HOWDY BOTS

FRC Team 6377

How it's made

Howdy Bots' 2019 Robot Code

Austin, TX - December 2019

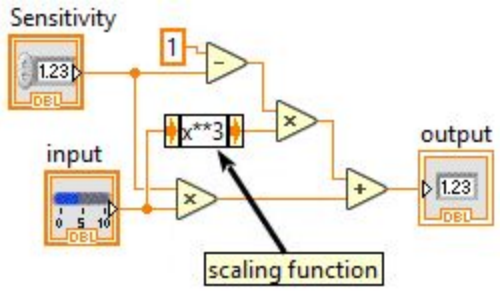
Index

The Decisions Behind the Design

Why LabVIEW?

LabVIEW is a graphical programming interface developed by National Instruments. The reason we chose LabVIEW was its ability to easily create and display control systems which works very well with FRC robots. Additionally, in a lot of respects it is easier to pick up and learn than syntax based languages like C++ or Java. LabVIEW also allows for easy development, debugging and troubleshooting, which cuts the testing time for the team.[1]

LabVIEW, because of its graphical nature, allows immediate tasks parallelization by allowing the code to run binded solely by dataflow[2]. This eases the effort required to maintain a good understanding about the code's functionality as more systems are added to it. Moreover, LabVIEW optimizes the code we write, and the development environment ensures that we do not have to spend as much time taking care of memory handling, syntax errors, and asynchronous race conditions.



Lessons learned from 2018 and preparing for the 2019's liftoff

During 2018's *FIRST* Power-Up challenge we learned a lot of very important lessons like simply keeping your project clean in labview and on your desktop helps immensely during the chaos of build season:

[1] "What is LabVIEW? - National Instruments." http://www.ni.com/en-us/shop/labview.html. Accessed 26 Feb. 2019.
[2] "Block Diagram Data Flow – LabVIEW 2018 Help", http://zone.ni.com/reference/en-XX/help/371361R-01/lvconcepts/block_diagram_data_flow/, Accessed 9/9/19

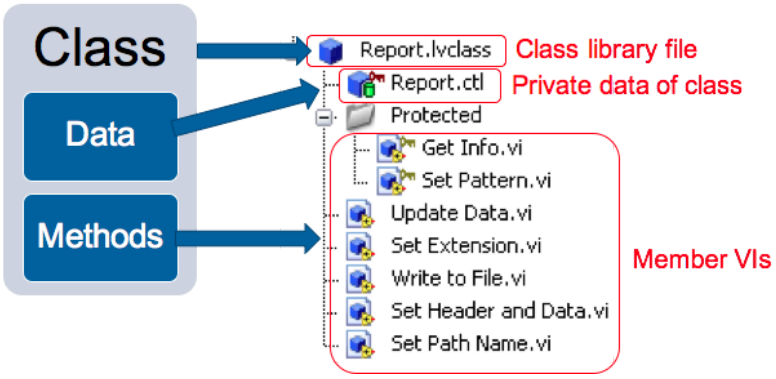
- We learned how to use PIDF[3] control mode which lets us actively compensate for our output on the drive train .
- We learned how to use classes and the fundamentals of object oriented programming .
- We learned about velocity closed-loop control modes letting directly map our joysticks to our velocity.
- We learned how to use vision to identify targets using the Limelight.[4]

Why LabVIEW Object Oriented Programming?

Object-oriented design has been a major trend in computer science for years, and with good reason. Using classes to represent systems and phenomenon allows programmers to abstract real life into code that is reusable, modular and easy to expand. LabVIEW allows for this programming using classes (lvclass) and libraries (lvlib).

The team adopted the LabVIEW Object-Oriented design[5] with two major goals:

1. Allow for code reusability and subsystems decoupling (one not depending on others), and
2. Enable better source code control and simultaneous development by multiple programmers.



Code reuse and class' decoupling

We worked hard to keep our classes decoupled this means they don't depend on each other. Coupling is the degree of interdependence between software modules; a measure of how closely connected two

[3] "FAQ: What are PID gains and feed-forward gains? - Motion Control Tips." 30 Dec. 2016, https://www.motioncontroltips.com/faq-what-are-pid-gains-and-feed-forward-gains/. Accessed 26 Feb. 2019.
[4] "Limelight for FRC: Limelight Smart Camera for FRC." https://limelightvision.io/. Accessed 26 Feb. 2019.
[5] "LabVIEW Object-Oriented Programming FAQ - National Instruments" http://www.ni.com/product-documentation/3573/en/

routines or modules are; the strength of the relationships between modules[6]. Our goal decreasing coupling is to allow our code modules (classes) to work independently of each other, allowing us to reuse them and test them independent to the other systems.

The main goal of class' decoupling is to make sure the classes are self contained and do not call into other classes, since this would mean that porting one of them to another project or test bench will bring the rest as dependencies. As a matter of fact, this is a common problem with all programming languages, however LabVIEW allows us to detect these couplings by showing them in the Dependencies section of the project explorer window, and locks them when they are used in more than one target or project.

> For example: One of the things we focused on in 2019 was being able to reuse code on different robots or different pods for instance using classes to set our implementation to simulated for the parts of the robot unique to Outlaw (e.g our arm) and than simply adjust the values of the .INI file and our 2019 code is able to run on our 2018 robot "Pacbot" without issue. Therefore, we developed the code in a way where if one of the systems was not present, it would not break the functionality of the rest of them.

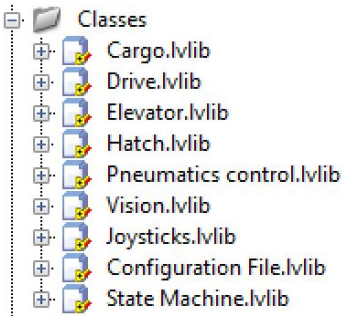
[6] Coupling (computer programming) https://en.wikipedia.org/wiki/Coupling_(computer_programming) Accessed 18 May. 2019.

The Architecture

From subsystems to classes

When a team designs a robot, they usually divide it into its different components: the drive train which will make it move; the arm or elevator that will help move game pieces; the game piece grabber; and so on. Similarly, it is natural to think of the robot's code as a delicate dance of multiple systems receiving commands and working with each other to accomplish tasks. Therefore, we decided to take this approach with our software architecture.

As the mechanics and CAD team worked on the robot's design, the programming team was able to outline and grow the code as the mechanism were defined and built. A big challenge in the engineering world that we also have in FRC is a constant change of requirements and specifications on how a certain assembly will work. Fortunately, having a flexible architecture that allows us to create, edit, and iterate quickly over multiple subsystems enables the team to respond effectively to this environment.

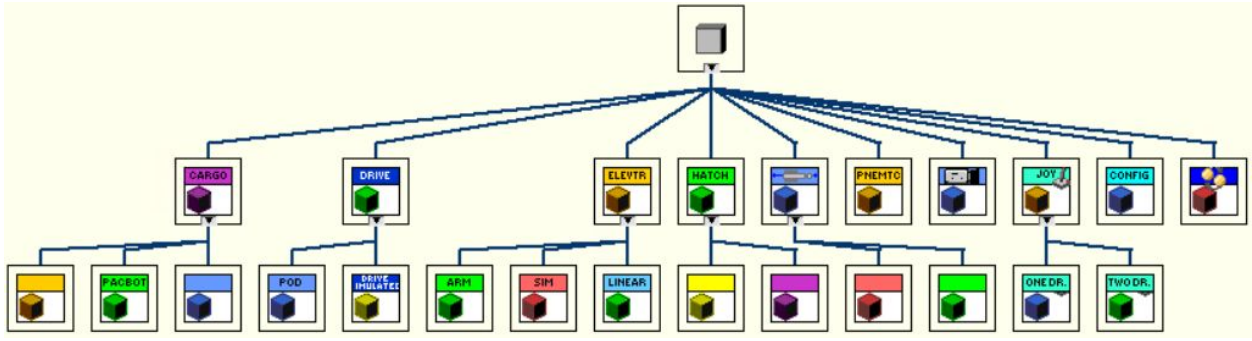


By seeing how the robot is physically wired and built, the team modeled the subsystems to reflect the real world. However, there was one last piece that needed to fit in the puzzle: those components that are not physical but provide the code with more functionality and allow modularity. You will notice that some classes, such as the Vision, Joysticks and Config have been modeled to show encapsulated code modules, which does not always reflect the physical world, but provide an abstraction for the team to work more efficiently.

Using inheritance to allow for simulation and flexibility

Utilizing classes lets us choose implementation based on which child class is picked for each of our subsystems (e.g drive, pneumatics, Vision) there is a parent (e.g Drive) and child classes that inherit from the parent like Pod Drive our main velocity closed loop drive that uses file reading to determine what

Configuration to use Via our .INI file using or Victor[7] drive which is a relatively simple PWM drive or simulated which is an implementation with that is completely empty what this lets us do is “simulate” code effectively



Our inheritance tree reassembled a Hardware Abstraction Layer (HAL) using the idea of abstract and concrete implementations of code. This translated into having parent classes that provide the function prototypes (which can be thought of as a Shell that contains no functionality, or hosts elements of logic that are constant for the system), and the child classes that concretely implemented the connection between the logic and the physical layer (which can be thought of as the driver layer).

> For example: the 2019 elevator parent abstract class holds the state machine logic for the arm positions, while the child concrete class implements the Talon SRX functions to submit the requested position to the physical motor controller.

Loading constants from a file to prevent code recompiling

Having to re-deploy the code to a robot after changing one value can be time consuming. Let aside the effort that sometimes has to be made to find where that value is hidden. Therefore, a solution is needed to ease the need to find all these values and keep prototyping, testing and competition easy to tweak as needed.

Instead of hardcoding values like sensor phase or PIDF[8] or even CAN IDs[9] values we read them from an .INI[10] file that lives on the roboRIO[11], which lets us edit those values in real time without editing the code or having to recompile it also lets easily switch between our drive pods because we don't have to edit the Talon IDs in the code we can just change the file.

---

[7] "Victor SPX - Cross the Road Electronics." 28 Nov. 2017, http://www.ctr-electronics.com/victor-spx.html. Accessed 26 Feb. 2019.
[8] "FAQ: What are PID gains and feed-forward gains? - Motion Control Tips." 30 Dec. 2016, https://www.motioncontroltips.com/faq-what-are-pid-gains-and-feed-forward-gains/. Accessed 26 Feb. 2019.
[9] "CAN bus - Wikipedia." https://en.wikipedia.org/wiki/CAN_bus. Accessed 26 Feb. 2019.
[10] "INI file - Wikipedia." https://en.wikipedia.org/wiki/INI_file. Accessed 26 Feb. 2019.
[11] "roboRIO - National Instruments." http://www.ni.com/en-au/support/model.roborio.html. Accessed 26 Feb. 2019.

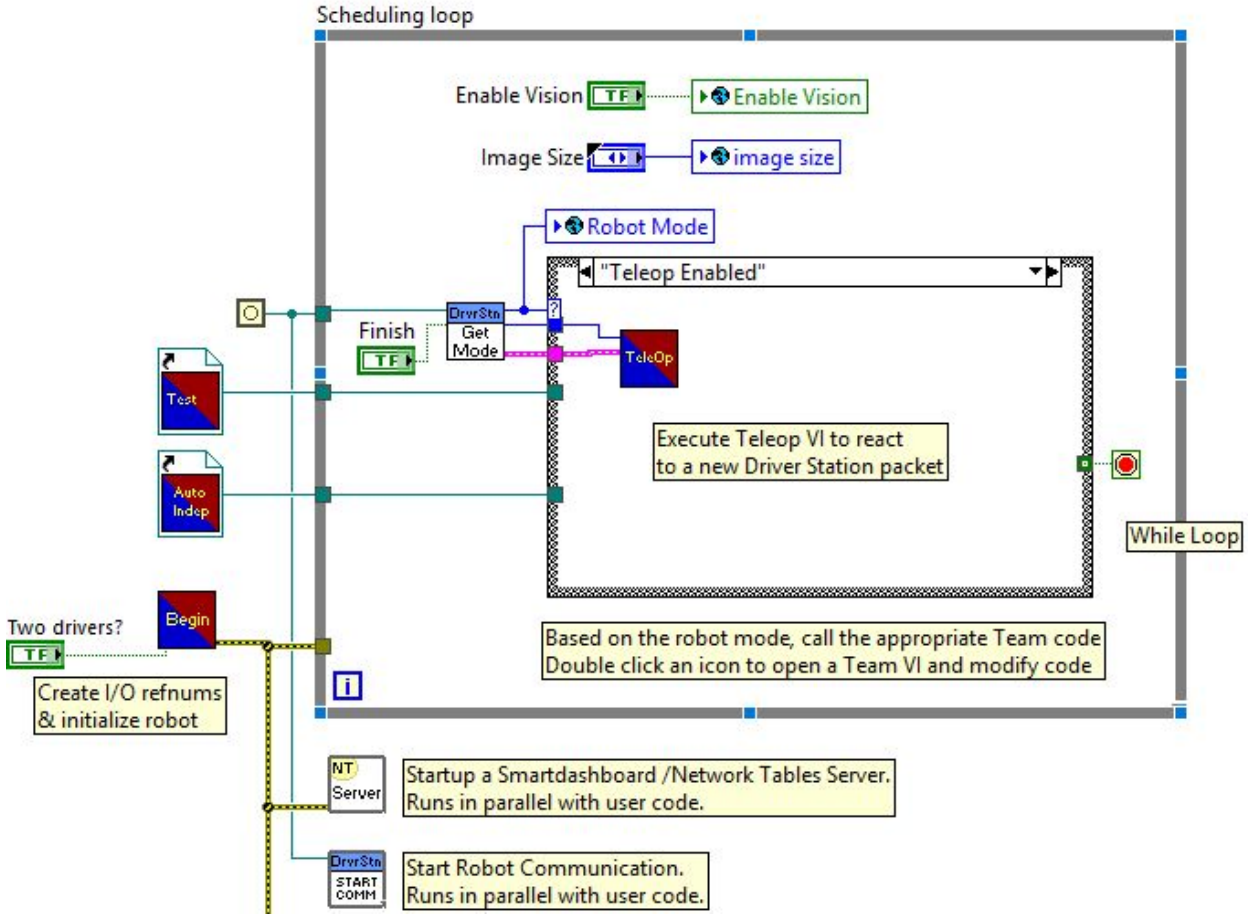
The configuration file is unique to each robot, which enables us to run the same codebase in multiple systems without having to edit the source, but just rely on the configuration file. Moreover, for those robots without specific mechanisms (either because they were not designed to have one, or because it is being fixed), we can select an abstract implementation of the different subsystems in the code. This feature allows for fewer errors in the console when something is missing, as well as more flexibility on the mechanisms that the code supports.

The file is divided by subsystem (or library), which makes it easy to navigate and find the value that's been looked for. Additionally, each configuration parameter contains the data type it uses, so both the code and the programmed editing the code can know what is expected from that value. Read more about the file parsing and code load in the file configuration class.

The Implementation

The FRC Code

We use the standard shipping architecture for the Arcade Drive robot in LabVIEW. Once we have these basic components we start developing on top of the different sections of code offered by the template.



Robot Main.vi

Team Code

The basic LabVIEW robot framework consists of the following VIs, which we use for different purposes, as detailed below:

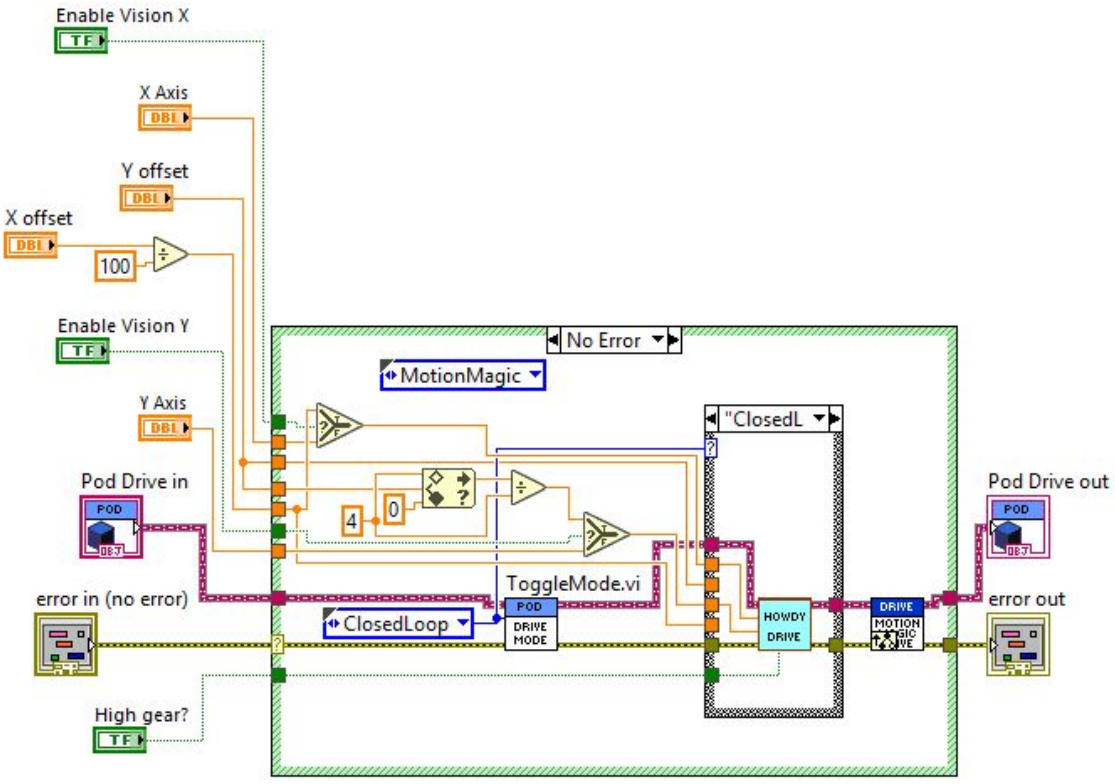
VI	Description and Use
Robot Main.vi	Robot Main is the robot's top level application. It calls the VIs located under TeamCode as they are needed, depending on the driver station's commands. This VI required minimal edits, and we disabled the VI that sends the camera from the roboRIO's USB port to the dashboard to preserve RAM memory, and because we use Limelight to provide this functionality.
TeamCode/Begin.vi	This is the first function that runs when the robot code starts running. We use this VI to load our configuration file and then instantiate each subsystem according to its parameters. This provides us with maximum flexibility in the systems we decide to use and the code constants that each of them will use through the running code. Read more about this process in the Configuration File.lvclass
TeamCode/Teleop.vi	The telop VI runs every ~20 ms (50 Hz) and connects the joystick controls to each of the robot's systems (objects). Because this VI is consistently called by the framework, we use it also to control all the state machines that are stored in each subsystem.
TeamCode/Autonomous Independent.vi	Because of the Sandstorm mode thatthe2019 game provided, we used Autonomous Independent to call into the Teleop VI and allow robot control through the joysticks. In future games, we will consider other approaches, as needed.
TeamCode/Finish.vi	This VI invokes all the Finish.vi methods for all the classes that were opened in Begin.vi. We use it to close references and deallocate resources that we might have required during operation. It is important to note that this VI will never run on the field and it is used when debugging the code when running interactively from the LabVIEW Development Environment.
TeamCode/RobotGlobal.vi	This is a global variable that holds all the robot's objects. This global variable is set in Begin.vi and then used through Autonomous, Teleop and Finish to preserve the state of the global data. We use a global variable to ensure data communication during Autonomous, since this VI is launched asynchronously by the framework, and traditional "wire" approaches would not work without editing the base framework.

The classes

Drive.lvclass

This class contains the initialization and implementation for our drivetrain it lets us seamlessly switch between pod drive, and simulated drive. The pod drive is based in the Howdy Bots pod system for 2019, which enables the mechanical team to fix the drive train easily by replacing a complete side of it (referred to as a Pod). However, this brings more challenges to the programming team, since it means not knowing which Talons (and Talon IDs) are going to be set for each side until the robot is ready for a match.

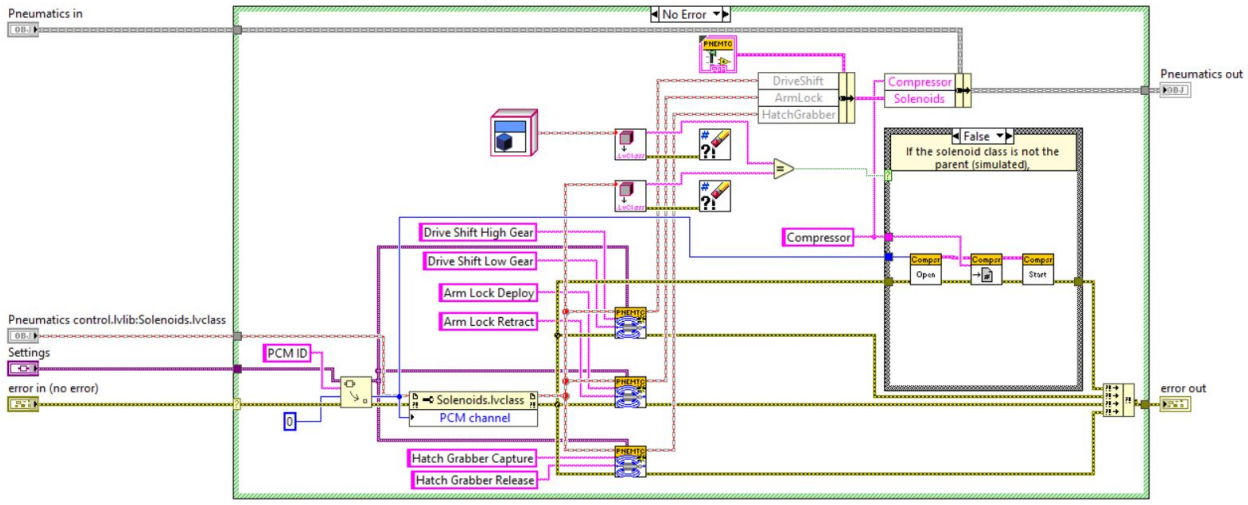
To overcome this challenge, we implemented a configuration file that declares the drive constants and initializes the code with the correct motor controllers. Moreover, the drive allows for pneumatic switching between high and low gear, as well as control of the robot using Motion Magic in the Talons.



Drive.lvlib: Pod Drive.lvclass:MotionMagicDrive.vi

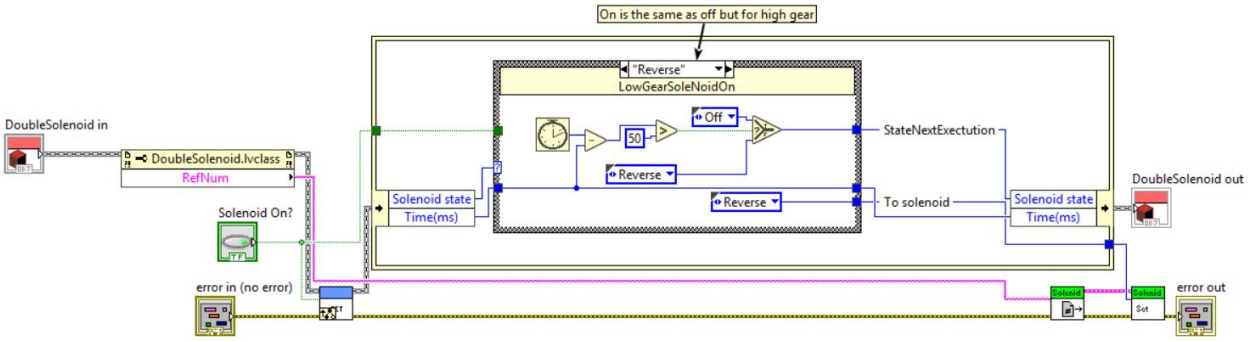
Pneumatics.lvclass

This class contains the initialization and implementation for our pneumatics like our “bird beak” or our ball shifters it contains an array of pistons we can toggle true or false based on button inputs it lets us switch between single channel solenoid and dual channel solenoid.



Pneumatics control.lvlib:Pneumatics.lvclass:Begin.vi

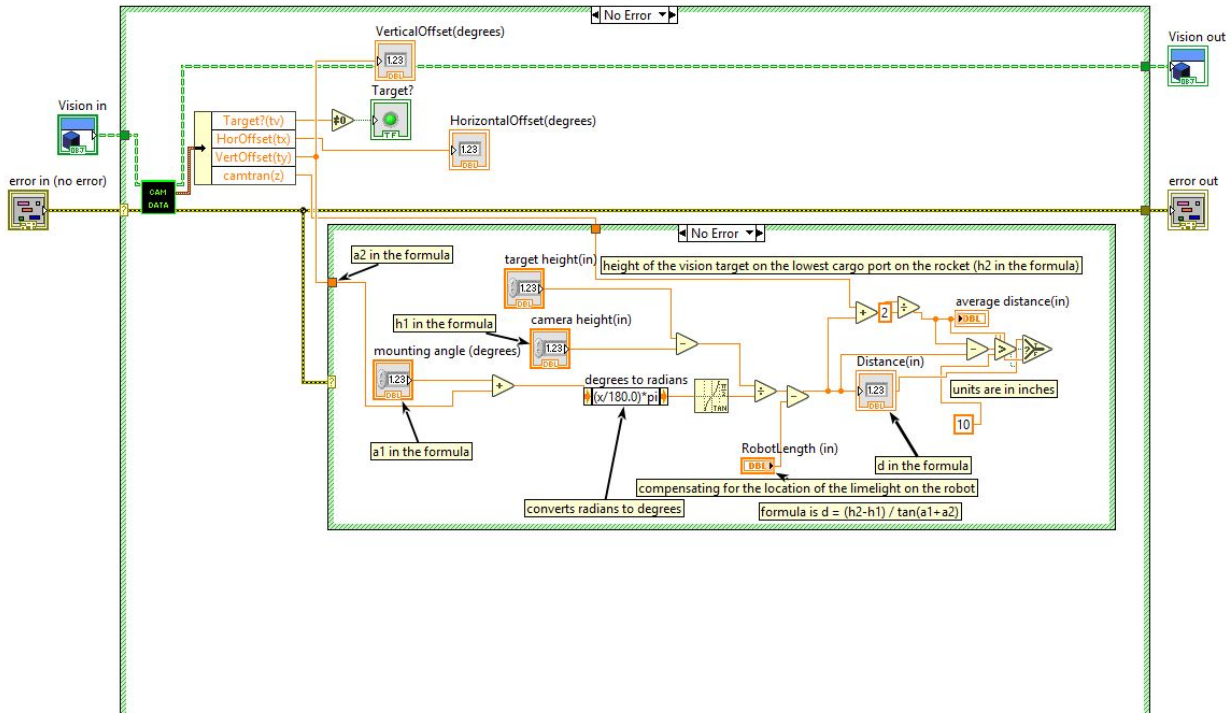
The Set Solenoid State VI is a VI that holds a state machine inside of the solenoid so we can switch between open and close without holding the solenoid high or low for more than 50 ms. The state machine switches between the states: Forward, On, Reverse, Off.



Pneumatics control.lvlib:DoubleSolenoid.lvclass:SetSolenoidState.vi

Vision.lvclass

The vision class interprets the data that we get from our Limelight 2[12] and turns it into the numbers that we can use, we turn target offset into a joystick input (-1 to 1) and then passes it to our drive

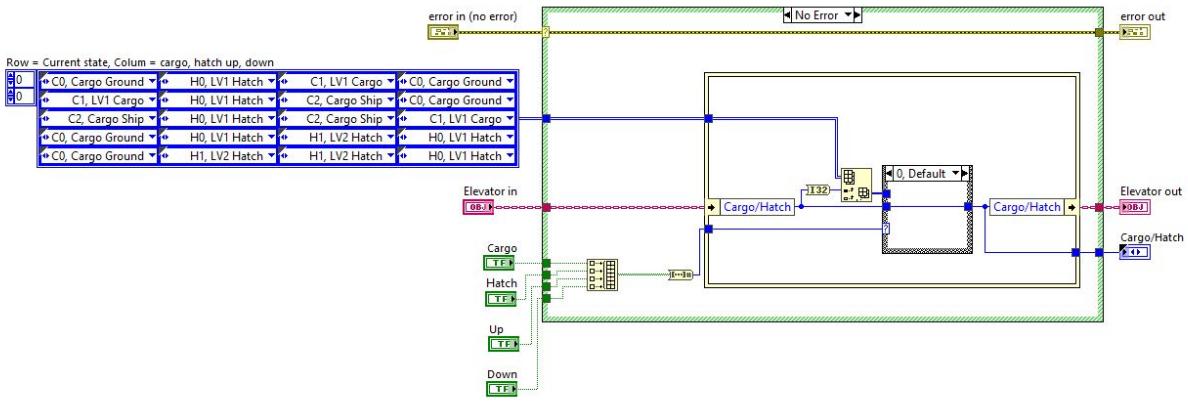


Vision.lvlib:Vision.lvclass:VisionTrack.vi

Elevator.lvclass

Uses a state machine to quickly determine when and where to move the arm. We moved the elevator by having three cargo positions and two hatch positions and setting each of them to a certain value depending on the specifications in the FRC field and then updating this value on the configuration file. Then we would use this class to control where we wanted the arm to go by giving all the possible places we wanted the arm to go. Using the d-pad on the controller you could switch between these cases to move the arm.

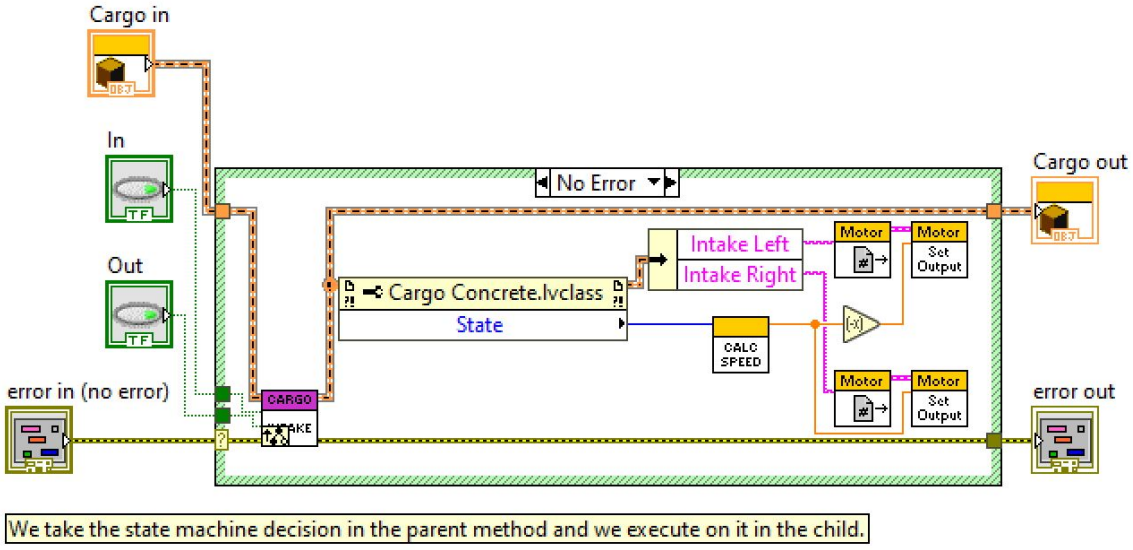
[12] "Limelight 2 Smart Camera for FRC – Limelight for FRC." https://limelightvision.io/products/limelight-2. Accessed 26 Feb. 2019.



Elevator.lvlib:Elevator.lvclass:Switching Levels Logic.vi

Cargo.lvclass

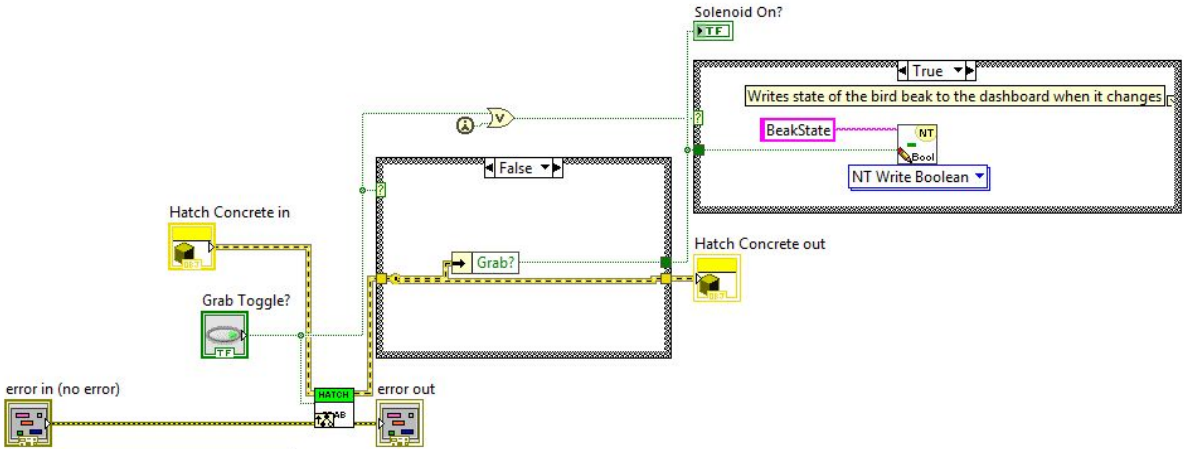
Controls the compliant wheels on the cargo manipulator which has an idle intake state to stop cargo from slipping. This idle intake state provides a low amount of power to the wheels so they retain the loaded cargo. This idle power is deactivated when the driver offloads.



Cargo.lvlib:Cargo Concrete.lvclass:Intake.vi

Hatch.lvclass

Controls the firing of the pneumatics in our “bird beak” and tells the wrist to lift up the hatch and published the state to a network table that provides the feedback to the drivers.

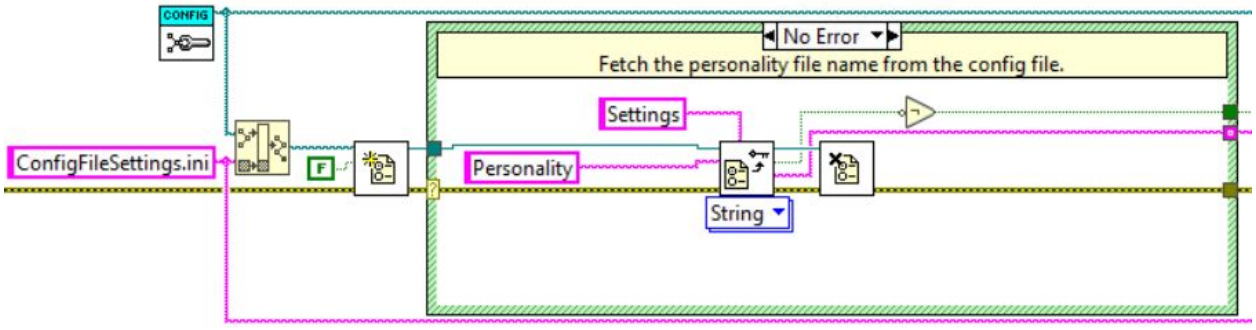


Hatch.lvlib:Hatch Concrete.lvclass:Grab.vi

Configuration File.lvclass

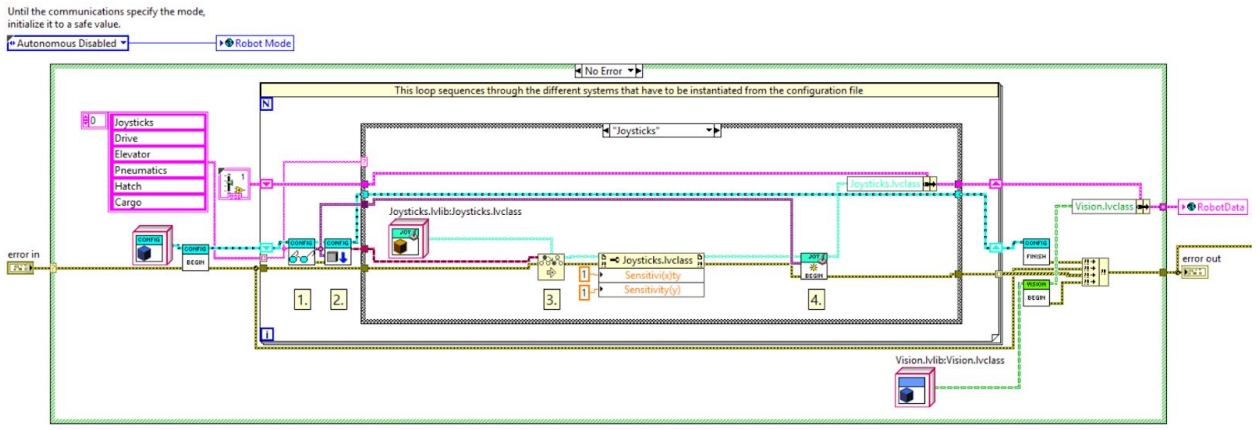
This class reads data from the .INI file that lives on the robot and passes its data to every class that requires parameters. This class takes the spotlight during code bootup time, since it will determine which classes to load, and the information to send to each one of them.

The class is first initialized by fetching two files: the first one will let the code know which configuration file to load, and the latter will be said configuration file that contains the parameters of the robot. This double approach allows the team to have more than one set of parameters in a single robot (for testing purposes mostly) without the danger of losing the previous data because of an overwrite. Therefore, by changing the file to read from an “index” file, we can swap values without having to edit more than one line.



Configuration File.lvlib:Configuration File.lvclass:Begin.vi

After the configuration file has been loaded (the file has been identified and a reference has been opened to its data), we then instantiate each subsystem according to its parameters. This process is done using a For Loop and iterating through all the systems we need to instantiate.



2019 Deep Space Project.lvproj/TeamCode/Begin.vi

The steps are as follows:

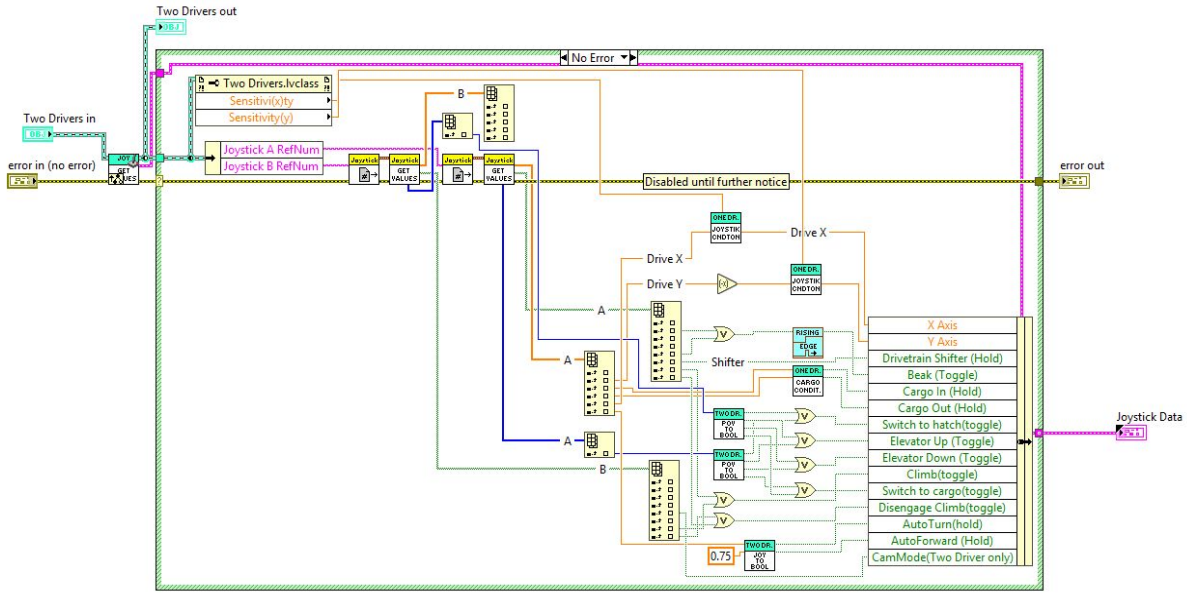
1. The configuration file class is commanded to read the data for a specific system (or library).
    a. We search for the section of the file that contains said class and read all the keys (parameters) that are inside it.
    b. Because every key has written its data type, we use the correct parsing method for each and store them in the data type they are intended to be.
    c. We load each one of the parameters (keys) that are read from the file into a Variant[13] as attributes. This mechanism creates a map of key and value now inside the Variant data[14] type.
2. The configuration file then searches for the *Implementation* key in the system (section or library) to learn which instance (child class) of the library needs to be loaded.
    a. We use the function Get LV Class Default Value By Name VI[15] to instantiate the class' instance that we need.
3. We use a Preserve Run-Time Class VI[16] to cast down the data type into the parent class we're interested in using (we do not use a specific type cast – known as: To Specific Class – since we want to make sure we keep any child class as-is).

[13] "Handling Variant Data – LabVIEW 2018 Help", https://zone.ni.com/reference/en-XX/help/371361R-01/lvconcepts/handling_variant_data/, Accessed 9/9/19
[14] "Variant Data – LabVIEW 2018 Help", https://zone.ni.com/reference/en-XX/help/371361R-01/lvhowto/variants/, Accessed 9/9/19
[15] "Get LV Class Default Value By Name VI – LabVIEW 2018 Help", http://zone.ni.com/reference/en-XX/help/371361R-01/glang/get_lv_class_by_name/, Accessed 9/9/19
[16] "Preserve Run-Time Class Function – LabVIEW 2018 Help", http://zone.ni.com/reference/en-XX/help/371361R-01/glang/preserve_runtime_class/, Accessed 9/9/19

4. We call the Begin.vi of the instantiated class, which takes in the Variant with all the parameters that were taken from the configuration file and loads them into its private data, as needed.
    a. A malleable VI (.vim)[17] is used to enclose the functionality of searching for each parameter in the Variant, and helps us throw errors whenever we cannot find a key element required for the component to work properly. Because of it being malleable, the same VI can handle multiple data types, adequating itself to the context and mutating itself into the first block diagram that can be compiled.

Joystick.lvclass

Do we want to drive with one or two joysticks? That's a common question we found ourselves dealing with. Since we are unsure of the performance using one or two drivers, we created the Joystick classes to allow switching from one joystick to two joysticks without having to modify any of the code.
We use parent class' dynamic dispatch methods in the code to read the controls we care about, which do not show as an array but as a cluster designed to work with our robot. This cluster will be populated with different data depending on the mode we are running, but the code does not care about it, since it has the information it needs from the custom cluster that always comes out of the function.



Joysticks.lvlib:Two Drivers.lvclass:Get Joystick Values.vi

[17] "Malleable VIs – LabVIEW 2018 Help", https://zone.ni.com/reference/en-XX/help/371361R-01/lvconcepts/malleable_vis_intro/, Accessed 9/9/19

Additional Tools

Using source code control with LabVIEW and GitHub

To enhance collaboration and parallel development, we use GitHub to manage our code and have our repositories synced up between the programming computers and the driver stations. To aid our development, we adopted a set of principles and tasks that ensure our success:

1. Understanding git is no easy task, so we encourage our teammates to play the "Learn Git Branching" game[18], which exposes them to the terms and commands available in this framework. After they are familiar with the terms, then they port the knowledge to our GUI git tool: SourceTree.
2. Using LabVIEW with git can be a challenge, due to the graphic nature of the programming environment. Nevertheless, the are able to mitigate that risk by following the recommendations cited by the NI's Systems Engineers in their whitepaper: NISystemsEngineering/GitHub-Hands-on[19].
3. Finally, because of our code modularity and order, we make sure no more than one programmer is editing the same files (VIs and Classes) at the same time, so we keep the code's merges clean and avoid conflicts when a graphic piece of code has been modified more than once.

Working with the CTRE Tools to enhance actuator's control

We used CTRE's fantastic motor controller the talon SRX[20] which let us easily implement advanced control modes like motion magic and closed loop velocity. You can find our methodology when working with motor controllers in our whitepaper: Don't Break Your Bot[21]

[18] "Learn Git Branching" https://learngitbranching.js.org/ Accessed 9/21/19
[19] "NISystemsEngineering/GitHub-Hands-on" https://github.com/NISystemsEngineering/GitHub-Hands-on/blob/master/Basic%20Concepts/Basic%20Concepts.md Accessed 9/21/2019.
[20] "Talon SRX - Cross the Road Electronics." http://www.ctr-electronics.com/talon-srx.html. Accessed 5/25/2019.
[21] https://howdybots.org/wp-content/uploads/2019/12/Dont_Break_Your_Bot_Whitepaper-V1.pdf

Sending files from the Dashboard to the roboRIO

To ensure each robot loads the correct settings it will use for competition, we developed a configuration file tool. Using some ingenious code in our dashboard we are able to read and write to the .INI file and to edit the version of the file that is hosted locally on our computers. This tool ensures we persist the file version that each robot uses and allows us to quickly iterate over parameters that otherwise would require a code recompile.

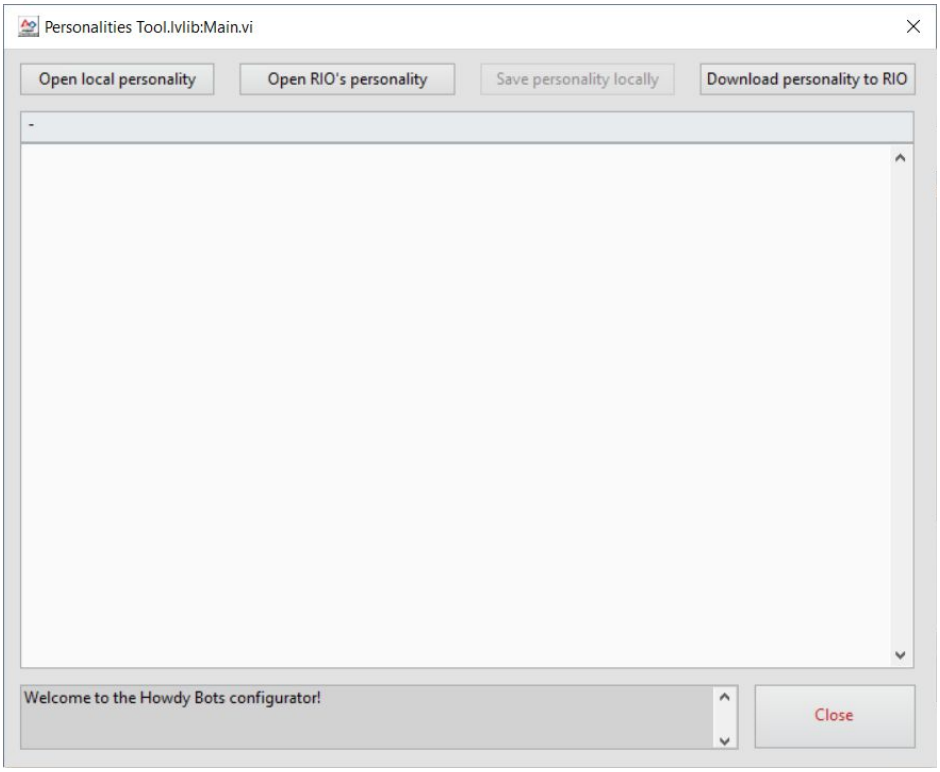


Image: Personalities tool front panel.

The Personalities Tool allows the team to read, edit and wrote the configuration files that the code uses to configure the robot. The tool also holds a small roboRIO serial numbers database that enables us to ensure the correct file is transferred even when the roboRIO is reformatted and reimaged.

Open local personality - This button allows you to see your personality on your machine/computer not the roboRIO. This is used to find your personality and then you can edit it on this tool.

Open RIO's personality - This button allows you to see the current personality on the roboRIO. Note - this only works when you're connected to the roboRIO. This is used to see what personality is currently on the robot. You can also edit this but also keep in mind that you have to download on to the RIO if you want the changes to actually save.

Save personality locally - This button does exactly what it says it does and downloads the personality to your computer not the roboRIO. This is used after you edit something and you want to save it but not upload it onto the robot because it might not be done yet.

Download personality to RIO - This button downloads your personality to the roboRIO AND downloads this locally as well. This is used to download something back to the roboRIO after you know the personality is right and all changes have been accounted for. This is the only way to upload something back onto the RIO.

Error Box - The text box on the bottom that says “Welcome to the Howdy Bots configurer!” is where all errors occur. One example of an error is when you click the “Open local personality” button and the tool can’t locate any .ini files on your machine. It will look for any .ini file after it can’t find the actual personality. Another very common example is when the tool can’t locate the roboRIO at all.

References and useful links

- https://phoenix-documentation.readthedocs.io/en/latest/ - Documentation for Pheonix CTRE
- https://www.ni.com/en-us/support.html - LabVIEW help
- https://johnvneun.com/calc - JVN calculator
- http://docs.limelightvision.io/en/latest/ - Documentation for Limelight
- https://wpilib.screenstepslive.com/s/currentCS/m/labview - WPI/Additional help on LabVIEW
- https://www.ni.com/en-us/landing/first.html - Resource Hub for FRC
- https://github.com/NISystemsEngineering/GitHub-Hands-on/blob/master/Basic%20Concepts/Basic%20Concepts.md - Git Hub Basic Concepts
- https://learngitbranching.js.org/?locale=en_US - Game to learn Git Branching

The team

About the Howdy Bots

We are a community robotics team from Austin, TX that participates in FRC (*FIRST* Robotics Competition) with students who range from 13 to 18 years old. We accept students who do not have access to FRC through their schools or existing organizations. In fact, our team was created because some students wanted to participate, but there was no team available for them! We are comprised of students with a passion for STEAM education, and mentors, whom are experienced professional engineers and business leaders.

The programming team

Students

- Alex Bui
- Rishik Boddeti
- Sean Cappleman
- Tyler Hedge

Mentors

- Austin Page
- Erik Skiles
- Evan Marchman
- Michael Brandt
- Oscar Fonseca