the test images supplied with this book, the field of view is 640 pixels. (It is probably a bad idea to use that high a resolution in competition.) Finally, the field of view is a physical distance, in centimeters for example, or inches if you are fond of that dead English king. This is the least well defined definition, but also an important one. It depends on how far the camera is from the things it is looking at, and also on what those things are. The simplest case is when the camera is aimed directly at a flat wall. Then the field of view is simply the measured distance from the left most point on the wall that is within the camera frame to the right most. And in fact, we are going to pretend that all the cases of interest are pretty much like this simplest case.

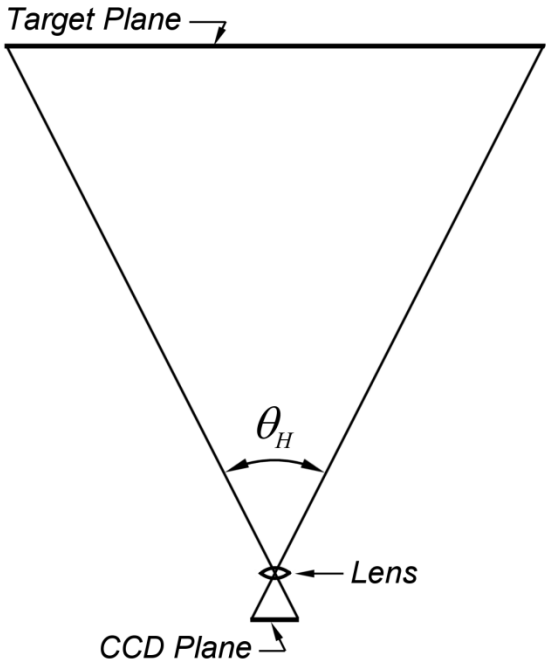


FIGURE 7.11 The basic camera geometry. In this view we are looking down on the camera and targets, so the field of view is 640 pixels wide at full camera resolution.

Finding the target distance

As you may have already guessed, we are going to continue to use the vision targets from the 2012 Rebound Rumble completion as our example for image processing. That was an exceptionally good year for using image processing, as the placement of four large targets in a single plane allowed for some very sophisticated and precise determinations of the robot position. From this one example, which we will look at in detail, you should be easily able to generalize and develop solutions for the actual competition you are trying to solve.

Let's start with the target geometry, which is shown in Fig. 7.12. Imagine that the camera is some distance back from this pattern and is pointing straight at it. We want to find that dis-

tance, labeled d in Fig. 7.13, which is now a *side* view. The height of the field of view is H. Because the CCD is only three quarters as tall as it is wide, the vertical angle of view is $\frac{3}{4}\theta_H$,

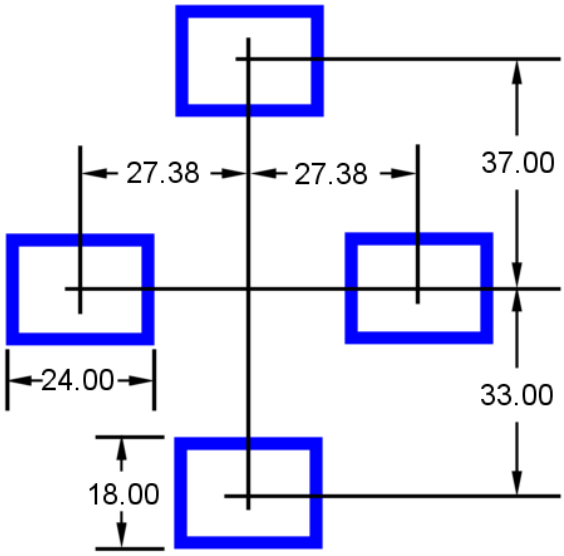


FIGURE 7.12 Target geometry from the 2012 Rebound Rumble competition. Units are in twelfths of the shoe size of a dead English king.

and thus the half-angle indicated in the figure is $\frac{3}{8}\theta_H$. The dots in the image represent two points that we can reliably find using our image processing routines. For example, they could represent the locations of the center-of-mass for the highest and lowest targets in the pattern of four in Fig. 7.12. The real world separation between these two points, y, is 70 inches. The vertical separation between these two points in camera pixels is y_p, which we can compute from the data output by our blob finding routine. Finally, we know how tall the field of view is in pixels: it's just the vertical size of the image, which we will call H_p. Because the camera lens scales everything by the same amount, the ratio of the actual size of any object and its size in camera pixels is a constant. This includes the height of the field of view. Therefore we can write

$$\frac{y}{y_p} = \frac{H}{H_p}. \tag{7.1}$$

The only thing in this equation we don't know is H, the real-world height of the field of view at the target distance d, so we can solve for it.

From Fig. 7. 13, it should be obvious that

$$\tan\left(\frac{3}{8}\theta_H\right) = \frac{H}{2d}. \tag{7.2}$$

In this equation, the only unknown is d (assuming we have already used the previous equation to find H), so we can solve for it and know how far our camera (and therefore our robot) is from the targets.

Actually, we can do way more than that. And we can start with using the camera to get a better value for the field of view. To do that, we need to process a whole bunch of images from

the package of test images that accompanies this book. Start by adding something like Fig. 7.14 to the end of your code from Fig. 7.10. What I've shown here represents my personal style, not some absolute requirement. You have some basic information on each blob. How you want to organize it is up to you: you have your own style.

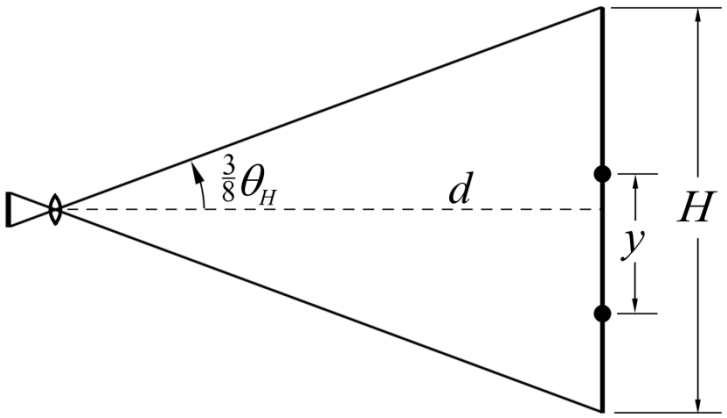


FIGURE 7.13 Geometry for using a known vertical distance to compute the distance to the targets.

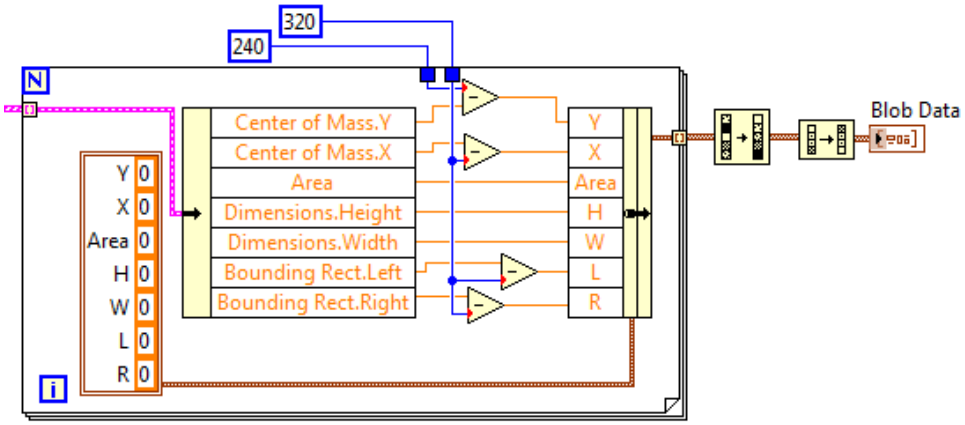


FIGURE 7.14 Organizing the data from the blob finding routine.

The first thing to notice in this code is that I have offset the data so that the point (0,0) for the coordinate system is in the center of the image instead of the upper left corner. Once the loop has completed, the code sorts the data so that it is ordered by the height of the blob y coordinate, from highest to lowest. Once this piece of code has been added, you can take the whole blob finding routine and package it into a sub-VI. Then you can use this sub-VI in a program like the one shown in Fig. 7.15.

There's a lot going on in this program, and it's positively crammed together to make it fit on the page, so we'll take it one step at a time, starting from the left. The images in the test package are all named by the distance from the camera to the target, in feet, and the angle of the line the camera was moved along to take the images. This routine reads in the zero degree images (the ones taken with the camera placed along a line perpendicular to the targets). If you are really paying attention, you will notice that the zip file contains images taken at 10,

11, and 12 feet from the targets, but that we do not read them in. That's because the lowest target is clipped in those images, and the code is much simpler if we don't try to analyze those images too.

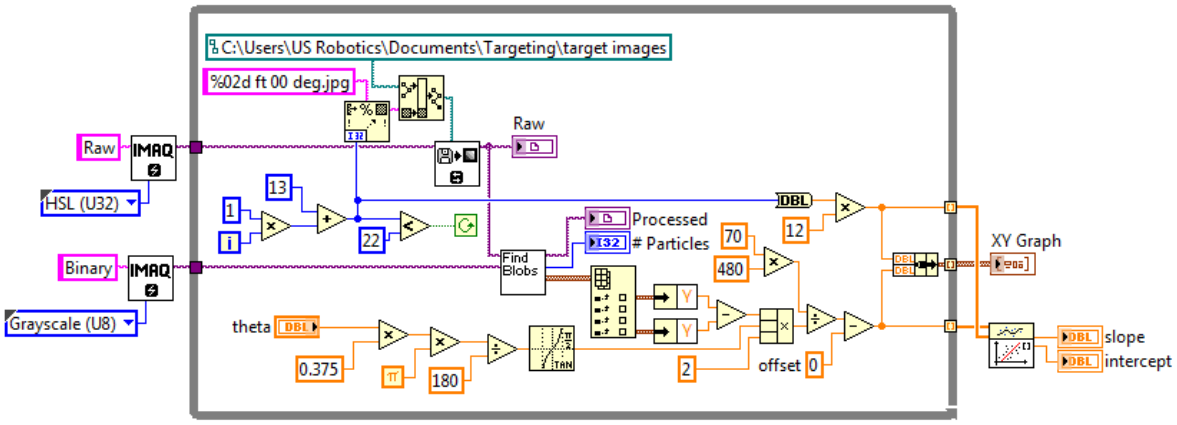


FIGURE 7.15 Code to read in a series of images, compute the distance from the camera, and make a graph of computed distance vs. actual distance.

Each image is fed to the Find Blobs routine, and out comes an array of clusters (the clusters built using the code in Fig. 7.14). The difference between the vertical position of the highest target and the lowest target is used to compute y_p. That difference then goes into a bunch of math. You should convince yourself that this math does exactly the computation Eqs. 7.1 and 7.2 to find d. Also in there is code to take the actual distance to the target, as given in the name of the image, and convert it from feet to inches. This actual distance is then bundled with the computed distance, and used to make a graph of computed d vs. actual d. At the same time, a VI from the Mathematics: Fitting sub-palette is used to find the slope and intercept of the resulting line. Note that the control "theta" represents θ_H, and we've made it a control because we plan on changing it. There is also a constant "offset" in there, which we will get to in a minute.

Fig. 7.16 shows the important part of the front panel for this program after it has been run assuming that the camera specified view angle of 54° is correct. But it is not correct. For example, when the actual distance is 180", the computed distance is 175". More importantly, the slope of the line is not 1, so that this error gets larger for larger distances. Because the slope is less than 1, it must be that our value for θ_H is too large. You doubt me? Work it out for yourself. In fact, it must be that

$$\frac{\tan\phi}{\tan 20.25°} = 0.925132. \tag{7.3}$$

If you are very clever, you will note that 20.25° is three eighths of 54° , so ϕ must be three eighths of the corrected view angle. If you do this math, you find that it would be more accurate to use a value $\theta_H = 50.25°$. Use that as theta and run the program again. Then take the intercept value (which should be 9.17449, or close to it) and use that as the offset constant in the program. When you run the program again, you should find that the computed distance matches the actual distance rather well. To test whether this is robust, modify your program so that it reads in the images taken at 10° or 20° off of perpendicular. You will have to be clever because these were taken only every 2 feet. It works terrifyingly well.

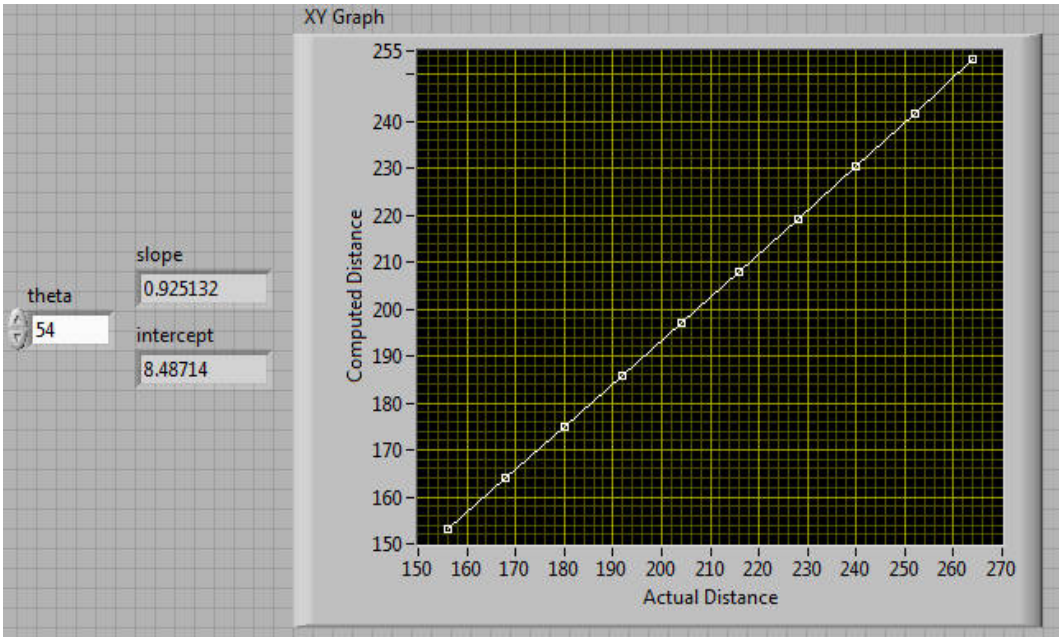


FIGURE 7.16 Partial front panel for the code in Fig. 7.15. It has been run with the "default" view angle of 54° and with the constant "offset" equal to zero.

Bear in mind that what we have done here is just the simplest thing. In "real life", which is to say, in a competition, the robot might be so close to the targets that one is clipped (like in some of the test images). But you could easily write code to figure out which target is missing and use the rest of the target information to compute the distance. You could even use the height of a single target as your known y, but in that case you should not expect to be able to compute the distance within 1%, as we can using a larger y.

Exercise E12: Write code that automatically computes the distance to the targets for *all* the images in the test package, including the ones with clipped targets. (Hint: You can improve the accuracy if you use a different value for the offset when using only the upper three targets.)

Advanced Stuff

Fig. 7.171717 shows a top view of the camera and the targets, plus a lot of other circles and arrows. All that's missing is a paragraph on the back explaining what it is. (If you are under the age of 50 *and* get that joke, you have just earned 10 Obscure Cultural Reference Points.) The centroids of the targets are represented by the solid black dots. You will notice that the camera is neither centered on the targets, nor pointed at the center of the targets. Instead it is pointed just inside the right hand target. Remember how I said we were always going to pretend the camera was pointed directly at a wall? That's the dotted line, perpendicular to the camera axis. The left and right edges of the camera frame are denoted by the little triangles.

In the plane of the targets, the left and right targets are separated by a distance x. But to the camera, they appear a little bit closer together than that, as indicated by the open circles. It should not be hard to convince yourself that this reduced distance is not exactly, but to a very good approximation, equal to $x\cos\beta$. Therefore we can construct the identity, analogous to Eq. 7.1,

$$\frac{x\cos\beta}{x_p} = \frac{W}{W_p}. \tag{7.4}$$

If we first use the *vertical* separation between the center targets to find d, as in the last section, then the only thing we don't know in Eq. 7.4 is β. (Because if we know H, then we can find W from $W = \frac{4}{3}H$ for the Axis camera.)

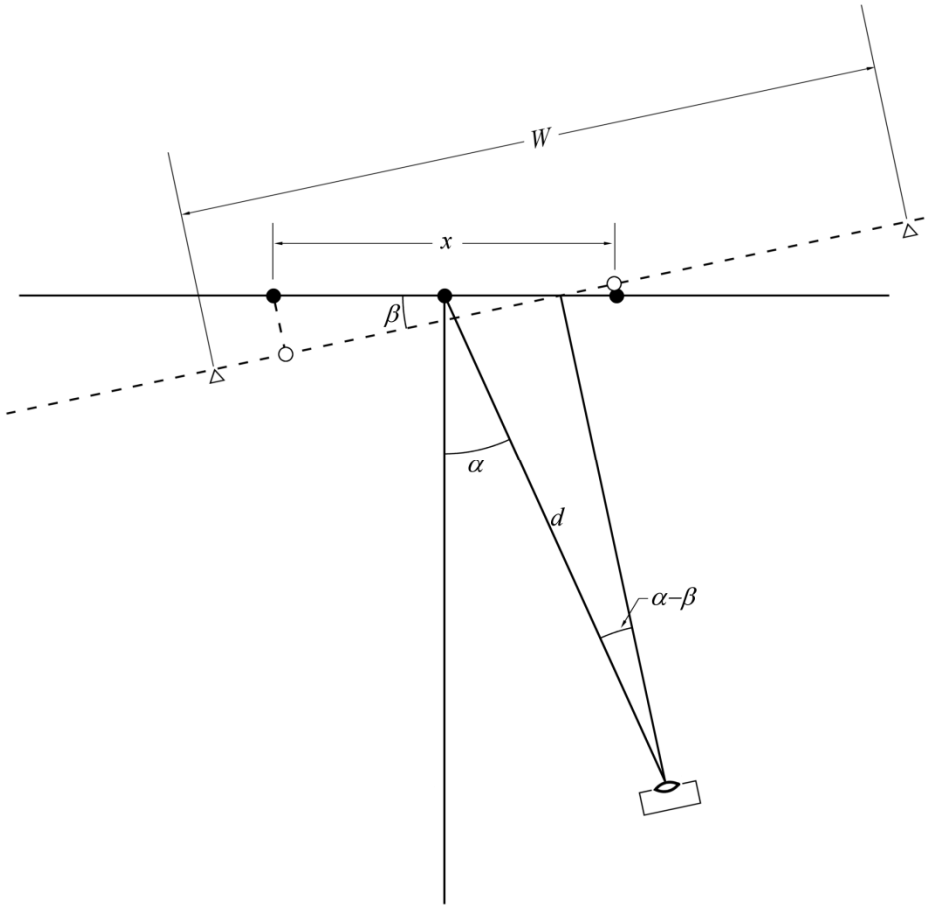


FIGURE 7.17 Analyzing the image in the case that the camera is off-axis and not pointing directly at the center of the targets.

Now, suppose that the horizontal distance, in pixels, from the center of the image to the center of the target pattern is o_p. Then you can easily find the angle $\alpha - \beta$ from

$$\frac{\alpha-\beta}{\theta_H} = \frac{o_p}{W_p}. \tag{7.5}$$

At this point we know everything there is to know about the camera: we know its position on the field (in a cylindrical coordinate system centered on the target pattern) and its orientation. If the camera is bolted to a robot, we know the same things about that robot, which might prove useful…

Exercise E13: Write code to determine α and β for the images in the test package, at which point you will have earned your Image Processing Zen Master merit badge.

Chapter 8 — The Dashboard

What is the Dashboard? It's a piece of LabVIEW code that runs on your laptop. It can receive and display data from the robot. It can receive and process images from your camera. And it can send data to your robot. This data can come from image processing, from auxiliary electronics plugged into your laptop and read by the Dashboard, or from front panel controls on the Dashboard itself. For example, you could process an image to determine the distance from a target, send that distance to the robot, which could then execute a move based on that distance.

The Dashboard was first mentioned way back in Chapter 3 where all we wanted you to do was ignore it. It is automatically launched when you start the Driver Station, and the default configuration looks like Fig. 8.1. If your camera is powered up and attached to your robot's wireless access point, you will get a live image in the upper left, although your camera might not see Rio unless you have the special lens installed.

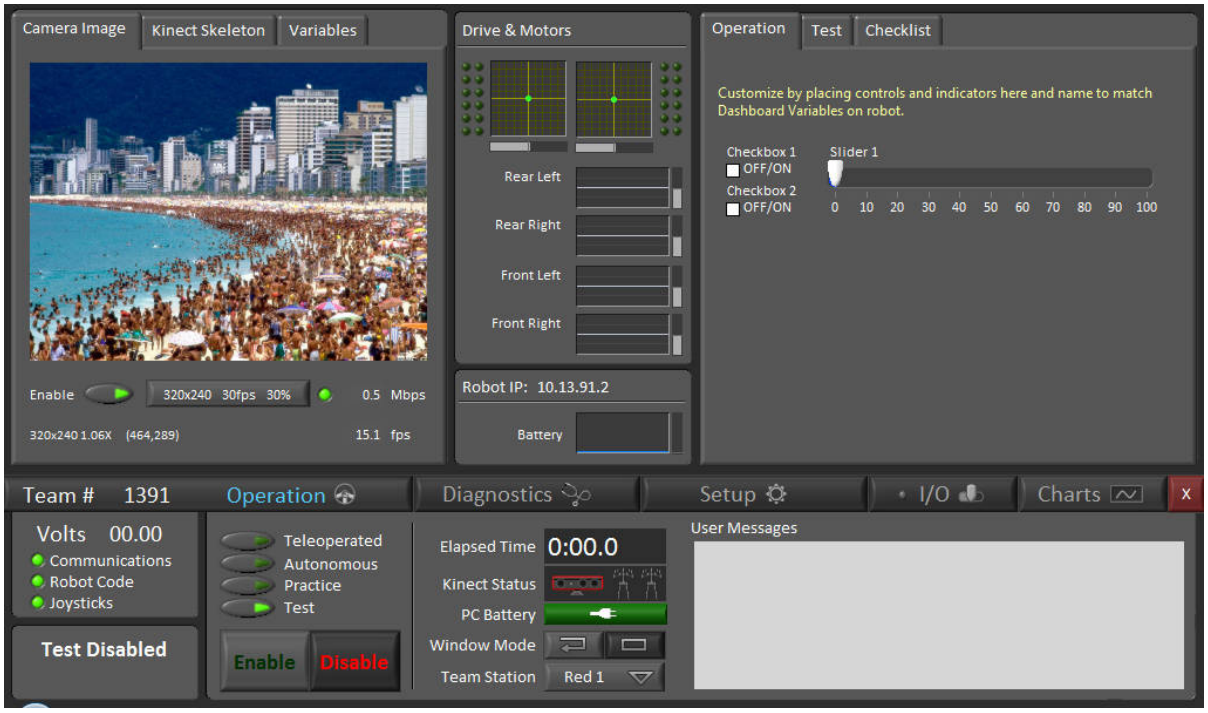


FIGURE 8.1 The Driver Station and the default Dashboard.

We will go into some detail on how the Dashboard actually works, so that you have the knowledge you need to modify it. But in order for any of that to make sense, you need to have a basic idea of how information is passed between the Dashboard and the robot.

Basic Dashboard Communication

As of the 2013 season, this is easy. The actual system that implements this communication (a network table) is complex, but you don't need to know anything about it. Suppose you have a Boolean value generated by your Dashboard code that you want to send to the robot. In your Dashboard, you implement something that looks like Fig. 8.2.

This bit of code has two essential parts: the text input you use to name this piece of data, and the data value itself. In the figure, the actual data control and the text field have the same name. That has a certain logical appeal, *but is not a requirement*. The data comes in on a wire, after all, and so is actually nameless.

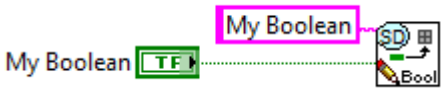


FIGURE 8.2 Putting a Boolean value into the network table. Note the pencil to indicate that we are writing data.

Down on your robot, to read this value you implement something that looks like Fig. 8.3. This kind of code can go in Periodic Tasks, Autonomous Independent, or Teleop. Here we again need the name under which the data was stored. If this name doesn't match the storage name exactly, then you will get an error instead of your data. (This method of naming for storage and retrieval should remind you of the way in which we name things like motors and sensors in Begin.vi so we can refer to them in other parts of the robot code.)

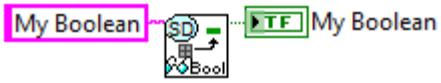


FIGURE 8.3 Reading a Boolean value stored in the network table. Note the cute reading glasses to indicate that we are reading data.

Now, suppose you want to go the other way. What do you do if the robot has a Boolean value (say, the status of a limit switch) you would like to be able to display on the Dashboard? *The code is exactly the same!* Just put the store code (Fig. 8.2) on your robot, and the read code (Fig. 8.3) in your Dashboard. Very simple. Very nice.

Probably not every piece of data you need to pass is a Boolean. Not to worry, if you look at the Dashboard palette (a sub-palette of the WPI Robotics Library palette), you will see (as in Fig. 8.4) that there are six different data types you can read and write: single Boolean, Double Precision Real, and String values, and arrays of these three types. There are also polymorphic VIs that let you choose which of the six types to use <u>after</u> you've dropped them into your diagram.

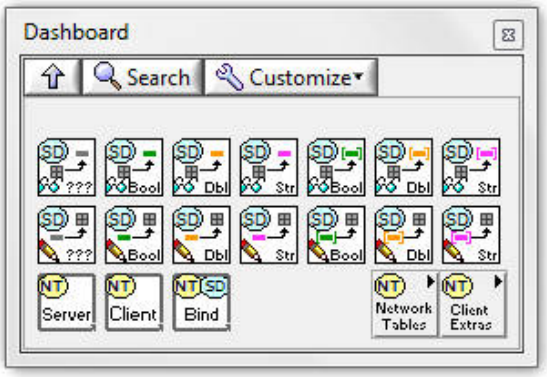


FIGURE 8.4 The dashboard palette. The polymorphic VIs are the ones with "???" as their type. We will not need to mess with the more advanced stuff in the bottom row.

The Dashboard Project

To understand the Dashboard, we have to look at the code. To look at the code, you need a copy, which you can get by creating a new Dashboard project. Open LabVIEW and from the Getting Started screen, under **New**, click on **FRC Dashboard Project**, which should bring up a window as shown in Fig. 8.5. As with your robot project, you need to name it and select a location for the file. I recommend the same location as your robot project. None of the file names conflict (unless you create that problem for yourself), and it becomes that much easier to back up everything associated with the project.

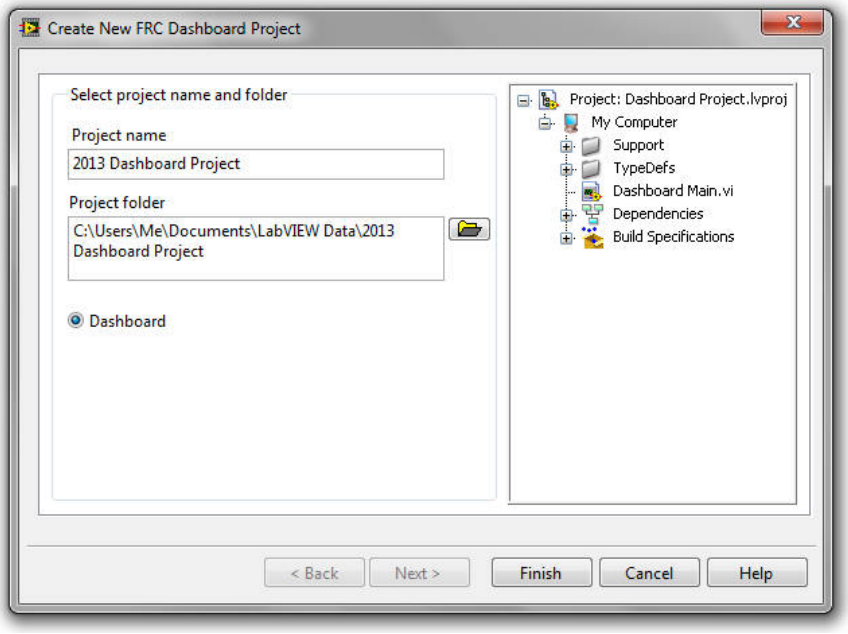


FIGURE 8.5 The Create New Dashboard Project window.

As with the robot, you will get a project tree, although in this case it's quite a bit simpler. The only file of interest to us is Dashboard Main.vi. Go ahead and open that file. The code you see will be terrifying at first. It's also the code you will need to modify to make your own Dashboard, so we'd better explain it enough so it's not quite so terrifying.

Under the Hood (or Behind the Dashboard)

I'm going to reveal here that I have an evil intent. A well designed Dashboard that is useful in competition will have just a few key indicators. These indicators will be very large, so that the information they contain can be read with a quick glance out of the corner of your eye. This is not the default Dashboard, which has been designed to provide you with a lot of information that might be useful in debugging your robot, but not in competition. So, I expect you to delete most of the provided code in the Dashboard, thus shoving a dagger straight through the hearts of the programmers who clearly worked very hard to create a complex and highly functional piece of software. Oops.

OK. That might be extreme. But at the very least, some serious renovations are in order. We'll discuss alternatives to deleting nearly everything at the end of the chapter when we discuss Dashboard strategies.

Loop 1: Simple Data Communication

The default Dashboard has seven while loops that run in parallel, plus some extra bits of code. Loop 1 is shown in Fig. 8.6. The stuff outside the loop on the left has mostly to do with inter-loop communication and doesn't concern us. The stuff to the right of the loop is mostly a way of labeling stuff on the front panel, and again doesn't concern us.

This loop executes every time a data packet arrives from the Driver Station. In the default configuration, it is used to update diagnostic displays, mainly shown in the central part of the Dashboard, as highlighted in Fig. 8.7. This is the most straightforward way to get data to and from the robot. You simply put one half of a read/write pair, as in Figs. 8. 2 and 8. 3 inside the case structure, and the other half where you need it in your robot code. If you open the default Teleop code, you will see exactly that read/write pairing, although as it happens all the writing is on the robot and all the reading is on the Dashboard.

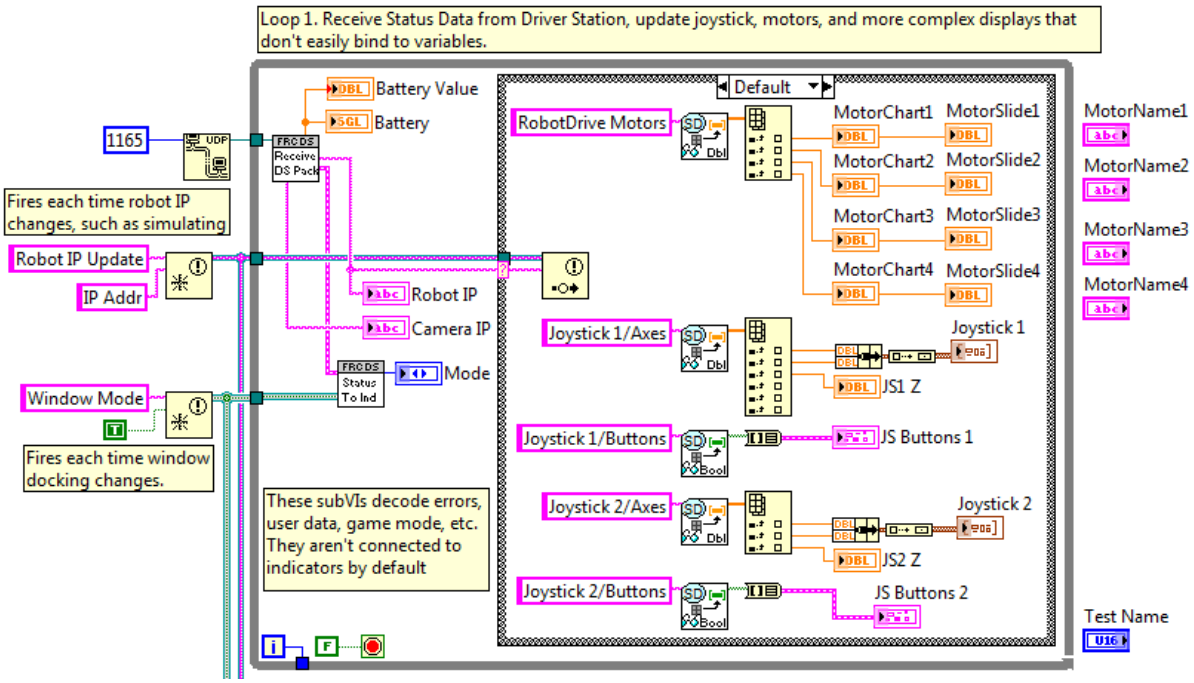


FIGURE 8.6 Dashboard Loop 1.

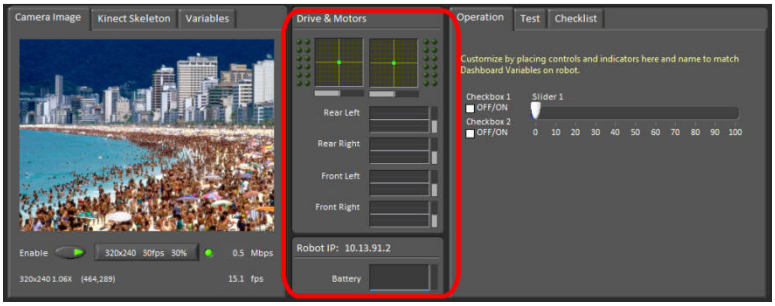


FIGURE 8.7 Data displays used by Loop 1.

In creating a custom Dashboard, you should keep this loop, but you don't need to keep much that is in it. If you don't want to display the joystick and motor data, you can delete all these indicators, and with them the reads from the network tables. The four MotorName controls

outside the loop on the right could go as well. Outside the case structure, there are indicators for the battery voltage and IP addresses. If you don't feel you need these on the Dashboard, they can go as well. Keep the two Driver Station related VIs. We haven't discussed Loop 3 yet, but bear in mind that if you decide you don't need Loop3, then you can remove the case structure in Loop 1. Once you do that, the "Robot IP Update" notifier and related wiring can go. I strongly recommend keeping the "Window Mode" notifier though. As you add your own code, don't be afraid to expand this loop to make as much room as you need to fit all your data writing and reading in.

Loop 2: Image Acquisition

Images are acquired and processed in Loop2, shown in Fig. 8.8. This loop acquires images from the Axis camera. Outside the case structure is some code that should remind you of the code in Fig. 7.5. It's a bit fancier because there is a front panel switch to turn image acquisition on and off, and because it is possible to change the camera IP and settings while the program is running.

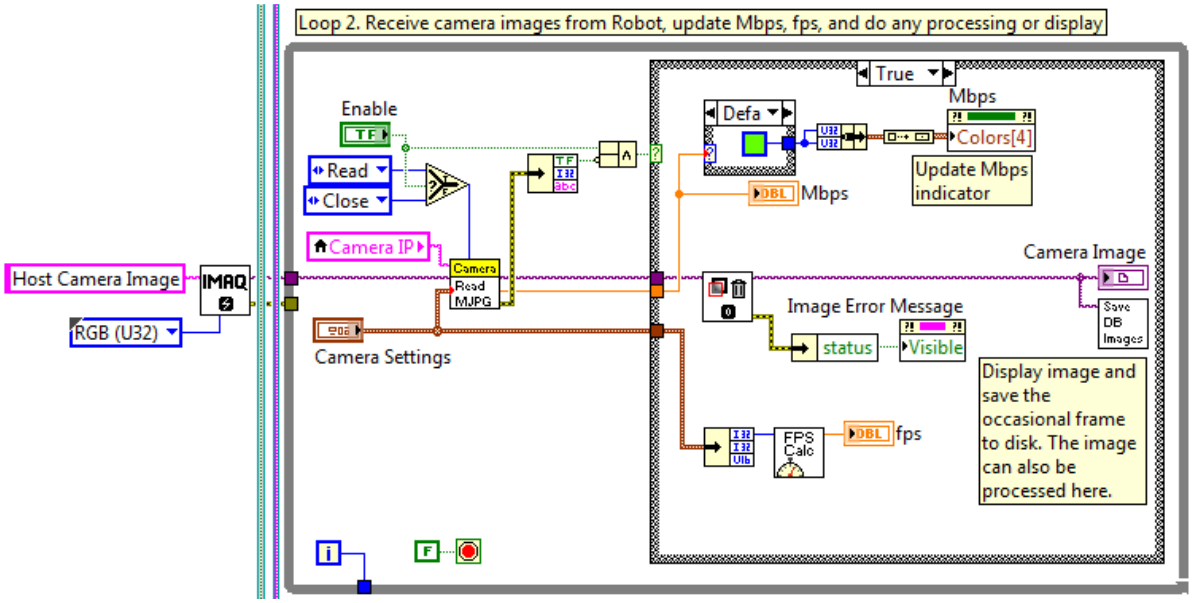


FIGURE 8.8 Loop 2: image processing.

Inside the loop there is code that is mostly not essential. The upper bit displays the data bandwidth being used by the images, and controls the color of a little Boolean. If your camera settings are using too much bandwidth (above 3.5 megabits per second) the indicator will go from green to yellow. If the demanded rate gets above 5.5 Mbps, the indicator goes red. At the bottom is a computed frame rate. If your demanded bandwidth is low enough to keep the indicator green, then you should see that the actual frame rate can keep up with the demanded frame rate.

Although I've just said that these bandwidth indicators are not essential, the reason they were added is very important. If you look way back at Fig. 3.1, you will see that there is one wireless communication channel between your robot and your laptop. That means that the basic control of your robot, in the form of data packets sent from the Driver Station, is fighting for bandwidth with the stream of images your Dashboard is demanding from the camera. Don't make this a fight that robot control is likely to lose. You can control the image size, the

amount of image compression, and the camera frame rate. Experiment some with your test target setup, and use the smallest image you feel gives you reasonable image processing. You can probably do a perfectly nice job working with 320 by 240 images. The default compression value of 30 seems to be a reasonable compromise, but you can adjust it as you see fit. Finally, ask yourself if you really need to acquire 30 frames per second. At 10 frames per second, you will be acquiring an image every 100 ms. That may be fast enough. If you need to, you can adjust any of these parameters "on the fly". Maybe you have a "low res" mode for cruising around and a "hi res" mode for shooting? It's up to you. You've earned that Image Processing Zen Master merit badge, haven't you?

In the middle of the case structure, you can see that the image is processed through IMAQ Clear Overlay. Since the image has no overlay, this does nothing but provide an easy way to detect whether there is an error associated with the image. If there is, the Image Error Message indicator is made visible, and the resulting error message will be displayed on top of your image. Also in here is a nifty little VI that will save an image to your laptop hard disk once every second. The saved images can be found in "MyDocuments/LabVIEW Data". The routine only keeps the most recent 60 images so that you don't fill the hard drive. If you are having problems with your image processing and targeting, these saved images could provide you with an extremely useful diagnostic. You would probably want to add some code so the VI was only called during times that you were actually interested in the results. Or, you could make your own code (using this VI as a reference) that saves images using a naming scheme of your choice at times of your choosing.

Since I have mentioned image processing, I should point out that this loop is the obvious place to do it. You can send the results of the processing down to the robot from here. There is no need to somehow get the value into Loop 1.

Loop 3: Autobound Variables

The default dashboard has two the tab controls, the front tab of the right hand control, highlighted in Fig. 8.9 is the Operations tab, and it is special. Any control or indicator you put onto the front panel inside this tab is automatically put into the network table using its actual name as the storage name. This saves you half of the work, but you still have to implement Fig. 8.2 or 8.3 in your robot code to read or write the value. And it only works for stuff you put here, in the Operations tab. It's something of a neat trick, but the lack of flexibility is a big price to pay to save a rather small amount of work. (If you look in the default Teleop VI, you can see the code used to read the checkboxes and slider on the default Dashboard.)

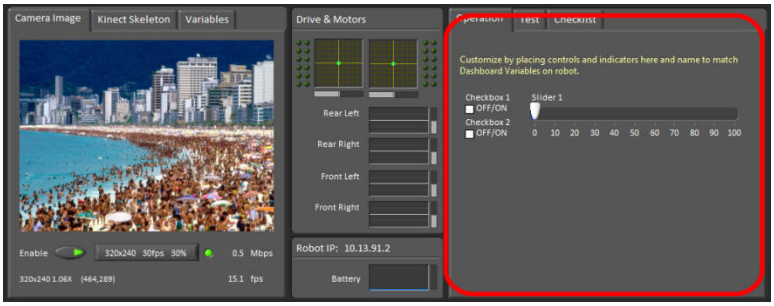


FIGURE 8.9 The Operations tab on the default Dashboard.

Loop 3 (Fig. 8.10) does the work of gathering up the controller and indicator names on the operations tab and putting them into the table, and also of continually monitoring the values and keeping everything up to date. (All the heavy lifting is hiding inside the "Bind" routine.) If you decide to do away with the Operations tab (or even the entire tab control), you can delete this loop. If you do so, you should also delete the notifier wiring as discussed in the section on Loop 1.

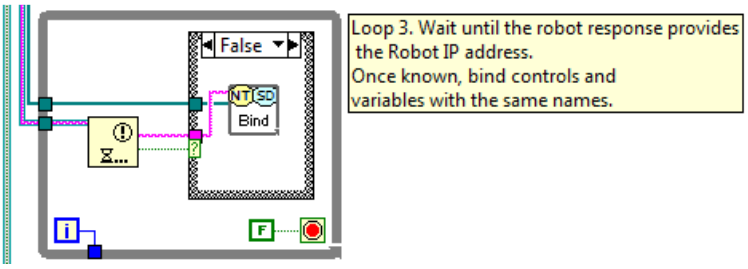


FIGURE 8.10 Loop 3, which sends data to and from the indicators and controls in the Operations tab.

Loop 4: The Network Table Tree

Fig. 8.11 shows the loop that updates a tree indicator on the Variables tab (left hand side of the Dashboard, behind the camera image). This tree shows updated values for all the variables in your network table, both the ones on the Operations tab, and the ones you've coded entirely yourself. It also shows how much bandwidth is being used by the network table. Possibly useful when debugging. Probably useless in a competition. From a functional point of view, there is no problem with deleting this loop and the Variables tab as well.

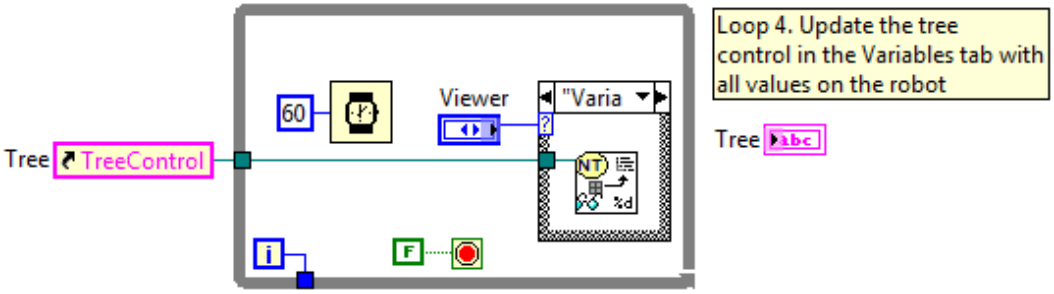


FIGURE 8.11 Loop 4, which updates the Variable tab and lets you see all the data you are passing to and from the robot in one place.

Loop 5: Kinect Data and Display

Here is where I confess that I have never programmed the Kinect. If you plan to use the Kinect, Loop 5 is where its data gets read. If you don't plan to use the Kinect, this loop can go.

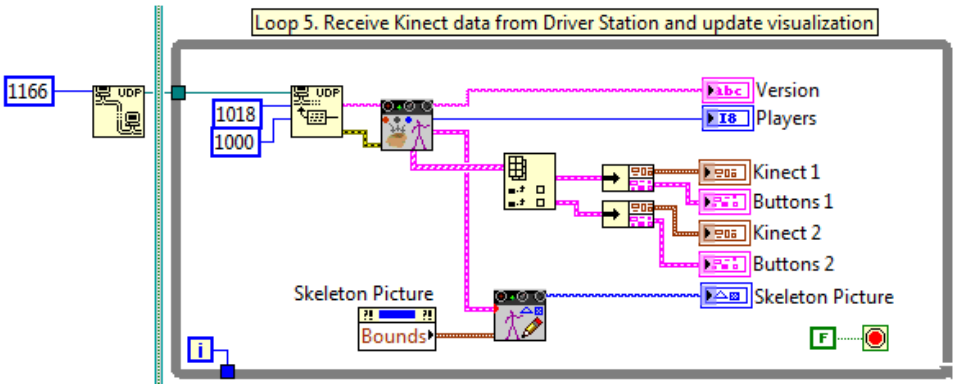


FIGURE 8.12 Loop 5, the Kinect.

Loop 7: Window Size and Position

Probably you notice that I skipped Loop 6. The reason for that will become apparent shortly. Loop 7 (Fig. 8.13) controls the size and position of the Dashboard window. Unless you have a very old laptop, it's screen is much bigger than the default Dashboard size. It would be a shame to waste all that real estate, which you could be using for bigger, more informative displays. If you modify the code to match Fig. 8.14, then your Dashboard will fill your laptop screen. Part of the front panel will be unusable because it's behind the Driver Station, but still, it's an improvement.

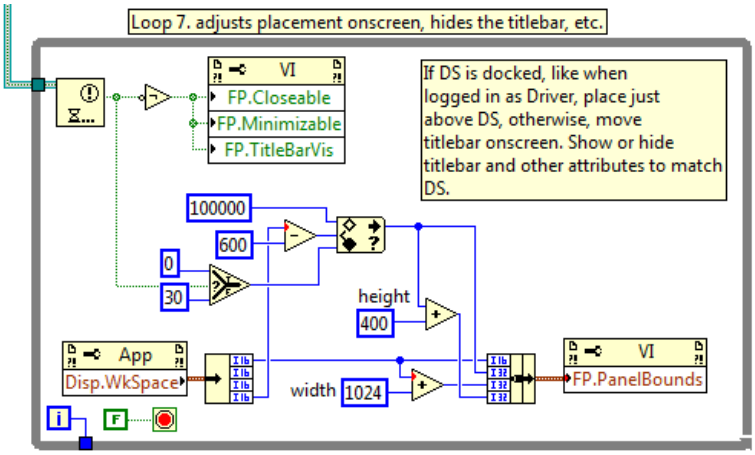


FIGURE 8.13 Loop 7.

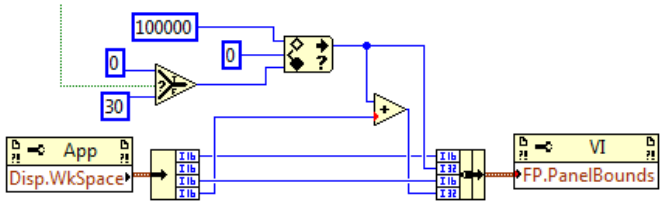


FIGURE 8.14 Modification to make a full screen Dashboard.

Loop 6: A Whole Lotta Stuff

There is so much going on in Loop 6 that it difficult to know where to begin. Let's try starting with the 30,000 foot view: Loop 6 is all about handling the user interface to the Dashboard. Let's face it: during a competition match, you absolutely will not be using a mouse to

click around the controls on the dashboard. So if all you have in mind is building a Dashboard for competition, you can just go ahead and delete this loop and all the code at the bottom of the diagram that feeds it.

On the other hand, there's some great stuff in here that will teach you a lot of LabVIEW if you have the patience to work through it. You might also find much of it useful in the building and debugging phase, and could adopt some or all of it for a custom debugging dashboard of your own. Or you could just use the default Dashboard as it is, and not worry about any of this. This last option is probably the least useful, as you won't be able to use all the features of the default Dashboard unless you learn something about how it works.

The Event Structure

Fig. 8.15 shows our first look at Loop 6 (we will need to take a lot of looks at this loop…). You can see that inside the While Loop is a rectangular structure with diagonally striped border. This is an Event Structure. It is a very close cousin to the Case Structure, with which you are already very familiar. A Case Structure has multiple frames, and a particular frame is selected by the data that is wired to the selector terminal. An Event Structure also has multiple frames, but no selector. Instead, an individual frame is selected for execution by things that happen, or "events." Possible events are the user clicking on a control, or the value of a variable changing, for example (the actual number of different events to choose from is quite large). The event that corresponds to the frame is shown in the bar at the top. Just as in a Case Structure, you can click the little down arrow to get a list of all the events in the structure. You will see, for example, "Camera Settings": Value Change, which indicates the particular frame that will execute of the value of the variable Camera Settings changes.

As you can see, there are 13 separate events, most of which are related to the (new in 2013) Test Mode. We will continue to take a high level view here: we'll look at what is going on in the structure and explain how to use it, but not go through everything in detail. Actually figuring out how this code works will be a good exercise in self-directed learning.

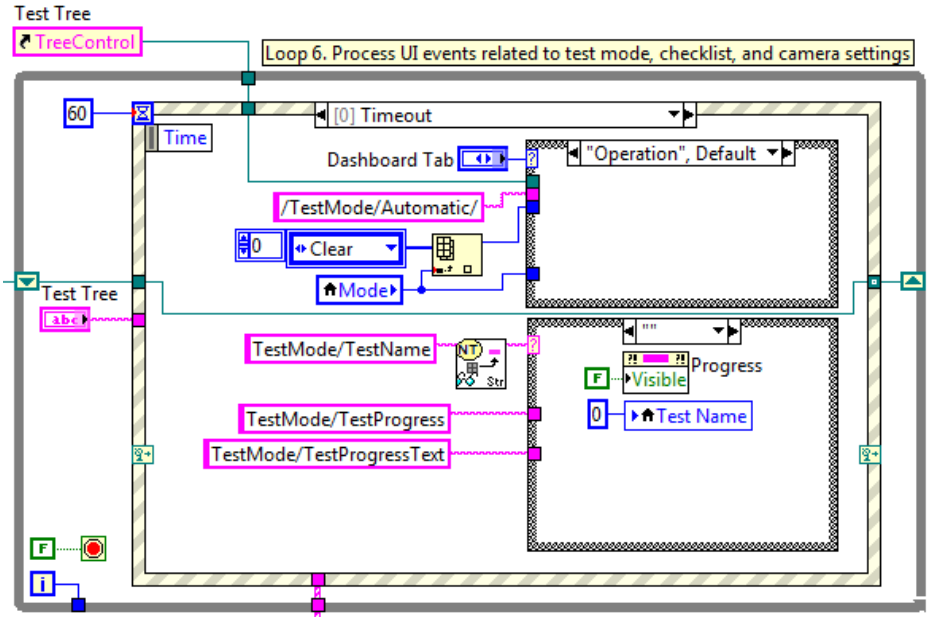


FIGURE 8.15 The Timeout event in Loop 6.

Event 0: Timeout

Timeout is the event that happens when no other events are happening. It happens every 60 ms, since that is the interval wired to the timeout terminal in the upper left corner of the structure. If the robot is not in Test mode, then this event doesn't do much.

Events 6 and 7: Camera Settings

As already discussed, the default Dashboard has indicators that show you the frame rate and bandwidth usage for your camera. You change them through a nifty little pop-up control. These two frames manage that control, and—as a bonus—save your settings to disk when you are done. On Windows 7, the saved settings are written to the text file "Public Documents\FRC\DashboardSettings.ini" (the Windows XP equivalent is somewhere under "Documents and Settings"). When the Dashboard first starts up, this file is read, and the camera settings are read from this file and applied to the camera, using code that is below and to the left of Loop 6 in the diagram. For competition, you might not want to allow the accidental changing of the camera settings via the Dashboard. In that case, you could hard-wire the settings you want, as in Fig. 7.5, having first used the default Dashboard to figure out what they are.

Events 4 and 5: The Checklist

If you've been involved in competitions already, you probably know that it's a good idea to have a pre-match checklist to help you remember all the critical items (freshly charged battery? drive train tight? drive station switches for autonomous match your start position?). The right hand tab control on the Dashboard has a checklist feature. The list is stored in the text file "Public Documents\FRC\Checklist.ini". Simply edit this file to replace the example list with your own items. The example list also explains how to check individual items and how to clear all the check marks at once. There is also an un-documented method that allows you to un-check (or check) an individual item in the list. See if you can figure out what it is from the code.

Test Mode (Events 0, 1, 2, 3, 8, 9, 10, 11, 12, and 13)

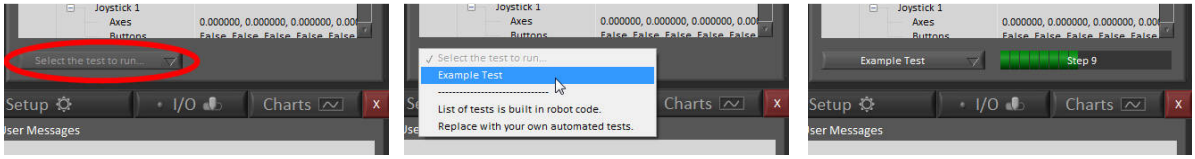
In 2013 a new mode, Test, was added to the FRC robot framework. The idea behind this mode is to provide you with quick way to verify the operation of your robot. You can do this either by directly reading and controlling robot components, or by writing specific tests of your own design. For example, you could create a test that runs the left drive motor at 75% power for half a second, and then does the same for the right drive motor. Or your test could close and then open a gripper.

Test mode works like Autonomous: it is an independent VI that is called by reference (as you can observe in Fig. 4.1). You launch this VI by pressing the Test mode on the Drive Station's Operations tab (see Fig. 4.20). Like Autonomous, Test mode runs inside a While Loop (there are actually two in parallel, as we'll see) that keeps it running, since it is not driven by the main Robot Framework state machine.

We'll start with the do-it-yourself style of tests. The default Dashboard comes with an example of this kind of test which is a good place to start because it uses no hardware. You can run it on your cRIO without anything robot-like attached.

Switch the Dashboard to the Test tab. If your "robot" (which could be no more than a cRIO for now) is in Test mode, you will see a tree full of information (which you can ignore for the moment). If you are not in test mode, there will be an informative message. Down below this tree of data is a drop-down menu (Fig. 8.16a). Press that and a drop down list of all your pre-programmed tests. If you are just running the default code, then the list will look like Fig. 8. 16b, since the only available test is the provided example. Select that test and it will run immediately, giving you a progress bar, as in Fig. 8. 16c.

To see what is going on and to understand how to create more meaningful tests, look at the code that is running the example, shown in Fig. 8.17. Inside the While Loop is a case structure. When you make a selection from the drop-down list on the Dashboard, you are selecting one of the cases in this structure. Your complete test code has to be contained in a single



(a) (b) (c)

FIGURE 8.16 Running a programmed test.

case and has to run to completion in one shot, as this is not a state machine (although you can get around this and turn it into a state machine. See below). The code in Example Test does nothing but update the progress bar (Fig. 8. 16c). The For Loop executes 20 times, causing the numbers 0, 5, 10… to be written in sequence to the network table variable named "TestMode/TestProgress". The Dashboard reads this value from the table and updates the progress bar. At the same time, a sequence of strings is created, "Step 0", "Step 1", etc. and written to the network table variable "TestMode/TestProgressText". The Dashboard reads this text variable and writes that text as an overlay on top of the progress bar. (If you are interested, the Dashboard is using the Timeout Event (Event 0) of Loop 6 to do this updating.)

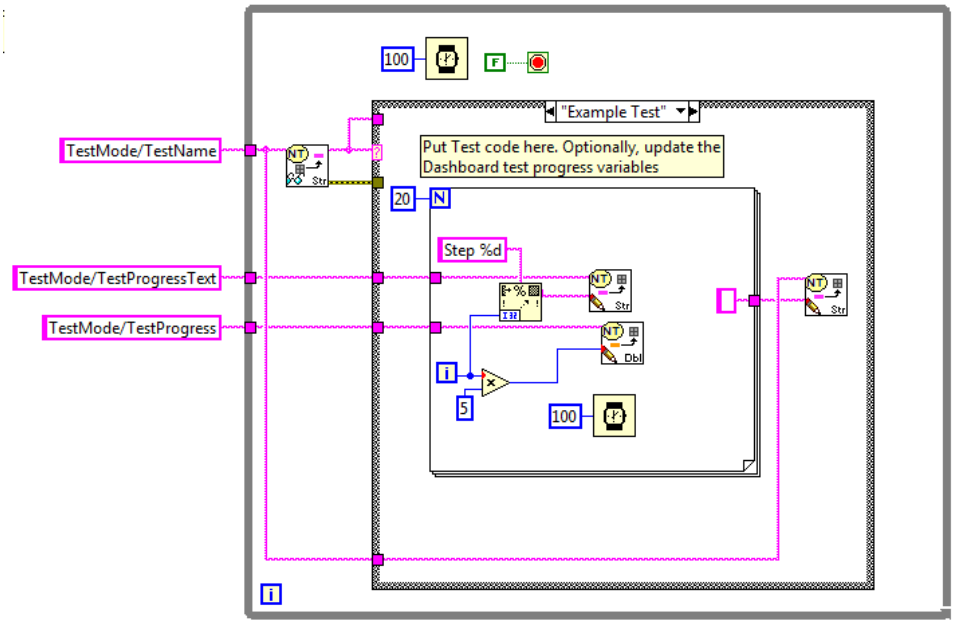


FIGURE 8.17 The code for the example test.

Note that when the For Loop completes, an empty text string is written to the network table string variable named "TestMode/TestName". *This is very important.* When your test completes, the next iteration of the For Loop will select whatever test name is sitting in "TestMode/TestName". If you forgot to update this variable, then your test name will still be there, and the Test program will go into an infinite loop. (I would have written this code differently: I would have put that network table VI *outside* the case structure. Then, when you add a new case for a new test, you automatically get an incompletely wired tunnel and a broken arrow, which would force you to remember that you have to wire something to control the program flow.)

Note that you are not *required* to use an empty string here. You could create a test that used multiple cases in the main case structure. The first case in your test could put the name of the second case in there, and so on, causing the program to work through the steps of your test in sequence. You just need to make sure that the last case writes that empty string.

Just so you have an example of a test that does something, Fig. 8.18 shows a new test (created by duplicating the Example Test case) that actuates a solenoid, first one way, and then the other. You can see that I've modified the flow control for the entire Test VI in the way I suggested. As written, the (imagined) gripper will close, the progress bar will update. When it reaches 100% and the test is over, the gripper will open. You could easily figure out how to write code that will open the gripper when the progress bar reaches 50%, if that suits you better.

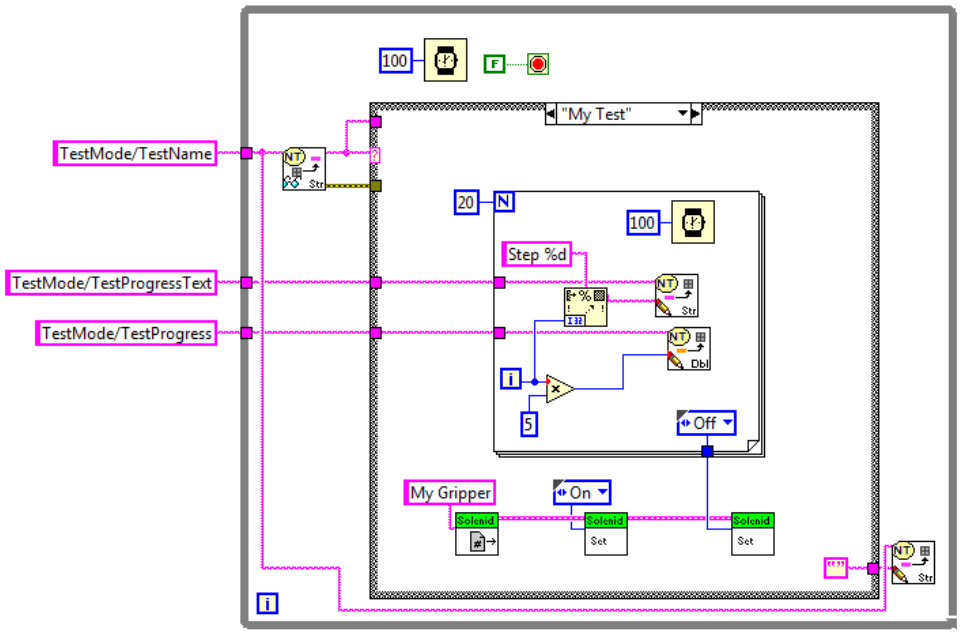


FIGURE 8.18 Another example.

Of course, you can't run your test if it doesn't show up in that drop-down list on the Dashboard. To make that happen, look at the diagram for Test.vi, above the main loop. Put your test names in this string array (Fig. 8.19) and they will appear on the Dashboard list so you can run them.

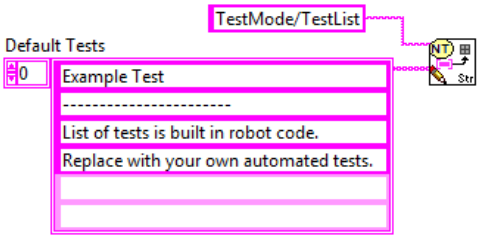


FIGURE 8.19 The list of tests. One test per line.

Finally, we return to that tree of data that takes up most of the Test tab. Test mode includes a secondary loop, which I'm sure you have noticed by now, that runs a VI labeled "Auto Publish". This VI searches through all the devices you registered in Begin.vi, reads their value (if they are a sensor or a joystick) and assembles the results (along with some other bits of information) into a tree display. In Test mode, you can mess with your joystick controls and watch the values change before your eyes. (Hint: to read values that extend beyond the tree display window, hover the mouse over the values *and keep it moving* while you test the joystick). You could also test that a limit switch is working by (safely!!) reaching into the robot an operating the switch. The corresponding digital input in the tree should change value if the switch is working.

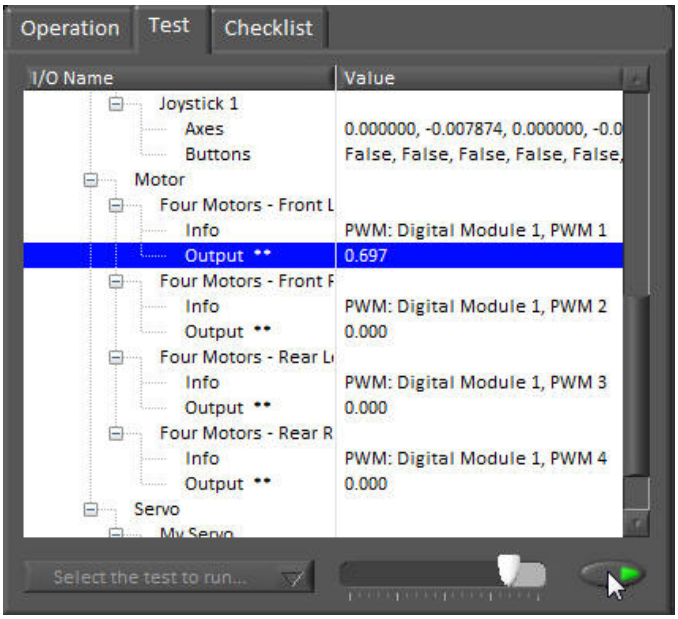


FIGURE 8.20 Test mode being used to operate the left front motor in a four motor drive system.

You can also control your actuator outputs from this tree. Things that you can control in this fashion show up with a "**" after them. So, for example, you can operate a motor, as in Fig. 8.20 by clicking on the output line. A slider and a button magically appear. Set the slider to the value you want, and press the button. The motor will run at that power as long as the button is pressed. If you chose a different actuator (solenoid, digital output, servo motor), you will be presented with a suitable control on the Dashboard to allow you to operate it in an appropriate fashion. Note that while you gave a single name to your drive system in Begin.vi, the motors appear here as separate items so you can test them one at a timet. (If

you really want a driving test, you could code that yourself as one of your programmed tests, but be careful!!) You can even use this tree to mess up the camera parameters you carefully set while optimizing your image capture and processing…

Dashboard Strategies

If all you want is a competition Dashboard that shows you key information, you need only Loops 1 and 2 out of the original seven. If you are not doing any image processing, you don't even need Loop 2 (although that would be sad). On the other hand, Test mode could be really handy in the pit… What to do?

The Dashboard has introduced you to the Tab Control, something we've not paid any attention to because it's a user interface feature, and before the Dashboard, we had no user interface to program. But as you can see, it is a convenient way to stack different front panel interfaces on top of each other, allowing them to share the same piece of screen real estate. Hmmm…

Building Your Dashboard

You can run your Dashboard by pressing the white arrow, provided you've stopped the default Dashboard that pops up when you start the Driver Station. But what you'd really like to do is have *your* Dashboard pop up and launch automatically when the Driver Station starts. The default Dashboard is an executable file located (along with a couple of related files) in the folder "C:\Program Files\FRC Dashboard". Make a sub-folder in there called "Original" or something equally original, and drag the executable and the other files in there. That saves the default so that you can drag it back out if you are having trouble with yours and you really need something that works *now, now, now!*

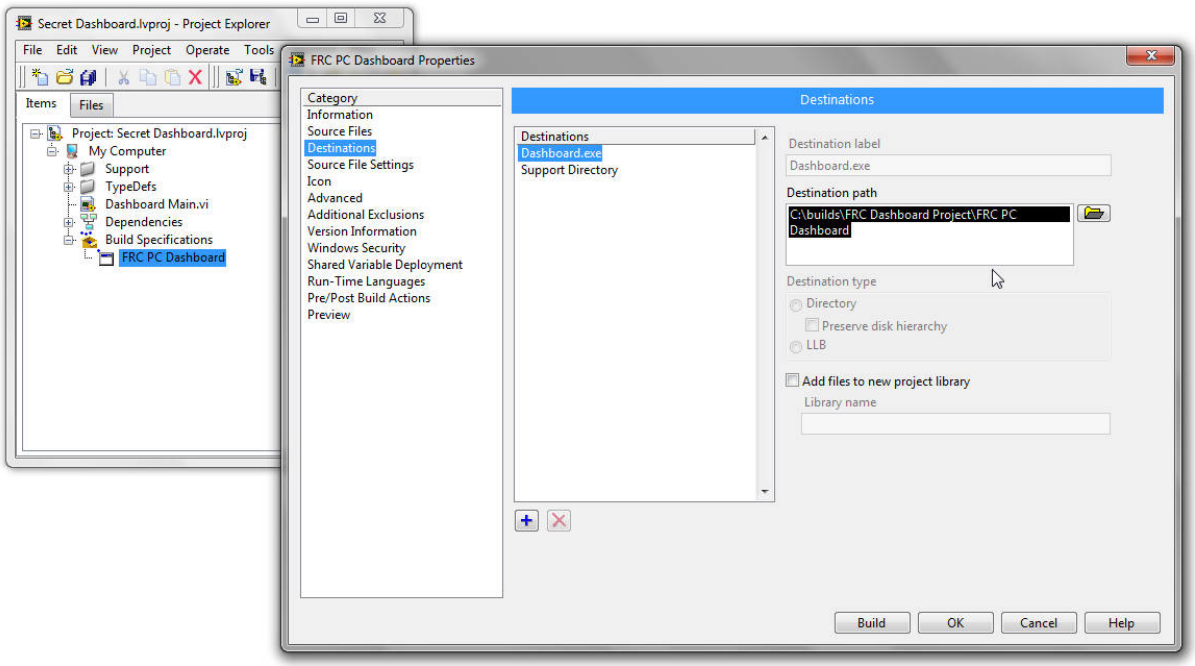


FIGURE 8.21 Changing the default directory for Dashboard.exe.

Now go to the Dashboard Project Tree, expand "Build Specifications", right-click on "FRC PC Dashboard", and select properties. Select "Destinations" in the left hand column and you

will get the view shown in Fig. 8.21. Change the Destination Path for Dashboard.exe to "C:\Program Files\FRC Dashboard". Then click on "Support Directory" and make that directory the same as for Dashboard.exe. Click the OK button and save the project file. Now, you can right-click "FRC PC Dashboard" and select Build, and an executable file will be created and automatically launched with your Driver Station.

That's it. Go bring home a blue banner!