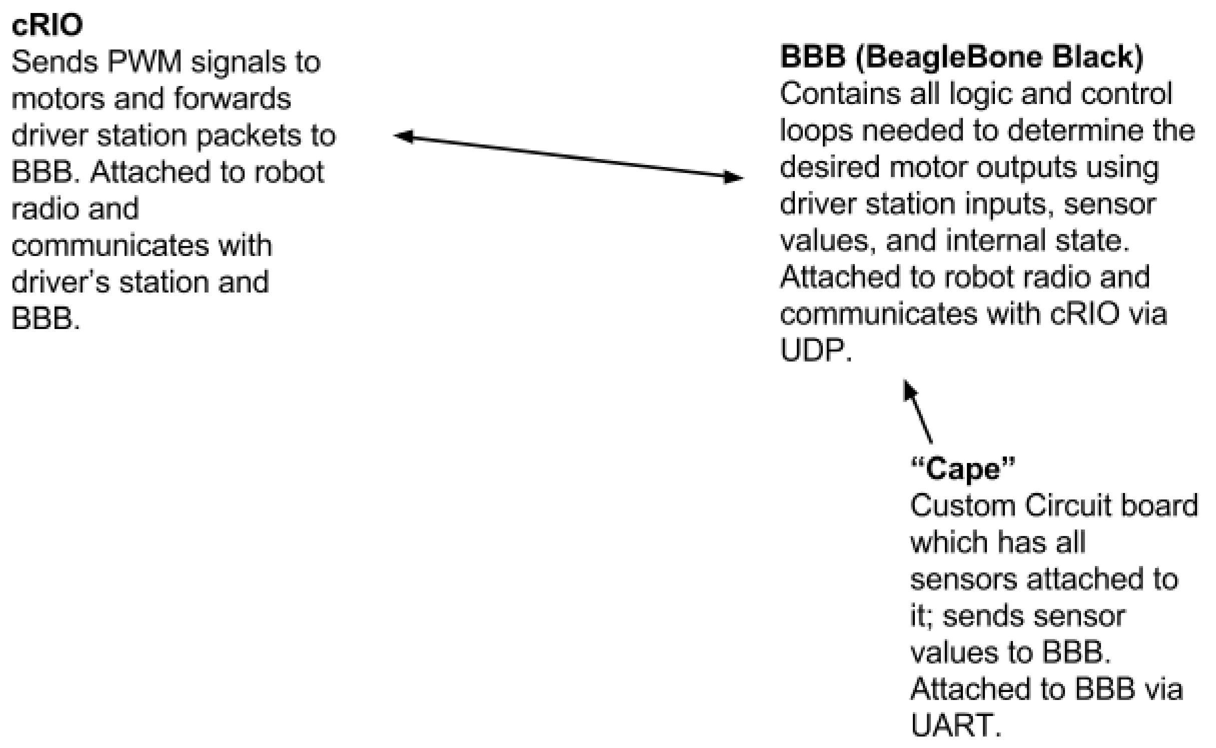
Spartan Robotics

2014 Controls Documentation

Design Goals

Create a reliable and effective system for controlling and debugging robot code that provides greater flexibility and higher performance than what is available through traditional methods.

Overall Control System


cRIO
Sends PWM signals to motors and forwards driver station packets to BBB. Attached to robot radio and communicates with driver's station and BBB.
BBB (BeagleBone Black)
Contains all logic and control loops needed to determine the desired motor outputs using driver station inputs, sensor values, and internal state. Attached to robot radio and communicates with cRIO via UDP.
"Cape"
Custom Circuit board which has all sensors attached to it; sends sensor values to BBB. Attached to BBB via UART.

Advantages of using a secondary processor over using just the cRIO

- *Logging:* We can log everything that happens on the BBB and store it on a microSD card on the BBB, allowing us to look after a match and see what happened in case we had any issues.
- *Faster code deployment times* (does not require rebooting cRIO to deploy code)—we are just limited by how fast we can compile the code on a laptop and send it over the network to the BBB.
- *Greater control over environment:* We have far greater flexibility in what we are able and allowed to do with the BBB than with the cRIO.
- *Greater opportunity for students:* Allows students to expand their skills beyond the FIRST-provided framework.
- *Unit-Testing:* We were able to use the gtest unit-testing system to create a framework for unit-testing such that we can create unit-tests for every major system and control loop on the robot, allowing us to debug certain issues before they ever reach the robot.

Roles of Various Processors

The cRIO

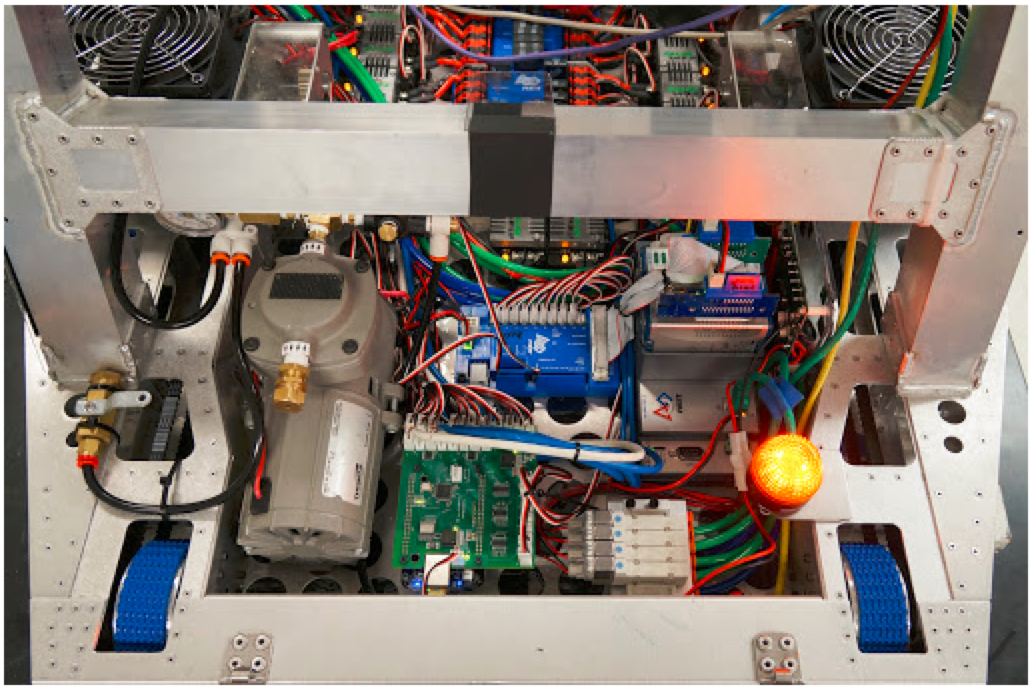
- Receives the joystick packets from the driver's station and forwards them to the BBB.
- Receives desired motor outputs from the BBB and sends them to the Digital Sidecar.

The "Cape"

- Receives electrical signals from various sensors (analog and digital) and sends them to the BBB.
    - Includes a gyroscope of higher quality than the typical one that we would buy off of AndyMark, allowing for improved feedback.
- Provides power for the BBB, taking power in turn from a COTS circuit board (originally designed by us) which provides a regulated 12V.

The BBB

- Runs real-time version of Linux
- Contains logic and instructions for autonomous and teleoperated modes.
- Logic for zeroing claw joints and shooter.
- Control loops for claw, shooter, and drivetrain.
- Logs all that happens on the robot.



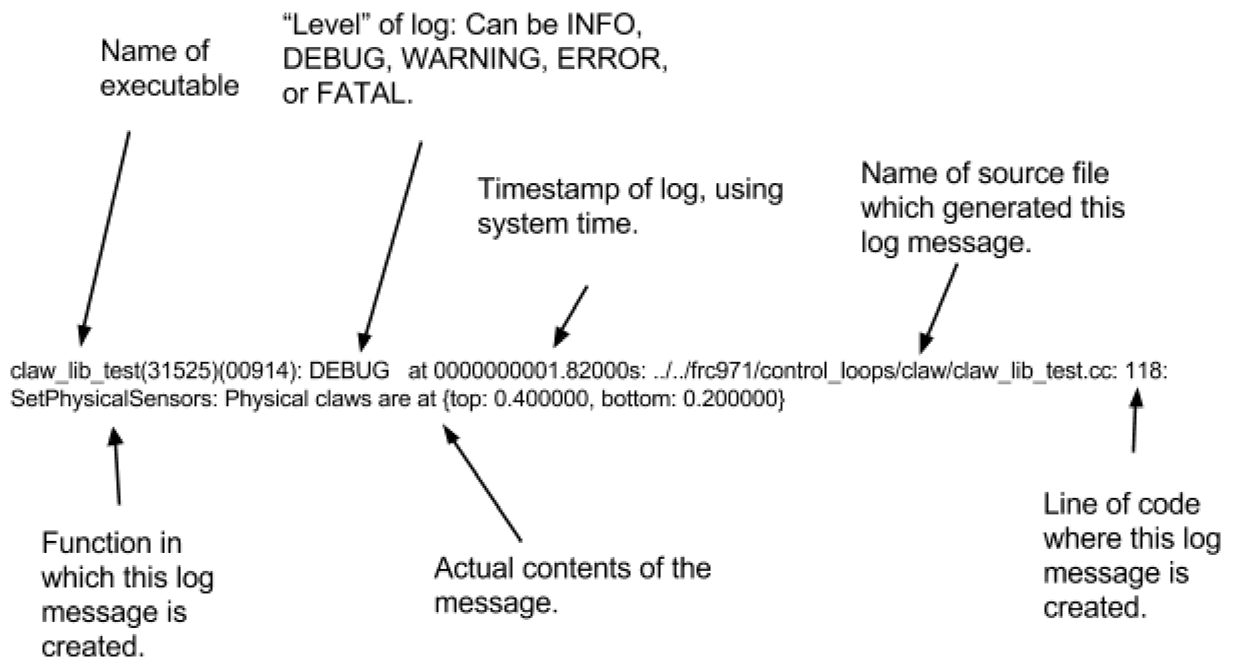
Electronics package inside the robot

Primary Features of the BBB

Logging

- Logging makes it much easier to debug issues and determine what the robot was doing at any given time.
- Among other things, we log:
    - Any significant warnings or errors.
    - Sensor values at every run of the control loops (which are run at 100 Hz)
    - The motor outputs being sent to the cRIO.
    - The internal state of the various control loops and zeroing logic.
    - The values of the various joysticks on the driver's stations, which buttons were hit and released when, when the robot entered and left autonomous or teleop.

An individual log message will look something like this (this log message is taken from the output of a unit test):


Name of executable
"Level" of log: Can be INFO, DEBUG, WARNING, ERROR, or FATAL.
Timestamp of log, using system time.
Name of source file which generated this log message.
Function in which this log message is created.
Actual contents of the message.
Line of code where this log message is created.

```
claw_lib_test(31525)(00914): DEBUG   at 0000000001.82000s: ../../frc971/control_loops/claw/claw_lib_test.cc: 118:
SetPhysicalSensors: Physical claws are at {top: 0.400000, bottom: 0.200000}
```

A single iteration of each control loop will generally generate 4 - 5 of these messages each, providing information about position, status, outputs, goals, etc. In order to create the log message above, all that had to be done was to call the LOG macro:

```
LOG(DEBUG, “Physical claws are at {top: %f, bottom: %f}\n”, pos[TOP_CLAW],
pos[BOTTOM_CLAW]);
```

In order to view robot logs, then you must ssh into the BBB and use the “log_displayer” executable which we wrote for this purposes. An example usage might call:

```
log_displayer -f -n claw -l DEBUG
```

This would look in the claw logs (-n claw) down to the level DEBUG (-l DEBUG; in this case, it would also display WARNING, ERROR, and FATAL logs, allowing us to filter by severity) and continuously display logs as they were being created by the robot (-f), allowing someone with access to the robot network to follow what the robot thinks it is doing in real time.

Queues

In order communicate between the various processes running at any given time (for instance, the autonomous routine, the drivetrain control loop, and the process which sends motor outputs to the cRIO), we wrote code which creates and manages pieces of shared memory which we refer to as “queues”. To whoever uses the queues, it appears that there is what amounts to a struct in each queue; processes can write and read to the queues to communicate with other processes. For instance, there is an output queue associated with the drivetrain which contains doubles for left and right voltages and two booleans to represent shifter positions. The drivetrain control loop writes values to the output queue whenever it runs and the process which sends values to the cRIO then reads from the queue and sends all the information from the drivetrain, claw, and shooter output queues to the cRIO. In a similar manner, the autonomous routine may write position goals for the drivetrain to the drivetrain goal queue; the drivetrain control loop will then try to reach these goals.

Control Loops

On the BBB, we run the three main control loops for the robot: The drivetrain, the shooter, and the claw. Each control loops presents its own unique set of challenges, although the control loops also share a great many features, such as the underlying state space models, as well as a set of custom libraries which allow for easy code-reuse from year to year and thereby quicker development of stable, effective control loops for robot mechanisms. In order to run these control loops, the robot relies on a total of 5 quadrature encoders, 10 digital hall effect sensors, and 4 analog digital hall effect sensors to determine its state and act upon it.

Common Features

State Space Control and Modeling

In order ensure optimal control and stability, we chose not to use the typical PID loop but rather to rely on a set of matrices and equations which allow us to both model the given system and to determine certain aspects of optimal control under ideal circumstances.

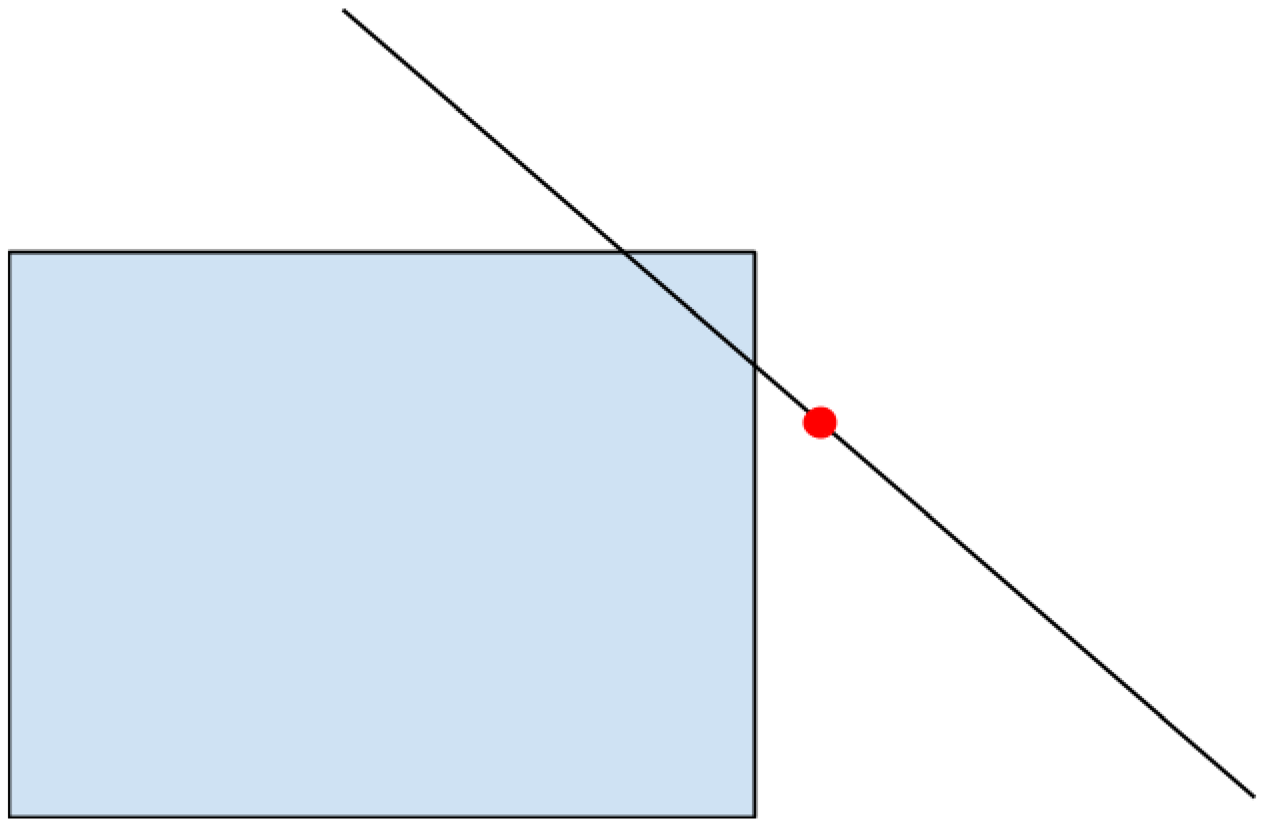
This allows us to:

- Model the response of a DC motor to a given input voltage, given certain common statistics about a DC motor, such as free speed, free current, stall torque, and stall current.
- Model any other linear time-invariant aspects of the system which may be present (for example, the springs on the shooter).
- Determine the optimal controller using a Linear Quadratic Regulator and appropriate weightings of error and voltage in the cost function.
- Determine the stability and controllability of a system.
- Perform certain other operations on the system, as are described below with the polytopes.
- Use the model in our unit tests to test that the state machines are able to control the various robot systems to do what we desire them to do, and debug complicated problems before ever running code on the real hardware.

Polytopes for setting goals to match certain conditions

In order to achieve certain desired characteristics in the behavior of a given system, such as constant radius turning in the drivetrain, we can use the state-space model of the system and certain constraints (such as a maximum of 12 volts to apply to the motors). We then end up with a situation where we have a region of possible goal states (for our purposes, these regions are two-dimensional, although the problem does generalize to n dimensions), a line which represents certain constraints which we are imposing on the system, a point which represents some sort of desired or optimal solution, and we must determine what action will fall within the provided constraints while being as close to the desired point as possible.

To provide a general example, consider the following box, line, and dot:



The “solution” point must lie inside the shaded region; if the line crosses the shaded region, the solution must fall on the line, otherwise the point in the shaded region and closest to the line is selected; once these constraints have been satisfied, the goal is to find a solution as close to the dot as possible.

We use this math in two main places on the robot:

- The drivetrain, where we want generally to achieve a constant radius turn (which creates the line) while going as close to the desired linear and rotational velocity (the dot) as possible, all while sending out voltages which are physically possible (the region).
- The claw, where we want to avoid allowing the separation of the two components to drift (the line) while aiming for the desired goal (the dot) and sending out physically possible voltages (the region).
    - Note: The line used for optimization does not have to be straight; by adding an angle into the line and making it into two rays, we were able to tune the claw controller such that, even when dealing with large positional error, the controller never allowed it to prioritize position over separation (ie, holding the ball in the claw).

Individual Control Loops

The Drivetrain

- Takes a throttle and steering for an input; then attempts to achieve a certain speed (the throttle) while turning at a constant radius which corresponds to the steering input. Uses the aforementioned polytope code to achieve this.
    - On the driver’s station, we have a steering wheel and a joystick; the steering wheel

    provides a turning radius (just as it would in a real car) and the joystick provides a desired velocity.

- State-Space model accounts for physics involved in the interactions between the left and right sides of the drivetrain—the control loop must account for the fact that, when one side of the drivetrain moves, the other side does not necessary stay in place.
- Uses quadrature encoders, one on each side of the drivetrain, to provide feedback; 4 analog hall effect sensors on the shifters provide feedback as to whether the dog shifter is fully engaged in either high or low gear on either side of the drivetrain.
- Gyroscope allows further feedback to ensure that drivetrain is driving straight and turning correctly during autonomous.

The Shooter

- When shooter plunger is not held back by latch, accounts for physics of springs—due to the linear nature of the spring force, allows for relatively simple incorporation of spring physics into state-space model.
- When the springs are held back by the latch, we use a different model and controller which knows that the springs are all the way back. (Gain Scheduling)
- Takes a desired shooter power as a goal and translates it into a positional goal for the hard stop.
- Uses two digital hall effect sensors to calibrate position of shooter (see zeroing section below).
- Uses a quadrature encoder to determine relative position of shooter.
- Uses two additional digital hall effect sensors to detect when the shooter plunger is pulled back and when it is latched, allowing us to detect potential jams or errors and thereby allowing the code to shut-down the shooter to avoid stalling and burning out the motors.
- All shooter motions have timeouts which monitor whether the shooter is acting properly and abort to avoid burning up the motors if something goes wrong.

The Claw

- By manipulating the state-space representation, we were able to create a controller which allowed for near-independent control of the position of the bottom of the claw and the separation of the two halves.
    - The “claw” is made up of two halves: A top and a bottom. In order to hold onto the ball when moving the claw, for instance, between the ground intake and the close shot positions, the separation of the two halves must remain almost constant in order to avoid dropping the ball. Ideally, we would be able to determine a solution which allowed us to control the position of the bottom half and the separations of the two halves entirely independently. Unfortunately, three things complicate the problem:
        - That the two halves have different moments of inertia (having different motors or different gear ratios would also affect this).
        - The discretization of the state-space model.
        - The limits on the voltages which we are able to apply to the motors.
    - In order to solve each of these issues, we did the following:
        - To deal with the different moments of inertia, we were able to manipulate the controller given our knowledge of the state-space representation until we had

certain constraints on the controller matrix; from here, we were able to tune the controller.

        - When the state-space model is discretized using larger and larger time-steps, it becomes more and more difficult to retain independence; in this case, the effects are not significant enough to cause us issues.
        - In order to deal with the constraints on the motor voltages, we use the polytope optimizations mentioned above and ended up making it such that the claw will always aggressively reduce separation error such that if it starts getting slightly off during a movement, the separation is quickly returned to normal.
- Feedback is provided by one quadrature encoder on each half of the claw and each claw is calibrated using three digital hall effect sensors (for a total of six digital hall effects on the entire claw).
- In order to avoid the time required to zero during autonomous, the claw allows for "autonomous" zeros which can be determined by hand-driving the joints past hall-effect sensors, from which the code can determine approximate calibrations—for finer calibration, the robot will follow a more elaborate routine.

Zeroing and Logic in the Claw & Shooter

In order to calibrate the claw and shooter, we use digital hall effect sensors which are at known positions; once we have passed over an edge with the hall effect sensor, we can determine an offset for the quadrature encoder which then allows for absolute control.

- **Shooter:** There are four hall effect sensors on the shooter. Two detect the position of the motor-driven stop, one detects whether the latch is down and one detects whether the shooter plunger is all the way back. When the robot starts up, depending on what set of hall effect sensors is triggered, it will move the stop at a capped voltage until it sees the edge on a hall effect sensor, at which point it can calibrate itself and operate at the full voltage.
- **Claw:** There are three hall effect sensors on each half of the claw, two near the front and back limits of the claws and one in the middle. Before autonomous of each match, we can manually run the claw halves over the edges of the sensors to allow it to calibrate itself based on what edges it sees pass. In order to ensure accurate zeroes during teleoperated mode, the robot follows a procedure where it first determines an approximate zero by running the claws over any edge and then going back to the middle sensors and running over exactly the same edge at the same speed every time, ensuring a consistent zero.

Other Notes

Unit Tests

By running unit tests on all of the individual systems in the robot, we are able to check whether changing the code in a certain way will cause any obvious issues in the functionality and safety of a given mechanism. We also adhere to test driven development practices where possible. Whenever the robot does something new which exposes a bug in the software, we read the log files to understand the bug, write a unit test to expose the bug in simulation, and then fix the bug in simulation. This means that once bugs are caught, they don't come back.

Improvements over previous years

In the 2012 and 2013 seasons, we used a Fit-PC as a secondary processor, rather than the BBB, but we ran into several issues which led us to switch (both run linux, so the code itself did not have to change much):

- *Expense:* The Fit-PC cost several hundred dollars and if one broke, it was expensive to replace; a single BeagleBone Black is ~$45.
- *Durability*: We occasionally had issues with the Fit-PC being insufficiently durable to survive and FRC environment. So far, the BBB has survived well through even relatively rough conditions; we replaced the BBB once during the season, at Championships, because we were noticing that some of the log files were getting corrupted, so we did not want to take the chance that there were any issues which could lose us a match later.
- *Reliability* in communication with the custom circuit board: On the Fit-PC, we also had a custom sensor board, but the USB interface we used to communicate with it tended to be unreliable; UART, which we use to communicate between the BBB and its sensor board, has proven more reliable. We have a robust custom protocol which very quickly recovers from errors.