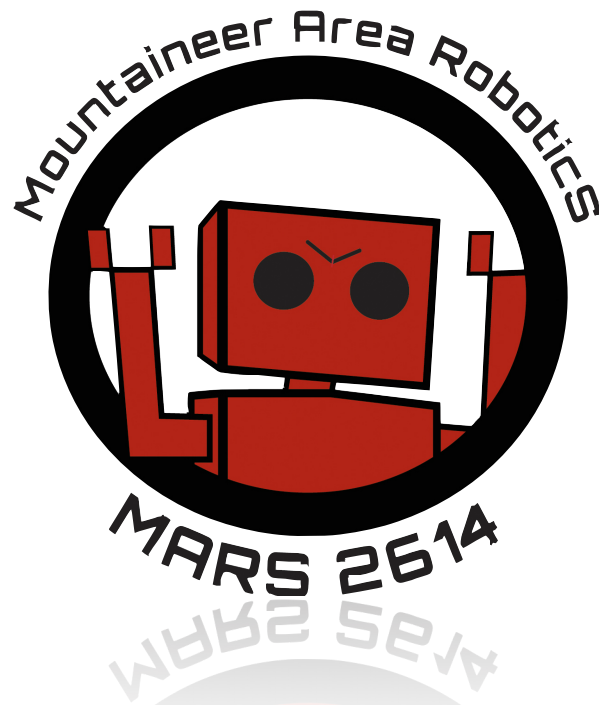


# Programming Handbook



# Programming Handbook

Introduction	3
FRC VIs and LabVIEW tricks	4
Structures	7
Global VIs	9
Architecture conventions	12

# Introduction

My name is Jonathan. Between beta testing, **Sensobot**, schedule difficulties that often prevented everyone from being at practice on the same days, and software incompatibilities **\*cough\*** *Cameron's crappy laptop* **\*cough\*** *dirty Mac users* **\*cough\***, I am a bit behind in terms of teaching programming. I think that a manual would be a handy way for people to figure out the MARS way of programming, even if I can't be there to answer a specific question. The key to successful programming is a firm grasp of two important concepts:

## 1. Robots are dumb as hell.

No matter how fast a robot's processor is, and how expensive all its sensors are, the robot doesn't have a brain. Robots cannot analyze an unexpected situation and figure out what to do. The robot will execute its code the same way every time it is run. At its core, a robot only *"thinks"* in terms of data. To get robots to do what we want, we need to write extremely specific code. Robots cannot "fill in the gaps". When a human is asked to throw a Frisbee, the human can be safely assumed to know that he must extend his or her arm, flick his or her wrist, and release his or her grip on the Frisbee. Unless the code is already written, for an arm robot, one would have to specify those three actions in a sequence (not concurrent, or else the Frisbee will be dropped) for the code to work. We thus need to closely examine what specific actions that performing a task entails. Also, robots cannot fix mistakes made in the code made by the programmer. If an extra zero is added to the distance that a robot must travel in a "Go to supermarket" method, you may find your robot in Pennsylvania. People would look for visual cues and can tell when Google Maps incorrectly calculated the distance to the supermarket but (unless the robot was programmed to look for visual cues) a robot would be completely fooled because it DOESN'T HAVE A BRAIN.

On the plus side, wouldn't it be hilarious to get a robot to look for a belt stretcher or a dip loop? Given enough battery power, it may literally search forever.

## 2. Other programmers are not geniuses.

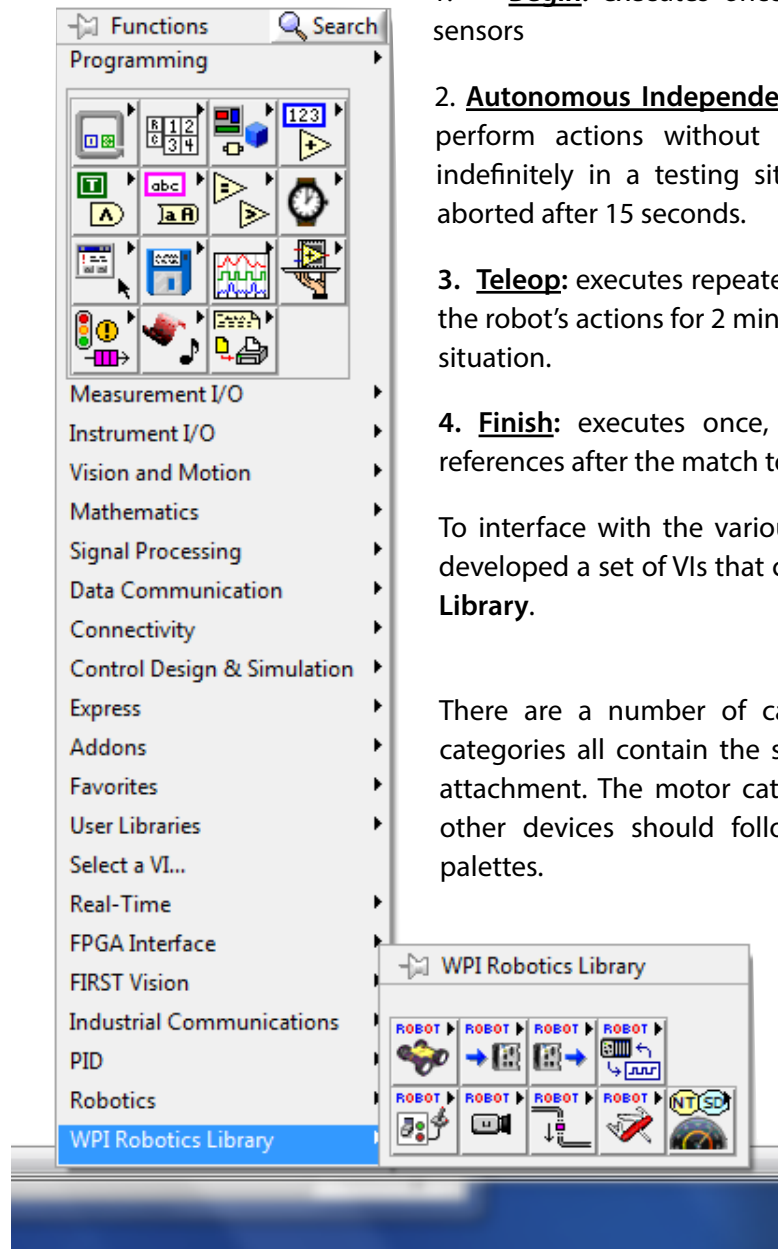
There is no question programmers are smarter than robots but they are not necessarily geniuses or telepaths. Code needs to be written and annotated in a way that makes sense to other programmers whom may be correcting the code if it does not work or needs to be modified to work with newer code. In addition, there are a number of steps that can be taken to limit creation of software errors as the code is being written. One misspelled device reference name can mean a total loss of functionality of a subsystem (see concept #1 for an explanation why).

The consequences for ignoring concept #2 are severe, and may include mascot duty at regionals and/or the eternal wrath of Steve. If I missed anything that should be covered, email me at [jglister96@gmail.com](mailto:jglister96@gmail.com).

# FRC VIs and LabVIEW Tricks

The guys at FRC designed a baseline set of code that allows the robot to drive and respond to commands from the Field Management Software (FMS) and from the Driver Station. The FMS runs the match by sending cues to the robot that specify which stage the match is in, giving the robot autonomous control for 15 seconds, allowing teleoperated control for two minutes, and stopping the robots at the end.

There are **four** key VIs that are executed in sequence during a match:

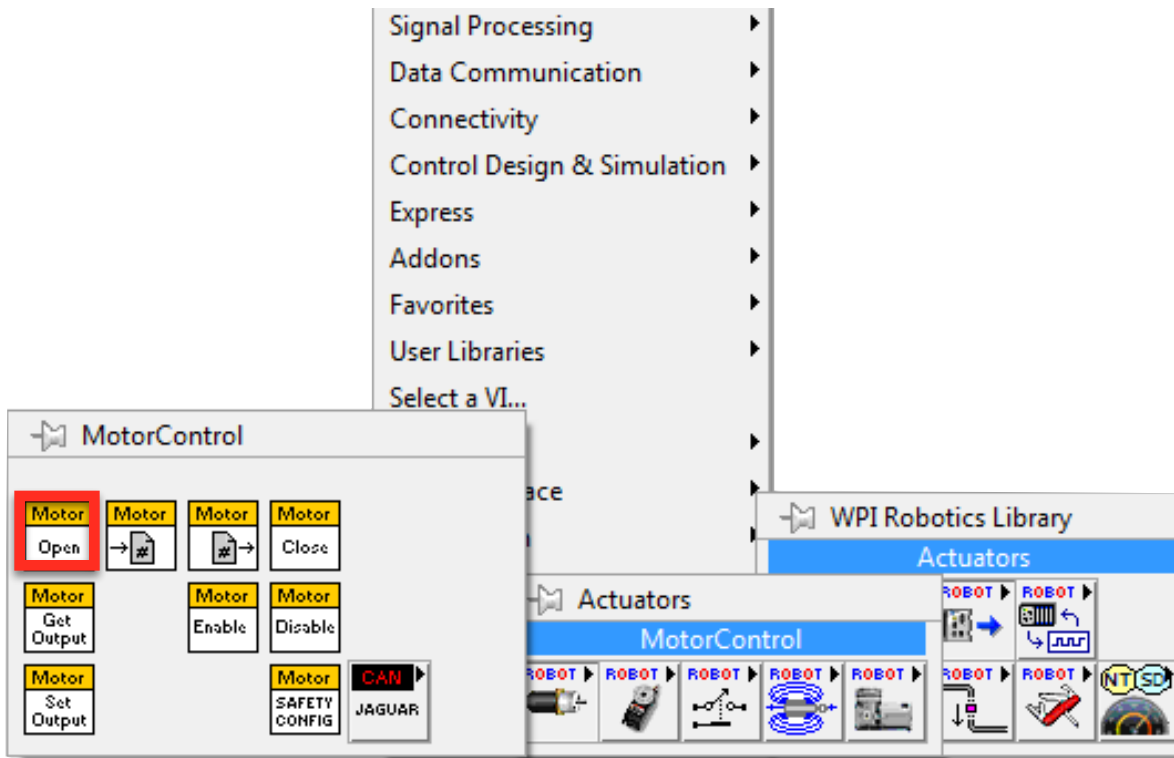


1. **Begin:** executes once, initializes all motors, actuators, and sensors
2. **Autonomous Independent:** executes once, robot is allowed to perform actions without driver input. The program can run indefinitely in a testing situation but during a real match it is aborted after 15 seconds.
3. **Teleop:** executes repeatedly within a while loop, drivers control the robot's actions for 2 minutes in a match, indefinitely in a testing situation.
4. **Finish:** executes once, deletes motor, actuator, and sensor references after the match to avoid memory leak.

To interface with the various attachments on the robot, FRC has developed a set of VIs that can be accessed from the **WPI Robotics Library**.

There are a number of categories and subcategories, but the categories all contain the same general functions for each robot attachment. The motor category will be used as an example, all other devices should follow the same pattern of VIs in their palettes.

In LabVIEW, each motor, sensor, or other actuator is represented as a cluster of data called a device reference (analogous to an object in Java or related languages.) In the **Begin** VI, device references are created using Open VIs.



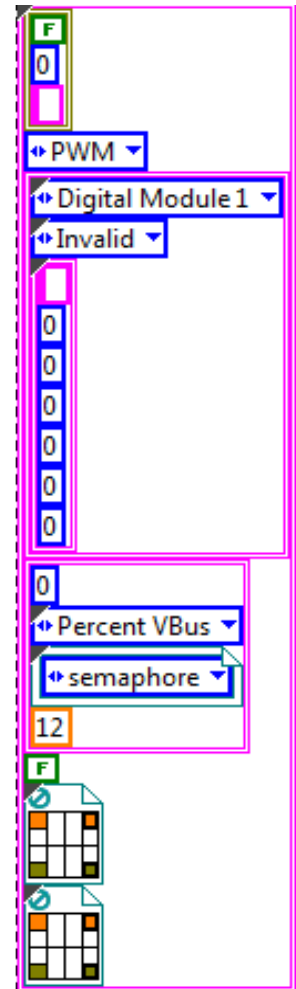
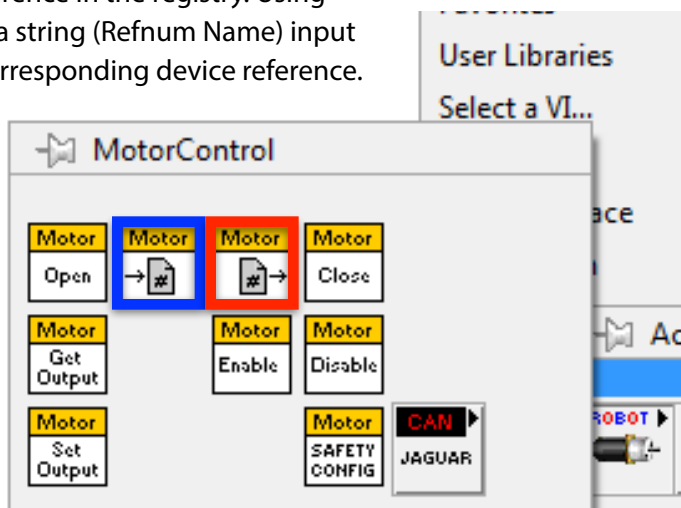
An Open VI takes a number of inputs (usually one or more slots, such as PWM outputs, through which the robot can either send or read data) and outputs a device reference using that information. However, upon looking at the data contained within the device reference, you will realize it's a bunch of incomprehensible crap, as seen to the right.

To improve readability, the device references can be compiled into a registry and identified by string names. (Recall that a "string" is a series of characters.) To do this, FRC made the Refnum Set and Refnum Get VIs.

**Refnum Set** takes a device reference and a string (Refnum Name) as inputs and sets the device reference in the registry. Using

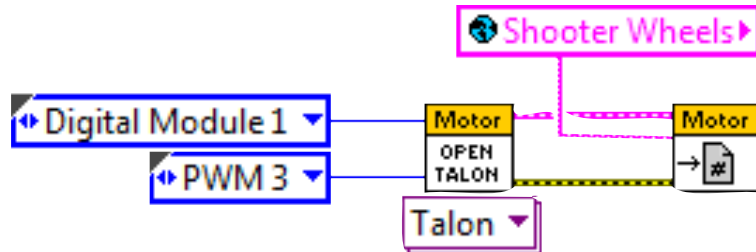
**Refnum Get** takes a string (Refnum Name) input and outputs the corresponding device reference.

Device references are set in Begin and they are used in Autonomous, Teleop, and Finish.



# Examples

## Proper use of Refnum Set:



## Proper use of Refnum Get:

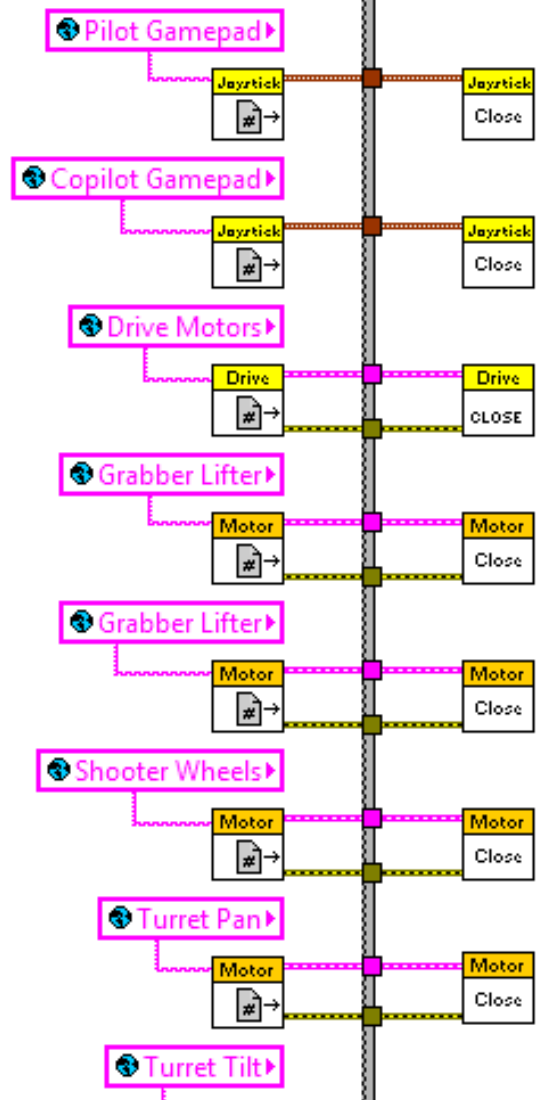


Yes, those strings do look funny. They're globals, but they're still strings like any other. We have our refnum names in a global file for neatness and consistency.

What to do with the device now that we can access it varies. We may be sending a certain power level to motors, or reading a sensor, or actuating a pneumatic, depending on the device we accessed.

Specifying the device reference with the Open and Refnum Set VIs would be declaring and instantiating the object in Java. The other VIs in the palette are analogous to methods in a Java class (example: the Set Output VI in the Motor Control palette can be thought of as an outputSet method in the class Motor.)

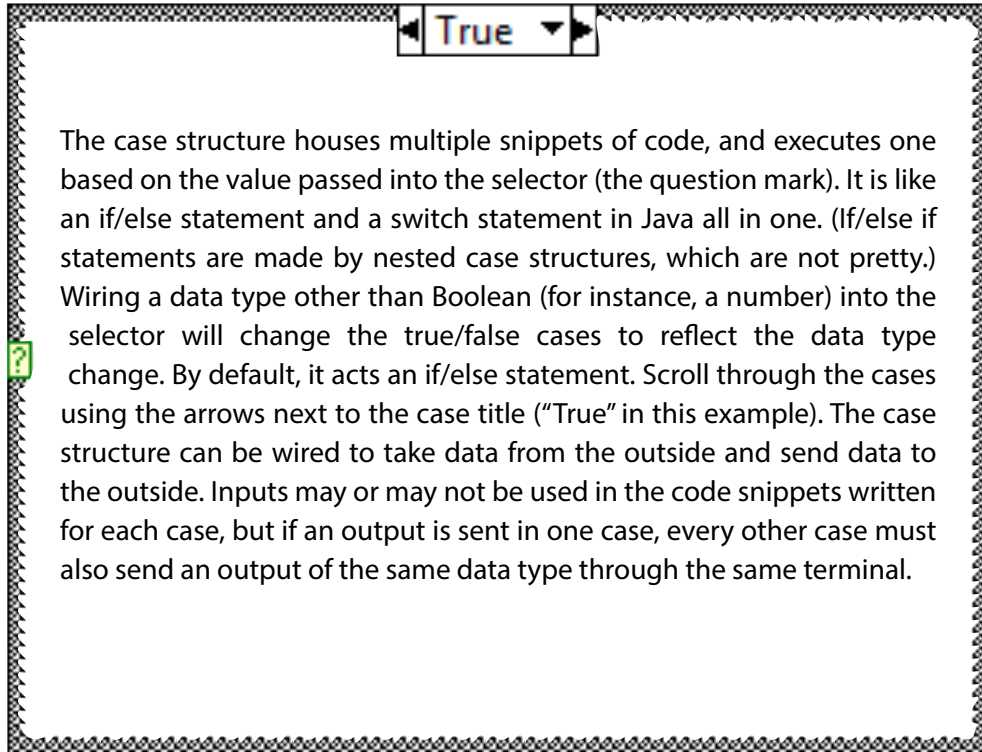
In the **Finish** VI close device references like in the picture to the right:



# Structures

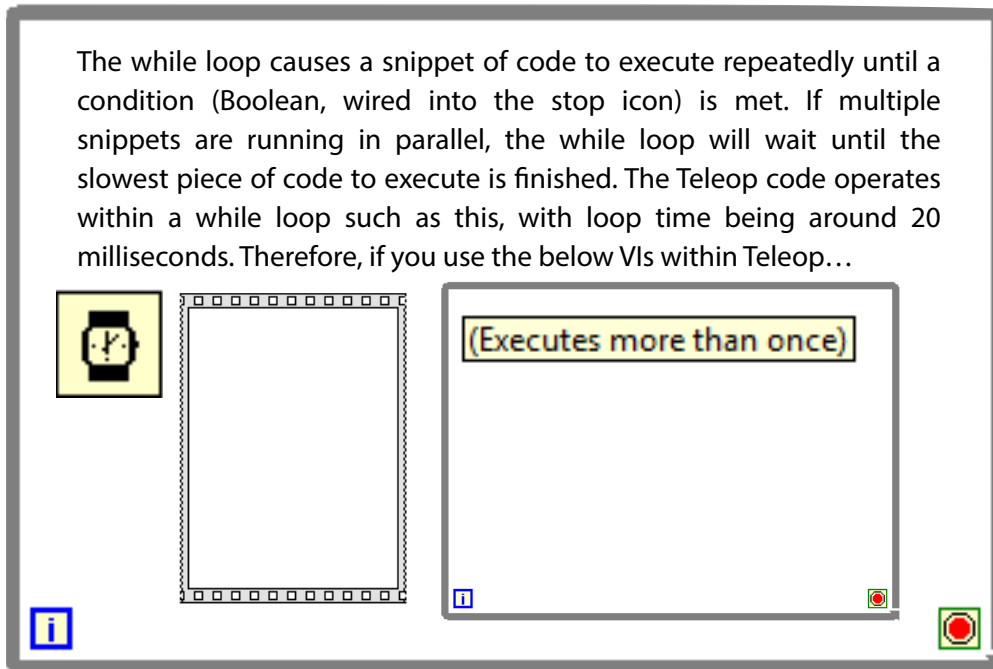
There are some other VIs and structures of note.

## Case Structure



The case structure houses multiple snippets of code, and executes one based on the value passed into the selector (the question mark). It is like an if/else statement and a switch statement in Java all in one. (If/else if statements are made by nested case structures, which are not pretty.) Wiring a data type other than Boolean (for instance, a number) into the selector will change the true/false cases to reflect the data type change. By default, it acts an if/else statement. Scroll through the cases using the arrows next to the case title ("True" in this example). The case structure can be wired to take data from the outside and send data to the outside. Inputs may or may not be used in the code snippets written for each case, but if an output is sent in one case, every other case must also send an output of the same data type through the same terminal.

## While Loop

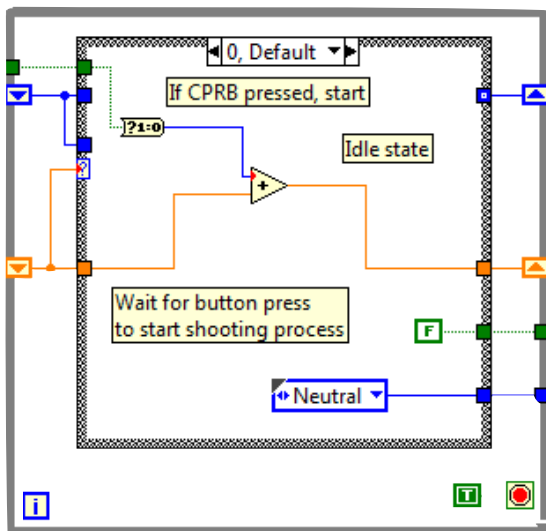


The while loop causes a snippet of code to execute repeatedly until a condition (Boolean, wired into the stop icon) is met. If multiple snippets are running in parallel, the while loop will wait until the slowest piece of code to execute is finished. The Teleop code operates within a while loop such as this, with loop time being around 20 milliseconds. Therefore, if you use the below VIs within Teleop...

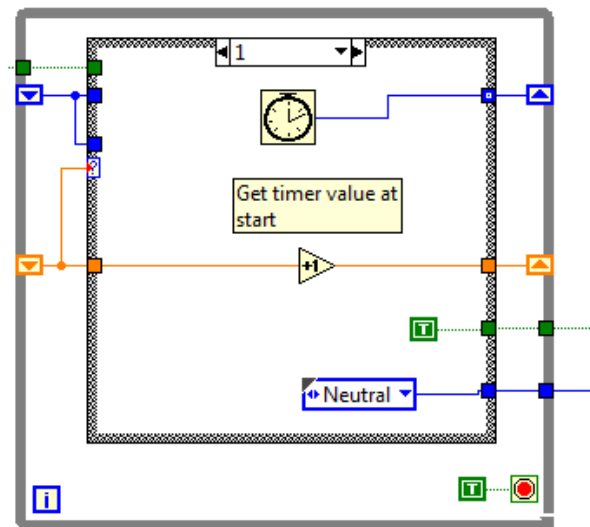
... you will cause lag in the drive code and not only will Matt and (probably) Wyatt, kill you, but Steve will kill you again. I have seen a lot of teams do this in their code at competition.

A single execution while loop is fine, however. If a "True" is sent to the loop condition, the code will execute once and stop. (Technically, a LabVIEW while loop is more like a Java do... while loop.) The while loop would normally be redundant, but while loops can hold shift registers. Shift registers (in red circles, below) do the same thing as feedback nodes (send the value from last cycle) but look neater, especially when multiple feedback nodes would be needed. Note that the down arrow on the left corresponds to the up arrow directly across from it.

Some processes take longer than a loop cycle to complete. (For instance, our shooting cycle takes about a second.) Therefore, we must spread the process across multiple loop cycles. A method of accomplishing this is using a single-execution loop, a case structure, and a timer (tick count) VI. Timers do not slow the loop cycle, instead they output the time elapsed in milliseconds since the robot started. (The tick count VI does not define a time 0 in fact it resets after  $2^{32}$  milliseconds, but this is over 1000 hours, so it is not a concern for FIRST robotics.)

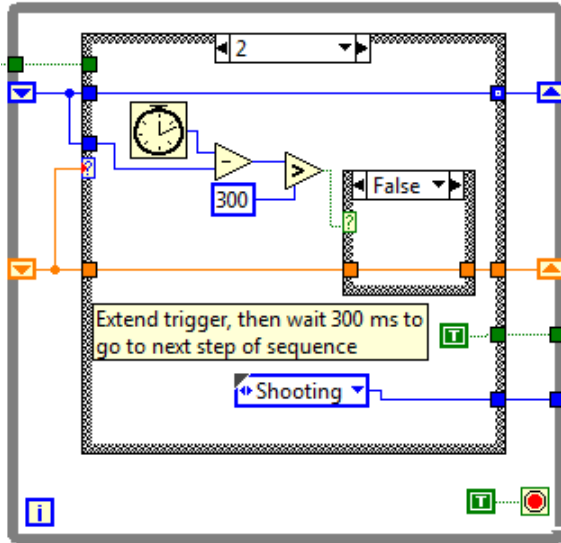


In **case 0**, the robot is idle. The while loop has a shift register for the value that is fed to the selector on the case structure. The logic that determines whether the case switches is held within the case structure itself. It increments the value if the copilot right bumper is pressed.



In **case 1**, the timer value is recorded, and the case structure automatically increments.





In **case 2**, the timer value recorded in case 1 is maintained on the shift register and compared with the current time. When it exceeds 300 (to give the trigger pneumatic time to slide out), 1 is added to the value fed into the selector.

In the last case (not shown), when the completion condition occurs, the selector value is set to 0 and the process can restart.

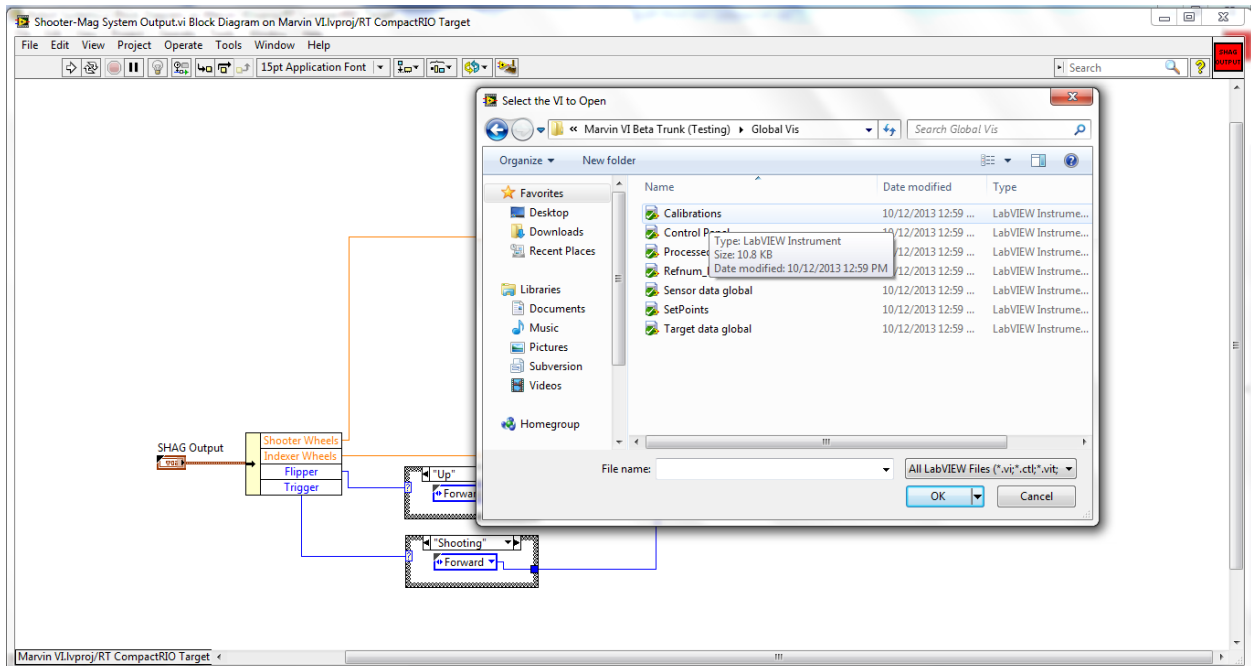
The robot is not waiting for certain conditions to occur that would slow down the loop. It is instead checking if a certain condition is true or false at every loop cycle, which does not slow the loop. This allows for timing and sequencing processes without angering the drive team with lag.

## Global VIs

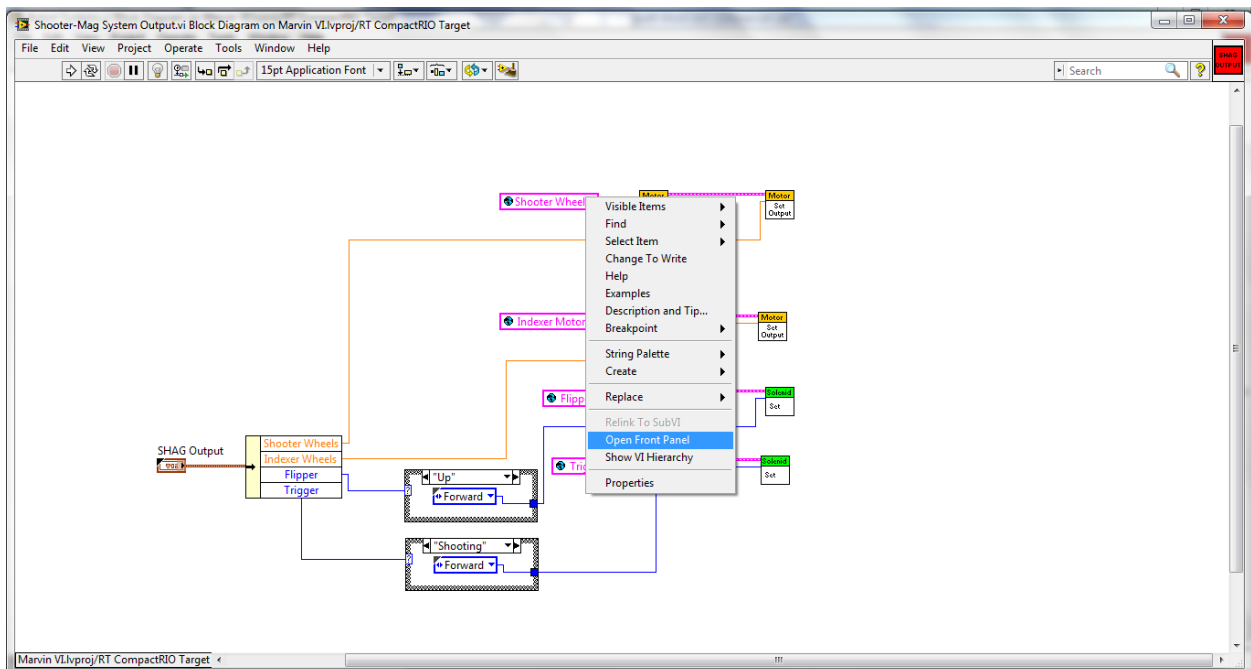
A global VI is a global file located in the project that houses one or more global variables. They have front panels, and can be saved within the project to be re-used elsewhere. They are accessed from the structures menu. We use them for refnum names and constants that need to be tuned (“Calibrations”, as we call them.) Why? There are two reasons for this:

1. Global variables are represented as drop-down lists, which makes it convenient to access variables by name, and eliminates the risk of someone mistyping a string name. Recall that robots are dumb as hell, and they cannot correct the string “Shoter Motor” to “Shooter Motor”. They also do not realize that “drive motors” is the same as “Drive Motors”. Using a global variable in this way ensures consistency throughout the code.

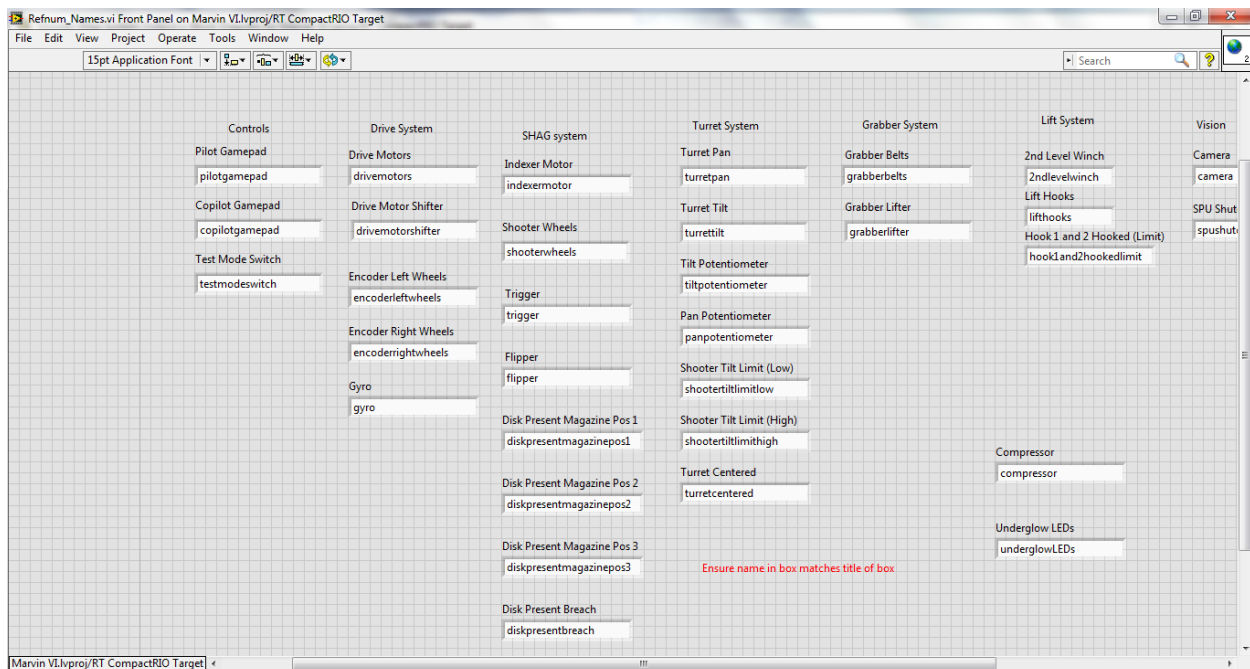
2. Changing the value of a variable in the global file will change the value everywhere in the project, no matter how many times it is used. This frees up time, as we (for example) would only have to change the shooter wheel speed once instead of five times.



Opening a global file



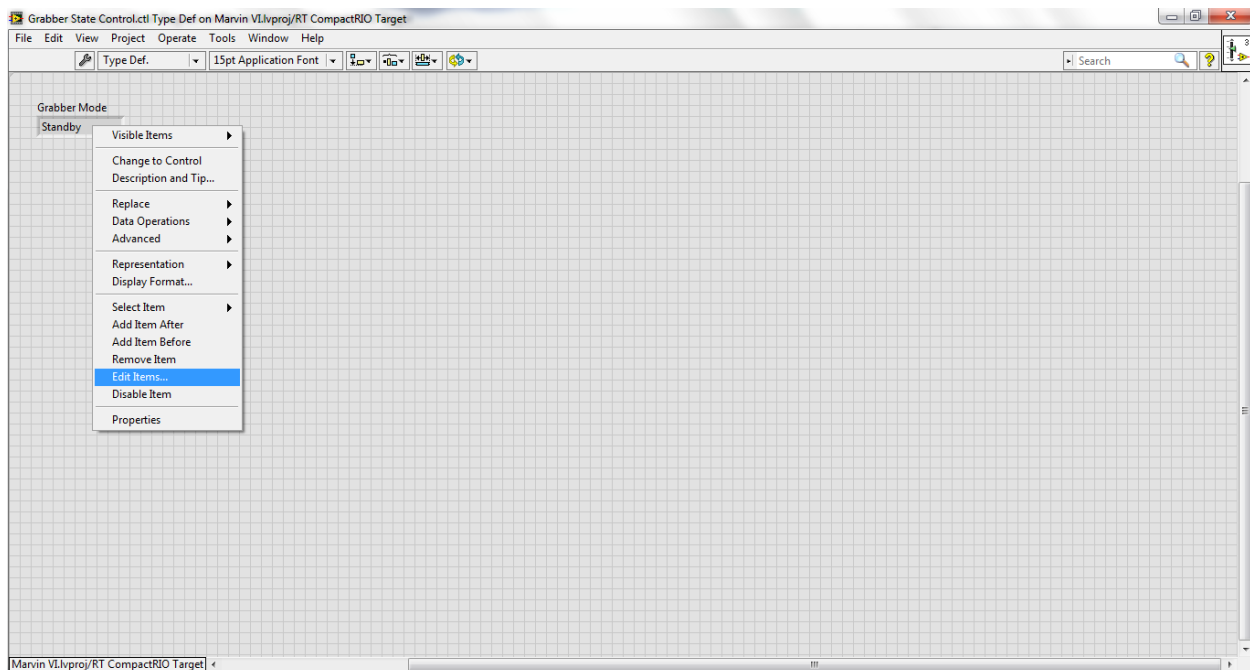
Accessing a global file



### Data contained within

For mode and state names, we use enumerated types, saved as typedefs, which follow the same principle. To create an enumerated type, select Enum constant, click Make Type Def, then Open Type Def.

### How to edit the values:

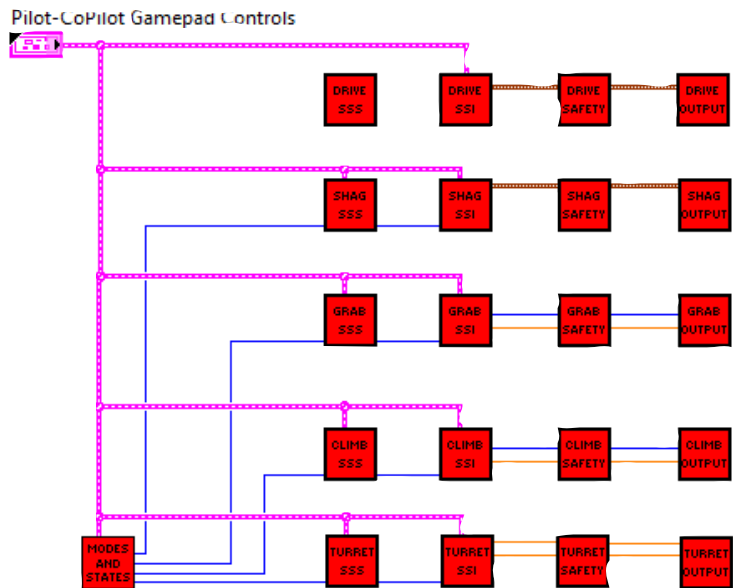


### Edit Items

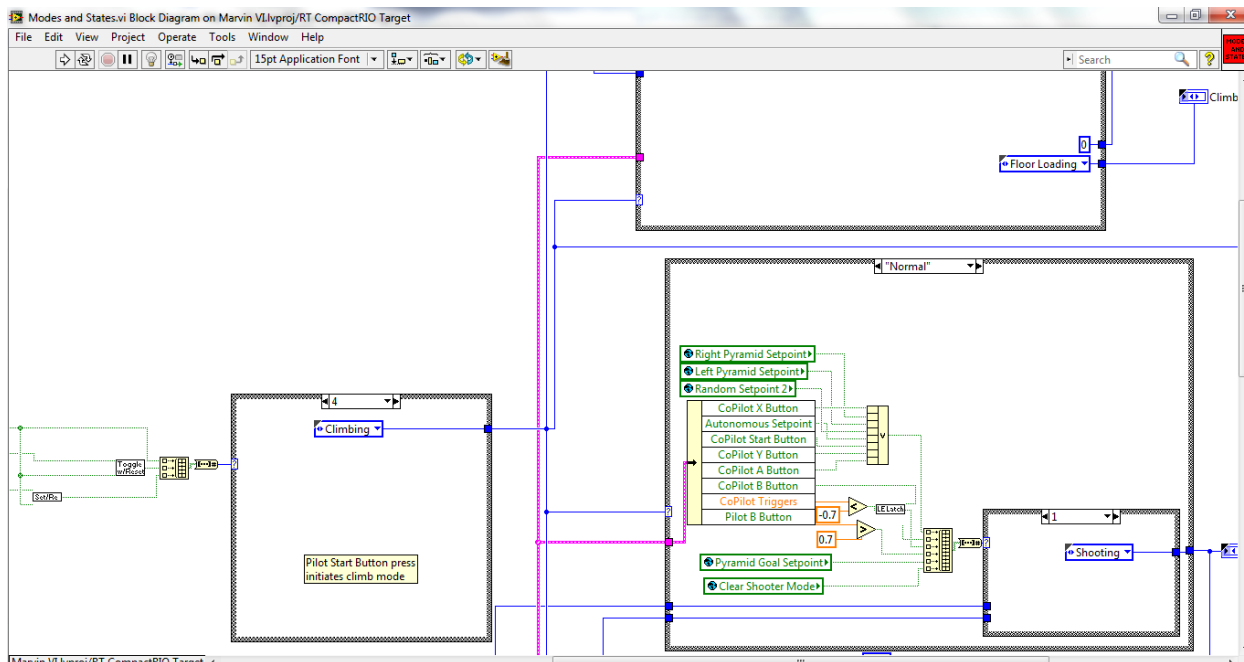
Then add values for every state you will need and assign each numerical value a string name. Be sure to put the default mode in as item 0 or the robot will boot in another mode. (Ensure subsystems have "Standby" or "Normal" as item 0 as well, for the same reason.) Like with global variables, save them within the project.

# Architecture Conventions

We touched on some basic architectural concepts already (such as using global VIs and enumerated types) but the way the robot operates follows a certain pattern. Every Marvin is a state machine its behavior depends on the state it's in. The states dictating overall robot behavior are called robot modes, and the states the govern subsystem behavior are called subsystem states. These modes and states are set and switched by the pilot and copilot pressing buttons on the joystick or control board. The mode governs which subsystem states are allowed to be selected by the pilot and copilot. The subsystem states are then sent to the SSI (subsystem state implementor) VIs, which take a state input and joystick axes and buttons and output numbers and other data types. The outputs from the SSI VIs are passed through Safety, where the robot checks to see if it's safe to pass said values to the Output VIs, where the data is actually translated into robot motion. But usually there is no safety check and the values go straight through.

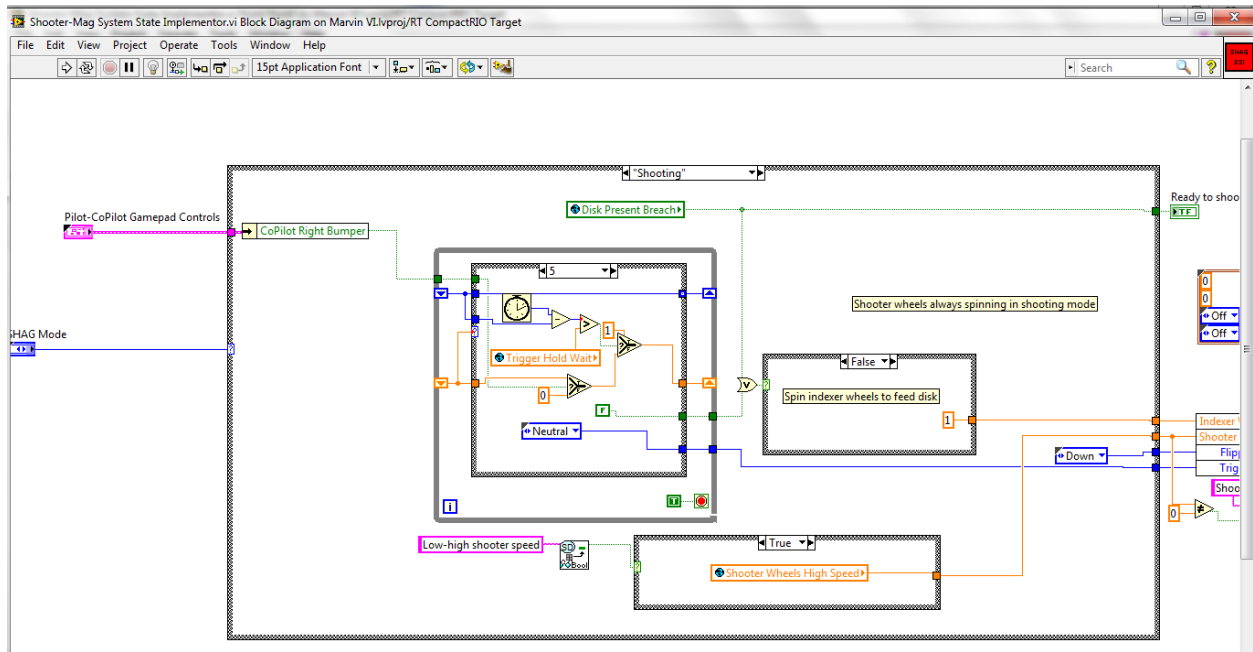


Overall robot architecture (note that SSS is no longer used, subsystem state selection is done in Modes and States)



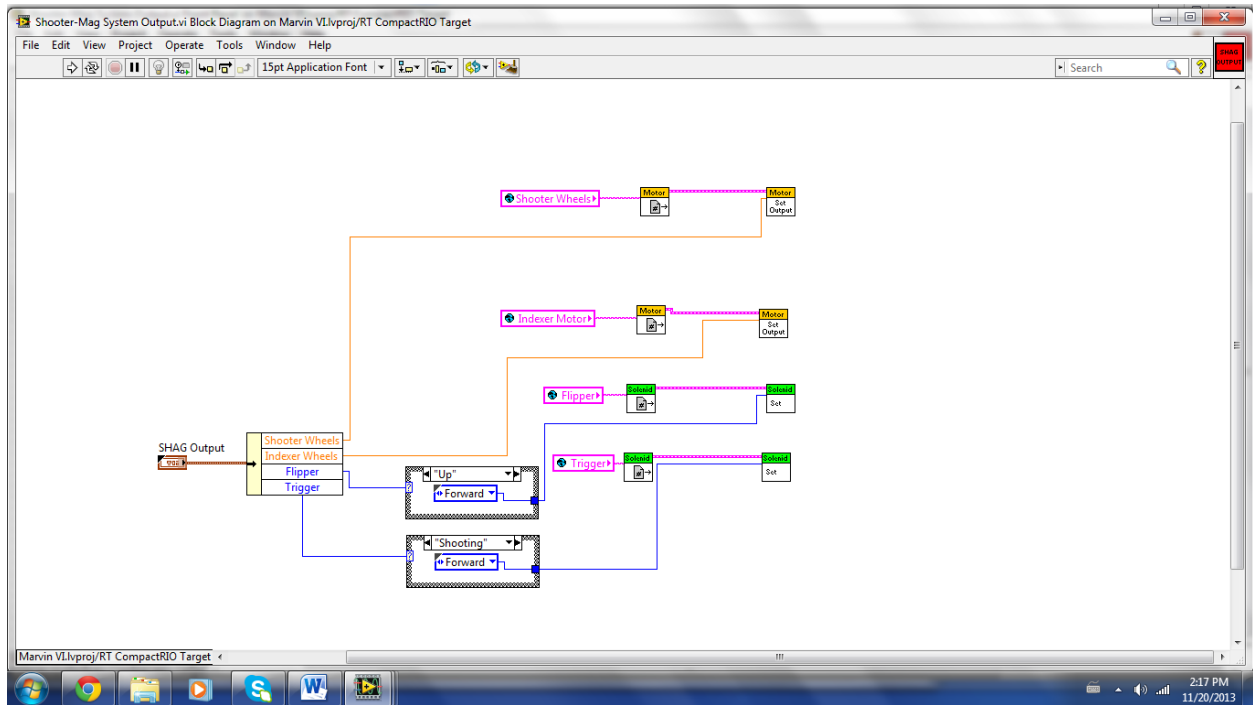
Jesus Christ, Modes and States can be a mess. On the left case structure is the major robot mode, and on the right are some subsystem states. It outputs an enumerated type for each subsystem. This VI only controls subsystem state selection, however, and the states do not have a meaning as of yet.

Please note that buttons on the joystick do not automatically latch or toggle they send True when held and False when not held. We use Boolean arrays and Boolean Array to Number to combine the cases for each button press into a case structure. To maintain a mode while no buttons are pressed, we put the state into a feedback node, which is maintained (it sends the value to itself continuously) in case 0 and set it to change in cases above 0 (cases above 0 correspond to button presses). This way it outputs the state it was already in if no buttons are pressed.



In each SSI VI, every state is assigned a different behavior for the subsystem using the case structure. Here is where the robot's behavior is actually determined. However, it only outputs numbers and other types of data.

Skip the safety VIs, and then...



The output VIs are where the outputs are actually sent to the motors and other actuators. Note that in these VIs are some of the few places where real FRC VIs are used.