Summary of Week Six

We should be in Week Seven now, but we have not yet picked a time to make up our lost session. If we don’t find a spot for a makeup, next week will be our next 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. Last week I said that if you get to the point of executing your code on the Raspberry Pi, you will run into two issues, it turns out that there are three.

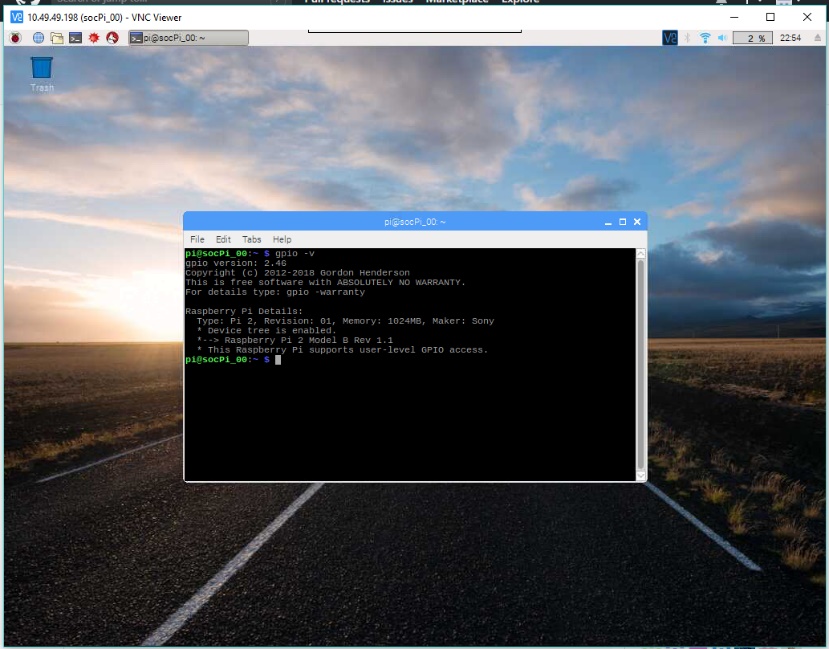
Your code is going to be run by either opening up a terminal shell with the Remote System Explorer or by connecting using RealVNC, Putty, or that MobaXterm program you guys found . In the command shell you have to execute the program

“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. If you end up with the new error, telling you that the program could not tell what kind of a Raspberry Pi you have, that means that I did not upgrade your operating system correctly.

expecting BCM2708 or BCM2709.  
If this is a genuine Raspberry Pi then please report this  
to [projects@drogon.net](mailto:projects@drogon.net). If this is not a Raspberry Pi then you  
are on your own as wiringPi is designed to support the  
Raspberry Pi ONLY.

The solution to this problem is to make sure you are using the Pi4J 1.2 library, which includes the new Wiring Pi, and that Wiring Pi on the Raspberry Pi is version 2.46. You determine the version on the Pi by opening up a command shell on the Pi and executing the command “gpio -v”.

This is the desktop on my Raspberry Pi using RealVNC. If you see 2.4.6 then I have been remiss in making sure that you have the proper Pi4J libraries.

You can also check the Pi4J version by typing the command “pi4j –version” in the Pi’s command shell. If you see 1.1 there you have the old version with the static linking that will not work for us. It will have to be upgraded on the Raspberry Pi. Use the instructions on the site <http://pi4j.com/install.html> to update to the current version. If you have problems with that, I am almost always online of an evening. You can reach me by phone at (240) 470-9885 for help.  
  
THE LISTENER

Your listener consisted of two parts. The first is the class Listener.java, or whatever you called it. When its constructor runs, it finds the current instance of the GpioFactory, sets up a definition for the pin, and adds or binds your instance of a listener program to the state of either GPIO Pin 7 or Pin 8.

You created a variable to store the current count. Integer is probably big enough, though I made it a long integer. Your interrupt code, the thing that called eight times every time the motor goes around once, exists only to update the count of ticks.

You put a line like this in the Main method of your program.

SparkFunListeners listener = **new** SparkFunListeners();

You used one or two calls to the public methods your wrote in Listener.jave to get the current count, which you display in your main loop using “System.out. printf(“Left %d Right %d”, left, right);” You found that you could either get the value for left and right and then print them or simply call them in the print statement.

In your listener code you first connected to the GpioFactory

**final** GpioController gpio = GpioFactory.*getInstance*();

You defined your pins

Pin pinOutA = RaspiPin.***GPIO\_07***;

Pin pinOutB = RaspiPin.***GPIO\_08***;

PinPullResistance pullOutA = PinPullResistance.***PULL\_UP***;

PinPullResistance pullOutB = PinPullResistance.***PULL\_UP***;

And then you had to get your code attached to the ups and downs of the pin  
  
 outA.addListener(**new** GpioPinListenerDigital() {

@Override

**public** **void** handleGpioPinDigitalStateChangeEvent(GpioPinDigitalStateChangeEvent event) {

countA += 1;

}

});

All you had to do after that is set up some public methods for other programs to retrieve your values and used them.

## MOTOR DRIVERS

Now you have to do the same thing with your motor drivers. Looking at the datasheet, you have to deal with two motor drivers, plus at least one parameter that drives the entire card. You disable the MOTOR DRIVER by taking pin STBY low. You enable the pin by bringing STBY to high.

You can tinker with the PWM settings. There is enough information at <https://raspberrypi.stackexchange.com/questions/4906/control-hardware-pwm-frequency> to get you into trouble – well worth digging into to get a way to work this problem right down near the wire. You get to change the PWM frequency by setting the range, the mode, and the clock.

Gpio.*pwmSetMode*(***PWM\_MODE\_MS***);

Gpio.*pwmSetRange*(***MAX\_SPEED***);

Gpio.*pwmSetClock*(2);

The frequency is Hz is 19.2e6 / pwmClock / pwmRange – so, in the sample we started from, we get 19.2e6/2/480, which equals .02 MHz, or 20 KHz. The boards we are using can go up to 100 KHz.

## CODE OUTLINE

So far, we have defined about four out of some 16 classes I have alluded to.

RobotMain. Java – contains our main code, initializes all of the subsystems, and runs a loop that takes our robot to its goal.

StateAssessment.java – figures out the state of the vehicle. It keeps track of the robot’s pose, defined as the (x,y) coordinates where the robot is sitting on our field, and Theta, the angle between the Robot’s central axis and the field’s x axis, in radians.

Planner.java – contains a map of the playing field and routines to plot paths through the obstacles to the goal.

Listener.java – establishes the monitoring of the input pins for the Encoders. Stores the current encoder counts. Reports the counts back to the calling class.

We are going to have to add a couple of classes to the list to allow us to drive.

Drive.java – initializes the MOTOR DRIVER, sets up the output pins to enable the device, defines a Motor object and a reverse motor object for the two motors, sets up methods to start and stop the motors.

Motor.java s Sets up methods to control the speed of the motors. The easiest way to implement this is to set up Motor.java and a ReverseMotor. Java that inverts the control for the motor on the other side. Handle the hardware by setting up a class that extends the Motor class and actually implements these methods.

GpioMotor.java – extends Motor.java and sets up the two control pins so we can set modes CW, CCW, Short Brake, and Stop. Sets up PWM parameters. Sets up methods to control the speed of the motors. The easiest way to implement this is to set up Motor.java and a ReverseMotor. Java that inverts the control for the motor on the other side.

Logging.java – sets up a log file for entries that we add to the classes to collect data. It overrides the native log features.

If we find time, we have the remaining classes that we can build.

These are classes to run Network Tables, talk to the Arduino, read the MPU 9250, and so on.