Summary of Week Five

We should be in Week Six now, but we have not yet picked a time to make up our lost session.

## Weekly Summary

We continued with wiring the chassis and a couple of people were able to get their code up to the Raspberry Pi and executing, but so far nobody has managed to read the encoders and print the results. If you get to the point of executing your code on the Raspberry Pi, you will run into two issues, one is that java can’t find your Main entry point, and the other is that it can’t find the runtime libraries.

Your code is going to be run by either opening up a command shell with the Remote System Explorer or by connecting using RealVNC or Putty. In the command shell you have to execute the program

“sudo java -jar PHSUserProgram.jar”. At that point you will either see the errors described or the program will begin to spit out your printed console messages.

## Fixing up the Runtime

You will fix up both of these issues by making a change in your build.xml file. The issue is in the section of your build.xml that creates PHSUserProgram.jar and prepares it to be sent to the Raspberry Pi.

<!-- Put everything in ${build} into the PHSUserProgram.jar file -->

<jar jarfile="${dist}/lib/PHSUserProgram.jar">

<fileset dir="${build}"/>

<manifest>

<attribute name="Main-Class" value="org.phs.code.robot.RobotMain"/>

<attribute name="Class-Path" value="/opt/pi4j/lib/pi4j-core.jar /opt/pi4j/lib/pi4j-devce.jar /opt/pi4j/lib/pi4j-gpio-extensions.jar /home/pi/Jama-1.0.3.jar"/>

</manifest>

</jar>

Note that “org.phs.code.robot” is the package name for my RobotMain.class. The Main-Class attribute tells java to begin execution at that point. We would normally be able to specify the class path for our program at execution time, but I never found a combination that would work on jar files, so we include that item in the Manifest. The Class-path attribute lists all of the libraries we have decided so far that we have referenced in our code. That last library is the NIST Java Matrix library.

## Collecting Execution Data and Unit Testing

One of the things I wanted to get to today was debugging code and running systematic tests. If you include the Logging.java class file that I placed on the Google share, you can print messages from your program to a text file. It writes to the directory /home/pi/log, which has to be created manually.

You can still test even if you don’t use the Raspberry Pi. You do this by invoking Junit.

---------------------------------------------------------------------------------------------------------------  
package org.phs.code.test;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

import org.phs.code.robot.Planner;

import org.phs.code.robot.StateAssessment;

class TestPlanner {

StateAssessment state = new StateAssessment();

Planner planner = new Planner();

@Test

final void testGetCurrentWayPost() {

double testWayPost[] = { -40, -40, 0 };

assertArrayEquals(planner.getCurrentWayPost(), testWayPost);

}

@Test

final void testCalculateNextMove() {

int[] nextMove = { -100, -100, -150, -150, 0 };

state.init();

state.setCurrentPose(-150, -150, 0, 0);

nextMove[0] = 100;

nextMove[1] = -100;

nextMove[2] = -140;

nextMove[3] = -140;

nextMove[4] = 45;

assertArrayEquals(planner.calculateNextMove( state.getCurrentPose(), state.getCurrentTarget()), nextMove);

}

}

---------------------------------------------------------------------------------------------------------------

This class file imports the StateAssessment.java and Planner.java class files, sets up the two necessary environments and then calls different methods in the classes. Once it gets the values back it sets up assertions to test the returned values. If the assertions fail, the test is failed and displays a banner in red. If all of the assertions come back true, a pretty green banner is displayed. If you prepare these tests well, you can rerun them at any time and verify that these modules are working properly.

This is a 5 Minute intro to Junit: <https://www.youtube.com/watch?v=LPPji9JESgQ>

## Getting the Motors Running

As I was checking out the different ways of setting up the motors, I visited Michael Vorburger’s Github site. He had an interesting concept there and I borrowed it. He set up a provider for the driver and then attached a GPIOMotor class to drive the motor.

<https://raw.githubusercontent.com/vorburger/dual-mc33926-motor-driver-pi4j/master/src/main/java/ch/vorburger/raspberry/mc33926/TwoMotorsProvider.java>

<https://github.com/vorburger/dual-mc33926-motor-driver-pi4j/blob/master/src/main/java/ch/vorburger/raspberry/mc33926/GpioMotor.java>   
  
I modified it a bit and put in the code I needed to talk to the SparkFun controller. Notice that the controller that Michael uses has some different features, but does basically the same thing. To design our code, we have to watch for the following items:

Channels – there are two, each with two input pins and two output pins. There is one pin, named STBY that has to be high for the motors to run. This suggest that we need one method, possibly called enable(), that raises the pin to high. My enable looks like this:  
  
 **public** **void** enable() {

enabled = **true**;

enableGpioPin.setState(PinState.***HIGH***);

}

You can see here that I had previously created on object called enableGpioPin that is the pin connected to the STBY port of the MOTOR DRIVER IC.

Similarly, there are the two input pins. We have to raise them high and low, per the documentation, to get the wheels moving in the right direction.

Once we have the STBY high, the AI1 and AI2, BI1 and BI2 in the correct state, we have to send the proper PWM signal to PWMA and PWMB.

Where Michael Vorbuger had one pin to set for direction and one to set for speed, you have two. The code that Michael used is this:  
  
 directionGpioPin.setState(direction);

pwmGpioPin.setPwm(speed);

Instead of a directionGpioPin, you are going to have to create and control the two input pins.

---------------------------------------------------------------------------------------------------------------

You can probably work with your mobile robot at home by just connecting it to a wired port on your router. You would have to log into the router to get the IP Address, or you could just open up a command prompt and execute the command “arp -a” – this will bring you a list of all of the other network cards your computer knows about at the moment. If you find a network card with a MAC Address starting with b8:27, you are looking at a Raspberry Pi.

|  |  |  |
| --- | --- | --- |
| Computer Name | MAC Address | IP Address |
| socPI\_12 | b8:27:eb:08:f8:f8 | 10.49.49.168 |
| socPi\_02 | b8:27:eb:fc:00:31 | 10.49.49.172 |
| socPi\_02a | b8:27:eb:a9:55:64 | 10.49.49.173 |
| socPi\_10 | b8:27:eb:0d:44:57 | 10.49.49.174 |
| socPi\_10a | b8:27:eb:58:11:02 | 10.49.49.175 |
| socPi\_08 | b8:27:eb:87:f4:ba | 10.49.49.176 |
| socPi\_08a | b8:27:eb:d2:a1:ef | 10.49.49.180 |
| socPi\_07 | b8:27:eb:1f:8a:b0 | 10.49.49.178 |
| socPi\_07a | b8:27:eb:4a:df:e5 | 10.49.49.179 |
| socPi\_03 | b8:27:eb:9e:65:48 | 10.49.49.182 |
| socPi\_03a | b8:27:eb:cb:30:1d | 10.49.49.183 |
| socPI\_12a | b8:27:eb:5d:ad:ad | 10.49.49.184 |
| socPi\_01 | b8:27:eb:b1:18:c7 | 10.49.49.185 |
| socPi\_01a | b8:27:eb:e4:4d:92 | 10.49.49.186 |
| socPi\_05 | b8:27:eb:e2:50:a2 | 10.49.49.196 |