Summary of Week Three

We did not get quite as far as we would like for week three, but we are still doing fairly well.

We took what we learned last week and constructed the shell of our control program. We are still working with our playing field of 4 meters by 4 meters. We are going to perform our computation in centimeters, and we are going to have our field divided by two axes, x going from -200 centimeters on the left to +200 centimeters on the right, and y going from +200 centimeters on the side away from us to -200 centimeters on the side right next to us. The vehicle will be placed -150 centimeters on the x axis and -150 centimeters on the y axis. The target will be 150 centimeters on the x axis and 150 centimeters on the y axis.

Looking at what we talked about last week, we created a couple of classes. We spent time playing with some of the eclipse functions that allow us to identify a method in the place it is called and have it written into the class file where it should be designed. We did the same thing with the package definitions, watching how Eclipse notified us of the discrepancies and offered us ways of keeping the code consistent. Be careful with this, Eclipse doesn’t know what you intend doing – it just offers you ways to reduce the number of errors. You probably don’t want to correct a spelling error by creating a new class or variable.

The first class is RobotMain.java, which I placed in package org.phs.code.robot. The first part of the class initializes a StateAssessment.java class and calls on the a method called init(). We use init() to set up two private variables “private double pose[] = {-150, -150, 0, 0}” and “private double target[] = {150, 150, 0, 0}”. Pose is the position and orientation of the vehicle on the field. Pose[0] is the x ordinate, pose[1] is the y ordinate, pose[2] is the z ordinate, and pose[3] is the angle of the vehicle with respect to the field – when the vehicle is aligned with the y axis, driving from negative numbers to positive numbers, it is at 0 degrees with respect to the field.

With these set up, we can create a while loop to contain and control our code.

// We initialize the variables

StateAssessment state = new StateAssessment(); 🡨 this calls the constructor for the class

State.init(); 🡨 this method sets up the arrays pose[] and target[]

Double x = -150; 🡨 test setup, the intrusive way. We will learn better ways to do this

Double y = -150;  
  
while (!state.atTarget()) { 🡨 this method checks to see if we should stop

// We are going to do some primitive testing here

X += 1;

Y += 1;

State.setPose(x,y,0); 🡨 I use x, y, and, z, but z is always 0 and we aren’t turning

}

We are going to tweak this a bit so that we collect our data at decent intervals, but the shell works like this. We start the vehicle at its designated pose and then use the setPose() method to move it 1 centimeter up and right each time through the loop. Since we put the initial pose and the target on the diagonal, we are going to reach the target in 300 passes through the while loop. The purpose of the method atTarget() is to determine when the loop should stop. We decided that we would consider ourselves close enough when the center of the axle got to within 15 centimeters of the target.

To achieve that goal, we have to figure out how to compute that condition. Since we are on a basic grid, we can calculate the shortest path to the target as the hypotenuse of a right triangle with one leg passing through the vehicle parallel to the y axis and the other passing through the target and parallel to the x axis. What we actually wrote was:

Public Boolean atTarget() {

Double x = target[0] – pose[0];

Double y = target[1] – pose[1];

Distance = Math.sqrt(x \* x + y \* y);

Return (distance < 15);

}

All we did is see if the hypotenuse, which can be treated as the radius of a circle the distance of the center of the axle from the target, is less than 15 centimeters.

Our next object is to replace our gimmick of incrementing x and y values to move closer to the target with a routine to tell us where we are. We need to know our last pose and the changes that took place between that point and time and this one. The gearmotor turns at a rate, without a load, of 90 RPM, or 1.65 revolutions every second. During this time, the wheels would travel 33.5 centimeters – its diameter of 6.5 centimeters \* pi() or its radius of 3.25 centimeters \* tau() multiplied by 1.65.   
  
The wheel base from center to center is about 15 centimeters, so a complete trip for one wheel around the center of the vehicle is 15 \* pi() or about 47 centimeters.  
  
If one wheel was turning backward full speed and the other turning forward, the vehicle would spin around its z axis (this is yaw) a full 360 degrees in about .8 seconds.

With that rotation speed on the wheel side of the gearbox, the motor is going 90 \* 48 RPM, or 4,320 RPM, and the encoder is ticking along at 8 times that, or 34,560 ticks per minute. That is going to generate about 576 interrupts every second.

If we sample for course corrections 20 times a second, about every 50 milliseconds, which, coincidentally, is the rate that the cycle runs in the FRC code, we would be integrating a max of about 10 ticks per encoder, and 23 degrees of rotation, during each interval.

Next Wednesday, we are going to put the ,motors and encoders on our chassis and connect to GPIO 4 and 7 on the Raspberry pi. In the code, we are going to set up a GPIO Factory. Since we are going to be executing code and working with the pi4j libraries, we need to make some adjustments to our build.xml files. If we had an actual main.java at the root of our code, the java compiler would find it and everything would be OK. Since we are following what the FRC does, we have to tell the compiler where it lies in the code. We also have to tell the compiler where the pi4j libraries are to be found. In the build.xml, the complier looks at the libraries to figure out where the different methods are and to compile the references properly into the code. This is referred to as the ClassPath. The entire set of libraries we intend to use are in the target definition. These are java archives that contain the methods you would find on a Raspberry Pi, even though the code is generated on a windows system. The compiled output has to find those same files on the Raspberry Pi, or you will get errors that certain things can’t be found or loaded.

<target name="compile" depends="init">

<javac includeantruntime="false" srcdir="${src}" destdir="${build}" >

<classpath>

<pathelement path="C:\Users\steve\ntcore-java\ntcore-java-4.0.0.jar"/>

<pathelement path="C:\Users\steve\ntcore-java\ntcore-jni-4.0.0-all.jar"/>

<pathelement path="C:\Users\steve\ntcore-java\wpiutil-java-3.0.0.jar"/>

<pathelement path="C:\Users\steve\pi4j-1.2\lib\pi4j-core.jar"/>

<pathelement path="C:\Users\steve\pi4j-1.2\lib\pi4j-device.jar"/>

<pathelement path="C:\Users\steve\pi4j-1.2\lib\pi4j-gpio-extension.jar"/>

<pathelement path="c:\C:\Users\steve\slf4j-1.7.25\slf4j-api-1.7.25.jar"/>

</classpath>

</javac>

</target>

The file that you are generating to pass to the Raspberry Pi is PHSUserProgram.jar. This file is built using the “dist” target. The Class-Path here has the libraries listed using the Linux file system notation. The Main-Class entry points to the class file that contains the Main() statement. Those of you who compiled your work so far and executed it on the Raspberry Pi found that java could not find Main() – this is why.  
  
 <target name="dist" depends="compile"

description="generate the distribution">

<!-- Create the distribution directory -->

<mkdir dir="${dist}/lib"/>

<!-- 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" />

</manifest>

</jar>

</target>

All that being said, we are going to build up our code to listen for the ticks on the encoders using the GPIO listener code. We will test it using the switch and then wire in the encoders, which should have the same results. If you don’t understand this the first time through, don’t worry about it. We will go over it several more times by the end of the summer.  
  
http://pi4j.com/example/listener.html