Table Of Contents:

Main Robot State Machine:

The majority of the firmware that the robot runs is found in the file: StudentsRobot.cpp

There is a large state machine inside of this file called: updateStateMachine. Here, the robot keeps track of its high level state. **The robot can be in one of the following states, found in the "RobotStateMachine" enumeration in StudentsRobot.h**:

- Starting up → "StartupRobot"
- Starting to run → "StartRunning"
- Idle → "Running"
- Going to safe stop → "Halting"
- Stop → "Halt"
- Waiting for motors to finish → "WAIT_FOR_MOTORS_TO_FINNISH"
- Wait for time (non-blocking delay) → WAIT_FOR_TIME
- Testing* (used for development of new features) → "Testing"
- Navigating* → "Navigating"
- Parking* → "ParkingRobot"
- Homing the lift mechanism* → "HomingLift"
- Going to a lift setpoint* (which was set from the UI) → MovingLiftFromGUI
- Performing a delivery of a bin* → DeliveringBin
- Returning a bin to the shelf* → ReturningBin

Changing the code in any of these states will change the behavior of the robot. One can add states to this enumeration freely, as long as they are handled in "updateStateMachine". The states marked with an asterisk (*) are those added for this project. The rest existed as part of the WPI RBE 2002 Template, which is used as the base firmware for the basebot used for this project.

Adding states to the main state machine:

Currently, within each state in the main state machine, there is another smaller switch/case statement to keep track of the internal state to a larger state. For example, the "Navigating" state, has its own little state machine, which actually calls yet another state machine in the Navigation routine. If you would like to add a state to the main state machine, follow this order of steps:

1. Add the name of the state to the "RobotStateMachine" enumeration in StudentsRobot.h
2. If you plan on implementing a state machine within the main state you just added, create another enumeration for the states of the smaller state machine. Name it something useful like "<state you just added>States". For example, if the state you added was for "Parking", then maybe your enumeration might be called something like "ParkingStates"

```
/**
 * @enum ParkingStates
 */
enum ParkingStates {
    SETTING_PARKING_GOAL = 0,
    GOING_TO_PARKING_SPACE = 1,
    PARKING = 2,
};
```

3. Create a variable that will keep track of the state. Name it something useful like: “<state you just added>Status”. Using our example in step three, maybe this could be something like “parkingStatus”.

```
// State variables for the enumeration of different routines.
ParkingStates parkingStatus = SETTING_PARKING_GOAL;
```

4. Add the state from the “RobotStateMachine” enumeration to the state machine in “updateStateMachine” in StudentsRobot.cpp

```
case ParkingRobot:
    switch(parkingStatus){
    case SETTING_PARKING_GOAL:
        parkingStatus = GOING_TO_PARKING_SPACE;
        navigation.setNavGoal(goalRow, goalColumn);
        break;
    case GOING_TO_PARKING_SPACE:
        status = Navigating;
        parkingStatus = PARKING;
        statusAfterNav = ParkingRobot;
       break;
    case PARKING:{
        myCommandsStatus = PRK;
        ParkingRoutineStates parkingRoutineStatus = parking.checkParkingStatus();
        if(parkingRoutineStatus == FINISHED_PARKING){
            parkingStatus = SETTING_PARKING_GOAL;
            status = Running;
            statusAfterNav = Running;
            robotParked = true;
        }
        else if(parkingRoutineStatus == TIMED_OUT_PARKING){
            status = Running;
            statusAfterNav = Running;
            parkingStatus = SETTING_PARKING_GOAL;
            robotParked = false;
            myCommandsStatus = Timed_out;
        }
    }
        break;
    }
    break;
```

Some of the states in “updateStateMachine” call functions such as “checkNavStatus” or “checkParkingStatus”. These methods actually contain another state machine, which is used to check the status of the routine. If everything was to be put in the “updateStateMachine” method, this method would be cumbersome to navigate. This allows for more modular and easier to read code. We’ll come back to this.

In the image above, we see within the “ParkingRobot” case, we have a switch on “parkingStatus”. The “PARKING” case calls “parking.checkParkingStatus”, which is another state machine in the parking routine itself, found in “Parking.cpp”

Communicating with the UI:

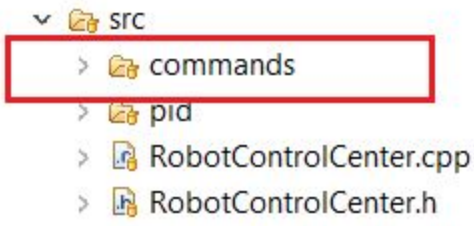
The firmware and the UI communicate via UDP. The UI commands the robot to do certain tasks, and the robot triggers UI actions by publishing its state. When the state of the robot changes, the UI updates. All of the robot states which are published to the UI are found in “StudentsRobot.h”, in the “ComStackStatusState” enumeration. **It’s important to note that these are different from the states found in the “RobotStateMachine” enumeration! The latter is only used for the internal state of the robot.** The firmware will change this state when appropriate in “updateStateMachine”, and it will get published to the UI every loop iteration. For example, when the robot has finished the return of the bin, it will update its “myCommandsStatus” variable to: “Return_Done”. The available states that are sent to the UI are:

- Starting up → “Starting_up”
- Starting to run → “Starting_to_run”
- Idle → “IDLE”
- Going to safe stop → “Stopping”
- Stop → “Stopped”
- Waiting for motors to finish → “WFMF”
- Wait for time (non-blocking delay) → “WFT”
- Testing (used for development of new features) → “Testing_Feature”
- Navigating → “NAV”
- Parking → “PRK”
- Homing the lift mechanism* → “Homing”
- Going to a lift setpoint* (which was set from the UI) → “Moving”
- Performing a delivery of a bin* → “DLV”
- Returning a bin to the shelf* → “RTN”
- Delivery is finished → “Delivery_Done”
- Return is finished → “Return_Done”
- Bin needs to be put on the cleat → “Bin_Not_on_Cleat”
- Bin is missing from the shelf at specified location → “Bin_Not_on_Shelf”
- Delivery was unsuccessful → “Delivery_Unsuccesful”
- Timed out during motion setpoint → “Timed_out”

It is critical that the numbering in this enumeration lines up with the numbering in the UI itself, otherwise the UI will not receive the correct command.

Adding commands to the firmware:

Adding commands to the UI starts with adding a command to the “commands” folder under “src”.



A command consists of a source and header file. For the purposes of explanation, we will use the command to set a navigation goal as an example. To create a new command:

1. Create a header file for the command
2. Create a source file for the command
3. Add the command to the set of commands

Creating the header file: To create a header file for the command, simply create the header under src/commands, and follow this example:

```
#ifndef SRC_COMMANDS_SETNAVGOAL_H_
#define SRC_COMMANDS_SETNAVGOAL_H_

#include <SimplePacketComs.h>
#include "../../StudentsRobot.h"

class SetNavGoal: public PacketEventAbstract {
    StudentsRobot* robotPointer;
public:
    SetNavGoal(StudentsRobot* robot);
    virtual ~SetNavGoal(){}
    void event(float * buffer);
};

#endif /* SRC_COMMANDS_SETNAVGOAL_H_ */
```

We must import “SimplePacketComs.h” and “../../StudentsRobot.h”. SimplePacketCommands is the packet structure that the UI and firmware use to communicate with one another. The “StudentsRobot” import is used to access the robot itself.

Create a new class with your command name, and make sure it inherits from PacketEventAbstract, as shown.

The functions that each command should have are a constructor (shown in red), a destructor (shown in blue), and an event handler (shown in green). These should be public.

Creating the source file: To create a header file for the command, simply create the source file under src/commands, and follow this example:

```
#include "SetNavGoal.h"
#include <Arduino.h>
SetNavGoal::SetNavGoal(StudentsRobot* robot) :
        PacketEventAbstract(1966) {
        robotPointer = robot;

}

void SetNavGoal::event(float * buffer) {
    float row = buffer[0];
    float column = buffer[1];

    robotPointer -> goalRow = (int) row;
    robotPointer -> goalColumn = (int) column;
    //robotPointer -> navigation.setNavGoal(row, column);
    robotPointer -> status = Navigating;
    robotPointer -> statusAfterNav = Running;

}
```

Each command must have a unique ID, and furthermore, it must match between the firmware and the UI. In the constructor for the class of the command, choose an ID and pass it into the constructor of “PacketEventAbstract” as shown in green. Additionally, assign the robotPointer variable to StudentRobot* robot passed into the constructor. This allows us to access variables from the robot.

As mentioned previously, each command class should have an event handler. What you choose to do with the data in the packet is up to you. In this case, the packet coming in contains data about the row and column that the robot must navigate to. The method extracts this data, assigns it to the goalRow and goalColumn variables in the robot object, and sets the robot’s status to “Navigating”.

Adding the command to the set of commands:

Now that the command is made, it needs to be added to the list of commands that the RobotControlCenter expects. In “RobotControlCenter.h”, add the header file associated with your new command, as shown in the figure below, in red.

```
#include "../StudentsRobot.h"
#include "commands/SetPDVelocityConstants.h"
#include "commands/SetPIDVelocity.h"
#include "commands/GetPDVelocityConstants.h"
#include "commands/GetPIDVelocity.h"
#include "commands/IRCamSimplePacketComsServer.h"
#include "commands/GetIMU.h"
#include "commands/GetStatus.h"
#include "commands/SetNavGoal.h"
#include "commands/SetBinDeliveryCommand.h"
#include "commands/SetBinReturnCommand.h"
#include "commands/SetParkCommand.h"
#include "commands/SetStartLiftHoming.h"
#include "commands/SetLiftHeight.h"
```

Next, in “RobotControlCenter.cpp” find the section that starts with “//Attach Comms”. In there, create an instance of your command, and use coms.attach to add it to the set of available commands. The example using our navigation command is highlighted below in green.

```
// Attach coms
coms.attach(new NameCheckerServer(name)); // @suppress("Method cannot be resolved")
coms.attach(new SetPIDSetpoint(numberOfPID, pidList)); // @suppress("Method cannot be resolved")
coms.attach(new SetPIDConstants(numberOfPID, pidList)); // @suppress("Method cannot be resolved")
coms.attach(new GetPIDData(numberOfPID, pidList)); // @suppress("Method cannot be resolved")
coms.attach( // @suppress("Method cannot be resolved")
        new GetPIDConstants(numberOfPID, pidList));
coms.attach(new GetPIDVelocity(numberOfPID, pidList));
coms.attach(new GetPDVelocityConstants(numberOfPID, pidList));
coms.attach(new SetPIDVelocity(numberOfPID, pidList));
coms.attach(new SetPDVelocityConstants(numberOfPID, pidList));
// Get the status of the robot
coms.attach(new GetStatus(robot));// @suppress("Method cannot be resolved")
// GettingNavGoals
coms.attach(new SetNavGoal(robot));// @suppress("Method cannot be resolved")
// Setting parking command
coms.attach(new SetParkCommand(robot));// @suppress("Method cannot be resolved")
coms.attach(new SetStartLiftHoming(robot));
coms.attach(new SetLiftHeight(robot));
coms.attach(new SetBinDeliveryCommand(robot));// @suppress("Method cannot be resolved")
coms.attach(new SetBinReturnCommand(robot));// @suppress("Method cannot be resolved")
```

Basic Motion Firmware:

All the firmware for basic motion, such as turning, stopping, and driving is found in “DrivingChassis.cpp” and its supporting header file: “DrivingChassis.h”.

Whenever setting an action such as driveForward, driveBackwards, or turnToHeading, make sure to check the status of the action by calling statusOfChassisDriving. Here is an example, from the navigation routine.

```
chassis->turnToHeading(180, 7500);
navStateAfterMotionSetpointReached = FINDING_ROW;
navState = WAIT_FOR_MOTION_SETPOINT_REACHED_NAVIGATION;
```

Here we call turnToHeading, and then set the state to “WAIT_FOR_MOTION_SETPOINT_REACHED_NAVIGATION”. In that state, once the robot reaches the correct orientation, the state will change to “FINDING_ROW”. The “WAIT_FOR_MOTION_SETPOINT_REACHED_NAVIGATION” case continuously calls statusOfChassisStatus to see if the robot has reached the orientation or has timed out.

```
case WAIT_FOR_MOTION_SETPOINT_REACHED_NAVIGATION:{
    DrivingStatus motionStatus = chassis -> statusOfChassisDriving();

    if(motionStatus == REACHED_SETPOINT){
        navState = navStateAfterMotionSetpointReached;
    }
    else if(motionStatus == TIMED_OUT){
        navState = TIMED_OUT_NAVIGATION;
    }
}
    break;
```

The documentation for each of the motion commands is found in “DrivingChassis.h”. Each command takes in a setpoint (angles or millimeters) and a timeout. **Note: our code for this project rarely used millimeter setpoints due to wheel slipping on the surface. Instead, we opted for the other option, described below.**

turnToHeading(float desiredHeading, int msDuration): Sets a motion setpoint to turn to a heading. msDuration dictates a timeout in case setpoint is not reached.

driveForward(float mmDistanceFromCurrent, int msDuration): Sets a motion setpoint to drive a set distance forward. msDuration dictates a timeout in case setpoint is not reached.

driveBackwards(float mmDistanceFromCurrent, int msDuration): Sets a motion setpoint to drive a set distance backwards. msDuration dictates a timeout in case setpoint is not reached.

statusOfChassisDriving(): returns what the status of the driving is. We really care about REACHED_SETPOINT and TIMED_OUT

If you are not trying to reach a certain setpoint, but instead would like to drive until a condition is met, you can call the driveStraight() method. driveStraight takes in a desired heading to maintain, and a direction. If direction is “DRIVING_FORWARDS” the robot will move forwards, and subsequently if it is “DRIVING_BACKWARDS” the robot will move backwards.

```
void driveStraight(float targetHeading, MotionType direction);
```

Line following and the “LineFollower” object:

The robot is equipped with two line sensors. One is located below the center of rotation and is used only for row and column tracking, and the other is used for line following and is at the front of the robot. The functions for line following are found in DrivingChassis.cpp. There is a function prototype for line following if the line sensor was to be mounted to the back of the robot, but its source code is commented out. In the future, if you need a redesign, feel free to re-implement this function.

There is also a LineFollower class type. The robot chassis has an instance of this. This class has the following methods:

onMarkerFront(): call this to determine if the front line sensor is over a line. It will return true if the line sensor is over the line, and false otherwise

onMarker(): call this to determine if the line sensor used to track lines is over a line. It will return true if the line sensor is over the line, and false otherwise

calibrate(): call this to read values of all the line sensors, and determine which thresholds should be used for a black line, a grey line (between white and black), and a white surface.

resetLineCount(): reset the line count that has been tracked.

There is another line following method with the function prototype “lineFollowForwards(int speed)**”.** This will line follow at the given speed in mm/s, **but will not track position!**

Lift Firmware:

The lift firmware is found in the files “LiftControl.cpp” and “LiftControl.h”.

The robot chassis uses an instance of this class called “lift”. Upon booting, the robot starts by performing a homing routine on the lift. It will move the lift mechanism to the bottom, until it detects the limit switch has been triggered. At this point, it will zero the encoder, and move up, until the other limit switch has been triggered. At this point, it determines how many encoder ticks exist per millimeter of motion, and setLiftHeight() can be used. Realistically, you will not need to modify the homing routine, but the explanation above gives you a high level overview of how it works. The functions below are ones you should use whenever controlling the lift.

SetLiftHeight(float mm): Use this function on the LiftControl object to set the lift setpoint

Once the lift setpoint has been set a checking function, similar to that of the basic motion commands should be called. Let's look at an example. This is taken from the bin procurement firmware:

```
case RAISE_LIFT_TO_SHELF:
    lift->SetLiftHeight(binHeight);
    binProcurementState = WAIT_FOR_LIFT_SETPOINT_REACHED_PROCUREMENT;
    binProcurementStateAfterLiftSetpointReached = APPROACH_BIN;
    break;
```

Here, we set the lift to the height of the bin, and then move into the "WAIT_FOR_LIFT_SETPOINT_REACHED_PROCUREMENT" state. This is very similar to the process for motion setpoints.

In the "WAIT_FOR_LIFT_SETPOINT_REACHED_PROCUREMENT" case, the firmware calls CheckIfPositionReached repeatedly, and once the position has been reached, it moves to "APPROACH_BIN" (from the image above). Any routine that uses the lift should follow this approach.

```
case WAIT_FOR_LIFT_SETPOINT_REACHED_PROCUREMENT:
    if(lift->CheckIfPositionReached()){
        binProcurementState = binProcurementStateAfterLiftSetpointReached;
    }
    break;
```

CheckIfPositionReached(): checks if the lift has reached the target position. Returns true if it has, false otherwise.

Navigation Routine:

All navigation code is found in "Navigation.cpp" and "Navigation.h". **The state machine that actually does the navigation is checkNavStatus(), which is called from the "Navigation" case in the main state machine.** The working principle of the navigation routine is as follows:

1. Determine the current row and column (more on this later)
2. If the goal row is not the current row, navigate to the outer column
    a. If it is, turn towards the goal column and find it
3. Once at the outer column, turn towards the goal row
4. Once at the goal row, turn towards the correct column
5. Once at the correct column, finish.

There are comments in the checkNavStatus routine, for additional clarification.

setNavGoal(int row, int col): sets the navigation goal to a specific row and column

The “Pose” object:

In order to keep track of the robot’s current pose, a Pose class was created. The code for this pose object is found in Pose.cpp and Pose.h

The initial pose of the robot is found in Pose.h

The position is updated in the line following routine, as the robot tracks the row and column markers. In order to determine whether rows should increment or decrement, the pose object has a method called “getOrientationToClosest90” which uses the current heading to get the closest ordinal direction, and from there, we know whether we can add or subtract rows and columns. The current heading of the robot is determined by the IMU.

getOrientationToClosest90(): returns the orientation to the nearest 90 degrees. (0, 90, -90, 180)

Parking Routine:

All parking code is found in “Parking.cpp” and “Parking.h”. **The state machine that actually does the parking routine is checkParkingStatus() and getOutOfParkingStatus(), which are called from the “Parking” and “Navigation” cases in the main state machine, respectively.** The working principle of the parking routine is as follows:

1. Turn to the parking space
2. Back up a little bit so that the line sensor is not on the outer edge
3. Back into the parking space

To maneuver out of a space, it’s virtually the same thing, but in reverse:

1. Drive forward so that the line sensor is not on the line used for parking
2. Drive up until we have reached the outer edge of the world

There are comments in the two routines, for additional clarification.

Bin Handling Routines:

All bin handling code is found in “BinHandling.cpp” and “BinHandling.h”. **The state machine that actually does the procurement and return routines is checkBinProcurementStatus() and checkBinReturnStatus(), which are called from the “DeliveringBin” and “ReturningBin” cases in the main state machine, respectively.**
The working principle of the bin procurement routine (checkBinProcurementStatus()) is as follows:

1. Turn to face the bin
2. Raise the lift to the center of the bin (either on shelf row one or two)
3. Approach the bin slowly by following a line
4. Grab the bin by raising the lift a little bit

    a. Make sure the bin is successfully grabbed. If not, go to step 3. Otherwise go to step 5. After 5 retries, report unsuccessful delivery. Go to step 5.
    b. If no bin exists, report that a bin is missing. Go to step 5.
5. Back up to the world
6. Lower the lift

The working principle of the bin return routine (checkBinReturnStatus) is the same thing, but in reverse:

1. Turn to the shelf
2. Raise the lift to the shelf height
3. Approach the shelf until the front line sensor detects a line
4. Lower the bin onto the shelf
5. Back up to the world
6. Lower the lift