Assignment 2
Microcontroller

Luke Doman
Team 10

Introduction

The purpose of this assignment was to familiarize ourselves with programming the microcontroller that we will be utilizing for all subsequent assignments. The microcontroller is part of the Vex robotics kit. Our task was to utilize the Squarebot robot we built in our previous assignment as a target platform for custom software we write in this assignment. There were three programming tasks to complete. The first is writing a simple function that calculates the hailstone sequence of a given number. The second is coding a motor to modify it's speed based of a light sensor. The third and final task is to utilize ultrasonic sensors to implement object avoidance in Squarebot.

Software Design

The first of our tasks was a simple problem to tackle. We implemented a helper function that when given a number, it calculates the next hailstone number. Then in main, we were able to use a while loop to repeatedly call this function on the same variable, which outputs the entire sequence of hailstone numbers starting the initial variable value. Please see implementation in Appendix a.

Our second task was a more interesting problem. First we had to configure the light sensor to properly interface with the hardware and software. Once we had it configured, to get some baseline readings and verify its operation we conducted tests evaluating the sensor values under the following circumstances: shining a flashlight at the sensor, covering the sensor, and just letting the sensor detect ambient light. Our idea was to implement this such that the motor speed correlates to a percentage of the light the sensor was detecting based off our measured values. The minimum and maximum sensor values we observed in test differed from the values specified in the sensor documentation. By calibrating our software to accommodate for these differences we were able to fine tune our robot's performance. Please see implementation in Appendix b.

The third task gave us some difficultly due to our initial software architecture approach. Our idea was to make the motor speed a function of the current sensor value. If the sensor returned an "out of range" value, then the robot continues full speed ahead. Otherwise if there was a valid reading the speed for the motors would be calculated based off of the following logic: speed = (sensor_value – distance)*2, where "distance" is the the variable that defines the distance in inches at which we want the robot to turn and avoid the object in front on it. The value for distance that we decided on was 10 inches. In order to understand the logic for multiplying that value by 2, I must first explain an edge case for this approach.

By acting on this logic the robot will slow down as it approaches an object. This allows for more data to be collected by the sensor in order to make a decision to avoid or stay course. The motors have "dead zone" of [-10,10] where at those speed values they do not turn. So when the robot approaches an object and the sensor value is <= 20, subtracting by our distance, 10, the robot would be left in a dead zone and just stay stationary. By using a multiplier we were able to provide the motors with enough speed to either continue its approach to the object or execute a turn. The decision to turn or not was a function of the calculated value for speed, where if the speed was below a threshold then execute a turn because we

are really close to the object. We did testing to experiment with what the optimal value for this would be, but experienced erratic behavior using this logic.

After several attempts to make our approach work without luck we then decided to dial back our logic and simply use an if else that calculates whether of not to turn based off the current sensor value. If the sensor read: < 0 or > 35 then move forward, and otherwise execute a turn. Please see implementation in Appendix c. Our initial approach can be seen in the commented out code of that section.

Performance Evaluation

Our initial approaches for tasks 1 and 2 performed nominally without much troubleshooting. We experienced a great deal of unexpected behavior from our initial architecture for task 3 though. The robot would slow down as expected as it approached an object, but would randomly charge full speed ahead right into it. In very well controlled instances of having the robot move directly towards an intentionally placed object, like a notebook, where it was directly perpendicular to the robots path, the performance was nominal. However outside of these controlled scenarios the behavior was too unpredictable to be deemed acceptable. I believe that from a high level perspective and in a perfect environment the robot would perform quite well, but with so many random variables in a real world environment the approach is unfeasible. Our simplified logic performed the task correctly without much modificiation.

Conclusion

The largest takeaway from this was that sometimes the simpler approach is a better option. While more sophisticated logic has its place some considerations need to be made as to whether or not that logic will behave as expected in real world testing. I think the reason our task 3 initial approach didn't work is due to the sensor not performing exactly as we expected in all instances. This is likely due to angles of surfaces that the ultrasonic waves hit that don't perfectly bounce back to the sensor. A possible solution to this would be to average sensor values over some number of readings and use that value for the speed calculations. This could eliminate "noise" from the sensor readings, but would also mean the robot would not respond to changes in the environment as rapidly.

Appendix

a.

```
int hailstone(int n)
{
	// If even
	if(n % 2 == 0)
	{
		return n/2;
	}
	// If odd
	else
	{
		return 3 * n + 1;
	}
}
```

```
task main()
{
  // Start at 7
  int n = 7;

  // Loop until n = 1
  while(n != 1)
  {
                // Print current number
                writeDebugStream("%d, ", n);

                // Calculate next in sequence
                n = hailstone(n);
  }

  // Print last number (1)
  writeDebugStream("%d \n", n);
}
```

b.

```
#pragma config(Sensor, in1,    lightSensor,    sensorReflection)
#pragma config(Motor,  port2,           leftMotor,     tmotorVex393, openLoop)
#pragma config(Motor,  port3,           rightMotor,    tmotorVex393, openLoop, reversed)

task main()
{
  wait1Msec(2000); // give stuff time to turn on

  while(true)
  {
                // Get current light sensor value
                int sensor_value;
                sensor_value = SensorValue(lightSensor);
                writeDebugStreamLine("Sensor: %d\n", sensor_value);

                // Declare min max
                int minLight = 800;
                int maxSpeed = 127;

                // Claculate percentage of light/speed
                float perc = 1 - (float)sensor_value/minLight;
                writeDebugStreamLine("Perc: %.6f\n", perc);

                // Multiply percentage by max motor speed
                int speed = perc * maxSpeed;
                writeDebugStreamLine("Speed: %d\n", speed);

                // Update motor speed values
                motor[leftMotor] = speed;
```

```
                motor[rightMotor] = speed;
        }
}
```

c.

```
#pragma config(Sensor, in1,    lightSensor,    sensorReflection)
#pragma config(Sensor, dgtl1,  sonar_value,    sensorSONAR_inch)
#pragma config(Motor,  port2,           leftMotor,     tmotorVex393, openLoop)
#pragma config(Motor,  port3,           rightMotor,    tmotorVex393, openLoop, reversed)

task main()
{
        // Declare variables
        int distance = 10;       // Distance to back off and turn
        int sensor_value;               // Sonar reading
        int speed;                                      // Value to pass to motor

 wait1Msec(2000);    // Give stuff time to turn on

 // Main loop
 while(true)
 {
                // Get current light sensor value
                sensor_value = SensorValue(sonar_value);
                speed = 50;

                writeDebugStreamLine("Sonar%d", sensor_value);
                // If sensor value out of range -> max speed
                if(sensor_value < 0 || sensor_value > 35)
                {
                        motor[leftMotor] = speed;
                        motor[rightMotor] = speed;

                }
                // Else set speed to the (sensor value - distance threshold) * multiplier
                // to increase speed out of motor dead zone
                else
                {
                        motor[leftMotor] = speed;
                        motor[rightMotor] = -speed;
                        wait1Msec(200);
                        //speed = (sensor_value - distance)*2;
                }

                //// If speed is less than motor dead zone, multiply again
                //// and set one motor value to negative to turn
                //if(speed < 12)
                //{
                //        motor[leftMotor] = speed*5;
```

```
                //      motor[rightMotor] = -speed*5;

                //      // Wait for turn to complete
                //      wait1Msec(500);
                //}
                //// Else set motors to calculated speed
                //else
                //{
                //      motor[leftMotor] = speed;
                //      motor[rightMotor] = speed;
                //}

                //// Take 5 readings per second
                wait1Msec(100);
        }
}
```