Vinayak Vedantam
02/03/2015

Microcontroller Programming

I. Intro

The purpose of the second lab was to give each group the opportunity to work with Robot C, the development IDE used to program the robot, discover how sensor inputs work (digital and analog), and program the SquareBot directly to view the results. This lab also allowed each group to designate a programmer and discover which members of the group are comfortable developing in C.

II. Mechanical Design

Since this lab was completely focused on Algorithm/Software Design and operated on the predesigned SquareBot, there was very little in the way of mechanical design. The only hardware change made to the SquareBot was the addition of the sonar sensor on the front of the robot in the place of the two bumpers. The bumpers were removed and the sonar sensor was attached using two of the same screws. In addition, I attached two partially threaded beams to each side of the sonar using one 3/4" screw on each side. The purpose of the beams was the protect the sonar from impact should the algorithm or sensor not divert collision in time.

III. Algorithm

The main focus of this lab were the three functions that had to be implemented in Robot C. The first assignment was to implement a function that print the results of the hailstone sequence beginning at 7 and ending at 1. The algorithm was given to us as part of the assignment, so the main point of this assignment was to test out some basic commands in C and start navigating the Robot C interface. I ran into my first issue when coding the 'writeDebugStream()' method. writeDebugStream() takes any value and prints it to the Debug Stream, but the value must be specified with a type first. For example, an integer but be specified with "%d". So the code would look like:

```
writeDebugStream("%d", 5)
```

However, after that brief confusion, I found C to operate much like java. Statements ended with the ';' character and 'if/else' statements were implemented the same way in both languages. Our hailstone function required a couple of rewrites to reach a level of efficiency and accuracy that I felt comfortable submitting. These are the changes that I made each time I rewrote the function:

1. I got the sequence and recursion properly working. At the end of this stage, the function properly printed the recursion to the debug stream albeit with some visual issues.
2. I edited the code to insert a newline at the end of each element. This allowed the sequence to be printed vertically without confusion. This change was implemented by adding a new line character, '\n', within the writeDebugStream method. This character is implemented write after the types of the values have been specified. e.g. writeDebugStream("%d \n", n);
3. I moved the hailstone method above the main task and implemented a call to hailstone from main(). This allowed the hailstone method to be executed as soon as the program ran.
4. The final step I took to make the code as efficient as possible was to completely rewrite the loops of the hailstone method. I moved the 'writeDebugStream("%d \n", n)' method

outside of the if/else statement so that the the function would both look aesthetically pleasing (cut down the total number of lines of code) and function more efficiently. Though this process seems arduous, in reality it took less than half an hour.

The next assignment was to write a program that continuously adjusted the speed of a motor so that it was directly proportional to the light intensity received by a light sensor. In other words, we had to make a program that made the motor spin faster when its sensor was under bright light and slow down when it was in the dark. The actual definition of the program was made infinitely easier thanks to the example code provided by SquareBot's original programing. We merely had to import the light sensor using the line:

#pragma config(Sensor, in1, lit, sensorReflection

This command imported the light sensor into Robot C so that we could use it for the program definition. It also provided the input for the sensor (in1). The motor speed expressions were easily moved from the examples using the bumpers. The motors operate under values ranging from 0 to ±127 (negative values indicate reverse motion). The light sensor, on the other hand, operates from 0 to ~1023. Thus, we needed to find a scale so that the motor would read a 0 value when the room was dark and a 127 value when the room was very bright. This all seemed easy enough, but I soon hit a snag. 1023 is the light sensor value for complete darkness, while 0 is the value for high intensity light. Thus, though my scale (the range of the motor ÷ the range of the sensor) was accurate, my motor ran faster when the room was dark and stop completely when under direct light. I struggled with this next step more than any other in the assignment. With the help of my teammates and the instructor, I finally came to the conclusion that reversing the polarity of the sensor during the calculations would be the simplest way of having the motor operate directly proportional to the light intensity. Because I had noticed light value reading higher than 1023, I subtracted the light sensor value from 1100. Theoretically, I should have been able to reverse the polarity by subtracting the light sensor value from its maximum possible value, 1023.

On our second day in the lab, my group set forth to tackle the third assignment. This assignment asked for us to extend the autonomous obstacle avoidance with an ultrasonic sensor. This would allow the SquareBot to anticipate and avoid obstacles prior to collision and thus avoid them without bumpers. After the hardware installation, we turned to Robot C to begin extending the obstacle avoidance methods. I first imported the sonar sensor into digital port 7 (dgtl7). I then changed the input for SensorValue to take values from the sonar sensor. Since the sonar measured in millimeters, I set the program to run whenever an obstacle was less than 150 mm away from the sonar sensor. I arrived at this number by measuring and estimating the amount of time the robot would need to compute the command and stop while going full speed and not smash into any obstacles. 150 millimeters seemed like a reasonable distance to prevent any collisions/damage, but also not far enough to prevent the program from ever running.

At this point I ran the program and encountered my very first hearth-wrenching reality about robotics. Sometimes things simply don't work. The robot would not function when connected to the computer. Neither my group nor the instructors could figure out what the problem happened to be. In the end we finally managed to get the program working. One possible hypothesis is that the debugger could have been stopping the motor since it was unable to find the wait1Msec() method at the beginning of the program. However, other programs I ran through Robot C functioned very well without this command. Once the program

was working, I stripped all of the light-detection methods out of the sonar function in order to simply the code.

IV. Performance evaluation

My first major insight during this lab was that C programming language functions much like Java. Though the instructors respectfully disagreed with me, I found C to be very easy to pick up, and on occasion, I almost forgot that I was not writing in Java (particularly important to me since I develop for Android in java). Another insight I had while working on the first assignment was that Robot C included only a few distinctive C methods. One such method was wait1Msec(). Robot C programs need to wait for a certain amount of time after running and before executing functions to account for the time needed by the firmware.

I ran into my first difficulty during assignment two. I was unable to figure out the math for reversing the polarity of the light sensor for the longest time. Though it seems simply to me now, I was unable to reason the problem at the time. In hindsight, I should have searched the internet for the solution after corresponding with my teammates, but I was unable to think of an accurate description of the problem. I did discover a cool feature of Robot C during the time I spent trying to reverse the polarity of the light sensor. The hardware inputs can be imported through a very elegant gui accessible in one of the Robot C settings. Thus, the import statements did not have to be handwritten each time. This also decreases the likelihood of error.

The biggest difficulty I ran into was also the most unpredictable. After programming the function for assignment 3, we simply could not get the robot to operate whilst connected to the computer. We followed the usually steps to debugging a technical issue with both the code and the robot. However, nothing could make the robot move. We even tried deleting everything and running 2 lines of code:

```
motor[leftMotor] = 127;
motor[rightMotor] = -127;
```

which should have caused the robot to rotate its motors at full speed, but alas even that did not work. Finally, the robot started to work after multiple resets while being disconnected from the computer. The most aggravating aspect of this malfunction is that we have no definite proof that we started the issue nor that we solved it.

Other than these issues, I quite enjoyed programming the SquareBot and operating Robot C. My only gripe is that our computer seems quite a bit slower than every other computer in the lab and takes a very long time to compile and download simple functions. In addition, Robot C has to redraw twice each time it wants to compile and download any program to the robot. I will likely be the programmer in our group, but I wish that we had a faster, more stable computer.

V. Conclusion

This lab has taught me not to be afraid of programming my robot. The syntax of C and the actual debugging process is very straightforward. In fact, I would go so far as to say the robot is easier to connect and debug than the average android device. As such, I feel very confident going into the next lab. Since I have been doing most of the mechanical and coding work, it will be nice to focus on only one aspect of the robot and leave my group members to figure out the other. Overall, this lab was a huge success.

Vinayak Vedantam
02/03/2015

1. Hailstone Numbers

```
int hailstone(int n) { //define the hailstone method
	writeDebugStream("%d \n", n); //prints the value of n after running through the if
statement
	if (n == 1) { //if n == 1 then the hailstone function is over
		return 1;
  } else if (n%2 == 0) { //if n%2 == 0 then n is even and the algorithm will follow the definition for
“if n is even”
		int next = n/2; // runs the hailstone function for the even integer
		return hailstone(next); //uses recursion to set n = to the next integer in
					//the hailstone function. Thereby, passing it back into the
function
					//to print to the debug stream and find the following integer.
  } else if (n%2 == 1) { //if n%2 == 1 then n is odd an the algorithm will follow the definition for “if
n is odd”
	int next = 3*n + 1; //runs the hailstone function for the odd integer
		return hailstone(next); //same as the past recursive step
	} else return 1; //returns 1 as a failsafe.
}

task main() // the main method
{
  wait1Msec(2000); //waits for the robot operating system
  hailstone(7); // runs the hailstone function with a starting value of 7
}
```

2. A Light-Controlled Motor

```
#pragma config(Sensor, in1,    lit,          sensorReflection)
#pragma config(Motor,  port1,           rightMotor,    tmotorVex393, openLoop)
#pragma config(Motor,  port10,          leftMotor,     tmotorVex393, openLoop)

task main() // define the main method
{
//  int bumper = 1;                  // normal bumper state = 1; when pressed = 0
int sensor_value; //created an instance variable for the sensor value

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

  while(true) //begins the loop
  {
	sensor_value = 1100 - SensorValue(lit); //reversed the polarity of the
                                                        // light sensor
	motor[leftMotor] = sensor_value / 8; // 8 is produced via 1023/127
                                                // (range of sensor/range of motor)
                                                // This provides an accurate scale
                                                   // for the motor to adjust
	motor[rightMotor] = - (sensor_value / 8); // right motor is negative when
```

```
                                                    // when the left is positive and
                                                    // vice-versa
            wait1Msec(1000);
  }
}
```

3. Anticipatory Obstacle Avoidance

```
#pragma config(Sensor, in1,    lit,           sensorReflection)
#pragma config(Sensor, dgtl7,  sonarSensor,    sensorSONAR_mm)
#pragma config(Motor,  port1,           rightMotor,    tmotorVex393, openLoop)
#pragma config(Motor,  port10,          leftMotor,     tmotorVex393, openLoop)

task main() //define the main method
{
//  int bumper = 1;                 // normal bumper state = 1; when pressed = 0
int sensor_value; //created an instance variable for the sensor value



motor[leftMotor] = 127; //started the left motor at full speed
motor[rightMotor] = -127; //started the right motor at full speed

wait1Msec(2000);                                    // give stuff time to turn on
while(true) //begin the loop
 {
        motor[leftMotor] = 127; //made sure the motors are running
        motor[rightMotor] = -127;


   sensor_value = SensorValue[sonarSensor];
   if (sensor_value < 150) {
        motor[leftMotor] = -127; //fullspeed backwards
          motor[rightMotor] = 127; //fullspeed backwards
          wait1Msec(1000);  // keep backing up for 1000ms (1 sec)

          motor[leftMotor]  = 0;  //stop motors
          motor[rightMotor] = 0;  //stop motors
          wait1Msec(1000);
    }
  }
}
```