Eagle Robot Design Document

Project Goal

Build a robot to compete in the First Tech Challenge for 2017-2018, placing well enough to compete in the Arizona & New Mexico FIRST Tech Challenge Championship in Flagstaff, Arizona.

Function

This section describes at a high level the functions the robot will perform.

Convert Joystick Input To Robot Heading

The Logitech game pad has two joysticks per game pad, which we plan to use for controlling the robot drive system. Each joystick provides a value between -1.0 and +1.0 on both the *x*- and *y*-axies. On the *x*-axis, -1.0 is to the far left, while +1.0 is to the far right. On the *y*-axis, -1.0 is toward the top, away from the player, while +1.0 is toward the bottom, closest to the player. Each position of the joystick gives a point (*x*,*y*) in a Cartesian plane.

The trigonometric function that computes the angle from the *x*-axis to the origin to a point on the plane is the *arctangent* function. In the Java programming language, it is the *atan2* function in the *Math* class. Now, because the sign of the *y*-axis is the opposite of what the *arctangent* function expects, we also have to adjust the inputs slightly.

Holonomic Drive System

A holonomic drive system allows the robot to drive forwards and backwards just like a car or a tank. But, unlike a car, it also allows the robot to move left or right without turning. The wheels are oriented in such a way that the force from the rotating wheels pushes in angles that are partly towards and partly away from the direction of travel. The trick is that the forces balance each other *except* in the direction of travel. This may be done by physically angling the wheels, as we have done with Omni wheels, or by designing the wheels to provide the force at an angle, as Mecanum wheels are designed to do.

Omni wheels are turned 45° inwards in the front, and 45° outwards in the back. When the wheels rotate in the normal direction for going forward, that is, counterclockwise on the left and clockwise on the right, the side forces balance each other, pushing the robot forward. Similarly, when the wheels turn in a backwards direction the side forces are balanced, leaving the robot to move backwards. A side motion happens when the two front wheels turn in the same direction and the two back wheels turn in the direction opposite from the front wheels. This is illustrated in the diagrams below.

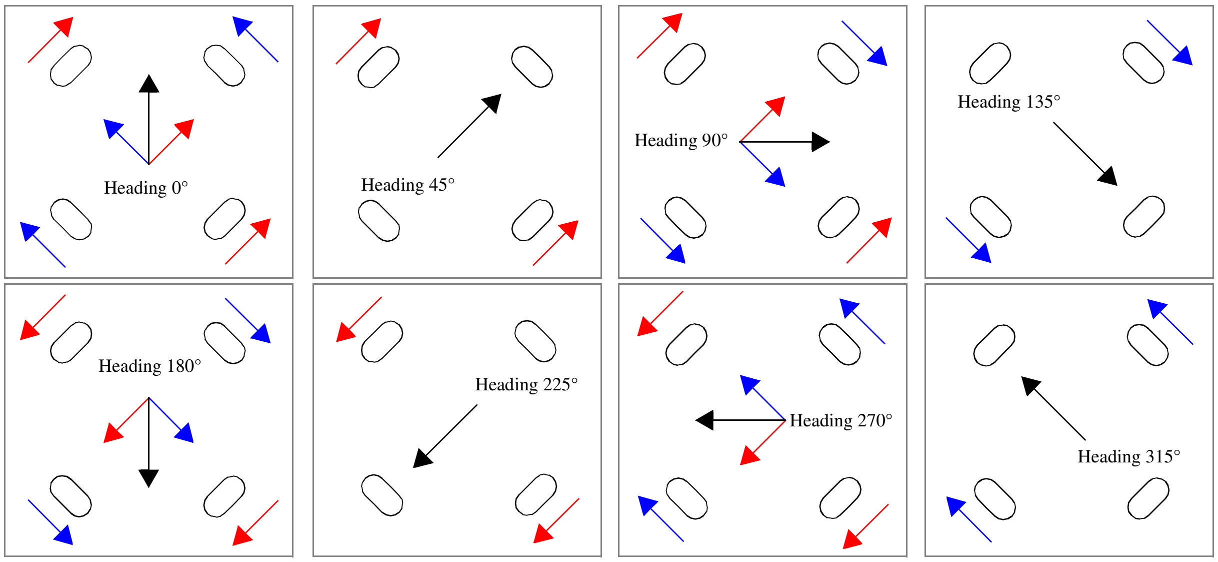

Figure 1 - Holonomic Force Diagrams

In the diagrams, the wheels with the red arrows turn in such a way as to push the robot in the direction of the arrows. The wheels with the blue arrows do the same. Adding the red and blue arrows together gives us the black arrow, which shows the robot's direction of travel.

The next step is to translate the heading we want to travel into power settings for each of the four wheels. We observed that *sine* and *cosine* functions seemed to have the behavior we wanted, so we used the force diagrams (above) with the Pythagorean Theorem to create the following table of headings and corresponding power settings. From that it was easy to identify which function was needed.

Table 1: Wheel Power Settings

Heading α	0°	45°	90°	135°	180°	225°	270°	315°	Function
Front Left	-0.7	-1	-0.7	0	0.7	1	0.7	0	$-sin(\alpha+45°)$
Front Right	0.7	0	-0.7	-1	-0.7	0	0.7	1	$cos(\alpha+45°)$
Back Right	0.7	1	0.7	0	-0.7	-1	-0.7	0	$sin(\alpha+45°)$
Back Left	-0.7	0	0.7	1	0.7	0	-0.7	-1	$-cos(\alpha+45°)$

Using these functions keeps the power setting between -1.0 and +1.0, which is required by the motors.

In the next figure we show the full translation from joystick input to drive motor power settings. The red angles are the heading of travel (or bearing) in radians, while the compass rose in the upper right corner shows the heading in degrees. The angles in black show the angles relative to the joystick. The dark blue square shows the joystick range of motion, while the light blue circle within the square shows the allowable joystick values. The drive motor power formulae are given in the lower right corner of the figure.

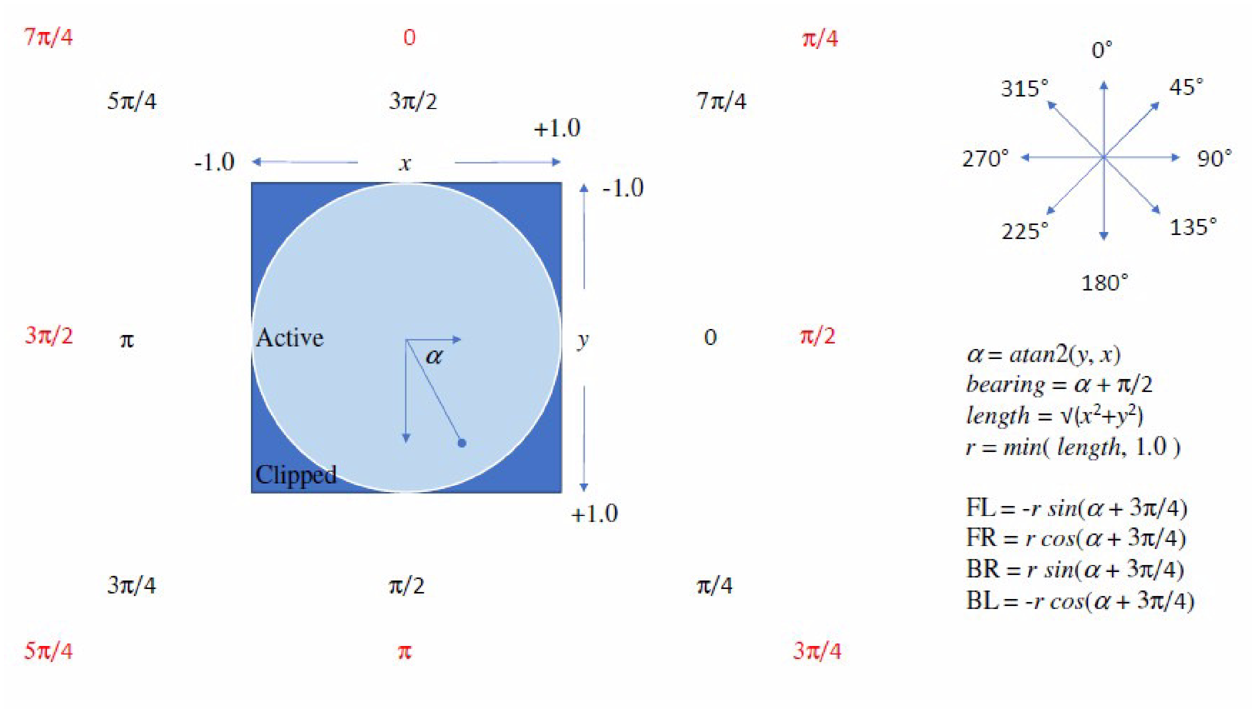

Figure 2 - Translation From Joystick Input To Drive Motor Power

Front Claw And Elevator System

Our plan to move crypto blocks in the game is to use a two-story claw and lift. The claw will be able to grasp, carry and lift one or two blocks at a time, high enough to stack four blocks in a column. The claw will have enough strength to hold the blocks in its grasp while the robot is moving and positioning the blocks. The lift will have enough strength to lift the claw and the blocks into a position where the blocks can be placed in the crypto box. According to the game rules, the claw and lift must fit within the 18” x 18” x 18” bounding box in *some* configuration. The lift can be lowered and the claw can be open or closed, as long as it fits in the box at the start of play.

Scorpion Tail And Color Sensor

Our solution to the part of the game where we dislodge a jewel from its stand is to use a long tail with a color sensor at its end. We call it a “scorpion tail” because it looks like the tail of a scorpion, where the color sensor is the stinger. In autonomous play, the tail lowers itself to sit between the two jewels. The tail will have a color sensor at its end that senses which of the two jewels matches the color of the opposing alliance. The robot will then swivel on its base to knock off the correct jewel, raising the tail into a safe position when it’s done.

Beam With Swinging Claw

We had also intended to mount a claw on an extendable beam that we could use to place the relic over the wall. The beam would need to lift the relic so it clears 12” to get over the wall, and the beam should extend up to 36” past the wall. The claw would swing the relic upwards, making it horizontal to the floor, rather than lifting it straight up to clear the wall. We completed most of the beam assembly, but ran out of time before it was completed.

Design

Mechanical Design

The robot base is a square frame made from 160 mm aluminum C-channel structural elements. The corners are 96 mm C-channel components below, with a 96 mm flat bracket cover on top. The motors mount to the under side of the 96 mm C-channel corner elements.

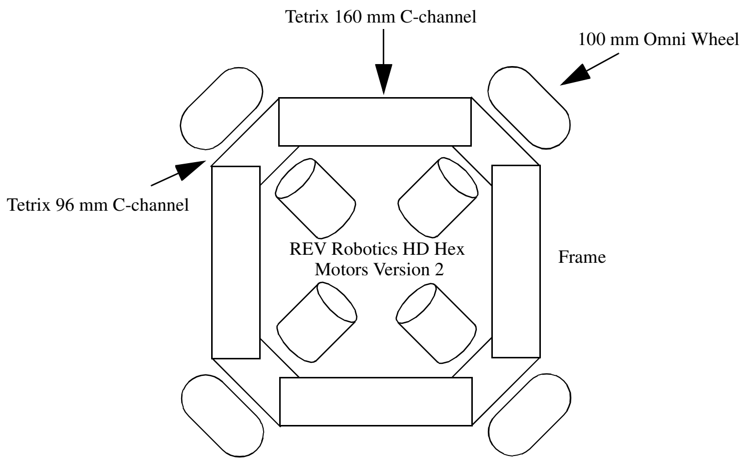

Figure 3 - Robot Base And Drive

The robot claw uses a worm gear in a worm gear box to open and close the claw, with a two-stage rack and pinion design for the lift. The claw is powered by a HITech HS-485HB servo. The claw lift mechanism is powered by a REV Robotics HD Hex Version 2 motor.

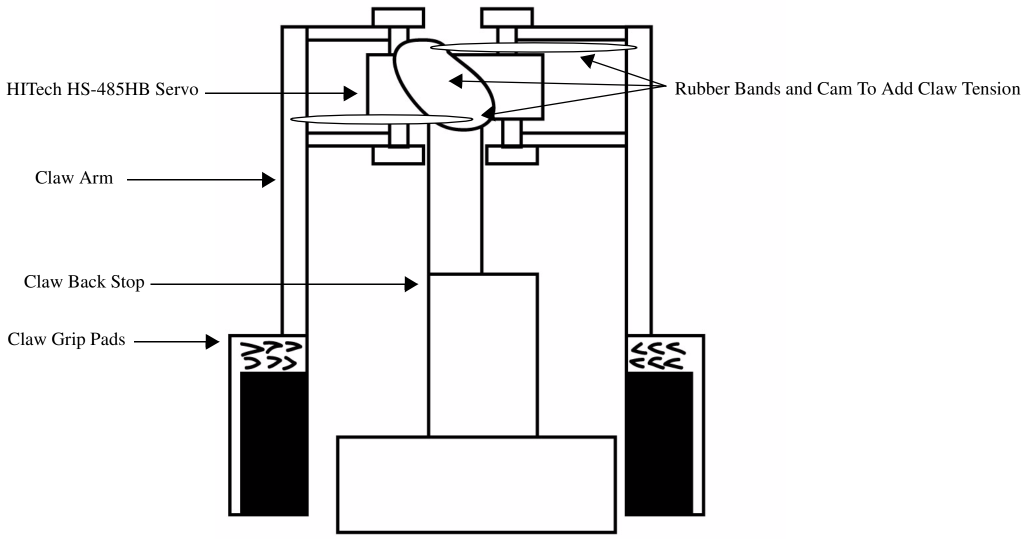

Figure 4 - Robot Claw Front View

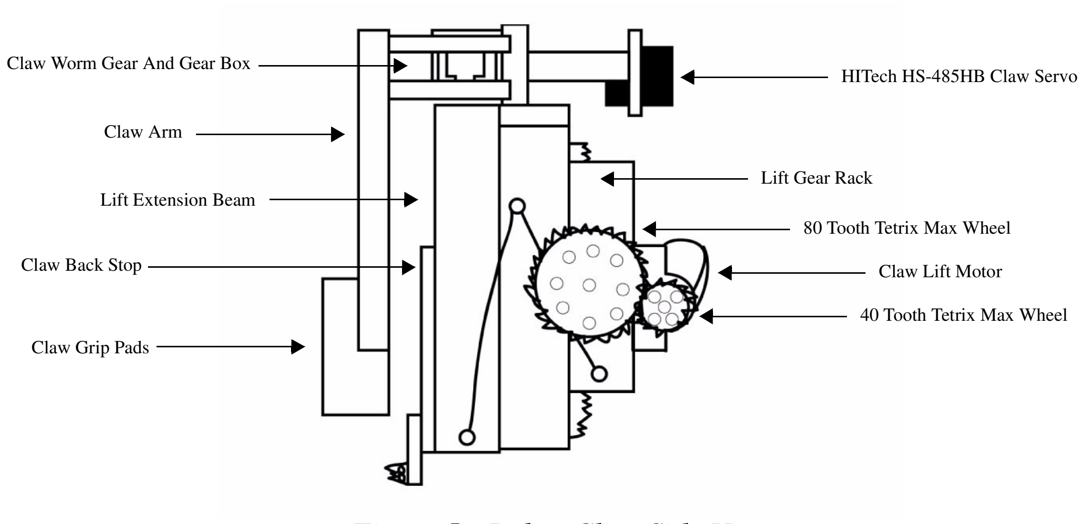
Claw Worm Gear And Gear Box
HITech HS-485HB Claw Servo
Claw Arm
Lift Gear Rack
Lift Extension Beam
80 Tooth Tetrix Max Wheel
Claw Back Stop
Claw Lift Motor
40 Tooth Tetrix Max Wheel
Claw Grip Pads

Figure 5 - Robot Claw Side View

The robot “scorpion tail” is a hinged design that fits within the 18” limitation by folding upon itself when it is in the upright position. It consists of a HITech HS-485HB servo with a 40-tooth cog wheel attached to its axle. The cog interlocks with another 80-tooth cog wheel. Turning the servo clockwise lowers the tail. The cog wheels protect the servo from being damaged in case the tail gets caught and bent. Without the wheels, the tail could be pulled from the axle, stripping the threads and damaging the servo.

The larger cog wheel is attached to the longer extension of the tail. The opposite end of the extension is loosely attached to a shorter extension, which holds the color sensor. The shorter extension is able to swing down under its own weight as the tail is lowered. The color sensor falls into position where it can read the color of the jewel it is facing. If it faces a jewel of the opposite color as its alliance the robot turns towards the jewel, knocking it off of its stand. If it faces a jewel of the same color it turns away from the jewel, knocking the opposing alliance’s jewel off of its stand.

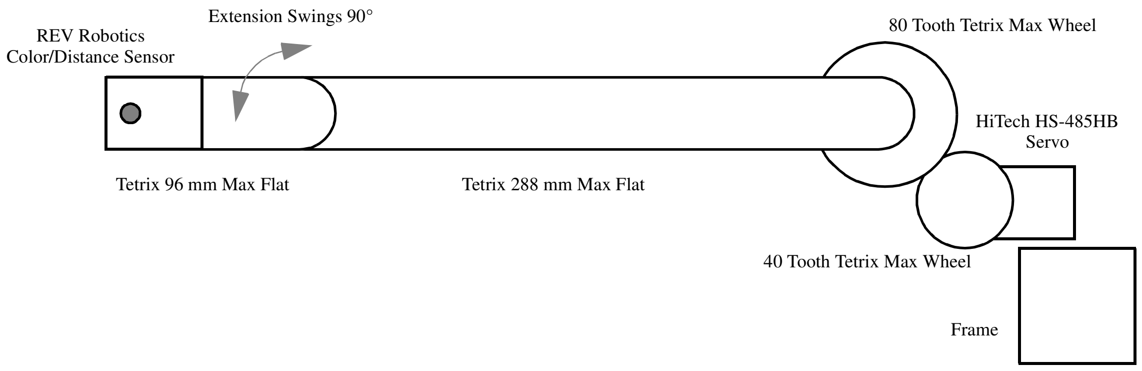

Figure 6 - Robot Tail, Color Sensor And Drive

Electrical Design

The robot electrical design is built around the REV Robotics Expansion Hub. The hub provides ports for four (9.6v to 12v) motors with encoders, six 5v servo ports, two +5v power ports, four analog ports, eight digital ports, and four I2C ports. There are also two RS485 ports for connecting additional hubs and two UART ports for debugging. (See “REV Robotics Expansion Hub Guide -- Rev 4,” http://www.revrobotics.com/content/docs/REV-31-1153-GS.pdf, for details.)

Our robot uses two expansion hubs in its design. The primary expansion hub (Hub 2) is used to control the four drive motors and two (claw and tail) servos. The additional hub (Hub 3) is used to communicate with the Robot Controller Phone and control the lift motor. The ports selected and the names given to each port in the robot configuration file are given in the next figure. For the sake of reliability and ease of assembly, all cable ends are marked with the port where they are to be connected.

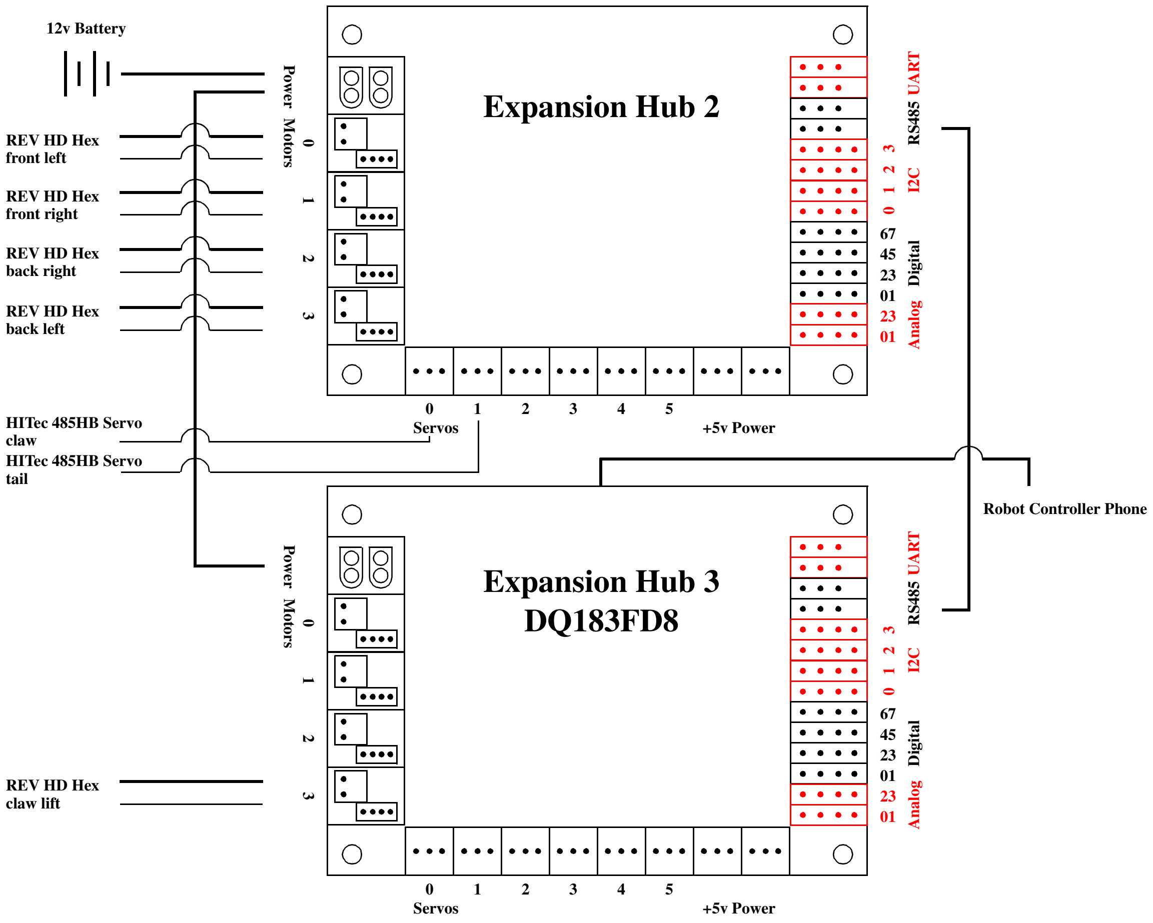

Figure 7 - Electrical Component Block Diagram

Program Design

The program has three components, namely the autonomous operation mode (or "op mode"), the driver (or "teleop") op mode, and a diagnostic op mode. We wrote all of our op modes in the Java programming language using the FTC library and Android Studio.

Autonomous Op Mode

The autonomous operation mode runs through a short list of problems to solve with no driver input. The robot may be told which alliance it belongs to (i.e., red or blue) and which quadrant of the play field it is in, but everything else must be done by the robot.

We had a choice of programming the autonomous op mode using either a "linear op mode" or an "iterative op mode". A linear op mode programs the robot actions in a linear (or sequential) way. The program executes a section of the code addressing one of the challenges of the game, after it's done with that challenge it falls into the next section of code, which addresses the next challenge, and so on. In contrast, an iterative op mode executes within a loop. The loop executes about every 1/10th of a second, during which the robot must make a decision about what it should do. While it is possible to write the autonomous mode in this way, and it may even be easier in some cases, we chose to write our autonomous mode using a linear op mode.

Our autonomous op mode also uses a unique feature. We created an abstract language to program our robot and an interpreter to execute that language. Our language consists of an operation code (or "op code") and a list of arguments specific to that operation. It's like an assembly language for robots. Each instruction is an array of Java type Object, and a program is an array of Object arrays. Organizing the program as an array of Objects makes it (almost!) easy to create the interpreter. The syntax may be a little unusual, but we can easily create a separate program for each quadrant (e.g., blue left, blue right, red left, and red right). Each quadrant's program can handle the unique sequence of actions in a few lines of code and in a very readable way.

The benefits we see from our abstract robot control language are:

1. More compact code. Our autonomous op mode is written in about 750 lines of code. If we expanded each operation back into its native Java, we estimate it would need around 2,000 lines of code.
2. More readable code. The instruction for each step of the autonomous mode fit on a single line. Op codes say what the robot will do in a clear way, such as "move forward 12 inches."
3. The code is easier to maintain. It is much easier to add or change a single robot instruction in a program of 10 instructions that relate to the robot than to add 30 Java statements to a program of 2,000 Java statements. (We did it both ways!)

An example program for the blue alliance, left quadrant is given below.

```java
private static final Object[][] blue_left_cmd  {
    {Command.ROTATE    90.0, AUTO_PWR, AUTO_TOL, ROTATION_RATE     },  // turn 90 degrees towards crypto box
    {Command.FORWARD   18.0, AUTO_PWR, AUTO_TOL, BALANCING_STONE    },  // move off the stone - 18.0"
    {Command.FORWARD    2.0, AUTO_PWR, AUTO_TOL, PLAYING_FIELD      },  // calibrate position
    {Command.BACKWARD   4.0, AUTO_PWR, AUTO_TOL, PLAYING_FIELD      },
    {Command.FORWARD    6.0, AUTO_PWR, AUTO_TOL, PLAYING_FIELD      },  // align with top of triangle
    {Command.STBD      15.0, AUTO_PWR, AUTO_TOL, PLAYING_FIELD     }, // move starboard to crypto box center
    {Command.ADJUST     6.0, AUTO_PWR, AUTO_TOL, PLAYING_FIELD      },  // adjust for VuForia VuMark
    {Command.FORWARD    5.0, AUTO_PWR, AUTO_TOL, PLAYING_FIELD      },  // place glyph in crypto box - 5.0"
    {Command.OPEN_CLAW                                              },  // release glyph
    {Command.BACKWARD   5.0, AUTO_PWR, AUTO_TOL, PLAYING_FIELD      },  // move away from crypto box - 5.0"
    {Command.CLOSE_CLAW                                             },
};
```

The language also accounts for the surface the robot is moving on. In the example, the second instruction tells the robot to move forward 18" and that it is on the balancing stone. The balancing stone is a more slippery surface than the playing field, so the robot has to account for that in its movements. The next statement tells the robot to move forward 2" and it is on the playing field, which has better traction.

Using a linear op mode makes the program pseudocode simple.

```
// initialize the drives and sensors
// select the alliance color and playing field quadrant
// wait for the start of the game
// reset the clock
// close the claw
// lower the tail
// read the jewel color
// dislodge the opposing alliance jewel from it stand
// execute the program for this quadrant
```

The last statement, "execute the program for this quadrant," looks like magic, but it is really a very simple loop that hands each one of the robot instructions to a command interpreter. The interpreter then executes the dozen-or-so Java statements that accomplish the instruction.

Driver Op Mode

The driver op mode takes commands from the two drivers using the game pads, and translates those commands into actions of the robot. Unlike the autonomous op mode, this op mode is more easily programmed using an iterative op mode. Each iteration allows the robot controller to poll each input and device, and make a decision about what needs to change. For example, if the left joy stick is pressed forward, the controller can read the joy stick position and translate that position into power settings for the drive motors. Each iteration must complete quickly, though, or the loop will throw a time-out error.

Iterative op modes have five functions that must be provided, namely, `init()`, `init_loop()`, `start()`, `loop()` and `stop()`. We only used the `init()` and `loop()` functions, though. The others kept their default values. The pseudocode for our two functions was quite simple.

```
// void init()
    // initialize the drive motors
    // initialize the claw servo and motor

// void loop()
    // get drive settings
    // get claw settings
```

Initializing the drive and claw motors/servos was straight forward. The direction was set to "forward" for all devices, and the motors were set to "run using encoder". "Get drive settings" implemented the holonomic drive code described in the section titled "Holonomic Drive," above. "Get claw settings" mapped individual game pad buttons to servo settings for the claw, and power settings for the claw lift motor.

The drive motor power settings had another complication. Sometimes we needed the robot to move quickly, other times we needed it to move very precisely. To solve this problem we used the two joy sticks on one of the game pads to control the robot at different speeds. We first calculated the power settings using the fast (left) joy stick, then if the precise (right) joy stick was in use, we re-calculated the settings using that joy stick, scaling down the power to a lower range. In that way, the precise motion always took precedent over the faster settings.

Diagnostic Op Mode

We created an additional op mode that we used to check out the hardware on our robot. We also used it, to a lesser degree, to check out our software ideas. The diagnostic op mode is a simple state machine where each state checks out a single device or device group on the robot. Nothing was initialized until it was used, and that prevented the program from aborting over devices you're not interested in checking out. This op mode was useful not only during robot development, but also during repairs after some component failed and had to be adjusted or replaced, and even for just checking out whether the robot was working correctly before a competition.

Configuration File

There were many constants throughout the program, and some of the same constants, such as encoder or servo settings, were used more than once within an op mode. What's more, those same constants appeared in each of the three op modes as well. A problem we faced was how to keep the different uses of these constants clear and up-to-date. We solved this problem by giving each of these constants a clear name and pulling it into a separate Java class that was imported by each of the diagnostic op modes. Giving each constant a name made it easier to understand why it was there and what it was used for. Pulling it into a separate class allowed changes to be made in a single location and have those changes propagate to all of the op modes that needed it, instead of having to change it in each op mode separately and possibly getting it wrong in some of them.

Testing

Our testing was extensive but perhaps a little less formal than it could have been. During development we tested each device individually as well as their operation together as a group. For example, we first tested the ability to apply power to a single drive motor, then as we got the holonomic drive system, we tested the balance of all drive motors working together as a group. The diagnostic op mode helped us isolate problems with individual components, while many runs of the robot in autonomous mode helped us refine our mappings of encoder values to distance on the balancing table and playing field.

Analysis

Our analysis of our tests fell into two categories: how well does it work, and what are the mappings of inputs to outputs. For example, early on we tested different motor settings using the diagnostic op mode to see whether they did what we hoped they would. Does this setting rotate the motor axle a certain number of times? How accurately does it make that many turns? Is it always exact, or is it off by a few turns? This is an example of "how well does it work?"

Another test we did was setting the drive motors to turn a certain number of times, using the encoders to measure the number of turns, and see how far the robot traveled in a certain direction. How far did it go? Was the distance the same each time? Did the surface matter? This is an example of mapping inputs (e.g., encoder settings) to outputs (distance on a surface).

In each case we used the results of our analysis to reexamine our assumptions about our design. We could have used it to re-think our goals and functional specifications, too.