



## Chapter #8: Tracking Distance Traveled

### Tracking Distance Traveled

Earlier we mentioned that the J-Bot wheel servo control operated without feedback. The Javelin can set the PWM object to generate pulses to move the wheel but sending the same pulse width to different servos will result in slightly different rotation speeds. This shows up in forward movements where the J-Bot drifts to one side or the other. Normally the J-Bot will continue drifting to one side.

If your J-Bot runs straight and true then you are lucky. Try running the J-Bot for fifteen or twenty feet to be sure. More than likely the J-Bot will drift at least an inch to one side or the other.

So what is a programmer to do? Resort to a closed feedback loop.

Actually, you have already encountered a closed feedback loop. The use of various sensors provides a way for the J-Bot to get feedback about its surroundings so the program can adjust the movements of the J-Bot. This is the same thing that will be done in this chapter except that the feedback is related to the wheel rotation.

### Introduction to Encoders

You may have wondered why the J-Bot's wheels are white and have slots and spokes. It is not because they look nice. The color and configuration allow the wheels to work with the QRB1113 infrared reflective object sensors. The sensors consist of an infrared LED and detector similar to the ones used in the prior chapter except that these sensors do not need a modulated signal for the LED. One reason this can be done is the way sensor is constructed.

The reflective sensors are designed to operate very close to the object being detected, in this case the wheel. The distance between the sensor and the wheel will be about  $\frac{1}{4}$  inch. If you look closely at the sensor you will see that the LED and detector are angled towards each other. If you draw a triangle near the tip with edges parallel to the edges of the sensor then endpoint will be about  $\frac{1}{4}$  inch away. Don't know which is the LED and which is the sensor? We'll take a look at these details in the next section.

The term encoder is used because the reflective sensors are used to encode information about the wheel movement into a binary form that the Javelin can use. Some wheel encoders can determine the absolute wheel position but this is not required for our application. We simply need to know the relative position so the

wheel rotation speed can be adjusted to keep both wheels moving at the desired rate.

The sensors will keep track of the wheel rotation by tracking the open slots in the wheels. The infrared LED light will be reflected back to the detector when the spoke is in front of the sensor. The light will not be reflected when a slot is in front of the sensor.

A quadrature wheel encoder uses two sensors per wheel. This architecture can determine direction in addition to position and rotation. This is not necessary with the J-Bot because it controls the direction of the wheel.

### Activity #1: Building and Testing the Encoders

The infrared reflective sensors are mounted on the J-Bot's frame underneath the JIDE as shown in Figure 8.1. To install the sensors it is necessary to remove the board. It may also be necessary to remove wheel servos as well. In this case, the position of the sensors should be noted using a permanent marker before removing the wheels and servos.

The sensors are delivered with four long leads that extend past the end of the plastic housing. The leads should be cut so they are about  $\frac{1}{2}$  inch long. The leads will then be in line with the end of the plastic housing. The leads should now be the proper length allowing the supplied cabling to be connected so most or all of the lead is covered.

The sensors are mounted so the label information is face up. This information will be needed when connecting the wires to the sensors. The sensors are mounted using a single screw, a nut and lock washer. The sensor should be placed close to the wheel but not touching it. Note that the wheels have a ridge on the rim. Try to place the edge of the sensor in line with the edge of the rim.

Figure 8.1: Remove the circuit board from the J-Bot. It is possible to assemble the infrared reflective sensors without removing the wheel servos but it is easier to do with the servos removed as well. If the servos will be removed then mark where the sensors will be placed. Attach the sensors to the J-Bot frame using a nut, lock washer and bolt. The S and E labels on the sensor should be face up.

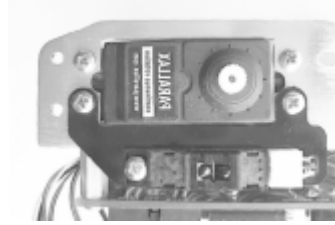


Figure 8.1

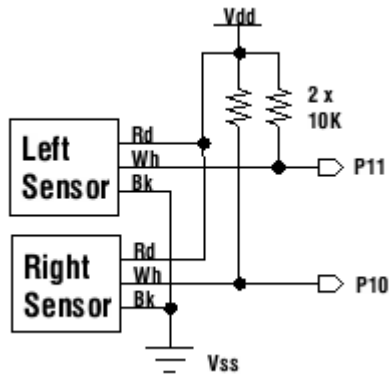


Figure 8.2a

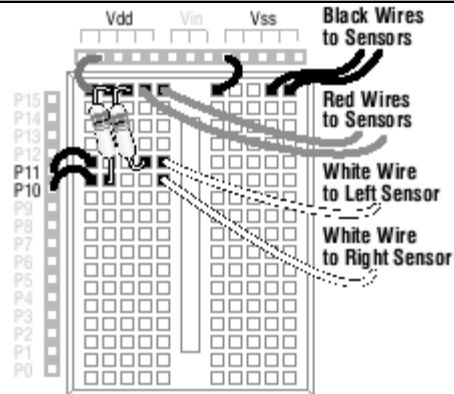


Figure 8.2b

The wheel encoder circuit always keeps the infrared LEDs on. This conserves Javelin output pins. The LED will not interfere with the other sensor because they are pointing in opposite directions. Likewise, they cannot interfere with the IR range finder that operates with a modulated 38.4 kHz infrared signal. This reduces the number of I/O pins needed to two. One input pin for each sensor, CPU.pin10 for the right wheel and CPU.pin11 for the left.

**FYI**

It is possible to install the infrared reflective sensors when initially building the J-Bot. They can be left disconnected until the wheel encoder support needs to be activated. It is relatively easy to install the wires on the sensors with the JIDE removed.

Build It!

- ❑ Construct the circuit shown in Figure 8.2a.
- ❑ Connect the sensors to the wheel encoder circuit on the JIDE.
- ❑ Replace the JIDE board.

Programming the Wheel Encoder

Testing the wheel encoders is relatively easy. Turn the wheels and check the inputs. The inputs should change as the path of the infrared light changes from being reflected by a spoke to not being reflected by an open slot. We use the FixedMovementJBot object to drive the wheels. In theory, this class was modified so the J-Bot should move forward in a straight line. We will take advantage of this in the next test.

8

- ❑ Enter WheelEncoderTest1 program.
- ❑ Lift the J-Bot so the wheels are not touching the ground.
- ❑ Run WheelEncoderTest1 program.
- ❑ This program makes use of the Message window, so leave the serial cable connected to the JIDE while program is running.

### How the Wheel Encoder Program Works

The jbot object will start the wheels running when the forward method is called. They run continuously until the Javelin is reset using the Reset button in the debugger or the power is removed from the J-Bot.

As the wheels turn, the slots and spokes move past the reflective sensors. The sensor outputs will be low, a logical 0, if light is detected as when the spoke is in front of the sensor. The value will be high, a logical 1, if no light is detected as when the slot is in front of the sensor.

The sensors and the Javelin operate faster than the wheels move. This means the numbers printed on the Message window will not toggle between 1 and 0 on every line but every few lines. Also, the left and right wheels are independent so the transitions will probably not be in synch.

### Your Turn

- ❑ If the wheels are turning but the output values for one or both sensors remain constant then check the sensor position, wiring, and circuitry. A typical problem is the connections to the sensors.
- ❑ See if the transitions between 0 and 1 are occurring at regular intervals. The difference should be one or two iterations at most.

### Seeing If Wheels Are Going Straight

The last program shows that the sensors and circuitry are working properly. A few minor changes to the program allows the Javelin to count the number of transitions. If the values remain in synch then the J-Bot will be moving forward. If one side is going faster then its count will be higher and the J-Bot will drift to the opposite side assuming the J-Bot is placed on the ground.

The following program counts the number transitions that are detected on both wheels.

---

The state of each pin is recorded and the counter associated with the pin is incremented when a change is detected.

### Your Turn

- ❑ The output may be scrolling too quickly to recognize the numbers. The output on the Message window can be stopped but it can be reduced in the program as well. Modify the program so it prints the counters every 10<sup>th</sup> count on the right wheel.
- ❑ Check and see if the wheels are moving consistently. Reset the counters when the left counter hits 100. Is the right counter value the same each time? Is it 100? If not, which way will the J-Bot drift?

Activity #2: Going Straight

The prior activity shows how the wheel encoder hardware is used to track the movement of the J-Bot. It is now possible to numerically determine how straight the J-Bot will go using the non-feedback settings. In this case, the rightCount and leftCount values can be compared. These are essentially odometer values indicating the distance traveled.

**FYI**

Keep in mind that the odometer values are only as accurate as the wheel rotation is with respect to the surface it is running on. If the wheels slip then the odometer readings will be more than the actual distance traveled. This can occur if the J-Bot is running on a thick carpet where the wheels do not have good traction. This is no different than a moving car slipping on a gravel road. Still, overall, the car's odometer is a fairly good approximation of the distance the car is driven.

In this activity the information from the wheel encoder hardware is used to keep the J-Bot moving in a straight line. This is done by comparing the distance traveled by each wheel based on the wheel encoder result. The speed of the wheels is adjusted if one is covering more distance than the other.

Compare this to the way a person keeps a car going straight. In this case, the wheels can be turned which has the effect of making the car turn as it moves forward. The J-Bot's wheels do not turn but as we have seen in prior chapters the same effect occurs by using different speeds for each wheel. Therefore, changing the speed of the J-Bot's wheels in response to changes in the distance traveled, courtesy of the wheel encoder feedback, results in the J-Bot making a very minute turn.

The end result is a J-Bot that moves forward in a relatively straight line. In actuality, it is weaving side to side but the turns are difficult to see. If the J-Bot winds up going perfectly straight then the adjustments will not have to be made and the J-Bot will not weave. In practice, the J-Bot will always deviate but the amount of deviation will vary based on the initial estimates made in driving the servos. The advantage of the feedback system is the J-Bot will go relatively straight even if the initial values are off considerably. Of course, excessive differences, such as using pivot values instead of forward movement values, will be very difficult if not impossible to correct using the program presented here.

We start with a relatively complete implementation instead of making incremental improvements. Instead, you can experiment by modifying the program to disable or adjust various parameters and algorithms to see how they affect the operation of the system.

❑ Enter the WheelEncoderTest3 program shown below.

Before getting into the algorithm, we take a look at the variables used as shown in the following table.

Variables	Description
transition	Indicates that an encoder transition has occurred
leftState, rightState	Last input values from wheel encoder hardware
leftCount, rightCount	Wheel encoder counters
leftRatio, rightRatio	Number of steps that should be taken with respect to the other side
leftSpeed, rightSpeed	Speed percentage (-100% to 100%)
leftAdjust, rightAdjust	Speed adjustments
LeftStep, rightStep	Increment for speed adjustments
leftLimit, rightLimit	Maximum speed adjustments
leftOdometer, rightOdometer	Odometer counters

The odometer counters keep track of the distance actually traveled by each while the leftCount and rightCount variables keep track of the relative difference in the distance traveled. The actually values are controlled by the leftRatio and rightRatio and the wheel encoder hardware input. The values in the leftCount and rightCount variables are relative odometer readings. Their use will become more apparent in the algorithm description.

The leftSpeed and rightSpeed values are the desired speed in percentages used in prior J-Bot wheel control classes. The leftAdjust and rightAdjust variables keep track of any percentage changes that should be made to the full speed settings. These values are changed when either the left or right wheel is falling behind. The values in these variables are changed in increments of leftStep and rightStep respectively. The maximum values for the adjustment variables are in the leftLimit and rightLimit variables.

The algorithm is implemented in the runTest method. The method keeps track of the number of transitions and exits when a limit, 200 in the listing, is exceeded. This will allow us to examine the output of the test that otherwise would continue forever.



This number of transitions is usually sufficient to show how the algorithm works. In a more practical implementation, movement would be stopped or changed when an external event occurred, such as the detection of an obstacle, or after a certain amount of time has elapsed or a specified distance covered as noted by the odometer values.

The main loop starts with a check of the right and left wheel encoder output pins as shown below.

```
if ( rightState != CPU.readPin(rightPin)) {  
    transition = true ;  
    rightState = ! rightState ;  
    -- rightCount ;  
    ++ rightOdometer;  
}
```

8

The condition compares the current wheel encoder output with the prior state. This allows the J-Bot to detect both edges of a spoke or hole in the wheel. This doubles the number of transitions detected compared to just checking for a spoke or hole. The transition variable is set so that both wheel encoder outputs can be checked. The loop continues if a transition is not detected on either input.

The state variable is toggled each time using `!`, the logical NOT operator. The `rightCount` value is decremented while the odometer variable is incremented.

The method behind this madness is that the `rightCount` and `leftCount` variables track the relative difference between number of transitions that are detected on each wheel. We could use the odometer variables but there is a drawback. First, the odometer values have an upper limit. It is large but it can affect calculations that would be necessary to compare the values. Second, the calculations are more complex and this takes time. The operations performed on the counter variables are simple comparisons, assignments and decrement operations. It is also easier to see the algorithm works.

If you look at the code before the main loop you will see that the `rightCount` and `leftCount` are set to the `rightRatio` and `leftRatio` respectively. These are positive values so eventually one of the two counters will be decremented to zero.

Things happen when one or both of the counters hit zero. Although it happens at the end of the code block, the one thing you can count on is the counters will be reset. In essence, both counters have their respective ratio values added to them but because the programmer already knows that one of the values will be zero a more efficient assignment statement can be used instead.

The reason the updates occur at the end is because the values need to be compared against the ratio values as shown in the next code snippet.

```
// Left side may be behind right
if ( leftCount >= leftRatio ) {
    // Adjust: left wheel is behind
    if ( leftAdjust == 0 ) {
        // No left adjustment. Slow right wheel
        if ( rightAdjust < rightLimit ) {
            rightAdjust += rightStep ;
        }
    } else {
        // Left wheel has adjustment. Reduce it
        leftAdjust -= leftStep ;
    }
    // Set new speed
}

setSpeed ( leftSpeed - leftAdjust, rightSpeed - rightAdjust ) ;
// Reset counters
leftCount += leftRatio ;
rightCount = rightRatio ;
```

If the counter value is less than the ratio value then nothing changes. This is akin to both counters reaching zero at the same time. It means that the wheels are exactly (when both values are zero) or close in synch or almost in synch within the limits of the ratios. A closer look shows that a ratio of 1:1 is essentially the same as 2:2 except in this algorithm the 2:2 will be less sensitive to small variations whereas a 1:1 setting, used in the example, will react more quickly to changes that occur.

If the counter value greater or equal to its respective ratio then it is time to make an adjustment to one of the wheels because one is going to faster than the other. One of two changes can be made. Either the slower wheel can be run faster or the faster wheel can be slowed down. The code checks for both conditions and determines which to do based on the current adjustment variable (leftAdjust or rightAdjust) values. This is done because we assume the maximum speed of a servo is 100%.

In the code listing shown above, the leftAdjust value is tested when the left wheel is going slower. If the leftAdjust value is not zero then it has been slowed prior to this point. Decreasing its value by subtracting the step value from it will result in the servo speeding upon when the setSpeed method is called. Otherwise, the right wheel must be slowed down by increasing its adjustment value. This is where a major optimization is included.

Note that the rightAdjust value is only changed if it is less than rightLimit. This prevents servo from going slower than the specified limit, otherwise the speed could be reduced to zero or even go negative in which case the wheel would be going backward!

Slowing the servo too much will cause the other wheel to catch up but it turns the J-Bot too quickly. This over steering will very quickly cause the J-Bot to require compensation in the opposite direction. The result is a J-Bot that weaves drastically from side to side.

The prevention of these drastic actions is called *damping*. Limiting how slow a wheel will be adjusted provides the damping.

- ❑ Raise the J-Bot so the wheels do not touch the floor and run the program with the serial cable attached. The Message window should show the state of various variables including the speed of both wheels. Notice how the wheel speeds are changed in response to differences in transition detected.
- ❑ Remove the PC cable and run the J-Bot using batteries. Place the J-Bot on the floor and see if it runs straight. Remember, it may waver side-to-side slightly.

#### Your Turn

- ❑ The program is designed to run the J-Bot in a straight line in a forward direction. Change the program so it will run the J-Bot in a backward direction instead. Hint: Change the percentage variables but not the transition counter variables. Remember, the spoke and hole transitions are detected in the same fashion regardless of the direction the wheel is turning.
- ❑ Make the same kind of change except allow J-Bot to pivot right or left. Remember that pivoting is done by running the wheels in opposite directions.
- ❑ The test program counts the number of transitions and terminates after a fixed value. Change the termination condition so it is based on the number of transitions detected on a particular wheel. This essentially controls the distance traveled. How does this approach differ from the initial test program?

### Activity #3: Wheel Encoder Class

Activity #2 presented a mechanism for running the J-Bot in a straight line. The additional experiments allowed for backward and pivot movements. These can be combined into a class that can be used to control the J-Bot in a single tasking environment. It is a more complex task to create a control system that will operate with the multitasking system but that is what will be done in this activity.

As you might expect, this wheel control class will be significantly more complex than the other classes defined in prior chapters. It actually requires two classes because a task is needed to monitor the wheel encoder hardware. The task will be hidden behind the wheel encoder class that inherits its interface from JBotInterface.

The two class architecture is similar to the ones used with the multitasking sensor systems used to support the photoresistor and infrared range finder hardware. The application interfaces with the main object and a second object, usually based on a Task class, operates in the background.

The main class for wheel encoder system is the WheelEncoderJBot class. The other class is the WheelEncoderTask. One object from both classes will be created and these two objects will interact to control the J-Bot servos. An application will interface with the WheelEncoderJBot object. The WheelEncoderTask only needs to run when the wheels are moving. The WheelEncoderJBot class will also provide odometer methods that could not be done using prior JBotInterface-based classes.

A slightly different interface is provided to control the WheelEncoderJBot class. This interface is based on the Event class. The WheelEncoderJBot class will call the event's notify method when a movement has been completed. This allows the event to immediately initiate another movement if necessary. This provides a mechanism for continuous servo control without requiring another task that will poll the status of the WheelEncoderJBot. Polling is still possible but less efficient.

The starting point is the WheelEncoderJBot class shown in the following listing.

---

The WheelEncoderJBot class starts with a number of constant definitions. These control the movement of the servos based on the wheel configuration. Use a wheel that is a different size or has a different number of holes and spokes and these numbers may

have to change. Keeping the constant definitions together make it easier to locate them when changes are necessary.

The `WheelEncoderJBot` constructor is relatively simple. It lets the superclass store the `startEvent`. This is typically a `FixedMovementJBot` or `MultitaskingJBot` event object.

The basic movement control methods, `movementDone` and `stop`, are available to an application along with a host of movement methods including `move`, `pivot` and so on. These are essentially identical. They setup the wheel encoder support and then call the matching superclass support. Note that the parameter passed to the superclass is increased. This is so the movement will continue passed the expected stop point allowing the wheel encoder support to mark the end of the movement. This also prevents the J-Bot from running forever should the wheel encoder hardware work improperly or if the wheels are slipping for some reason.

The new methods not required by the `JbotInterface` class include the odometer methods. The odometer operation is relatively simple. The odometer can be reset to 0 and the values can be obtained. The left and right values are available independently.

The bulk of the work is done in the `movementDone` method. This method will be called periodically by the `startEvent`. It will check the wheel encoder hardware and keep track of the transitions detected adjusting the speed as necessary. The `setRealSpeed` method is used so any changes made by the ramping support via the `setSpeed` method will not be affected.

To test the `WheelEncoderJBot` and `WheelEncoderTask` classes using the following program.

---

The `WheelEncoderTest4` class may be a bit of a surprise since it is based on the `Event` class and not the `Task` class. This is because only one task is necessary at this point, the background `MultitaskingJBot`.

The main method starts by creating a `WheelEncoderTest4` event object. An object variable, `jbot`, is assigned a reference to a new `WheelEncoderJBot` object that in turns creates a `MultitaskingJBot` object. This is the task that will actually be run by the `Task.TaskManager` method call.

Hopefully this will not get too convoluted so follow along closely. The constructor method creates the appropriate objects as just mentioned. The constructor then calls the `jbot's setEvent` method and passes a reference to the `WheelEncoderTest4` event object. Since the task is not running (remember, its constructor

will stop it) the call to the `setEvent` method will cause a subsequent call to the event's `notify` method. It is important to set the `i` and `state` variables *before* the `setEvent` call because these variables must be initialized *before* the `notify` method is called.

The `notify` method is simple because the path being followed by the J-Bot is a square. All four sides of the square require the same actions to be performed: move forward and pivot. In this case there are two states that handle these actions: `moveForward` and `pivotLeft`. These names do not conflict with the ones used in other classes because they are specific to the `WheelEncoderTest4` class.

The `jbot` methods are called to initiate each movement. The states change to the other state after initiating the movement. Unlike the `task execute` method that is called repeatedly by the `TaskManager` method, the `notify` method will only be called when a movement has completed.

The `moveForward` state in the `notify` method will be called 4 times to traverse the square. Two squares will be traversed by keeping track of the number of times the state is entered and by exiting after 8 iterations.

And that's it. Once the `notify` method is called 8 times it will not call the `jbot.forward` method. The background task will remain stopped so the `Task Manager` method will eventually return so the final "All Done" text can be printed using the `System.out.println` method call in the `main` method.

- ❑ Connect the PC cable and raise the J-Bot so the wheels do not touch the floor. Run the program and watch the wheels to see if they rotate as anticipated.
- ❑ Remove the PC cable and run the J-Bot using batteries. Place the J-Bot on the floor and see if it runs around in a square, twice. Remember, it may waver side-to-side slightly.

So what needs to be changed if the J-Bot will follow a more complex path? The `WheelEncoderJBot` and `WheelEncoderTask` classes should remain intact. The event class can be changed to make different movement calls.

#### Your Turn

- ❑ The program moves in a simple square. Change the figure to a rectangle so one pair of sides is twice that of the other sides.
- ❑ Run the figure in the reverse direction. This means going backwards and pivoting to the right.

- ❑ The Task class is a subclass of Event so it can be passed to the WheelEncoderJBot's setEvent method. Implement the WheelEncoderTest4 class by extending Task instead of Event. Keep in mind that the default Task action for notify is to start the task. This means the task's execute method should start a movement and then stop. It will be restarted when the movement is done in the state set before the stop method is called.



## Summary and Applications

This chapter makes use of infrared reflective sensors to keep the J-Bot on the straight and narrow. They are used to implement a wheel encoder class that can be used to track how far the J-Bot travels as well as when to adjust movements to keep the

J-Bot moving in the desired direction.

Odometer support is also provided. This is an offshoot of the sensor system used to provide feedback for controlling the J-Bot servos. Odometer readings provide significantly more accurate movement information than is possible using an open feedback loop as in prior chapters.

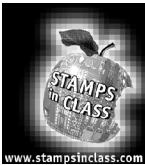
8

Knowing where the J-Bot moves and moving in prescribed directions will be important to solving many problems such as mapping and maze traversal.

### Real World Example

Feedback systems are used almost everywhere computers are used. Wheel encoders, also called shaft encoders, are used in a variety of areas. They are used in robotics to control wheels. The technology can also be used with servo control of robotic arms.

Encoders are used to track movements that may be generated from other sources. For example, an automobile's speedometer and odometer uses wheel encoder technology. The wheels drive a flexible speedometer shaft. The number of turns corresponds to the distance traveled.



## Questions and Projects

### Questions

1. What components are in the infrared reflective sensors?
2. How does the infrared reflective sensors work?
3. What would happen if the white wheel was replaced with a black wheel?
4. How can a black wheel be modified so it would work with the wheel encoder hardware?



### Exercises

1. Activity #2 introduced feedback control of the J-Bot servos. Describe what damping is and how it was utilized in the sample program and what happens when this support is removed.
2. Develop a single tasking class based on the JBotInterface that uses polling instead of the multitasking system used in Activity #3. Examine the CheckForWait and movementDone methods.

### Projects

1. The wheel encoder object can be used to keep the J-Bot going straight or turning as desired but it helps if these operations are started with values that are very close to the optimum values. The J-Bot servo control classes were calibrated manually in prior chapters.
  - ❑ Create a program to calibrate the J-Bot's forward movement using the wheel encoders.
  - ❑ Create a similar program for turns and pivots.
  - ❑ It is possible to create a self calibrating program. Do so if the prior programs require user intervention.
2. The event mechanism was used to traverse simple path using a fixed set of movement calls. Create an event class that can be passed a set of movements in a string or array. This class will be more useful if it also has a method, like setEvent, that takes a reference to an event. This event should be notified when all the movements in the array or string have been performed. Keep in mind that a task is a subclass of Event so a typical implementation has a task that will use this new movement event. The task will sleep after initiating a sequence of movements.