The Team

We were team "Get the job done", featuring Darryl Hill on Station 1, Me on Station 2, and Frank Perks on Station 3. We each played the following roles:

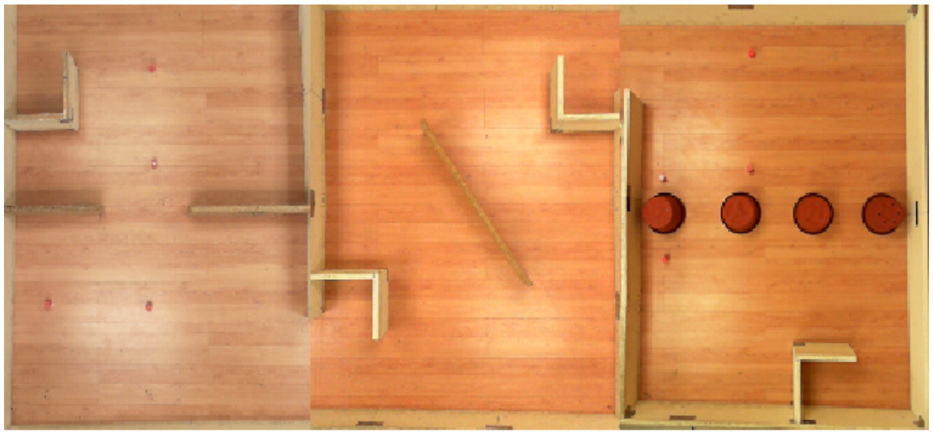
1. Darryl was in charge of finding blocks nestled amongst the upside down flower pots. This involved goal navigation guided by the Tracker, wall following and collision avoidance around the flower pots, and block sensing by the robot to find blocks.
2. My job in the middle station was to move blocks delivered by Darryl to be dropped off to the bottom where they could be picked up by Frank, and to also move blocks delivered by Frank and drop them off where Darryl could pick them up. This involved mostly goal following from the Tracker, along with some block sensing, and lots of dynamic course correction.
3. Frank's task was similar to Darryl. He had to find blocks laid out in his section which also was bifurcated by a wall with a small gap. Frank had to use wall following and collision avoidance to avoid smacking into the wall.

The Competition

The competition environment was set up as in the image below. There are pots to navigate in Station 1, a board and corners to avoid in Section 2, and there's a wall with a small gap to navigate in Station 3.

The gist of the task we had to accomplish was to exchange blocks from Station 1 and Station 3. As I was in charge of the middle station, I was in charge of doing the exchange between the top and bottom stations. My robot was essentially the fulcrum of the operation. The completion of the task depended on my robot.

As both team members were supposed to drop off 4 blocks each, I would have to deliver 8 in total: 4 from the top to the bottom, and 4 from the bottom to the top. All the while avoiding collisions with the environment.



My Approach

The approach our team aimed for was *extreme simplicity*. At first, we were trying to decide how best to accomplish the task. Some obvious issues with coordinating such a task (as described above) are problems with having a block for my robot to deliver (that is, if my robot wants to pick up a block from Station 1 and bring it to the bottom for Station 3, he has to know there IS a block to be picked up!), and making sure the robots wouldn't collide if there was such a block available. How would we know when Frank's robot had come to drop off a block and how would I avoid colliding with his robot before he'd safely exited my station's area?

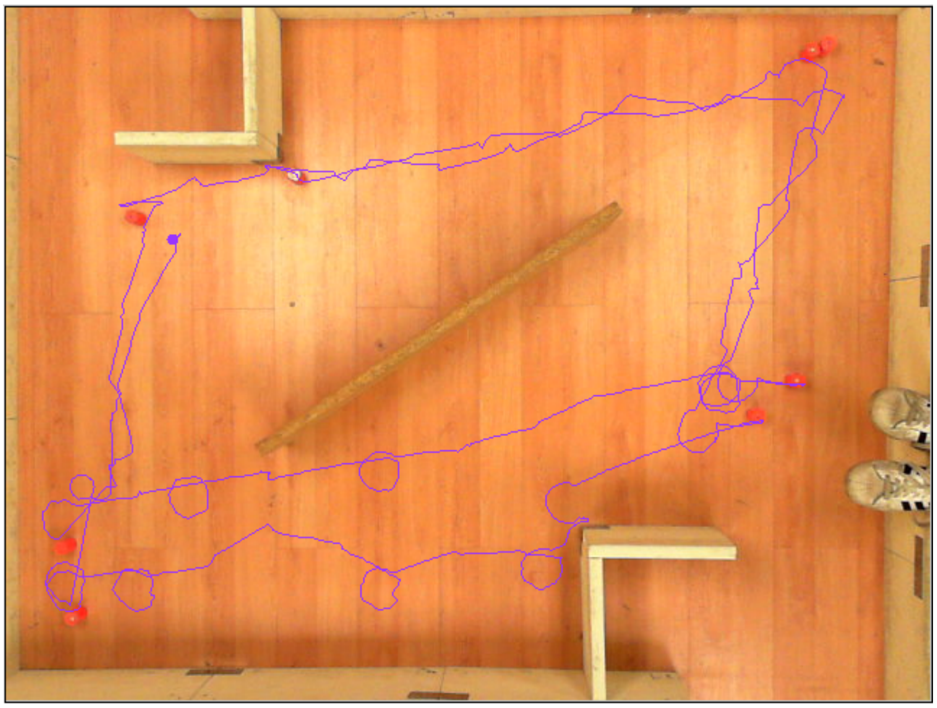
We decided to take the simplest possible approach, at the expense of completing the task in the allotted time. The approach was simple: Do everything in a lockstep way. There are three main stages to this appraoch:

1. Robots 1 and 3 hunt for their blocks and deliver them one by one to the designated areas in Station 2, informing my Tracker the exact drop off location of each block (and from which station the message came). While this is happening *my robot was to be completely idle*, safely out of the way of either of the other robots.
2. After each of Robots 1 and 3 had delivered 4 of their blocks, they were to return to their stations, and become idle. Then and only then, my robot in station 2 would begin its task. It would first **Goal Navigate** using the Tracker to the most recently dropped off block from Station 1 (we had the blocks dropped off farthest first, so that I'd pick them up in reverse order, so I wouldn't knock down any other blocks in the way). When my robot was close enough to the Goal (given a small distance threshhold so it didn't have to be exact), it would orient itself so it was angled properly at the Goal.

   It would then go into `Block Sensing Mode` and try to pick up a block. After it found the block, it would then Goal Navigate again, this time to a designated drop off location for Station 3 to pick up later. The robot would also message Station 3 with the exact location of this drop off. Then the robot would start **Goal Navigating** again, to get the most recently dropped off block from Station 3 (just like how it started by navigating for Station 1 blocks earlier). When it reached its goal location, it would again orient itself, enter block seeking mode, and capture a block. With its block captured, it was to **Goal Navigate** again to the Station 1 drop off area, drop off the block, and inform Station 1 of its location.

   After completing one such cycle, it was to check to see how many more blocks were remaining. If there were no more blocks remaining, it was to Goal Navigate out of the way again, and inform the other Stations. This was a completion of its task. If it *had not* completed its task, it was to make another cycle and keep checking, etc.

3. After Station 2 had finished its task and the other 2 robots had been notified, those robots would begin collecting the dropped off blocks and returning them to their stations. While this was happening, my robot was to remain idle. He was done his job.



Implementation Details

For the implementation of my Station's task, there were three main parts: The `MiddlePlanner.java` code, which handled most of the robot's logic; the `Navigate.spin` code which handled following Tracker instructions on the robot itself; and finally a messaging protocol established with the rest of the team for knowing when certain robot events have occurred and where (e.g., "Robot 1 dropped off a block at x,y"). Each of these will now be explained in detail.

MiddlePlanner

The tracker is where the majority of the work was done for my Robot's task. The gist of it is I used the planner to do **all** robot related logic, and then I would send these logic commands to the Robot itself, who would then perform the tasks and report back.

The Planner code was implemented as a finite state machine, with the following possible states:

```
public enum RobotState {
    RobotStateInvalid, // Used as default value before a state is set
    RobotStateStart, // The entrance state..robot waits to begin
    RobotStateSeekTop, // Goal navigate to Top pickup location and orient to it
    RobotStatePickTop, // Robot should be looking for a block and snatch it if found
    RobotStateDropBottom, // Robot navigates to the Bottom drop off target
    RobotStateDoActualDropBottom, // Robot performs the block drop and backs up slowly
    RobotStateSeekBottom, // Robot Navigates to the bottom Pickup Zone and orients
    RobotStatePickBottom, // Robot locates a block and picks it up
    RobotStateDropTop, // Robot navigates to the Top dropoff goal
    RobotStateDoActualDropTop, // Robot drops off the block at the top and backs up
    RobotStateFinishedCycle, // Evaluate if we're done. If not, do another cycle
    RobotStatePark, // If done, move out of the way and stop
    RobotStateEnd // Finally parked, shutdown and tell other stations
}
```

The robot stays in `RobotStateStart`, which is idle, until both Stations 1 and 3 have told me they have dropped off 4 blocks each and their locations.

The basic runloop of the Planner code is recording the Robot's latest pose in `receivedPoseFromTracker(Pose p)`. It adds the pose to a list of poses maintained by the class. The main logic then occurs in the `receivedDataFromRobot(int[] data)` method. Here's the way it works:

1. On the Robot, it enters a runloop forever, and it clears out some flags, then it sends some data to the planner, either reporting what it's just done OR asking the Planner for its next task.
2. The `receivedDataFromRobot(int[] data)` gets called on the Planner, which is essentially one large `switch` statement, with a `case` for every possible robot state (this is how the transitions between the robot's finite state machine are handled).
3. The Planner decides what the robot needs to do next. It looks at the most recent robot Pose and the current state, and determines what the robot should do, whether that's move, rotate, or enter a new mode (like block seeking or block dropping, for example). When the Planner has decided what to do, it responds by sending the robot a set of bytes back and it returns from this method.
4. The robot just sits and waits for a response from the Planner after it asks. Then it reads what the Planner has told it to do and it performs the task. More detail will be given below in the robot's code section.

This means the Robot does very little logic on its own. The Planner code is what determines what needs to be done. The robot is essentially a dumb terminal performing these tasks. In order to avoid things like wall collisions, the Planner code sends very small instructions to the robot very rapidly, so the robot never has to come near a wall. The move or rotate instructions are for small amounts of movement, with the idea the robot will request many of these as it's moving. This allows for really quick and precise course corrections while Goal navigating, for example.

When the robot was in a state where it needed to *Goal Navigate*, it called a single method to perform the navigation. Having all goal navigating states refer to a single Navigation method meant my code could be changed in one place and every place it was needed automatically got the new functionality. This made debugging much simpler. It also gave me consistent behavior.

The method also allowed me to transition to the next state when the navigation had completed and oriented properly. When doing this, it would send a message to the robot to just "ASK_AGAIN", meaning the robot would do nothing, loop, and just make another request to the Planner. When this new request came in, the Planner's finite state machine had already transitioned to the next state. I found this was an elegant way to transition states and keep the robot and planner in sync. The method looked like the following:

```
void goalNavigateAlongCurrentPathForRobotPoint(Pose pose, RobotState nextState, boolean forceNextState) {
    // This is Goal-navigation.

    // First see if he's close enough to the goal, in which case
    // transition to the next State.
    Point robotPoint = new Point(pose.x, pose.y);
    Goal currentGoal = goals.get(0); // Only dealing with 1 goal. If paths had more goals, change this

    if (currentGoal.isPointCloseEnoughToGoal(robotPoint)) {

        // Make sure we're properly oriented to the goal so we can just move forward when seeking a block. Then transition in this method.
        if (forceNextState) {
            // No need to orient
            // Change state and reset some internal flags.
            setRobotState(nextState);

            // Just so we have some instructions to reply with
            sendInstructionsToRobot(MOVE_NONE, ROTATE_NONE, ASK_AGAIN);
        } else {
            // Not done navigating.. still need to orient to the goal.
            orientToGoalForPose(pose, currentGoal, nextState);
        }

    } else {
        // We need to tell the robot to (keep) moving towards the Drop Goal.
        int robotCurrentAngle = pose.angle; // degrees
        int angleToTheGoal = (int)getAngle(currentGoal.location, robotPoint);
        int distanceToGoal = distance(currentGoal.location.x - robotPoint.x,
                                      currentGoal.location.y -robotPoint.y);
        System.out.println("Distance to goal is: " + distanceToGoal);

        int RANGE = 5;
        byte rotation = ROTATE_NONE;
        byte movement = MOVE_NONE;

        int angleDifference = angleToTheGoal - robotCurrentAngle;

        if (Math.abs(angleDifference) < RANGE) {
            rotation = ROTATE_NONE;
            movement = MOVE_SMALL;
        } else if (angleDifference > 0) {
            rotation = angleDifference < 180? ROTATE_BACK_SMALL : ROTATE_SMALL;
            movement = MOVE_NONE;
        } else {
            rotation = ROTATE_SMALL;
            movement = MOVE_NONE;
        }

        // Now send this command to the robot
        sendInstructionsToRobot(movement, rotation, GO);

    }
}
```

Orienting towards the goal worked in a similar way, as can be seen in the Planner file.

When transitioning to a state where the robot would need to goal follow, I also invoked the `computePathFromRobotPoseToEndGoal(Pose currentRobotPose, RobotState state)` method so that a new path could be computed. This path would guide the robot from its current location to its destination (e.g., picking up a block to where it needs to be dropped off). In this method, I used a really simple way for making a "path" which really only had a destination (like taking the next pickup location, for example). I could have made this more sophisticated for better dodging obstacles (like creating a safer path), but I went for utmost simplicity in the name of just getting things done.

Navigate.spin

This was the code which ran on the Robot, which was basically responsible for listening for commands from the Planner code and executing them. It had a main runloop where it would perform its most recent task if any (e.g., Rotate, move, seek a block, drop a block, etc.), and then it would report back to the Planner, either saying which task it had just completed, or it would ask for a new task. It would immediately stop and wait for a response from the Planner before doing anything else.

After the planner did its logic as described above, the Robot would read the response and perform any task needed, and the loop would happen over again.

This continued until the robot received the Complete message, in which case it would beeb a shutdown sound and exit the runloop.

The movement commands are fairly straightforward, just entering a loop and either moving or spinning as required.

The Block seeking code is a little more involved. It first opens the robot's grippers and tilts the head down, and then it must use the camera to try and find a block.

In a loop, it searches until it finds a block. If given more time, it would have been a good idea to integrate a timeout to this code, so if the robot didn't find a block after so long, it would just return a failure condition. Instead, my robot will look until the end of time (or its battery dies, whichever comes first!). This is less than ideal. On every iteration of the searching loop, the robot uses its block sensor to see if it's got a block. In this case, it breaks from the loop, closes its grippers, sets an internal flag for reporting back to the Planner, and returns.

If it doesn't detect a block, it uses the camera to see if it can find a block. Because the robot has already been oriented towards the block (hopefully) at this point, then it should quickly find the block. It uses the camera data to see if it needs to move to the right or the left, or just move forward to find the block. And then it loops again.

Message format between stations

The message format we agreed on to talk between stations was pretty simple and can be seen in the `StationMessage.java` and `MessageType.java` classes included below.

Essentially, we've used specially formatted strings which can be read and created, sent among the stations, indicating what kind of message was sent (like "Station 2 dropped off a block to the bottom at (x, y)").

There was a significant problem with this, however. **The Tracker was unreliable at delivering messages between stations 1 and 2**. This means I could not always get the corret messages from Darryl's station. It's incredibly frustrating because there's no way around that. I realize in this course there are "errors" with the robot, and the Tracker finding the robot. That's understandable because there exists noise, and the solution is to either smooth data or work around by asking again, etc. But these errors, as far as I can tell, are just bugs in the Tracker which I can neither fix (we're not allowed to modify the Tracker) or work around (how am I to reliably get messages from another tracker when the only software interface to accomplish this task is broken? My robot has absolutely **no way** to work around this problem and this should not be penialized).

The code is all below in the "Software" section.

Problems Encountered

The biggest problem we encountered was a lack of time and coordination. Not only were we limited for lab time, but of course, as students we each had our share of other assignments which ate away at our time in the lab, too.

We also lacked time when actually running the competition itself. Using our super simple lockstep method, of course we didn't have enough time to complete the whole task in the 15 minutes allotted. My robot didn't even get a chance to perform his task before the time ran out.

My plan was to get our task completed and then optimize it where we could so we'd get more done in the 15 minutes, while keeping things as simple as possible. It was hard enough just getting the SIMPLE version of our robots working, let alone trying to do anything complicated like interacting all three robots at the same time. I've learned from my other programming courses that Simple is the best, so we wanted something as elegant as possible. In the end, it didn't prove fast enough. Given more time for us to prepare, I'm convinced we could have tweaked our solution enough to finish within the 15 minutes we had to do our task.

As for my Robot, I had several problems, though thankfully most of them were minor. The biggest problem was again timing. I did my best to take my time and *think* out all my code as best I could before writing it. I drew out my finite state machine on paper so I could visualize exactly what the robot had to do. I took my time and care writing the Planner code to follow this.

Originally, I had all my logic in the `receivedPoseFromTracker()` method, but I found when sending responses to the robot, the RBC would crap out on me and I'd lose messages, so my robot would miss my commands and end up spinning or moving forever and that was very frustrating.

Instead, I re-wrote my code to only send a command to the robot exactly when he had explicitly asked for a new command, by doing this in the `receivedDataFromRobot()` method. This simplified things even more, and made sure the robot never missed a command. But this a long time to re-write properly.

As for other hardware problems, the biggest problem was getting the robot to properly detect blocks. In the end, I don't feel like I had an entirely satisfactory solution, but it worked well enough. The best I could do was to aim/orient my robot to be exactly in line with the Goal location for the block (this goal was to be given to me by another Tracker Station). This helped by giving the robot better odds at being directly aligned with the block, and thus much more likely to detect it. Before adding the orientation code, my robot would very much likely not find the block at all.

The Tracker was also a source of much agony throughout the project, having to constantly manually reload the Planner code (suggestion: update the Tracker so that it remembers the last file loaded, and have it poll to see if the modification date on that file changes -- reloading it automatically if so. That probably would have given me an extra hour or two onto my life which has otherwise been wasted constantly reloading the same class over and over again!). It also had difficulty trying to locate the robot when updating poses. This meant at times my robot would sit idle for many seconds, as the Tracker waited for a new pose to arrive to better command the robot with. If the robot was in a darker area of the map, this would severly slow it down.

If I Could Turn Back Time

I'm confident in my code and the solutions I've found to most of these problems. They're all on the right track. All I could do better would be to have many, many more hours given to me for lab time, and having no other course work to eat up the rest of my time. I believe I had good solutions, but they needed tweaking, and that needed more time.

Also, because I was trying to coordinate with two other members, and because we each had our share of troubles, we never were able to fully test the complete running solution to the code. By the time I left on the last night of lab time, one of my team members was still writing code... he would not have a solution until nearly the end of the night! That wasn't possible to test against. It was a problem of coordination, as is to be expected in a project like this.

We could have improved our team approach again by tweaking things. My robot could have moved faster and with better paths. We also could have introduced very simple *concurrency* among the robots to make more efficient use of our time, but again, we didn't want to do this until we had the simple base-case working first.

Robot Hardware Problems

Most of the hardware problems I encountered were already described above. The encoders were never once correct for me, but to overcome this I just had to tweak them every single time I came into the lab. Frustrating, but I dealt with it.

The RBC was also flakey at best for me through the whole term. It's extremely easy to swamp it with data and have it essentially crash on you.

In the end, I found it was easiest if I did the least amount of work possible on the Robot. Spin code is attrocious and nearly impossble to debug. I found things were simpler if I did my work on the Planner code and simply made the robot do its biddings.

Video of the Robot

The video of my robot doing its task can be found here: https://vimeo.com/40411292 I ended up hard-coding values into the planner in order to record this video. You can see this happening in the constructor of my Planner code. I spin off a thread which waits for a few seconds. When the thread fires, it sends the Planner messages, simulating messages coming from the other Trackers about the positions of Blocks being dropped off.

I do it this way to best simulate what would actually happen if interacting with the other robots. While the thread is paused, the robot is actually running, and he's in the StartState, so he waits until after he's got all the blocks announced and ready to be dealt with. Then he starts to move.

While recording this video, after 2 complete cycles the robot dies (another random hardware problem???) but I ended the video there. There was another team coming in after me and I didn't have a chance to make another recording. As you'll see in the code, the robot would have continued to do 2 more cycles before completing his task, and they'd look identical to what's shown in the video.

Software

The software which is included below can also be found at: https://github.com/jbrennan/COMP4807

MiddlePlanner.java

```
import java.util.*;
import java.lang.Math;

public class MiddlePlanner extends Planner {

    // byte indices
    final int ROTATION = 0;
    final int MOVE = 1;
    final int CONTROL = 2;

    // Movement instructions
    final byte MOVE_NONE = 0;
    final byte MOVE_SMALL = 1;

    // Rotation instructions
    final byte ROTATE_NONE = 0;
    final byte ROTATE_SMALL = 1;
    final byte ROTATE_BACK_SMALL = 2;


    // Command instructions
    final byte STAY_STILL = 0;
    final byte ASK_AGAIN = 1;
    final byte GO = 2;
    final byte SEEK_BLOCK = 3;
    final byte DROP_BLOCK = 4;
    final byte ALL_DONE = 100;


    // From the robot
    final int STATUS = 0;

    // Status codes
    final int STATUS_COMMAND_REQUEST = 0;
    final int STATUS_BLOCK_FOUND = 1;


    final int TOTAL_BLOCKS = 8;



    public enum RobotState {
        RobotStateInvalid,
        RobotStateStart,
        RobotStateSeekTop,
        RobotStatePickTop,
        RobotStateDropBottom,
        RobotStateDoActualDropBottom,
        RobotStateSeekBottom,
        RobotStatePickBottom,
        RobotStateDropTop,
        RobotStateDoActualDropTop,
        RobotStateFinishedCycle,
        RobotStatePark,
        RobotStateEnd
    }
```

```
boolean firstPose;

ArrayList<Pose> poses;
ArrayList<Goal> goals;


// The goal areas for where I pick up the cylanders from other bots
Goal topPickupZone = new Goal(new Point(0, 0));
Goal bottomPickupZone = new Goal(new Point(480, 240));

Goal topDropZone = new Goal(new Point(0, 240));
Goal bottomDropZone = new Goal(new Point(480, 0));


RobotState currentRobotState;

int _numberOfBottomBlocksReady;
int _numberOfTopBlocksReady;

int _numberOfDeliveredBlocks;


//Goal currentGoal;
//int currentGoalNumber;

ArrayList<Goal> topPickupGoals;
ArrayList<Goal> topDropoffGoals;
ArrayList<Goal> bottomPickupGoals;
ArrayList<Goal> bottomDropoffGoals;


// Flags
boolean _allDone;
boolean _haveSentRobotCommand;
boolean _latestCommandAcknowledged;
boolean _robotFoundBlock;

boolean _bottomZoneUnlocked, _topZoneUnlocked;

// Constructor for the planner
public MiddlePlanner() {

    firstPose = true;
    poses = new ArrayList<Pose>();
    goals = new ArrayList<Goal>();

    Pose[] poses = getUserDefinedPath();
    for (Pose p : poses) {
        goals.add(new Goal(new Point(p.x, p.y)));
        System.out.println("Added a new goal at: " + p.x + ", " + p.y);
    }

    if (goals.size() > 0) {
        //currentGoal = goals.get(0);
        //currentGoalNumber = 0;
    } else {
        //currentGoal = topPickupZone;
        //currentGoalNumber = 0;
    }

    this.currentRobotState = RobotState.RobotStateInvalid;
    setRobotState(RobotState.RobotStateStart);

    topPickupGoals = new ArrayList<Goal>();
    topDropoffGoals = new ArrayList<Goal>();
    bottomPickupGoals = new ArrayList<Goal>();
    bottomDropoffGoals = new ArrayList<Goal>();


    // Fill out the drop goals.. these are the destinations for drop zones
    // Offset these goals for where the robot should actually stop?
    for (int i = 0; i < 4; i++) {
        int x = 550;
        int y = 665 + (i * 40);
        bottomDropoffGoals.add(new Goal(new Point(x, y)));
    }


    for (int i = 0; i < 4; i++) {
        int x = 60;
        int y = 850 - (i * 40);
        topDropoffGoals.add(new Goal(new Point(x, y)));
    }



    // Hack just to get the robot to switch modes on its own for testing
    final MiddlePlanner that = this;
    new Thread(new Runnable() {

        @Override
        public void run() {
            // TODO Auto-generated method stub
            try {
                Thread.sleep(3000);
                System.out.println("Thread done sleeping. Going to trick the robot into starting!");

                String d;

                d = StationMessage.FormatToMessage(MessageType.STATION_1_RED_BLOCK_DROPPED_OFF, 60, 556);
                that.receivedDataFromStation(1, d);

                d = StationMessage.FormatToMessage(MessageType.STATION_1_RED_BLOCK_DROPPED_OFF, 60, 577);
                that.receivedDataFromStation(1, d);

                d = StationMessage.FormatToMessage(MessageType.STATION_1_RED_BLOCK_DROPPED_OFF, 60, 600);
                that.receivedDataFromStation(1, d);

                d = StationMessage.FormatToMessage(MessageType.STATION_1_RED_BLOCK_DROPPED_OFF, 60, 620);
                that.receivedDataFromStation(1, d);


                d = StationMessage.FormatToMessage(MessageType.STATION_3_BLUE_BLOCK_DROPPED_OFF, 568, 929);
                that.receivedDataFromStation(2, d);

                d = StationMessage.FormatToMessage(MessageType.STATION_3_BLUE_BLOCK_DROPPED_OFF, 568, 912);
                that.receivedDataFromStation(2, d);

                d = StationMessage.FormatToMessage(MessageType.STATION_3_BLUE_BLOCK_DROPPED_OFF, 568, 894);
                that.receivedDataFromStation(2, d);

                d = StationMessage.FormatToMessage(MessageType.STATION_3_BLUE_BLOCK_DROPPED_OFF, 568, 875);
                that.receivedDataFromStation(2, d);



                //_numberOfBottomBlocksReady = TOTAL_BLOCKS;
                _topZoneUnlocked = true;
                _bottomZoneUnlocked = true;
                //setRobotState(RobotState.RobotStatePickTop);


                // add some blocks
                //topPickupGoals.add(new Goal(new Point(60, 542)));

                //bottomPickupGoals.add(new Goal(new Point(560, 916)));

            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                System.out.println("There was an error in the thread!!");
                e.printStackTrace();
            }
        }
    }).start();
    System.out.println("Thread started!!");
}



void setRobotState(RobotState newState) {
    if (newState == currentRobotState)
        return;
    System.out.println("Transitioning from " + this.currentRobotState.toString() + " to " + newState.toString());
    this.currentRobotState = newState;

    // Reset some flags.
    _haveSentRobotCommand = false;
    _latestCommandAcknowledged = false;
    _robotFoundBlock = false;

}


void sendInstructionsToRobot(byte movement, byte rotation, byte command) {
    byte[] outData = new byte[6]; // the data buffer to send to the robot.

    outData[ROTATION] = rotation;
    outData[MOVE] = movement;
    outData[CONTROL] = command;

    sendDataToRobot(outData);
}


void orientToGoalForPose(Pose pose, Goal goal, RobotState nextState) {
    int robotCurrentAngle = pose.angle; // degrees
    Point robotPoint = new Point(pose.x, pose.y);
    int angleToTheGoal = (int)getAngle(goal.location, robotPoint);


    int RANGE = 10;
    byte rotation = ROTATE_NONE;
    byte movement = MOVE_NONE;

    int angleDifference = angleToTheGoal - robotCurrentAngle;
    System.out.println("Angle diff: " + angleDifference);
    if (Math.abs(angleDifference) < RANGE) {


        // We're good!!
        // Change state and reset some internal flags.
        setRobotState(nextState);

        // Just so we have some instructions to reply with
        sendInstructionsToRobot(MOVE_NONE, ROTATE_NONE, ASK_AGAIN);
        return;

    } else if (angleDifference > 0) {

        // Still need to rotate;

        rotation = angleDifference < 180? ROTATE_BACK_SMALL : ROTATE_SMALL;

    } else {
        rotation = angleDifference < -180? ROTATE_BACK_SMALL : ROTATE_SMALL; // might need to do like above...
    }

    sendInstructionsToRobot(movement, rotation, GO);

}



void goalNavigateAlongCurrentPathForRobotPoint(Pose pose, RobotState nextState, boolean forceNextState) {
    // This is basically Goal-navigation now.

    // First see if he's close enough to the goal, in which case
    // transition to the next State.
    Point robotPoint = new Point(pose.x, pose.y);
    Goal currentGoal = goals.get(0); // TODO: Change this if we're dealing with multiple goals!!!

    if (currentGoal.isPointCloseEnoughToGoal(robotPoint)) {
        // Really, we have to make sure we do this for the WHOLE PATH OF GOALS
        // Not just 1 goal.
        // We're close enough to change states
        System.out.println(this.currentRobotState.toString() + ": Close enough to goal, going to orient.");


        // Make sure we're properly oriented to the goal so we can just move forward when seeking a block. Then transition in this method.
        if (forceNextState) {
            // We're good!!
            // Change state and reset some internal flags.
            setRobotState(nextState);

            // Just so we have some instructions to reply with
            sendInstructionsToRobot(MOVE_NONE, ROTATE_NONE, ASK_AGAIN);
        } else {
            orientToGoalForPose(pose, currentGoal, nextState);
        }

    } else {
        // We need to tell the robot to (keep) moving towards the Drop Zone.
        int robotCurrentAngle = pose.angle; // degrees
        int angleToTheGoal = (int)getAngle(currentGoal.location, robotPoint);
        int distanceToGoal = distance(currentGoal.location.x - robotPoint.x,
                                      currentGoal.location.y -robotPoint.y);
        System.out.println("Distance to goal is: " + distanceToGoal);

        int RANGE = 5;
        byte rotation = ROTATE_NONE;
        byte movement = MOVE_NONE;

        int angleDifference = angleToTheGoal - robotCurrentAngle;

        if (Math.abs(angleDifference) < RANGE) {
            rotation = ROTATE_NONE;
            movement = MOVE_SMALL;
        } else if (angleDifference > 0) {
            rotation = angleDifference < 180? ROTATE_BACK_SMALL : ROTATE_SMALL;
            movement = MOVE_NONE;
        } else {
            System.out.println("Less than 0..." + angleDifference);
            rotation = ROTATE_SMALL; // might need to do like above...
            movement = MOVE_NONE;
        }


        // Now send this command to the robot
        sendInstructionsToRobot(movement, rotation, GO);

    }
}



boolean receivedCommandAck = true;
public void receivedPoseFromTracker(Pose p) {


    // Just keep updating to the latest pose... I suppose we could just keep a reference to the latest one. Whatever.
    if (p.x < 0 || p.y < 0) return; // skip invalid poses
    poses.add(p);


}
```

```
void announceCompletionToOtherStations() {

    String data1 = StationMessage.FormatToMessage(MessageType.ZONE_2_1_UNLOCKED);
    sendDataToStation(1, data1);
    sendDataToStation(3, data1);

    data1 = StationMessage.FormatToMessage(MessageType.ZONE_2_2_UNLOCKED);
    sendDataToStation(1, data1);
    sendDataToStation(3, data1);


    String data = StationMessage.FormatToMessage(MessageType.STATION_2_DONE);
    sendDataToStation(1, data);
    sendDataToStation(3, data);
}


void announceDropoffAtPointToStation(Point point, int stationID) {

    MessageType messageType;
    if (stationID == 1) {
        messageType = MessageType.STATION_2_RED_BLOCK_DROPPED_OFF;
    } else {

        messageType = MessageType.STATION_2_BLUE_BLOCK_DROPPED_OFF;

    }

    sendDataToStation(stationID, StationMessage.FormatToMessage(messageType, point.x, point.y));
}


private int[] processedRobotData(int[] data) {
    int length = data.length;
    if (length % 2 != 0) length += 1; // make sure it's an even number

    int[] processed = new int[length/2];
    int currentProcessedIndex = 0;
    for (int currentDataIndex = 0; currentDataIndex < data.length; currentDataIndex += 2) {
        processed[currentProcessedIndex++] = data[currentDataIndex]*256 + data[currentDataIndex+1];
    }

    return processed;
}


private void computePathFromRobotPoseToEndGoal(Pose currentRobotPose, RobotState state) {

    // Based on the current state and the current location, compute a path the robot needs to take to reach
    // the goal (the goal depends on the current state!).

    goals = new ArrayList<Goal>();
    switch (state) {
        case RobotStateDropBottom: {



            // Move in almost a straight line? Or just a straight line...
            goals.add(bottomDropoffGoals.get(0));

            // Must remember to remove this goal from the bottomDropoffGoals list... do it now?
            bottomDropoffGoals.remove(0);

            break;
        }


        case RobotStateSeekBottom: {


            // Pickup the last item in this list
            goals.add(bottomPickupGoals.get(bottomPickupGoals.size() - 1));

            bottomPickupGoals.remove(bottomPickupGoals.size() - 1);

            break;
        }


        case RobotStateDropTop: {

            goals.add(topDropoffGoals.get(0));
            topDropoffGoals.remove(0);

            break;
        }


        case RobotStateSeekTop: {

            goals.add(topPickupGoals.get(topPickupGoals.size() - 1));
            System.out.println("top goal is: " + goals.get(0).toString());
            topPickupGoals.remove(topPickupGoals.size() - 1);

            break;
        }
        case RobotStatePark: {
            goals.add(new Goal(new Point(160, 750)));

            break;
        }


        default: {
            System.out.println("Unhandled plan computation!!!!! " + state.toString());
            System.exit(-1); // force a crash
            break;
        }



    }

    System.out.println("New goal path looks like:");
    for (Goal g : goals) {
        System.out.println(g.toString());
    }
}


// Called when the planner receives data from the robot
public void receivedDataFromRobot(int[] data) {

    Pose latestPose = poses.get(poses.size() - 1);
    int[] commandData = processedRobotData(data); // basically just intifies the data from the robot.


    switch (this.currentRobotState) {
        case RobotStateStart: {
            if ((_numberOfTopBlocksReady + _numberOfBottomBlocksReady) == TOTAL_BLOCKS && _topZoneUnlocked && _bottomZoneUnlocked) {
                // transition to the next state


                setRobotState(RobotState.RobotStateSeekTop);
                String data1 = StationMessage.FormatToMessage(MessageType.ZONE_2_1_LOCKED);
                sendDataToStation(1, data1);
                sendDataToStation(3, data1);
                data1 = StationMessage.FormatToMessage(MessageType.ZONE_2_2_LOCKED);
                sendDataToStation(1, data1);
                sendDataToStation(3, data1);
                computePathFromRobotPoseToEndGoal(latestPose, RobotState.RobotStateSeekTop);
            } else {
                // Stay in the same state... don't tell the robot?
                //this.currentRobotState = RobotStateStart;



                /* The robot will then stay still for a certain timeout and then ask again
                    At which point we will try again to determine his current instruction
                */

            }

            // Always stay still. The robot will send a call when he's done
            // At which point, if we've transitioned, then we'll tell him his new instructions
            sendInstructionsToRobot(MOVE_NONE, ROTATE_NONE, STAY_STILL);
            break;
        }


        case RobotStateSeekTop: {
            // The robot needs to be moving towards the top pickup zone
            // This is basically Goal-navigation now.

            // First see if he's close enough to the goal, in which case
            // transition to the next State.

            goalNavigateAlongCurrentPathForRobotPoint(latestPose, RobotState.RobotStatePickTop, false);


            // Point robotPoint = new Point(latestPose.x, latestPose.y);
            // Goal currentGoal = goals.get(currentGoalNumber); // TODO: verify this
            //
            // if (currentGoal.isPointCloseEnoughToGoal(robotPoint)) {
            //  // We're close enough to change states
            //  System.out.println("SeekTop: close enough to PickupZone");
            //
            //  // Change state and reset some internal flags.
            //  setRobotState(RobotState.RobotStatePickTop);
            //
            //  // Just so we have some instructions to reply with
            //  sendInstructionsToRobot(MOVE_NONE, ROTATE_NONE, ASK_AGAIN);
            //
            //
            //
            // } else {
            //  // We need to tell the robot to (keep) moving towards the Pickup Zone.
            //  int robotCurrentAngle = latestPose.angle; // degrees
            //  int angleToTheGoal = (int)getAngle(currentGoal.location, robotPoint);
            //  int distanceToGoal = distance(currentGoal.location.x - robotPoint.x,
            //                                    currentGoal.location.y -robotPoint.y);
            //  System.out.println("Distance to goal (" + currentGoal.location.x + ", " + currentGoal.location.y + ") is: " + distanceToGoal);
            //
            //  int RANGE = 5;
            //  byte rotation = ROTATE_NONE;
            //  byte movement = MOVE_NONE;
            //
            //  int angleDifference = angleToTheGoal - robotCurrentAngle;
            //  System.out.println("Angle diff: " + angleDifference);
            //
            //  if (Math.abs(angleDifference) < RANGE) {
            //      rotation = ROTATE_NONE;
            //      movement = MOVE_SMALL;
            //      System.out.println("Should move");
            //  } else if (angleDifference > 0) {
            //      rotation = angleDifference < 180? ROTATE_BACK_SMALL : ROTATE_SMALL;
            //      movement = MOVE_NONE;
            //  } else {
            //      rotation = ROTATE_SMALL; // might need to do like above...
            //      movement = MOVE_NONE;
            //  }
            //
            //
            //  // Now send this command to the robot
            //  sendInstructionsToRobot(movement, rotation, GO);
            //
            // }


            break;


        }


        case RobotStatePickTop: {

            // We need to tell the robot to go into BLOCK_SEEK mode

            // We'll send the BLOCK_SEEK command, which he'll then do.
            // After he's found the block (OR A TIMEOUT?) then he'll message again, reporting
            if (commandData[STATUS] == STATUS_COMMAND_REQUEST) {
                // He's asking for what to do, so tell him to go seek
                System.out.println("PickTop: going to tell robot to SEEK_BLOCK");
                sendInstructionsToRobot(MOVE_SMALL, ROTATE_NONE, SEEK_BLOCK);
                break;
            } else if (commandData[STATUS] == STATUS_BLOCK_FOUND) {
                // He's found a block
                System.out.println("PickTop: Robot found a block. Transitioning...");
                setRobotState(RobotState.RobotStateDropBottom);
                computePathFromRobotPoseToEndGoal(latestPose, RobotState.RobotStateDropBottom);
            } else {
                // he must not have found it in time... sad face?
                System.out.println("PickTop: Robot did not find a block... transitioning anyway.");
                setRobotState(RobotState.RobotStateDropBottom);
                computePathFromRobotPoseToEndGoal(latestPose, RobotState.RobotStateDropBottom);
            }


            // He's either found or hasn't found a block, but he's already looked.
            // Now we tell him to just wait a sec and then ask again for the next state
            sendInstructionsToRobot(MOVE_NONE, ROTATE_NONE, ASK_AGAIN);
            break;

        }


        case RobotStateDropBottom: {
            // Moving towards the bottom drop zone
            goalNavigateAlongCurrentPathForRobotPoint(latestPose, RobotState.RobotStateDoActualDropBottom, true);
            // When do we tell the robot to ACTUALLY drop it?
            break;
        }


        case RobotStateDoActualDropBottom: {
            if (commandData[STATUS] == STATUS_COMMAND_REQUEST) {
                // ASking us what to do.. say drop the block!
                System.out.println("DoActualDropBottom: telling robot to drop");
                sendInstructionsToRobot(MOVE_NONE, ROTATE_NONE, DROP_BLOCK);
            } else {
                // He's done it
                System.out.println("DoActualDropBottom: Robot has dropped the block.");
                _numberOfDeliveredBlocks++;
                announceDropoffAtPointToStation(new Point(latestPose.x, latestPose.y), 3);

                setRobotState(RobotState.RobotStateSeekBottom);
                computePathFromRobotPoseToEndGoal(latestPose, RobotState.RobotStateSeekBottom);
                sendInstructionsToRobot(MOVE_NONE, ROTATE_NONE, ASK_AGAIN);
            }

            break;
        }


        case RobotStateSeekBottom: {
            // Seek to the bottom area
            goalNavigateAlongCurrentPathForRobotPoint(latestPose, RobotState.RobotStatePickBottom, false);
            break;

        }


        case RobotStatePickBottom: {


            // We need to tell the robot to go into BLOCK_SEEK mode

            // We'll send the BLOCK_SEEK command, which he'll then do.
            // After he's found the block (OR A TIMEOUT?) then he'll message again, reporting
            if (commandData[STATUS] == STATUS_COMMAND_REQUEST) {
                // He's asking for what to do, so tell him to go seek
                System.out.println("PickBottom: going to tell robot to SEEK_BLOCK");
                sendInstructionsToRobot(MOVE_SMALL, ROTATE_NONE, SEEK_BLOCK);
                break;
            } else if (commandData[STATUS] == STATUS_BLOCK_FOUND) {
                // He's found a block
                System.out.println("PickBottom: Robot found a block. Transitioning...");
                setRobotState(RobotState.RobotStateDropTop);
                computePathFromRobotPoseToEndGoal(latestPose, RobotState.RobotStateDropTop);
            } else {
                // he must not have found it in time... sad face?
                System.out.println("PickBottom: Robot did not find a block... transitioning anyway.");
                setRobotState(RobotState.RobotStateDropTop);
                computePathFromRobotPoseToEndGoal(latestPose, RobotState.RobotStateDropTop);
            }


            // He's either found or hasn't found a block, but he's already looked.
            // Now we tell him to just wait a sec and then ask again for the next state
            sendInstructionsToRobot(MOVE_NONE, ROTATE_NONE, STAY_STILL);
            break;
        }


        case RobotStateDropTop: {

            // Move towards the dropoff zone, then transition to the FinishedCycle state, where the block is dropped and we decide what's next
            goalNavigateAlongCurrentPathForRobotPoint(latestPose, RobotState.RobotStateDoActualDropTop, true);

            break;
        }


        case RobotStateDoActualDropTop: {

            if (commandData[STATUS] == STATUS_COMMAND_REQUEST) {
                // ASking us what to do.. say drop the block!
                System.out.println("DoActualDropTop: telling robot to drop");
                sendInstructionsToRobot(MOVE_NONE, ROTATE_NONE, DROP_BLOCK);
            } else {
                // He's done it

                _numberOfDeliveredBlocks++;
                System.out.println("DoActualDropTop: Robot has dropped the block.(" + _numberOfDeliveredBlocks + "/" + TOTAL_BLOCKS);


                // Tell the top station
                announceDropoffAtPointToStation(new Point(latestPose.x, latestPose.y), 1);

                setRobotState(RobotState.RobotStateFinishedCycle);
                //computePathFromRobotPoseToEndGoal(latestPose, RobotState.RobotStateFinishedCycle);
                sendInstructionsToRobot(MOVE_NONE, ROTATE_NONE, ASK_AGAIN);
            }


            break;
        }


        case RobotStateFinishedCycle: {

            if (_numberOfDeliveredBlocks == TOTAL_BLOCKS) {
                System.out.println("All the blocks have been delivered. Move out of the way and STOP");
                setRobotState(RobotState.RobotStatePark);
                computePathFromRobotPoseToEndGoal(latestPose, RobotState.RobotStatePark);
                sendInstructionsToRobot(MOVE_NONE, ROTATE_NONE, ASK_AGAIN);
            } else {
                System.out.println("Finished a cycle. More blocks remaining.");
                setRobotState(RobotState.RobotStateSeekTop);
                computePathFromRobotPoseToEndGoal(latestPose, RobotState.RobotStateSeekTop);
                sendInstructionsToRobot(MOVE_NONE, ROTATE_NONE, ASK_AGAIN);
            }

            break;
        }


        case RobotStatePark: {
            goalNavigateAlongCurrentPathForRobotPoint(latestPose, RobotState.RobotStateEnd, true);

            break;
        }


        case RobotStateEnd: {

            System.out.println("The robot is all done... why is it asking again?");
            announceCompletionToOtherStations();
            sendInstructionsToRobot(MOVE_NONE, ROTATE_NONE, ALL_DONE);
            break;

        }
    }
}


public void receivedDataFromStation(int stationId, String data) {
    //System.out.println("Got " + data + " from station: " + stationId);

    //System.out.printf("Station ID: %d, Data: [ %s ]\n", stationId, data);
    System.out.println("StationID: " + stationId + data);

    StationMessage message = StationMessage.Parse(stationId, data);
    int x = message.getX();
    int y = message.getY();

    switch (message.getMessageType()) {
        case STATION_1_RED_BLOCK_DROPPED_OFF: {
            _numberOfTopBlocksReady++;
            System.out.println("Got a new top block");

            // add it to the pickup list

            // Maybe have to offset these points to make it easier for bot to find them?
            topPickupGoals.add(new Goal(new Point(x, y)));

            break;
        }


        case STATION_3_BLUE_BLOCK_DROPPED_OFF: {
            _numberOfBottomBlocksReady++;
            System.out.println("Got a new bottom block");

            // add it to the pickup list

            // Maybe have to offset these points to make it easier for bot to find them?
            bottomPickupGoals.add(new Goal(new Point(x, y)));
            break;
        }


        case ZONE_1_LOCKED: {
            _topZoneUnlocked = false;
            System.out.println("Locking the top zone");
            break;
        }


        case ZONE_1_UNLOCKED: {
            _topZoneUnlocked = true;
            System.out.println("Unlocking the top zone");
            break;
        }


        case ZONE_3_LOCKED: {
            _bottomZoneUnlocked = false;
            System.out.println("Locking the bottom zone");
            break;
        }


        case ZONE_3_UNLOCKED: {
            _bottomZoneUnlocked = true;
            System.out.println("Unlocking the bottom zone");
            break;
        }


        default: {
            System.out.println("Got a message we don't care about: " + message.getMessageType().toString());
        }

    }

}


public int distance(int a, int b) {
    int c2 = (int) ((Math.pow(a, 2)) + (Math.pow(b, 2)));
    return (int)(Math.sqrt(c2));
}
```

```java
    public double getAngle(Point goalPoint, Point robotPoint) {
        System.out.println("Goal: " + goalPoint.toString() + " robot: " + robotPoint.toString());
        int gx = goalPoint.x;
        int gy = goalPoint.y;
        int rx = robotPoint.x;
        int ry = robotPoint.y;


        double dx = 0;
        double dy = 0;

        dx = gx - rx;
        dy = gy - ry;


        double inRads = Math.atan2(dy,dx);
        double deg = Math.toDegrees(inRads);
        if (deg < 0) {
            deg = 360 + deg;
        }
        return deg;
    }


    class Goal {
        public Point location;
        public boolean isReached;

        public Goal(Point loc) {
            location = loc;
            isReached = false;
        }

        public boolean isPointCloseEnoughToGoal(Point p) {
            final int RANGE = 30;
            return (inAbsoluteRange(p.x, this.location.x, RANGE) && inAbsoluteRange(p.y, this.location.y, RANGE));
        }

        private boolean inAbsoluteRange(int a, int b, int range) {
            return (Math.abs((a - b)) < range);
        }

        public String toString() {
            return location.toString();
        }
    }


    class Point {
        public int x, y;

        public Point(int x, int y) {
            this.x = x;
            this.y = y;
        }

        public boolean equals(Point p) {
            return p.x == this.x && p.y == this.y;
        }

        public String toString() {
            return x + ", " + y;
        }
    }

}
```

Navigate.spin

```
CON
  _clkmode = xtal1 + pll16x     ' This is required for proper timing
  _xinfreq = 5_000_000          ' This is required for proper timing

  CLK_FREQ = ((_clkmode-xtal1)>>6)*_xinfreq
  MS_001 = CLK_FREQ / 1_000

  SERVO_STOPPED_LEFT = 750
  SERVO_STOPPED_RIGHT = 750

  LEFT_GRIPPER_CLOSED = 208
  LEFT_GRIPPER_MID = 175
  LEFT_GRIPPER_OPEN = 158

  RIGHT_GRIPPER_CLOSED = 101
  RIGHT_GRIPPER_MID = 134
  RIGHT_GRIPPER_OPEN = 155

  HEAD_TILT_DOWN = 110
  HEAD_TILT_MID = 126
  HEAD_TILT_UP = 143

  HEAD_TWIST_LEFT = 81
  HEAD_TWIST_MID = 146
  HEAD_TWIST_RIGHT = 215
  SENSOR_FRONT_LEFT = 3
  SENSOR_FRONT_CENTER = 2
  SENSOR_FRONT_RIGHT = 1

  SENSOR_SIDE_FRONT_RIGHT = 7
  SENSOR_SIDE_FRONT_LEFT = 4
  SENSOR_SIDE_BACK_RIGHT = 6
  SENSOR_SIDE_BACK_LEFT = 5

  BEEP_SHORT = 250
  BEEP_TONE_LEFT = 1000
  BEEP_TONE_CENTER =3000
  BEEP_TONE_RIGHT = 5000

  YES = 1
  NO = 0
  RED = 185
  BLUE = 16
  GREEN = 16
  HALF_TURNING_AMOUNT = 150
  TURNING_AMOUNT = 310
  BACKWARDS_AMOUNT = 175
  WANDERING_AMOUNT = 500

  ' Arbitration
  PRIORITY_WANDER = 1
  PRIORITY_PERFORM_TASK = 2
  PRIORITY_COLLISION_AVOID = 3

  STATE_ORIENT = 0
  STATE_FOLLOW = 1
  STATE_ALIGN  = 2

  WHEEL_SPEED_LEFT = 17
  WHEEL_SPEED_RIGHT = 17

  BLOCK_LEFT_SIDE = 55
  BLOCK_RIGHT_SIDE = 30

  ' byte indices
  ROTATION_INDEX = 1
  MOVE_INDEX = 2
  CONTROL_INDEX = 3

  ' Movement instructions
  MOVE_NONE = 0
  MOVE_SMALL = 1

  ' Rotation instructions
  ROTATE_NONE = 0
  ROTATE_SMALL = 1
  ROTATE_BACK_SMALL = 2


  ' Command instructions
  STAY_STILL = 0  ' We're still waiting for the team to finish... set a timer and then ask again
  ASK_AGAIN = 1   ' Transitioning states, just ask again on the next iteration through
  GO = 2          ' Some kind of movement command (move or rotate)
  SEEK_BLOCK = 3  ' The robot needs to go into seek mode and then pick up a block... collision detection too!
  DROP_BLOCK = 4  ' The robot just needs to drop the block
  ALL_DONE = 100  ' The robot has completed the tasks and should shut down.


  ' Status codes
  STATUS_COMMAND_REQUEST = 0
  STATUS_BLOCK_FOUND = 1
  STATUS_BACKUP_DONE = 2

  TURN_COUNT_SMALL = 10
  TURN_SMALL = 5
  MOVE_SMALL_COUNT = 50
  STAY_STILL_COUNT = 100

  GOOD_CONFIDENCE = 20
  CAMERA_SENSITIVITY = 30


VAR
  long current_state, edge_follow_left, edge_follow_right
  long detect_front, detect_right

  byte dataIn[7]
  byte out_packet[6]
  long cur_x, cur_y, cur_a, estimation_counter, pulse_left, pulse_right, turning_count, forward_count, turning_left
  byte turn_val, move_val
  long collecting_data, sonar_reading, ir_reading
  long stay_still_counter

  long did_find_block, have_reversed

  long pc_move_command
  long pc_rotate_command
  long pc_control_command

OBJ
```

```
  RBC: "RBC"
  Beeper: "Beeper"
  Servos: "ServoControl2"
  IRSensors: "IR8SensorArray"
  Sonar: "PingSensor"
  Dirrs: "DirrsSensor"
  BlockSensor: "BlockSensor"
  Camera: "CMUCam"


PUB main

  'Beeper.Startup
  RBC.Init

  turning_count := 0
  forward_count := 0
  turning_left := NO
  did_find_block := NO
  have_reversed := NO


  turn_val := 0
  move_val := 0

'  RBC.ReceiveData(@dataIn)
 ' cur_x := dataIn[1]*256 + dataIn[2]
 ' cur_y := dataIn[3]*256 + dataIn[4]
 ' cur_a := dataIn[5]*256 + dataIn[6]

  Servos.Start(SERVO_STOPPED_LEFT - 5, SERVO_STOPPED_RIGHT, true, true, true, true)
  Servos.SetRightGripper(100)
  Servos.SetLeftGripper(200)
  Servos.SetHeadYaw(150)
 ' Servos.SetHeadPitch(125)

  Camera.Start
  Camera.SetTrackColor(RED, GREEN, BLUE, CAMERA_SENSITIVITY)

  Beeper.OK

  current_state := STATE_FOLLOW
  nslog(string("about to move"))
  repeat 'main loop



    'see if we're already in a turn, in which case we should turn!
    if (turning_count > 0)
      turning_count := turning_count - 1
      RBC.DebugLong(turning_count)
      'nslog(string(" :in the turning count loop"))
      pauseMS(50)
      do_turn

      next


    'move forward as needed
    if (forward_count > 0)
      forward_count := forward_count - 1
      'nslog(string("in the moving loop"))
      pauseMS(50)
      do_move_forward
      next


    if (stay_still_counter > 0)
      stay_still_counter := stay_still_counter - 1
      'nslog(string("staying still...."))
      pauseMS(50)
      next


    'wait and get data from the RBC
    nslog(string("waiting for planner data"))


    ' Stop the servos and then ask for a new command
    set_wheel_speeds(0, 0)
    ask_pc_for_instructions(did_find_block)

    ' Now process the response
    process_pc_instructions


    ' reset any control counters
    turning_count := 0
    forward_count := 0
    stay_still_counter := 0
    did_find_block := NO
    have_reversed := NO


    ' now deal with the instructions based on the control bit
    case (pc_control_command)
```

```
      STAY_STILL:
        stay_still_counter := STAY_STILL_COUNT
        next

      ASK_AGAIN:
        next

      GO:
        ' some kind of movement... need to check
        case (pc_rotate_command)
          ROTATE_NONE: 'do nothing important
          ROTATE_SMALL:
            turning_count := TURN_SMALL
            turning_left := NO
            next
          ROTATE_BACK_SMALL:
            turning_count := TURN_SMALL
            turning_left := YES
            next


        ' not rotation, so let's check movement then
        case (pc_move_command)
          MOVE_NONE:
            forward_count := 0
          MOVE_SMALL:
            forward_count := MOVE_SMALL_COUNT
            next 'ignoring the "REVERSE_BIG" command when we're all done?


      SEEK_BLOCK:
        do_seek_block ' enters its own run-loop where it doesn't break out until it's found a block and clasped the grippers
      DROP_BLOCK:
        do_drop_block ' enters its own run-loop where it moves and drops off a block and breaks out of it when it's done.
      ALL_DONE:
        set_wheel_speeds(0, 0)
        Beeper.Shutdown
        return




PUB ask_pc_for_instructions (found_a_block)

  if (found_a_block == YES)
    out_packet[0] := STATUS_BLOCK_FOUND / 256
    out_packet[1] := STATUS_BLOCK_FOUND // 256
  elseif (have_reversed == YES)
    out_packet[0] := STATUS_BACKUP_DONE / 256
    out_packet[1] := STATUS_BACKUP_DONE // 256
  else
    out_packet[0] := STATUS_COMMAND_REQUEST / 256
    out_packet[1] := STATUS_COMMAND_REQUEST // 256
  RBC.SendDataToPC(@out_packet, 6, RBC#OUTPUT_TO_NONE)


PUB process_pc_instructions

  RBC.ReceiveData(@dataIn)

  ' Does this need to be offset???

  pc_move_command := dataIn[MOVE_INDEX]
  pc_rotate_command := dataIn[ROTATION_INDEX]
  pc_control_command := dataIn[CONTROL_INDEX]



PRI do_seek_block | found_block, cam_x, temp_holder, temp_con

  Servos.SetHeadPitch(HEAD_TILT_DOWN)
  open_grippers
  found_block := NO
  cam_x := 0
  temp_holder := 99
  temp_con := 98

  nslog(string("GOING TO SEEK A BLOCK"))
  repeat until (found_block == YES)

    ' try looking for a block
    temp_holder := BlockSensor.Detect

    RBC.DebugLongCR(temp_holder)
    if (BlockSensor.Detect)
      close_grippers
      found_block := YES
      nslog(string("YESSSS"))
      quit

   ' nslog(string("gmmmmmm"))
    ' see if we can find where a block is using the camera
    Camera.TrackColor
    temp_con := Camera.GetConfidence
    nslog(string("confidence"))
    RBC.DebugLongCR(temp_con)

    if (temp_con > GOOD_CONFIDENCE)
      cam_x := Camera.GetCenterX
      nslog(string("SEE A BLOCK SOMEWHERE"))
      RBC.DebugLongCR(cam_x)


      if (cam_x > BLOCK_LEFT_SIDE)
        nslog(string("turn left"))
        do_left_turn
      elseif (cam_x < BLOCK_RIGHT_SIDE)
        nslog(string("to the right"))
        do_right_turn
      else
        do_slow_forward

    else
      ' don't see the block... keep moving forward. it might appear
      do_slow_forward


  set_wheel_speeds(0, 0)
  nslog(string("out of the loop"))
  close_grippers
  Servos.SetHeadPitch(HEAD_TILT_MID)
  did_find_block := YES ' so this can be sent to the PC on the next iteration


PRI do_drop_block | back_count
  nslog(string("dropping block and backing up"))
  ' drop the block, back up a bit, and then close the grippers
  open_grippers
  pauseMS(2000)
  back_count := 15
  repeat until (back_count == 0)
    back_count := back_count - 1
    pauseMS(300)
    set_wheel_speeds(-8, -8)

  set_wheel_speeds(0, 0)
  have_reversed := YES
  close_grippers


PRI open_grippers
  Servos.SetLeftGripper(LEFT_GRIPPER_OPEN)
  Servos.SetRightGripper(RIGHT_GRIPPER_OPEN)


PRI close_grippers
  Servos.SetLeftGripper(LEFT_GRIPPER_CLOSED)
  Servos.SetRightGripper(RIGHT_GRIPPER_CLOSED)
'moving functions for the block seeking


PRI do_left_turn | i_turn_count


  i_turn_count := TURN_COUNT_SMALL + 6
  repeat until (i_turn_count == 0)
    i_turn_count := i_turn_count - 1
    turnLeft
    nslog(string("L"))

  ' reset the speeds so the robot isn't still turning :)
 ' set_wheel_speeds(0, 0)



PRI do_right_turn | i_turn_count

  i_turn_count := TURN_COUNT_SMALL + 6
  repeat until (i_turn_count == 0)
    i_turn_count := i_turn_count - 1
    turnRight
    nslog(string("R"))

  ' reset the speeds so the robot isn't still turning :)
  'set_wheel_speeds(0, 0)

PRI do_slow_forward | i_move_count

  i_move_count := 10
  repeat until (i_move_count == 0)
    set_wheel_speeds(8, 8)
    nslog(string("m"))
    i_move_count := i_move_count - 1

  ' stop him again
 ' set_wheel_speeds(0, 0)


PUB do_moving_as_needed
  case (current_state)
    STATE_FOLLOW:
      'move forward
      set_speeds(WHEEL_SPEED_LEFT, WHEEL_SPEED_RIGHT)
    STATE_ALIGN:
      'turn right
      set_speeds(WHEEL_SPEED_LEFT + 5, -5)
    STATE_ORIENT:
      'turn left
      set_speeds(-3, WHEEL_SPEED_RIGHT + 8)


PUB do_move_forward
  set_wheel_speeds(WHEEL_SPEED_LEFT, WHEEL_SPEED_RIGHT)


PRI set_speeds(left, right)
  Servos.SetSpeeds(left, right)


PRI do_turn
  if (turning_left)
    nslog(string("lft"))
    turnLeft
  else
    nslog(string("rght"))
    turnRight


PUB turnLeft
  'Servos.SetSpeeds(0, 17)
  set_wheel_speeds(-5, WHEEL_SPEED_RIGHT)


PUB turnRight
  'Servos.SetSpeeds(14, 0)
  set_wheel_speeds(WHEEL_SPEED_LEFT, -5)


PUB set_wheel_speeds(left_s, right_s)
  Servos.SetSpeeds(left_s, right_s)



  'now write them out


PRI nslog(str)
  RBC.DebugStrCr(str)

PUB pauseMS(ms) | t
 {{ i borrowed this function from
http://www.parallax.com/Portals/0/Downloads/docs/cols/nv/prop/col/nvp7.pdf
}}
 t := CNT
 REPEAT ms
   WAITCNT(t += MS_001)
```

StationMessage.java

```
public class StationMessage
{

    private int m_X;
    private int m_Y;
    private MessageType m_MessageType;

    public static StationMessage Parse(int stationId, String messageData) {
        int x = -1;
        int y = -1;
        MessageType messageType;

        String[] tokens = messageData.split(":");
        messageType = MessageType.values()[Integer.parseInt(tokens[0])];

        if (messageType != MessageType.STATION_2_DONE) {
            x = Integer.parseInt(tokens[1]);
            y = Integer.parseInt(tokens[2]);
        }

        return new StationMessage(x, y, messageType);
    }


    public static String FormatToMessage(MessageType messageType, int x, int y) {
        return String.format("%d:%d:%d", messageType.ordinal(), x, y);
    }


    public static String FormatToMessage(MessageType messageType) {
        return String.format("%d", messageType.ordinal());
    }


    public StationMessage(int x, int y, MessageType messageType) {
        m_X = x;
        m_Y = y;
```

```java
        m_MessageType = messageType;
    }


    public int getX() {
        return m_X;
    }


    public int getY() {
        return m_Y;
    }


    public MessageType getMessageType() {
        return m_MessageType;
    }


    @Override
    public String toString() {
        return String.format("[MessageType=%s][X=%d, Y=%d]", m_MessageType, m_X, m_Y);
    }
}
```

MessageType.java

```java
public enum MessageType {
    STATION_3_BLUE_BLOCK_DROPPED_OFF,
    STATION_1_RED_BLOCK_DROPPED_OFF,
    STATION_2_BLUE_BLOCK_DROPPED_OFF,
    STATION_2_RED_BLOCK_DROPPED_OFF,
    STATION_2_DONE,
    ZONE_1_LOCKED,
    ZONE_1_UNLOCKED,
    ZONE_2_1_LOCKED,
    ZONE_2_1_UNLOCKED,
    ZONE_2_2_LOCKED,
    ZONE_2_2_UNLOCKED,
    ZONE_3_LOCKED,
    ZONE_3_UNLOCKED,
    STATION_1_FIRST_ROW_CLEAR,
    STATION_3_FIRST_ROW_CLEAR
}
```