ECE 118: Introduction to Mechatronics

Final Project: Slug World Cup

Leo King, Mathias Eriksen, and Ariyan Kalami

December 8, 2023

Introduction

This is a project to make a fully autonomous robot that accurately defeats Sammy the slug in the Slug World Cup. This is a class competition where we design a bot to shoot ping-pong balls at a goalie, Sammy, and depending on shooting location we score additional points. The bot is required to have the ability to resolve collisions with obstacles blocking our road to the goal.

We wanted to be able to work on multiple tasks at one time and make testing modular so we decided on a three platform design. The bottom layer handles actuation and events, the middle layer handles control, and the top layer handles shooting. They can be attached together with four all thread bolts making the three layers detachable. Having this design allows for electrical, mechanical, and programming design worked on in parallel. This makes integration of components easy because we can progressively add onto each layer.

To actuate and find the locations to score the bot uses IR light sensors and motors to move to target locations. After accurately getting to each designated location the robot's second set of motors drive the flywheels to blast a ping-pong ball past Sammy. All this logic is done in C programming with multiple state machines that use the event service routine. The program is loaded onto a PIC32 microcontroller that handles the autonomy of the robot.

After choosing our design we created a loose target timeline such as when each layer is completed, when the shooter is completed, and when the coding framework is completed.

1 Mechanical Design

For our robot's design, we decided to go with a 3-layer stacked octagon. Each level houses different components of the bot, in order to utilize the space as efficiently as possible. We used four beams to keep each level together, which also allowed for more modularity and easier access to the components. The shooter is housed on the third level, along with the power and left/right side switches. The second level stores the computing components, such as the PIC32. The first level is where all of the sensors reside along with the motors for the wheels. In order to save on weight, we made various cuts on the second and third levels, which allowed for more mobility due to the decreased weight.

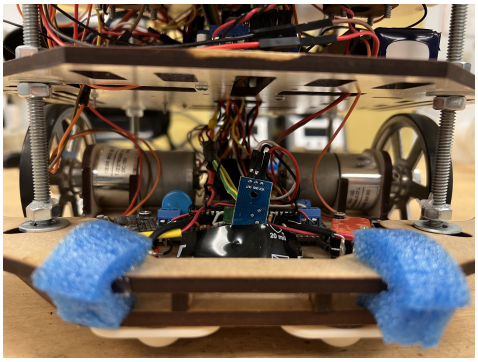


Figure 1: Sensors and motors of first level

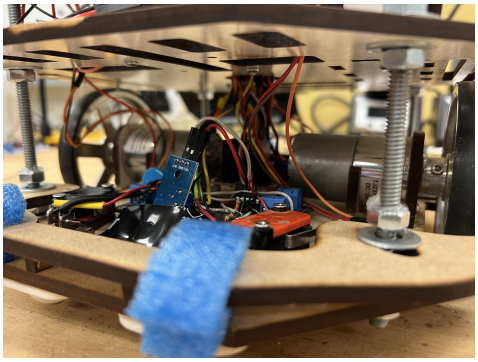


Figure 2: Side view of first level

1.1 Level One

We put all of our tape and track wire sensors on the first level, along with the bumper sensors and bumpers themselves. Originally, we had placed the IR sensors responsible for detecting the tape on the field directly under the bot. However, we eventually found that making a cutout for the sensor and placing it from the top yielded better and more consistent readings from the sensors. The side IR sensors used for detecting the wall were also positioned at an angle in order to yield better results. The bot was kept stable using ball bearings attached with adhesive on both the front and back undersides. In order to prevent hard collisions from rattling the bot too much, we also placed foam on both ends so that there's extra cushion. We also housed the motors for our bots' wheels on this level.

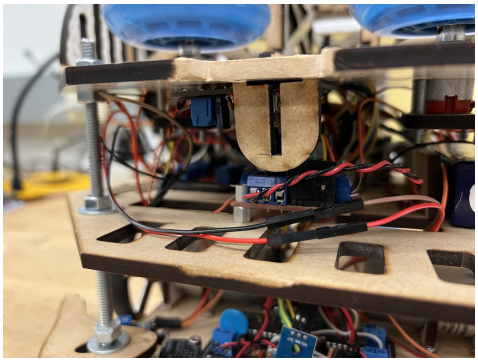


Figure 3: Beacon detector and second level weight-saving cuts shown

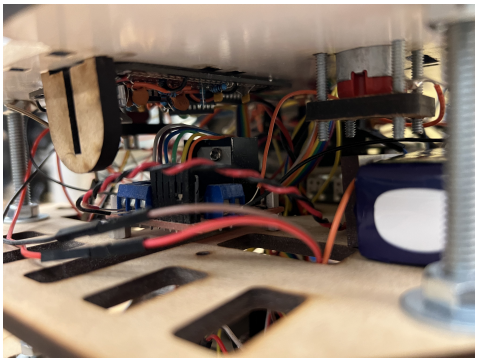


Figure 4: In between the second and third level

1.2 Level Two

On this level, we housed the "brains" of our robot, including the PIC32 and battery. We made extruded cuts throughout the base of this level in order to save on weight, along with a hole in the center for the wires to pass through. The battery can be slid in between and out of two MDF walls which hold it in place.

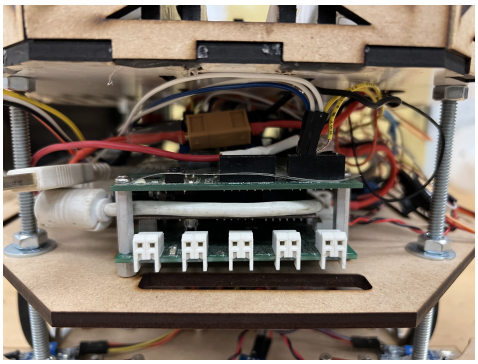


Figure 5: Backside of second level

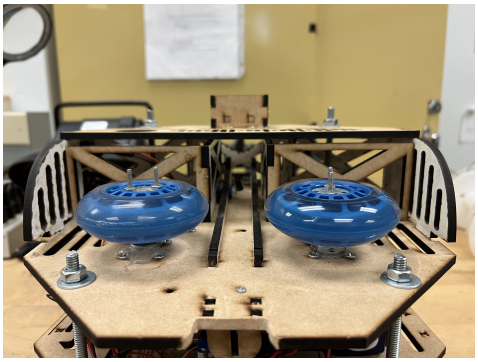


Figure 6: Front of third level

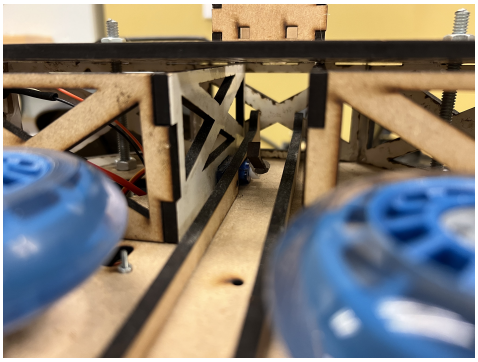


Figure 7: Servo arm

1.3 Level Three

Our flywheel shooter and beacon detector are both housed on this third level. The beacon detector is attached on the underside of the level toward the front, along with a slight ridge cut above it to improve the radius of our detector. The ramp is mostly encased by a box in order to allow the ping-pong balls to stay in place. The balls are held in place and released by a servo attached to the side of the ramp, with an extension made of MDF connecting to the servos' arm. We also added a small box at the very top in order to store all three balls.

The flywheels are in the open at the front of the bot. The flywheel motors are held in place using a cut piece of MDF to hold it from below, with bolts going from the MDF piece through the base. For extra stability, we also added hot glue around the motors so that they do not spin out of place. Comparing it to other methods of keeping the two motors in place, we eventually decided that this was the safest and most reliable option.

Both the base of this level and the walls of the box had extruded cuts in order to reduce the bots' weight.

2 Electrical Design

The microcontroller is the control system for the motors and circuits. To actuate and receive events, requires understanding the electrical aspects of every component. This includes knowing the I/O voltage needed to drive our components without breaking them. Each electrical component has it's own data sheet describing the minimum and maximum input voltage for a high and low. Figure 8 shows the electronics input and outputs to the microcontroller.

PIC32 Microcontroller

To use the PIC32 microcontroller input and output (I/O) ports, the microcontroller needs the tri-state ports enabled. Then set to input or output in code. The I/O ports have digital high signal at $3.3\,V$ and

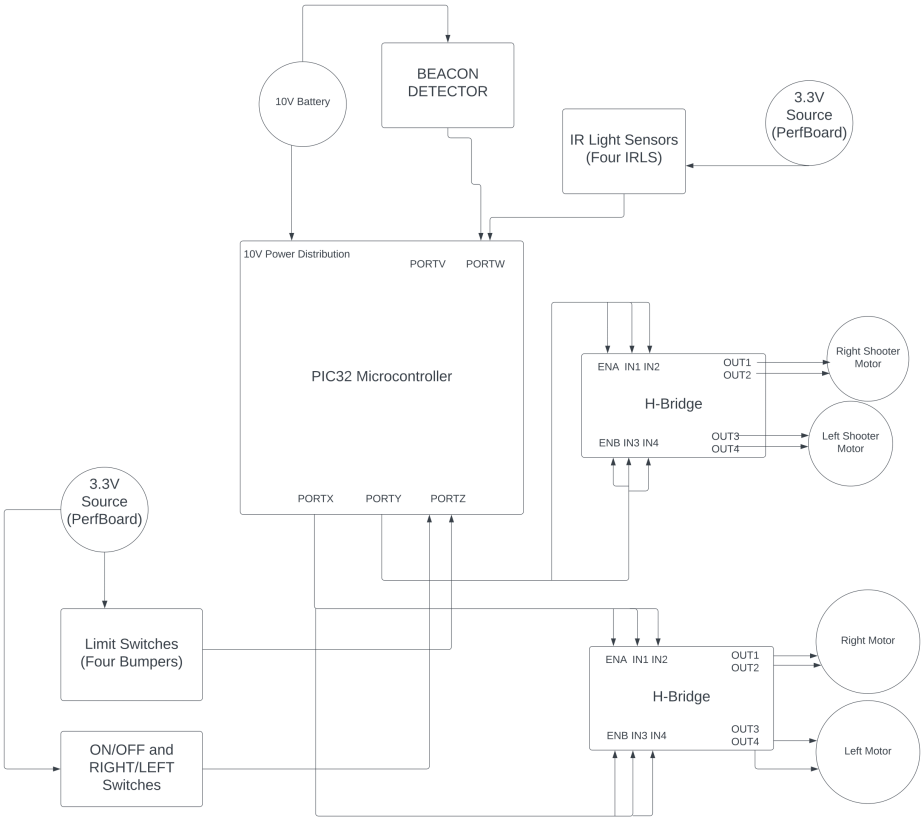


Figure 8: Complete electronic block diagram of the control layer

low at $0\,V$. Input values greater than the digital high voltage would blow the pin. The microcontroller has many load contingents to protect the pins such as:

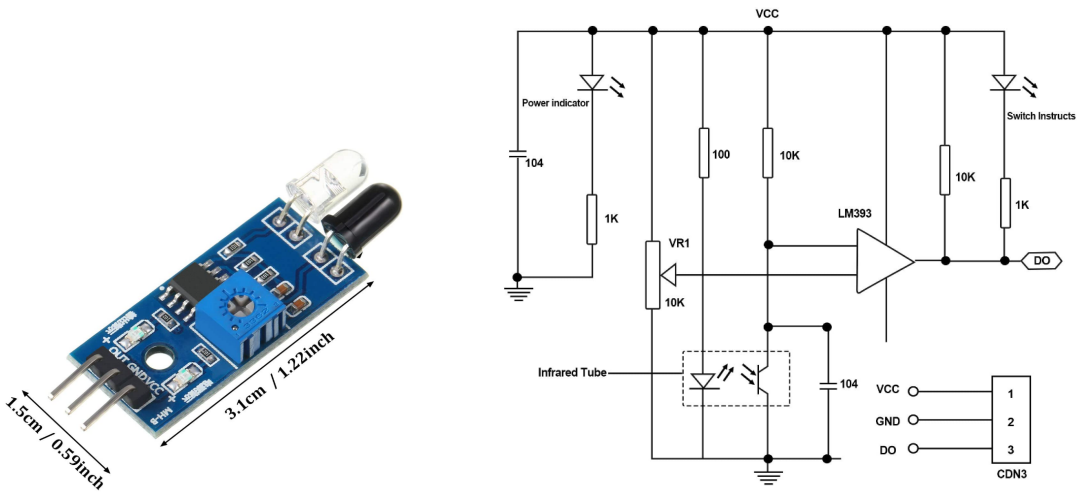
1. Built in clamp diodes to protect from negative voltage at the ports
2. Each pin has a resistor in series to protect from high input current
3. $3A$ fuses for protecting each half of the power distribution board, viable for $6\,V$ to $30\,V$

IR Light Sensor

The schematic shows that light sensor uses a comparator to swing to a digital high and low depending on input voltage, V_{cc}. Since the microcontroller takes inputs highs as $3.3\,V$ then V_{cc} on the IR sensor can be $3.3\,V$. The light sensor can take up to $5\,V$ so if the goal was increase distance range the input voltage needs to increase then dropping the output voltage will drive the microcontroller without breaking a pin. The sensor also has a potentiometer to adjust the range detection.

H-Bridge

The H-bridge has six input pins and four output pins. The H-Bridge uses an enable pin and two pins for direction control. This controls the output direction that drives the motors. The microcontroller drives the enable pin with a PWM output port to control the speed of the motors. The benefit of the H-bridge is it allows for control of two wheels on one board and had built in inductive kickback protection from drive motors. All the input pins take $3.3\,V$ as active high and outputs the voltage from the power supply to the motors.



(a) Physical IR Light Sensor

(b) IR Light Sensor Schematic

Figure 9: IR Light Sensor for sensing the tape and wall

Drive Motors

The drive motors are 150 rpm motors which are connected to the H-Bridge driving the wheels forward and backward. They are low in rpm which makes shooting our ping pong balls ineffective. However, since they are high torque they work well to drive the bot forward because it can drive the weight. At $10\,V$ they draw about $700mA$ of current with no load. The motors can be driven faster given the motor is supplied higher voltage.

Fly-Wheel Motors

The Fly-Wheels are driven the same way as the drive motors. The main difference is that the Fly-Wheels are driven by a separate H-Bridge driving and have limited drive at $3\,V$ to $12\,V$. Whereas the drive wheels need greater than $3\,V$ to drive and can drive way above 12V. They are both PWM controlled.

Power Distribution Board

This power distribution board is built on a perfboard specifically to drive our $3.3\,V$ components. It uses a LDO 3.3V regulator and must be given greater than $5\,V$ to be $3.3\,V$ stable. It uses large capacitors to ensure there is not noise at the input nor output pins. The output and ground is connected to five screw terminals to distribute voltage. The screw terminals provide a highly robust power supply for the other components (IR light Sensor, limit switches, and control switches).

Limit Switches

The goal is for a 3.3V output when the bumper is pressed. This is done with a $100k\Omega$ resistor in series to remove any high current going into the microcontroller. The schematic for the bumper limit switch in figure 10.

ON/OFF and Left/Right Switches

The ON/OFF and Left/Right switches are designed the same as the limit switches. Only the switch component active use is different. When the ON/OFF switch is flipped the "ON" the microcontroller power distribution board receives the voltage from the battery and fully powers the microcontroller. At the "OFF" position we get an open circuit. The Left/Right switch simply represent static values we can read for high or low in the microcontroller. The schematic for the limit switch in figure 10 is the same for the ON/OFF switches.

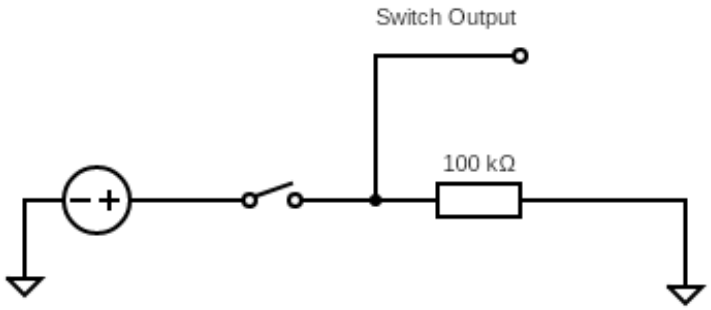
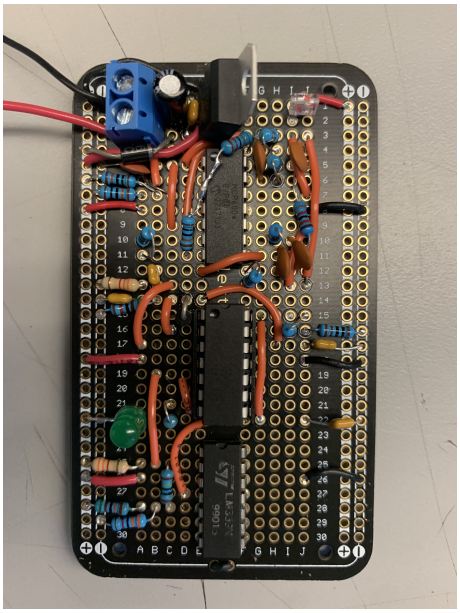


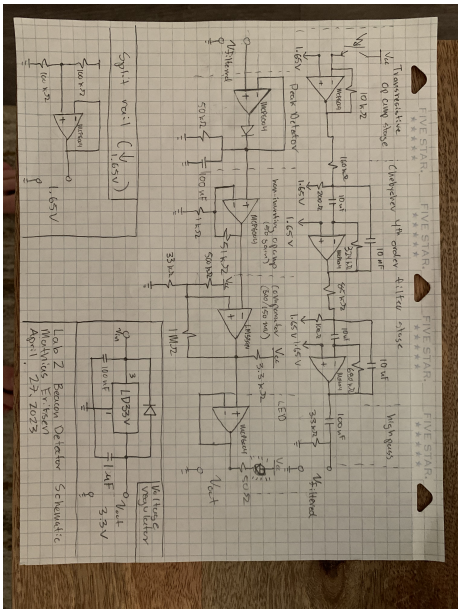
Figure 10: Schematic of the switches at active high

Beacon Detector

The beacon detector takes a $10\,V$ V_{cc} from the microcontroller distribution board and powers the beacon at 3.3V. The beacon uses many stages of gain and an fourth order Chevychev filter to get a usable signal for a comparator. The comparator provides a digital signal for the microcontroller. Figure 8 shows the microcontroller connects to analog pins we use in our program.



(a) Physical Beacon Detector



(b) Beacon Schematic

Figure 11: IR Light Sensor for sensing the tape and wall

3 Program Design

In order to write the software for the robot, we used the Event Services Framework provided. This meant writing event checkers to interact with hardware inputs in order to generate events in response to real world stimuli, as well as writing a hierarchical state machine to decide what to do in response to events. In this section, we will go through the thought process behind the design of the software, starting from the event checking code through to the lower level sub state machines.

3.1 Program Design Goals

The goal for the software, and our overall strategy for the game was to follow the wall up to the one point line, use a hard coded turn to shoot in one side of the goal, follow the wall back and reload. In response to collisions, we would retrace our steps, and follow the opposite wall down to the one point zone and shoot. This meant that event checkers were needed for many different sensors in order to generate the events needed in order to move through the state machine correctly. Each of the event checkers is described below.

3.2 Event Checking Code

The event checking code is all called on external ES timeout events in the main state machine. The event checking timer was on a 3 ms loop, so each event was being checked on every 3 ms rollover. All of the code was written in the event checker file, with functions for checking each sensor block. The subsequent sections will describe the strategy for checking of each sensor block. Each event checker directlt posts events to the main state machine.

Front and Back Tape Sensors

For this event checker, digital I/O ports were used to read the digital output pin from the two IR sensors on the bottom front and back of our bot. The events generated in this event checker are front and back tape tripped and untripped events. We did not find a need to add any further filtering through polling or moving average in this event checker as the reading from the IR sensor when crossing the tape was stable enough with the 3 ms timer the event checker was being called on. In order to

read the value from both IR sensors and handle it with only one variable, we used bit shifting in our read function, in the following manner:

```
unsigned char Read_DigitalTape(void){

    return (TAPESENSOR_F << 1) | (TAPESENSOR_B);

}
```

This way, we could look at the current value of both IR sensors using only one variable, and compare to previous value of each IR sensor individually using bit masking.

Wall IR Sensors

For the wall IR sensors, we had one further forward on the bot and one further back on the bot on each side. These were tuned to read give a high reading when within a desired distance from the wall and are used in tne state machine to implement a simple wall following algorithm. For these sensors, we used AD ports since we wanted the capability of reading directly from the phototransistor in order to get an analog reading proportional to distance from the wall, but we ended up just using the digital output from the IR sensors.

Since we have an analog reading coming from an A/D pin on each of the wall IR sensors, we had to implement our own noise margins like those on the digital pins of the pic32. For our noise margins, we ended up using 1.1V as the voltage input low maximum value, and 2.58V as the voltage input high minimum level. These threshold values worked well for these sensors, and we raised events only when an individual IR sensor changed readings.

Bumpers

The bumpers posed more of a filtering challenge to the software than the IR sensors did because of the nature of the switches themselves. Just like any physical button or switch, the physical press causes a large amount of bouncing in the voltage level as the state of the switch moves from one state to another, and this bouncing voltage can lead to an event checker spamming events which aren't actually real events unless you are careful with your code. In order to debounce our buttons, we implemented a polling strategy. To do this, we polled the reading from the bumpers on each call to the event checker, and checked it against the bumper reading from the last call. If it was the same reading, we would increment a variable by one. If the counter reached 10 counts without the bumper reading changing, we would pass an event if one had occured. In this manner, events will only be passed in instances where the bumper reading is stable for at least 10 different event checks, or 30 ms. If there is any bouncing at all in the period, and the previous and current readings do not match, the counter variable is reset. We did this in code in the following manner:

```
if (Bumper_Curr_Reading == Bumper_Prev_Reading){
      i++;
  } else {
      i = 0;
  }

  if (i >= 10) {
      Bumper_Curr_Event = Bumper_Curr_Reading;
  }
```

To actually read the bumper values and pass the bumper event, we once again used bit shifting. In both reading and outputting the state of the bumpers, we used an binary pattern with bit positions 0-3 representing on of the four bumpers. We read the bumpers using the following code:

```
    uint8_t BumperRead(void) {
    uint16_t PORTXPINS = IO_PortsReadPort(PORTX);
    uint8_t Bumper = 0;

    if ((PORTXPINS & BumperInFrontLeft)) {
        Bumper |= BumperFrontLeft;
    }
    if ((PORTXPINS & BumperInFrontRight)) {
        Bumper |= BumperFrontRight;
    }
    if ((PORTXPINS & BumperInBackLeft)) {
        Bumper |= BumperBackLeft;
    }
    if ((PORTXPINS & BumperInBackRight)) {
        Bumper |= BumperBackRight;
    }
    return Bumper;
}
```

Finally, the event checker will pass a bumper event when any of the four bumpers has changed states, and will pass a parameter with the binary pattern corresponding to the state of the bumpers.

Beacon

The beacon event checker was extremely easy to write, since the board with the beacon detector on it uses a comparator to output the digital signal, hystereses bounds were already built in to the circuit itself. Therefore, the reading from the beacon was smooth enough to just read the digital input pin and check against last events in the event checker.

TrackWire

The track wire code was very simple, just reading from one digital pin, but needed some filtering. The track wire code needed filtering due to some interesting hardware reasons. Since the inductor which picks up the track wire signal was placed in close proximity to the drive motors, the large current running in the DC motor coils was enough to cause a voltage response in the inductor. This issue could have been fixed with filtering on the detection board, but was easier to fix in software. Since the motors alternating magnetic fields cause a low frequency AC signal to be induced in the inductor, and the track wire circuit already had a low pass filter on it, the noise induced by the motor would only cause flickering in the track wire output, not a solid high signal. However, the signal produced when actually driving over the track wire was a solid logic level high signal. So, simply polling 5 counts of the trackwire signal using the same strategy as used for the bumpers was enough to remove the motor noise from the circuit completely on the software side. However, this event checker did not end up being used in the final iteration of the project since we did not shoot 3 pointers, and thus did not need to detect the signal.

3.3 State Machine

The state machine for this task needed to be robust, capable of handling any edge cases that may occur during a run. This mainly added complexity in the main state machine due to needing to handle collisions, and multiple different sensor events in many different states. The following state machine was built from extensive testing and therefore is quite complicated to follow, but each block will be described in the following sections.

Forward Half of State Machine

The following image shows the state machine for the forward half block of the state machine:

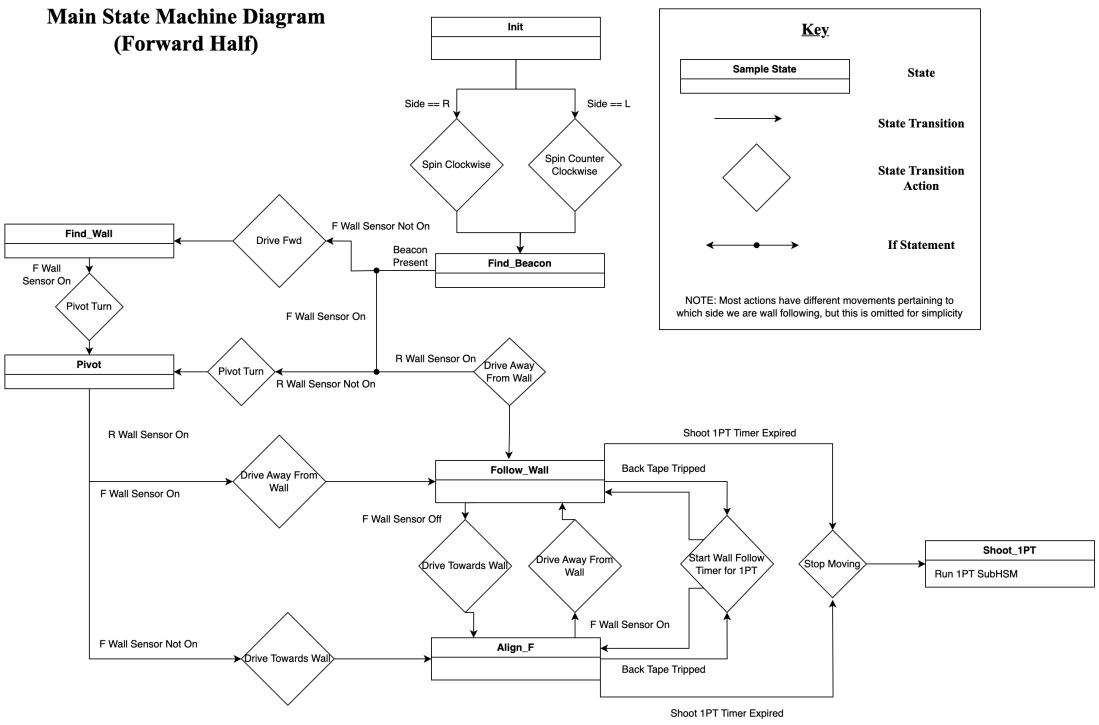


Figure 12: State Machine for Forward Half

The state machine for the forward moving half of the state machine is somewhat simple. We begin by looking for the beacon, and based on which side we are on we spin clockwise or counterclockwise. We spin in a different direction due to the robot skidding after attempting to stop once the beacon has been spotted. Using this spinning technique, the robot will skid and be angled towards the wall we want to find once the beacon has been detected. Therefore, we can simply move forward after the beacon has been detected, as we can see in the state machine above. There are some edge cases to be considered when moving from detecting the beacon to following the wall. For example, you can start very close to the wall, and the finding of the wall and pivot to align with the wall can be omitted in some cases. As we can see in the above state machine, these cases are accounted for using both direct reads from wall sensors as well as events from each sensor. Once we have aligned with the wall, we use the follow wall and align wall states to follow the wall with a simple algorithm. The algorithm simply uses the front wall sensors in order to move away from the wall until the wall is lost, and then towards the wall when the wall is lost. Then, once the wall is found again, we move away and repeat. This strategy worked the best for us to get to a reliable position for shooting one point shots. The implementation of this wall following meant that the aiming for the shot could be hard coded, simplifying our hardware dependency when shooting considerably.

Shooting Sub State Machine

Below is the sub state machine used for shooting one and two pointers. The two point state machine is omitted for simplicity since it is essentially the same as the one pointer, but with a different hard coded turn, and only releases one ball with a different release timer.

As we can see, the sub state machine is a timer based sub state machine which follows some simple steps. First, it uses a defined amount of timer ticks to execute a hard coded turn. The number of ticks for a left and right turn are experimentally determined and are quite different due to some hardware differences. the hard coded turns we used for the one pointers were as follows.

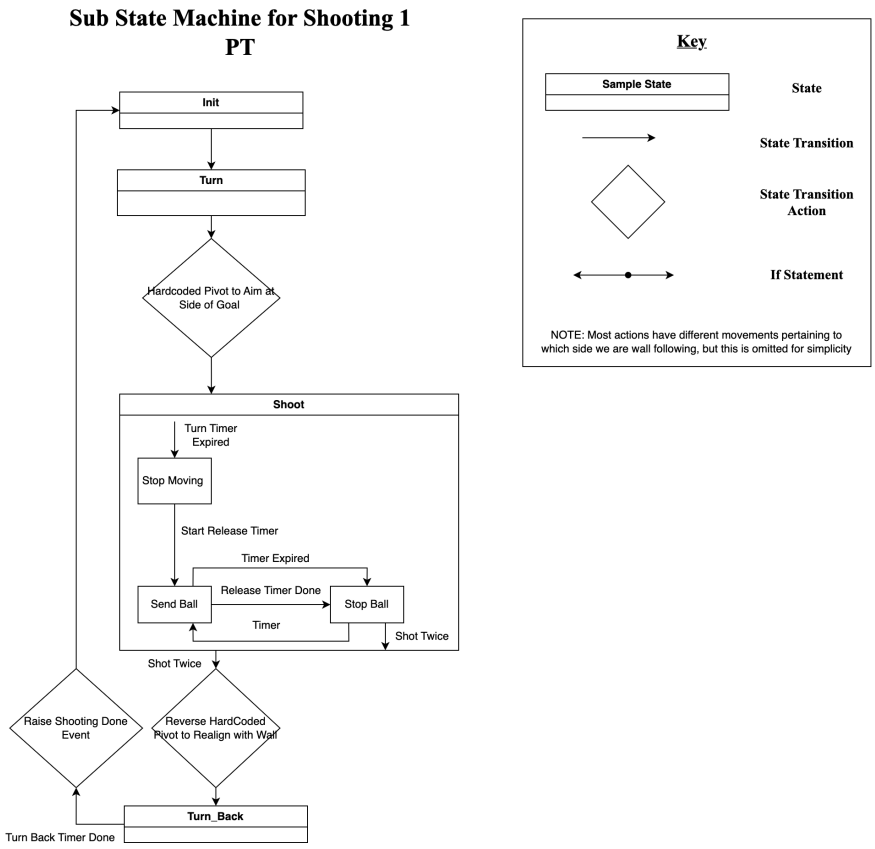


Figure 13: Sub State Machine for Shooting

```
#define TURN_1PT_TIMER 2
#define TURNL_1PT_TICKS 100
#define TURNR_1PT_TICKS 15
```

As we can see, due to the way the robot comes to a stop while wall following, the turn needed when shooting from the right side is considerably less than on the left. This is most likely due to a difference in the drive motors causing the robot to drift slightly left whenever it comes to a stop.

Using this shooting technique, we could reliably shoot in the side of the goal using nothing other than a hard coded turn to aim at the goal. This made our state machine much simpler and allowed us to significantly lower the amount of time one run took when running the robot.

For the two point shot, the state machine is the same except for a different hard coded turn, the turn times for the two pointers were significantly smaller because the robot is further way, leading to a smaller angular turn being needed to aim at the side of the goal. In order to make the two point shooting more accurate, we also took a reading from the back wall sensor before deciding how much to turn, since a wall present signal would indicate the robot already being angled towards the goal and away from the wall, and vice versa for a wall absent signal. So, in software, we would turn less if the back side wall sensor was high, and turn more if it was low. This addition led to much more accurate two point shots. The code used to implement this was as follows:

```
if (Side == LEFT){
        LeftWheelSpeed(300);
        RightWheelSpeed(0);
        if (Analog_TapeRead_L() < 450){
```

```
            ES_Timer_InitTimer(TURN_2PT_TIMER, 10);
            CurrentState = Shooting;
        } else {
            ES_Timer_InitTimer(TURN_2PT_TIMER, TURNL_2PT_TICKS);
            CurrentState = Shooting;
        }
    }
```

As we can see, we take a read of the back left tape sensor if we are on the left side, and turn less if it is already on (only 10 ms turn), since this means we are already angled towards the goal.

Reverse Half of State Machine

The image below shows the reversing half block of the state machine:

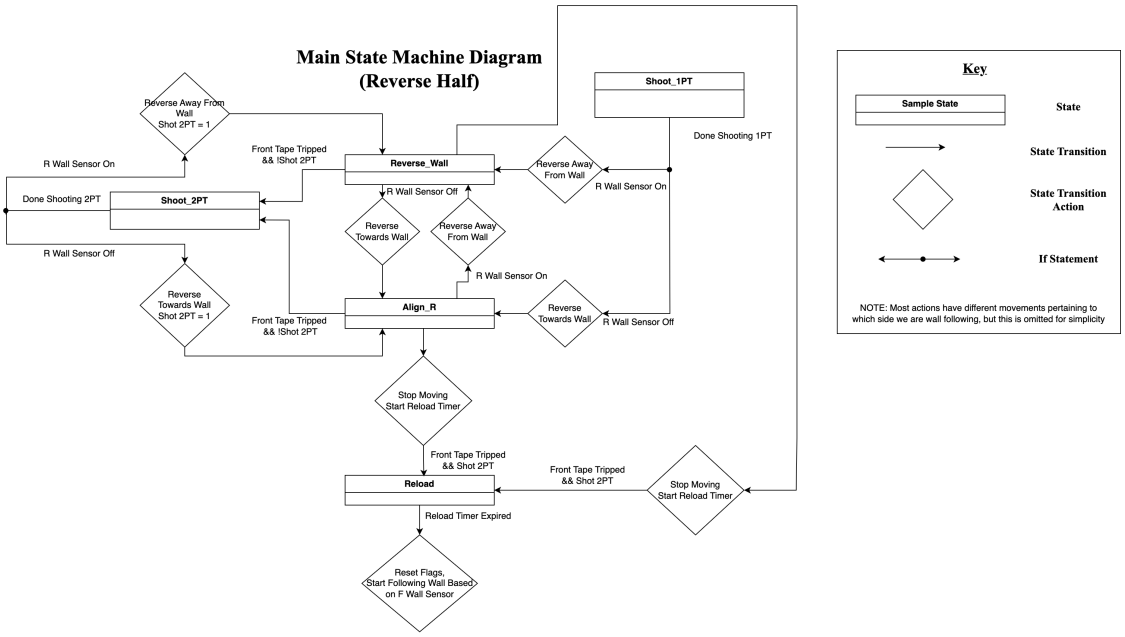


Figure 14: State Machine for Reverse Half

This state machine shows the state movements continuing from the forward block. This block of the state machine is entered once the done shooting one point event has been passed from the 1PT shooting sub HSM. Once we are done shooting the one pointer, we use the same wall following algorithm in reverse using the back wall sensors. Then, once the one point line has been fully passed, we will get a front tape sensor event which means it is time to shoot a two pointer. We then use the two point shooting sub HSM and continue following the wall backwards once we are done shooting. Since the same front tape sensor event is used for both shooting the two pointer and signaling a reload, we raise a flag once the two pointer is done being shot so the next front tape event will cause a state transition into the reload state. If the whole state machine is executed correctly without any collisions, the state machine will reset all the flags and go back into following the wall forwards after the reload timer has expired. This completes the loop and allows the state machine to loop back around repeatedly.

Handling Collisions

In order to handle collisions, the state machine becomes a bit more complex, which is why we omitted the collision cases from the state machines shown above. The way we handle collisions is based on a

simple principle, and the simplified collision state machine is as follows.

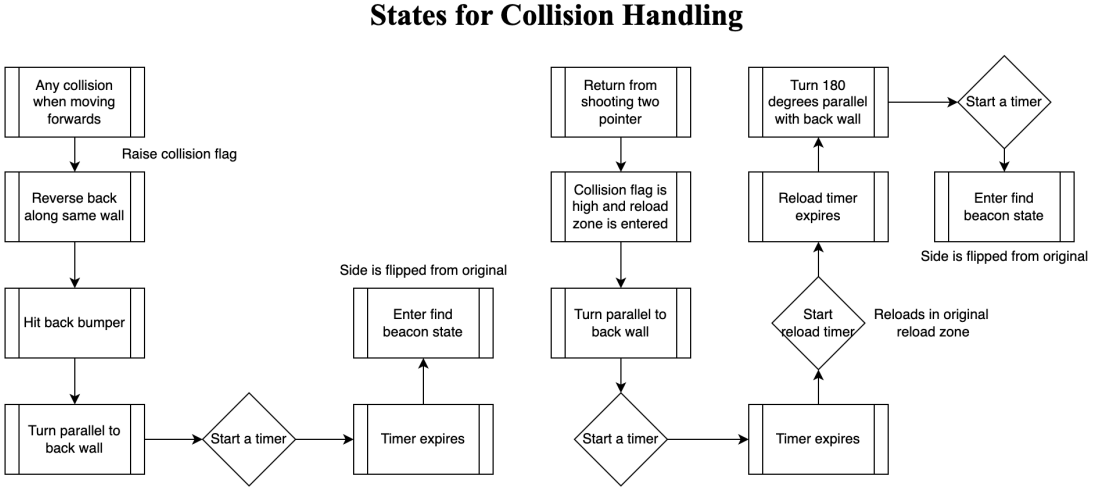


Figure 15: Simplified State Machine for Collision Handling

Essentially, our collision handling is built on the premise that our wall following is good enough that we never hit the wall using the IR sensors, so that we can use any bumper event as an indication that we have had a collision. And, in practice, this works, as long as the calibration with the wall sensors is good enough and the bot is not skidding too much in its pivots.

Since we are wall following, and the state machine just runs up and down the wall of whatever side we start on, we can check only the front bumper for a collision when moving forwards, and assume the path is clear on the way back since we just drove over the same path moments before. Because of this, we can use any front left or right bumper event whilst moving forwards as a collision. Once we have a collision of this type, we begin the state machine above, and permanently raise a collision flag. The state machine above is built into the prior state machine blocks in code, and is omitted from previous sections for simplicity.

Once a collision has been detected, we simply use the reverse wall states to follow the same wall we started on back to the reload zone. We know we are back in the reload zone when we hit the back wall with the back bumpers. Once the back wall has been hit, we follow the back wall on a timer until we get to the opposite sides reload zone, and enter the find beacon state from the forward block of the state machine shown in the previous section. When this happens, we change the value of the side global variable to the opposite side we started on, so we will follow the opposite side down to the shooting zones. This is the state diagram shown on the left side of the figure above.

Next, once we are done shooting on the opposite side to the one we started on, we use the collision flag to determine whether or not to reload in the side we are shooting from or the opposite side. In a no collision case, we would just get to the reload zone, reload and follow the wall back up. But, if we have had a collision previously, we know we are shooting from the opposite side to the one we are reloading in (must reload in the starting side). So if we return to the reload zone with a collision flag, we must follow the back wall to the other zone, reload, turn back and enter the find beacon state again to go down the same side and shoot. This is the state diagram shown in the right side of the figure above.

In practice, this collision handling is reliable and pretty efficient. The way it is written, the collision flag global variable allows us to only need to make the actual collision once in order to determine which side to shoot from, and all subsequent runs simply avoid the obstacle by going down the other side. The combination of wall following using IR sensors and bumpers for collision detection made

this code optimal for use in the min spec checkoff in this lab. This strategy also helped us save time on opponents who were using only bumpers and trying to resolve the collision by moving around the object.

3.4 Software Conclusions

In conclusion, the software we wrote for this task worked quite well, especially in the min spec setting. Using the very efficient shooting techniques and collision handling we were able to be the first team to finish the min spec checkoff, getting it on our first try. The most difficult part of writing the software for this project was really just difficulties with hardware while testing. There were so many moving parts and hardware concerns that made it harder to test the actual software than it was to write it. Looking back, if we were to rewrite the software, we would use many more sub HSMs to handle things like wall following, as the state machine we wrote ended up getting way to complicated and hard to follow. In addition, we would like to have added some more comments and drawn out more code before writing it, as this would have saved us more time in debugging and helped us understand what was going on in the code more. Overall, the software we ended up writing was quite robust, and in our opinion, was a good solution to the task we were given.

3.5 Bill of Materials

The total cost of our materials to create the robot ended up being 67.76. Most of this cost is due to buying all thread rods, nuts and washers for building the robot, and a lot of things that we used we acquired for free from lab services. The BOM is shown below.

Item	#	Price
IR Sensors	6	$5.27
Robot Wheels	2	$14.07
Mini Caster Whe	4	$5.79
H-Bridge	1	$3.20
1/4" All Thread	4	$10.36
1/4" Nuts	100	$5.49
1/4" Washers	100	$5.79
130 DC Motors	4	$4.80
MDF Sheet	2	$0.00
Limit Switches	4	$0.00
Perfboard (BB)	7	$12.99
Micro Servo	1	$0.00
Threaded Bolts	42	$0.00
Nuts	22	$0.00
Rubber Wheels	2	$0.00
12V DC Motor	2	$0.00
Switches	2	$0.00
Total		$67.76

Figure 16: Project BOM

Conclusion

While we encountered countless problems throughout the entire build of our robot, we were able to get it functioning the way we had hoped in the end. Each section had its own set of challenges,

which became easier and easier to deal with as more time was spent working on the robot, though that did not stop other challenges from arising either. If we were to do this project again, we would approach the software design with more sub-state machines in order to declutter the main file and improve efficiency. We would also have better wire and cable management, as there were times when pins would fall out and figuring out which component it led to was a bit challenging. Overall, from starting from scratch to building the fully functional bot, this project strengthened our understanding of mechanical, software, and hardware design, especially due to the fact that we were in charge of most things on our own. This project forced us to build upon the fundamentals we already have, along with pushing us to work with a team in a new environment.