

Chapter #2: Writing Programs to Control the J-Bot's Servos

Chapter #2: Writing Programs to Control the J-Bot's Servos

The J-Bot's servos are attached to wheels that drive the J-Bot. J-Bot applications can control the wheels to move the J-Bot in any direction or have the J-Bot pivot or turn in any direction. How the servos are controlled dictate how the J-Bot moves.

This chapter addresses how the servos work and how to calibrate them for use in subsequent chapters. Those who have already done this type of work, possibly using PBasic and a BOEBOT, may want to skip to Activity #5 that deals with servo calibration. The other activities provide a step-by-step introduction to servo operation and calibration. An interactive calibration program is used in Activity #5 that allows you to get the J-Bot configured quickly.

The J-Bot comes with a special modified servo that allows the wheel to turn continuously. This is different than normal hobby servos.

Normal (un-modified) hobby servos are very popular for controlling the steering systems in radio-controlled cars, boats, and planes. These servos are designed to control the position of something such as a steering flap on a radio-controlled airplane. Their range of motion is typically 90° or 180°, and they are great for applications where inexpensive, accurate high-torque positioning motion is required. The position of these servos is controlled by an electronic signal called a pulse train, which you'll get some first hand experience with shortly. An un-modified hobby servo has built-in mechanical stoppers to prevent it from turning beyond its 90° or 180° range of motion. It also has internal mechanical linkages for position feedback so that the electronic circuit that controls the DC motor inside the servo knows where to turn to in response to a pulse train.

A Parallax pre-modified servo does not have the position feedback and mechanical stoppers you find in normal hobby servos. You can send the same electronic signals (a pulse train) to a Parallax pre-modified servo as you would normally send to a hobby servo. In a hobby servo, a given pulse

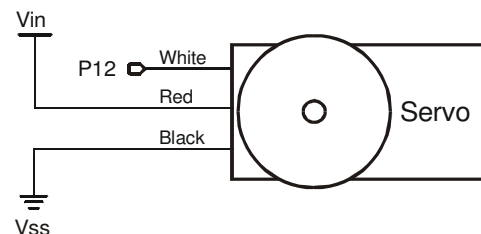


Figure 2.1: Servo connection schematic.

train makes it turn to a certain position and stay there. The same pulse train causes a Parallax pre-modified servo to turn continuously. The pulse train also sets the pre-modified servo's speed and direction. So, instead of controlling airplane flaps, the Parallax pre-modified servos are used as Javelin controlled motors that make the J-Bot's wheels turn.

Figure 2.1 shows the circuit that is established when a servo is plugged into the servo port labeled 12 on the BOE's Rev B's top right corner. The red and black wires connect to the servo's power source, and the white (or sometimes yellow) wire is connected to a signal source. When a servo is plugged into servo port 12, the servo's signal source is Javelin I/O pin P12.



Only use the Vdd sockets above the BOE's breadboard for the Activities in this workbook. Do not use the Vdd on the 20-pin app-mod header.

Activity #1: Connecting and Testing The Servos

The control signal the Javelin sends to the servo's control line is called a "pulse train," and an example of one is shown in figure 2.2. If this looks familiar it is because it is the same pulse width modulation (PWM) support presented in the prior chapter.

The Javelin can be programmed to produce this waveform using any of its I/O pins. In this activity, we'll start with I/O pin P12, which is already connected to servo port 12 by a metal trace built into the Board of Education. First, the Javelin sets the voltage at P12 to 0 V (low) for 20 ms. Then, it sets the voltage at P12 to 5 V (high) for 1.0 ms. Then, it starts over with a low output for another 20 ms, and a high output for another 1.0 ms, and so on.

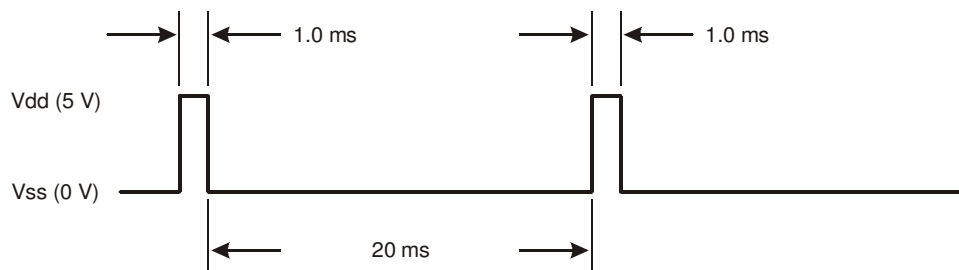


Figure 2.2: Pulse train.

This pulse train has a 1.0 ms high time and a 20 ms low time. This differs from the symmetrical pulse train used for the tone generator in Chapter 1. In that case, the object was to simulate a sine wave that is typically used to generate a tone.

In this case, the high time is the main ingredient for controlling a servo's motion, and it is most commonly referred to as the pulse width. In this example, we are working with 1 ms wide pulses. Since these pulses go from low to high (0 V to 5 V) for a certain amount of time, they are called positive pulses. Negative pulses would involve a resting state that's high with pulses that drop low. Pulse trains have some other technical descriptions such as duty and duty cycle. These are described in BASIC Analog and Digital, Experiment #6.

Remember

Pulse width is what controls the servo's motion. The low time between pulses can range between 10 and 40 ms without adversely affecting the servo's performance.

A pre-modified servo can be pulsed to make its output shaft turn continuously. The pulse widths for pre-modified servos range between 1.0 and 2.0 ms for full speed clockwise and counterclockwise respectively. If you give a pre-modified servo 1.25 ms pulses, it will turn clockwise at roughly half of full speed. If you give a pre-modified servo 1.90 ms pulses, the servo will turn at almost full speed counterclockwise. The "center pulse width" is 1.5 ms, and that makes the servo stay still. If the servo turns very slowly in response to 1.5 ms pulses, you will learn how to adjust the servo to stay still using a JAVA program in Activity #1.

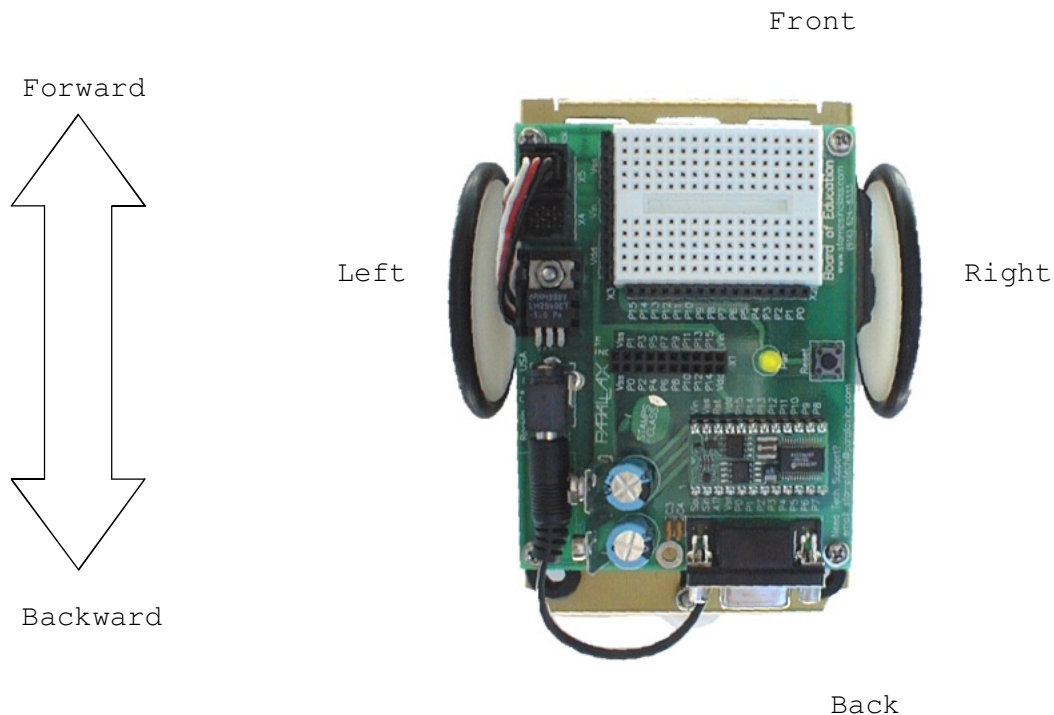


Figure 2.3: J-Bot from the driver's seat.

Figure 2.3 shows the J-Bot's front, back, left and right. Use this diagram as your guide when you see instructions about making the J-Bot move forward/backward, examining the right or left wheels, etc.

Converting Instructions to Motion Using the PWM Virtual Peripheral

Let's start by programming the J-Bot's right wheel to turn full speed ahead. For the right side of the J-Bot, this means the wheel has to turn clockwise, which means it needs to receive 1 ms pulses every 20 ms or so.

- ❑ You may want to set the J-Bot on something to keep it's wheels from touching the ground during these tests. Otherwise, you will see the J-Bot spin around in circles since only one wheel is turning.
- ❑ Enter JbotServo1.java listing into the Javelin IDE.

```
import stamp.core.*;

/**
 * Drives right servo at high speed
 *
 * @version 1.0 5/7/02
 * @author Parallax, Inc.
 */

public class JBotServo1 {
    static PWM pwmR = new PWM(CPU.pin12); // create right servo

    public static void main() {
        pwmR.update ( 110, 2000 ) ;
        pwmR.start () ;

        CPU.delay(10000);    // run for one second

        pwmR.stop () ;
    }
}
```

- ❑ Save the program using as JbotServo1.java. You can do this by clicking the File Menu and selecting Save (or Save As... if you are renaming the file). Then enter the "JbotServo1" into the File name: field, and make sure that the Save as type: field is set to "Java source (*.java)".
- ❑ Make sure the J-Bot has power and the serial cable is connecting the PC to the J-Bot. Run the program by clicking Project and selecting Program.

- ❑ Verify that, as you're looking at the wheel from the J-Bot's right side that it is turning clockwise fairly rapidly (about 37 RPM).

How the Right Wheel Full Speed Ahead Program Works

As with previous program examples, the first line of the program imports class definitions for objects that will be used in the program. In this case it is the PWM class. This is followed by a comment that describes the program. We forego major comments in the rest of the program because it is so short and it will not be reused.

The main static method is required for a Java program. It is contained within the JbotServo1 class definition. The first line in the class definition is a static variable, `pwmR`. That's an abbreviation of pulse width modulation right (wheel). The variable is a PWM reference used to keep track of the new PWM object that is created in the same line. The object is setup to use pin 12 by passing the `CPU.pin12` parameter to the PWM constructor.

FYI

Do not use integer values where a pin ID is required as an argument. For example, use `CPU.pin12` for the PWM constructor as in `new PWM(CPU.pin12)` instead of `new PWM(12)`. The latter will not work because `CPU.pin12` does not convert to a number 12.

It is possible to put the variable definition within the main method but this would restrict its access to that method. This is possible here but not typically done in the rest of the book where other methods would have to access the variable. The variable is static so it can be accessed by the static methods. If this class definition were to be used by other classes then it would be a better idea to use object methods and object variables. This is what will be done later in this book.

The creation of the PWM object simply sets up the virtual peripheral object. It does not start sending a pulse train to the servo. This is done using the update method and the `pwmR` object within the main method. The parameters to the update method are the length of time that the pulse should be high followed by the length of time the pulse should be low. The high voltage is `Vdd`, 5 volts, while the low voltage is `Vss`, 0 volts.

The update method starts the PWM virtual peripheral sending a series of pulses. These pulses will continue until the stop method is called. The pulse train can also be changed with a subsequent update method call.

The `CPU.delay (10000)` method utilizes the `CPU class`' to pause the program for 1 second otherwise the program would immediately call the PWM object's `stop` method and the wheel would not turn much since only one or two pulses may be sent. The PWM virtual peripheral operates in the background so pulses will continue to be sent while the program does other things. In this case, it waits.

The parameters in the PWM update method are measured in $8.68\mu\text{s}$ ticks. The high time is set to 110 ticks and the low time is set to 2000 ticks.

1.0 ms	=	1000 μs	=	$1000/8.68$ ticks	=	115 ticks
1.5 ms	=	1500 μs	=	$1500/8.68$ ticks	=	173 ticks
2.0 ms	=	2000 μs	=	$2000/8.68$ ticks	=	230 ticks
20 ms	=	20000 μs	=	$20000/8.68$ ticks	=	2304 ticks

Your Turn

JbotServo2.java is really JbotServo1.java with one small change. Instead of `pwmR.update(110,2000)`, the JbotServo2 class uses `pwmR.update(240,2000)`. This should make the right wheel turn full speed counterclockwise.

❑ Run JbotServo2 as shown.

```
import stamp.core.*;

/**
 * Drives right servo at high speed
 *
 * @version 1.0 5/7/02
 * @author Parallax, Inc.
 */

public class JBotServo2 {
    static PWM pwmR = new PWM(CPU.pin12); // create right servo

    public static void main() {
        pwmR.update ( 240, 2000 );
        pwmR.start () ;

        CPU.delay(10000);    // run for one second

        pwmR.stop () ;
    }
}
```

❑ To make the right wheel stay still, modify the `update` method so that it reads `pwmR.update(175,2000)`, and run the modified program. Note, the wheel may move very slowly. This is due to minor variations in the servos which is expected.

JbotServo3.java is JbotServo2.java with some minor changes:

- The variable `pwmR` was changed to `pwmL`
- `CPU.pin12` was changed to `CPU.pin13`

❑ JbotServo3.java as shown to make the left wheel turn full speed counterclockwise.

```
import stamp.core.*;

/**
 * Drives left servo at high speed
 *
 * @version 1.0 5/7/02
 * @author Parallax, Inc.
 */

public class JBotServo3 {
    static PWM pwmL = new PWM(CPU.pin13); // create left servo
```

```

public static void main() {
    pwmL.update ( 240, 2000 ) ;
    pwmL.start () ;

    CPU.delay(10000);    // run for one second

    pwmL.stop () ;
}
}

```

- ❑ To make the left wheel turn full speed clockwise, modify the **pwmL.update** method so that it reads **pwmL.update(110,2000)** and run the modified program.
- ❑ Now, change the first argument of the **update** method from **110** to **175**, and the wheel should stay still.

Activity #2: Running both servos together

When you assembled the J-Bot in Chapter 1, you plugged the servo on the right side of the J-Bot into P12 and the servo on the left side of the J-Bot into P13. Figure 2.4 shows a schematic of the circuit you created by doing this. The servo on the J-Bot's right side is connected to I/O line P12 and the servo on the J-Bot's left is connected to P13. Each servo is also connected to Vin (the battery pack's positive terminal) and Vss (the battery pack's negative terminal).

The easy part about making the J-Bot roll forward is that you use two **PWM** objects, one for each servo. The difficult part can be figuring out what the **update** method arguments should be.

Take a look at the right side of the J-Bot. To make this wheel turn forward, the servo has to turn clockwise. This means first argument to **update** is less than center value. Now look at the left side of the J-Bot. To make this wheel turn forward, the servo has to turn counterclockwise. Now instead of the first argument being less than the center value it must be larger.

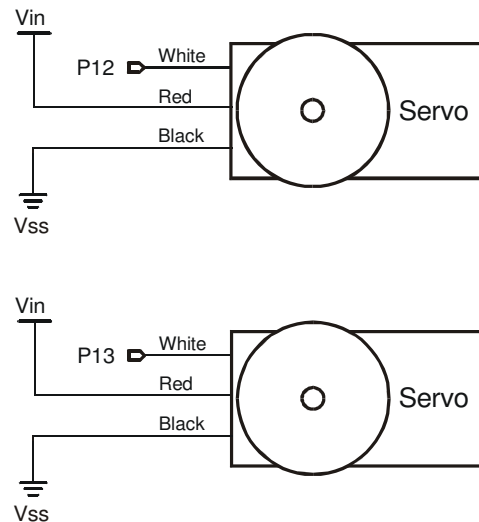


Figure 2.4: Servo connection schematic.

❑ Enter and run `JbotServo4.java` and observe the results.

If the J-Bot rolled backward instead of forward, the servo lines were swapped. It means that the servo plugged into servo port 12 should be plugged into servo port 13 and visa-versa.

```

import stamp.core.*;

/**
 * Drives both servo at high speed
 *
 * @version 1.0 5/7/02
 * @author Parallax, Inc.
 */

public class JBotServo4 {
    static PWM pwmR = new PWM(CPU.pin12); // create right servo
    static PWM pwmL = new PWM(CPU.pin13); // create left servo

    public static void main() {
        pwmR.update ( 110, 255 ) ;
        pwmL.update ( 240, 255 ) ;
        pwmR.start () ;
        pwmL.start () ;

        CPU.delay(10000);    // run for one second

        pwmR.stop () ;
        pwmL.stop () ;
    }
}

```

How JbotServo4.javaPulses Both Servos

JbotServo4.java program pulses the left and right servos at the same time because the PWM objects operate in the background. The right servo is started a fraction of a second before the left but this has little affect on the actual operation of the servos. Likewise, the right servo is stopped just before the left servo.

The program only runs the servos for a few seconds so the J-Bot will not run too far but you can get it to continue on by running it again from the IDE or by pressing the reset button the J-Bot. This approach to timed operation of the J-Bot's servos is how distance movement is controlled. We have not precisely calibrated the movement or the distance the J-Bot moves. This will be done later in this chapter and subsequent chapters.

Your Turn

- ❑ After you make each change listed below, make sure to run the modified version of the program and observe what the J-Bot does differently. Save your changes as new files but remember to change the class name to match the file. A good class names would be JbotServo100, JbotServo101, etc.
- ❑ Swap the first arguments of the **update** methods to make the J-Bot to roll backward. In other words, instead of using the

commands `pwmR.update(240,2000)` and `pwmL.update(110,2000)` use the commands `pwmR.update(110,2000)` and `pwmL.update(240,2000)`. This should make the J-Bot travel backward instead of forward.

- ❑ Try setting both arguments to the center value of 175 to make the J-Bot stay still.
- ❑ Try setting both arguments to 240 and run the modified program. It will make the J-Bot rotate counterclockwise in place.
- ❑ Try setting both arguments to 110 and run the modified program. It will make the J-Bot rotate clockwise in place.

Chances are that you noticed your J-Bot didn't go perfectly straight forward when you ran JBotServo04. For that matter, it probably didn't go perfectly straight backward in response to the modifications you made to JBotServo04. You can adjust the first parameter of the `update` method to straighten out the J-Bot's travel. This practice is called "calibration in software". We take a look finer control and calibration of the servos in the next few activities.

Activity #3: Centering the J-Bot's Servos

The first place to start in software calibration is the center point where no movement occurs. The center point is important because we will be making adjustments to speed based upon this value in the basic servo class in Activity #4.

It is possible to make incremental changes to the program and download them as we did in the prior activity and enhancements but this tends to get tedious especially when a value may change only a little. Instead we make use of the terminal interface between the Javelin and the IDE message window.

To start we need to enter the JbotServoCalibrate1 program shown next.

```
import stamp.core.*;

/**
 * Calibrate the servo center point
 *
 * @version 1.0 5/7/02
 * @author Parallax, Inc.
 */

public class JBotServoCalibrate1 {
    static PWM pwmR = new PWM(CPU.pin12); // create right servo

    public static void main() {
        int i = 175 ;
```

```
pwmR.update ( i, 2000 ) ;
pwmR.start() ;           // start servo

mainloop: while ( true ) {
    System.out.print ( "Servo value is " ) ;
    System.out.println ( i ) ;
    switch ( Terminal.getChar () ) {
        case '+':
            i ++ ;
            break ;

        case '-':
            i -- ;
            break ;

        case 'Q':
        case 'q':
            break mainloop;

        default:
            continue mainloop;
    }

    pwmR.update ( i, 2000 ) ;
}

pwmR.stop () ;
}
```

This program assumes that the center point for the right servo is about 175. If not, the value being sent to the servo can be changed using the keyboard. This done using the Terminal class and its static method, `getChar`. `Terminal.getChar()` returns a character when it is entered. It waits if there are no characters in the input buffer.

The program assumes that one of three characters will be entered: +, - and q. Any others will be ignored. The local method variable, `i`, stores the current high pulse width. The variable is incremented when the + character is typed and decremented then the - character is entered. The mainloop is exited if the 'q' or 'Q' character is entered.

This program shows off some of the power of Java. The while loop is labeled with `mainloop:`. This allows it to be referenced within the select statement that handles the terminal character input. The annotated break statement, `break mainloop;` after the case statements checking for the 'q' or 'Q' allows the loop to be exited. A lone break statement as in the + and - cases only exits the break statement. This allows the PWM object's pulse width to

be changed since the speed variable is changed within the select statement.

Start the program. Once the program starts the Message window will be presented and the text "Servo value is 175" will be displayed in the window. Press the + key and a new line containing "Servo value is 176" will be displayed. Pressing the - key should change the servo pulse width back to 175 with the appropriate change in servo rotation speed.

The servo may be moving slowly with the default value of 175 but it may not. Change the value using the + and - keys until the best result is attained. This may be a very slow movement or no movement at all.

Record the servo center point value. This will be used in the next activity.

Your Turn

□ Modify the program to get the center point of the left servo.

Run the program and determine the center point for the left servo. You should now have the center point value for both the left and right servo.

Activity #4: Basic servo class

The previous examples explicitly manipulated the respective PWM objects but this has a number of disadvantages. First, you need to know what values to set the pulse width to. Second, the values for moving the J-Bot forward are different for the left and right servos. Finally, the center point for each servo may be different.

Here we deviate from the simple program scenario and create a class for controlling servos. This is only a basic version because we will refine this later in the book. Still, this class and the sample application show how the J-Bot can be more easily controlled.

First enter the BasicWheelServo class shown next.

```

package JBot;

import stamp.core.*;

/**
 * Wheel servo control class
 * <p>
 * Handles PWM support for a free running wheel servo on the J-Bot.
 * Start movement using start() and stop it using stop().
 *
 * @version 1.0 7/23/02
 * @author Parallax Inc.
 */

public class BasicWheelServo extends PWM {
    public int forward;
    public int center;
    public int backward;
    public int low;
    public int pin;

    /**
     * Setup a wheel servo control object that uses PWM for movement.
     * Units are in PWM units.
     *
     * @param pin the pin to generate the PWM signal on (i.e. CPU.pin0)
     * @param forward pulse width for moving forward
     * @param center pulse width for staying still
     * @param backward pulse width for moving backward
     * @param low low time following a pulse
     */
    public BasicWheelServo ( int pin, int forward, int center, int
backward, int low ) {
        super ( pin ) ;
        this.forward = forward-center ;
        this.center = center ;
        this.backward = center-backward ;
        this.low = low ;
    }

    /**
     * Set wheel speed to forward/backward (+/-).
     *
     * @param percent relative speed
     */
    public void move ( int percent ) {
        if ( percent == 0 ) {
            stop() ;
        } else {
            update(center+(((percent>0)?forward:backward)*percent)/100
, low ) ;
            start () ;
        }
    }
}

```

The first difference is the between this and most of the other programs presented thus far is the package statement. This indicates the class is part of a larger package so it can be more easily imported. Also, the package file must be in the proper directory. In this case the BasicWheelServo.java file should be in the JBot directory of the Javelin's project directory.

The class extends the PWM class. This allows the PWM methods to be accessed directly. An alternative is to allocate a PWM object when the BasicWheelServo object is created. In this case, the PWM object can be hidden. This may be desirable but more on that later.

The BasicWheelServo class has four object variables. These track the pulse width values used with the servo. They are public so they can be changed but normally the values are set using the class constructor. Why? Because the values will be different for the left and right servos and because the center value will be the ones obtained in the previous activity for your J-Bot. The forward and backward values will be changed based on the next activity as well.

Note that the forward and backward variable values are set with respect to the center value. This is done to simplify the move method, otherwise each call to the move method would have an additional subtract operation to get the value to multiply by the percentage argument in the move method.

The class constructor calls the super constructor method. In this case, this calls the matching constructor in the PWM class. The parameter is the pin to use for the PWM class. The rest of the constructor simply stores the parameters in the respective object variables. The low variable is the time duration for the low part of the pulse. The other values are the high pulse time duration.

FYI

A conditional expression is used to compute the argument to the update method called in the move method definition. A conditional expression is similar to an if statement except that a result is required for both a true and false result of the condition. The general syntax is <condition> ? <true result> : <false result> where values are required for the items in <angle brackets>. For example, (a == b) ? 10 : 11 will have a value of 10 if the variables a and b are equal, otherwise the value will be 11.

The magic shows up in the move method. The move method turns the servo so the wheel movement is forward or backwards depending

upon the parameter. The big difference between this method and the ones used with the PWM is that the PWM values are absolute time values (proportional to 8.68 microseconds). Instead the parameter to the move method is a percentage (-100 to +100) of the maximum speed that should be used. A 0 value results in the center point value being sent to the PWM object. A -100 or +100 value results in the respective maximum rotation backward or forward. Values outside of the range will result in adverse results.

The approach used to calculate the PWM high pulse width works because of the range of values involved. If the typical range of PWM high pulse values were too large then the result of the calculation would be too large and exceed the value of that can be stored in the Javelin's integer variable.

For now, we use the limit values used in prior experiments. Start by entering the BasicWheelServoTest1.java program listed next but, instead of using the center values listed here, use the ones obtained in the previous activity. The actual values used for the forward and backward values will be determined in the next activity.

```
import stamp.core.*;
import JBot.* ;

/**
 * Wheel servo control class test program
 * <p>
 * Handles PWM support for a free running wheel servo on the J-Bot.
 * Start movement using start() and stop it using stop().
 *
 * @version 1.0 7/23/02
 * @author Parallax Inc.
 */

public class BasicWheelServoTest1 {
    public static void main () {
        BasicWheelServo leftWheel =
            new BasicWheelServo (
                CPU.pin12 // pin
                , 110      // forward
                , 175      // center
                , 240      // backward
                , 2000     // low
            ) ;
        BasicWheelServo rightWheel =
            new BasicWheelServo (
                CPU.pin13 // pin
                , 240      // forward
                , 175      // center
                , 110      // backward
                , 2000     // low
            ) ;
    }
}
```



```
leftWheel.move ( 100 ) ;
rightWheel.move ( 100 ) ;

CPU.delay(10000);    // run for four seconds

leftWheel.move ( -100 ) ;
rightWheel.move ( -100 ) ;

CPU.delay(10000);    // run for four seconds

leftWheel.stop () ;
rightWheel.stop () ;
}
}
```

Notice that the forward and backward constructor parameters for the left and right servo are reversed. This is necessary because they rotate in the opposite direction to go forward. Subsequent movements will be based upon the servos rotating in the proper direction.

The program is designed to drive the J-Bot forward for 4 seconds and then backward for the same amount of time returning to its starting point. In practice this J-Bot will not return to the exact same spot because the rest of the servo calibration process is incomplete. This will be handled in the next activity.

If the J-Bot veers to the right when it is programmed to go straight forward, either the left wheel needs to slow down, or the right wheel needs to speed up. Since the servos are pretty close to top speed as it is, slowing the left wheel down will work better. You can do this by making the pulse period to the left servo, which is connected to P13, smaller.

You can adjust the constructor parameters and run the program again to see about getting the J-Bot to move in a straight line. Otherwise, move onto the next activity where we have a program that will help you to do it.

Activity #5: Software Calibration - Navigating a Straight Line

The calibration done in Activity #3 was to determine the center point. This Activity will determine the forward and backward limits for both servos. This set of values should then be used with examples presented throughout the rest of this book.

There are two ways to approach the calibration issue. The first is to continue the approach started in Activity #4 and continued here. The other is to use a more complex, interactive application

that handles all the calibration process. The CalibrateWheelServos program is this application. It utilizes the multitasking system described in Chapter #4 so we will not go into what the program does internally. Instead, the interactive interface will be briefly described.

When you run the CalibrateWheelServos.java program it will start by printing the following in the Messages window:

```
Press ? for command list.  
>
```

Entering ? will display the following help message:

```
>?  
? - Help/current status  
C - current status  
H - halt, turn off wheels  
; - comment  
L.6 - increase left speed  
l.4 - decrease left speed  
R.3 - increase right speed  
r.1 - decrease right speed  
F.7 - forward (100)  
S.8 - stop (0)  
B.9 - backward (-100)  
Q - Quit. Exit program
```

Entering C (current status) will display the current settings for the two wheel servos as in:

```
Left:  F-C-B = 285-175-80  
Right: F-C-B = 80-175-285
```

The idea behind this program is the as with the other interactive calibration programs: to get the wheels running at the same rate. The F, S, and B keys will set the servo movement to forward, stop, or backward. The L, l, R and r keys will increment or decrement the left or right servo settings for the currently selected movement. The current status is displayed when the program quits. It will also stop the servos. The three results for each servo can be used when creating a BasicWheelServo.

If you are going to work this out for yourself then the first step is enter this program.

```

import stamp.core.*;
import JBot.*

/**
 * Wheel servo control class calibration program
 * <p>
 * Handles PWM support for a free running wheel servo on the J-Bot.
 * Start movement using start() and stop it using stop().
 *
 * @version 1.0 7/23/02
 * @author Parallax Inc.
 */

public class BasicWheelServoCalibrate1 {
    public static void main () {
        BasicWheelServo leftWheel =
            new BasicWheelServo (
                CPU.pin12    // pin
                , 110         // forward
                , 175         // center
                , 240         // backward
                , 2000        // low
            ) ;
        BasicWheelServo rightWheel =
            new BasicWheelServo (
                CPU.pin13    // pin
                , 240         // forward
                , 175         // center
                , 110         // backward
                , 2000        // low
            ) ;

        mainloop: while ( true ) {
            System.out.println( "Enter key

            switch ( Terminal.getChar () ) {
            case ' ':
                break ;

            case '+':
                leftWheel.forward ++ ;
                break ;

            case '-':
                if (leftWheel.forward > 1 ) {
                    leftWheel.forward -- ;
                }
                break ;

            case 'Q':
            case 'q':
                break mainloop;

            default:

```

```

        continue mainloop;
    }

    System.out.print ( "Left forward value is
    System.out.println ( leftWheel.forward + leftWheel.center ) ;

    leftWheel.move ( 100 ) ;
    rightWheel.move ( 100 ) ;

    CPU.delay ( 2000 ) ;

    leftWheel.stop () ;
    rightWheel.stop () ;
}
}
}

```

This program will determine the forward value for the left wheel. It is assumed that the right wheel limit for forward movement can be matched by the left wheel. If this is not the case then its value may have to be changed within the program.

The BasicWheelServoCalibrate1 program creates the left and right servo objects. It then waits for a character to be entered in the Message window that will be presented the first time as message is sent to it. In this case it will be "Enter key".

The +, - and space bar will cause the J-Bot to move forward. The spacebar uses the current settings while the other two change the value of the forward object variable of the left wheel object. The idea is to get the J-Bot to move forward in a straight line. When it does you have the settings for the forward direction. The program does not simply print the value of leftWheel.forward because this has been adjusted with respect to leftWheel.center. Instead it prints the value that is the forward argument to the left servo constructor.

If the J-Bot veers to the left then decrease the pulse width to speed up the left wheel. If the J-Bot veers to the right then increase the pulse width to slow down the left wheel. If the pulse width

This takes care of the forward direction parameters. The backward direction parameters can be determined using the following steps.

- ❑ Make the necessary changes in BasicWheelServoCalibrate1 so that it makes the J-Bot go full speed backward.
- ❑ Change the program so keyboard input adjusts the rightServo's **backward** object variable. Remember to change the

```
System.out.println ( leftWheel.forward + leftWheel.center ) ;
line accordingly.
```

- ❑ Save the new program as BasicWheelServoCalibrate2.java and change the name of the class accordingly.

Perform the same test as before to calibrate the J-Bot's servos. The only difference will be that the J-Bot will be going backward. Write the final results for both servos below for future reference.

	Left	Right
Forward		
Center		
Backward		

We now have settings for forward and backward movement. Although these settings will keep the J-Bot on the straight and narrow for at least a few inches we are not done with this process. Two other aspects will need to be handled first. These include circular movement and movement using feedback.

Circular movement includes pivoting and turns. The J-Bot can pivot on its axis. That is turning without moving from the same spot. This is covered in the next chapter.

Feedback controlled movement is something that has not been used thus far. It will be covered in Chapter 8. All movement thus far has been without feedback. The programs simply move the J-Bot using arbitrary parameters for a fixed amount of time. The distance traveled and the accuracy of movement is based upon the calibration performed in this chapter. The problem is that the calibration is an approximation and movement of the servos is not exactly repeatable, only approximately repeatable.

The amount of error may be minor but it is cumulative. This can lead to relatively minor deviations at long distances such as when the J-Bot tries to go straight for two yards. Unfortunately, the problem is more severe when rotation is involved. A 90 right degree turn may really be an 88 degree turn. The difference is a fraction of an inch if the J-Bot is pivoting but do this a couple dozen times and the J-Bot orientation is off by 45 degrees. Add a little linear movement error and the J-Bot can be completely lost assuming it is using dead reckoning, keeping track of its position based upon its desired movement.

Feedback movement is not simple which is why it is left to chapter 8. It also requires additional hardware that will be installed using instructions in that chapter.

In the meantime, we'll stick with movement that does not utilize feedback. It is possible to do quite a lot without this type of feedback. Instead we'll use other sensors for feedback to determine the desired direction or distance to an obstacle. The lack of wheel control feedback is often not a problem in these kinds of environments. For example, if the J-Bot is trying to navigate its way out of a maze then it needs to avoid the walls of the maze. As long as it stays away from the walls it does not really care about its position. In this case, how far the J-Bot moves and what its orientation is does not really matter.

Before checking out these other feedback mechanisms we'll take a look at how the J-Bot can turn and rotate.



Summary and Applications

Congratulations on the construction and operation of your J-Bot! Through following the procedures in this chapter, you may have had your first taste of testing and troubleshooting at the system and subsystem levels. Lots of other essential topics were covered that will get used and re-used throughout this text. For example, the Debug Terminal will be your best and most used tool for testing and troubleshooting each circuit as well as many upcoming programs.

The aspects of the Java programming language and virtual peripherals were introduced along with some example programs to get you started with the J-Bot. Software calibration and the terminal interface also was introduced.

Real World Example

From the space shuttle all the way down to the J-Bot, isolating and testing subsystems during each phase of development is critical to make sure the whole thing runs when it's put together. More importantly, isolating and testing each subsystem minimizes the time spent on, and difficulty level of, troubleshooting. At the beginning of the chapter, the problems associated with not iteratively developing and testing were discussed. Imagine if nobody tested the Space Shuttle's subsystems before putting it together. It would take hundreds of years for NASA to get all their problems sorted out!

Whether it's robotics competitions, product development, or space programs, subsystem and system level development and testing is the way to avoid unnecessary delays when working from the beginning to the end of a project. Especially in product development, groups of engineers develop systems and subsystems. Often, it's not until late in the design cycle that the system level testing and system integration occurs. Sometimes, all a design team knows are the input and output (I/O) requirements of their particular module in the project. Regardless, engineering design teams still have to iteratively develop, simulate (which we did not do here), and test the subsystems within the project module they are working on.

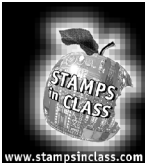
Software calibration also was introduced. This is currently a hot topic because many appliance makers are working on incorporating microcontrollers that can communicate across the Internet into their products. Remote diagnostic programs can then enhance the ability of the microcontrolled devices to self calibrate. Also, technicians can perform software calibration, diagnosis, and in some cases repair, all remotely. Imagine your TV picture going bad a few years from now. Getting it fixed might involve the pressing of a few buttons on your TV remote.

Then the microcontroller in your TV will log onto the Internet and report the problem to the company that makes the TV. Before sending out a TV repair person, your TV's problem will be diagnosed, perhaps by a computer program. It might even be fixed by the program sending special control instructions back to the microcontroller. Otherwise the TV repair person will show up with the right parts to fix it.

J-Bot Application

One item you'll investigate in the Questions and Projects section is what happens when the wiring of the servos gets changed. How does this get handled? It involves more changes than you might think. For example, if you were to unplug a servo from servo port 12 and then plug it into servo port 13, you can't just change the drawing that shows what port to plug it into. The schematic, which is the preferred method of communicating wiring information, has to be changed, but so do all the program listings. At some point you might want to add more servos to function as grippers. Although a gripper design is not included in this text, Questions and Projects has exercises that will prepare you for connecting servos to different ports.

So far, the J-Bot can be programmed to roll forward or backward or to rotate in place. During some of your testing, small variations in servo performance were discovered and corrected in software. The advantage of calibration in software is that, instead of mechanically adjusting the J-Bot, the JAVA program is modified to make the correction. This software calibration was all done at only two or three speeds: full speed ahead, full speed reverse, and full stop. In the projects section you'll get a chance to further research and generalize the software calibration for a variety of speeds.



Questions and Projects

Questions

1. Assuming the circuit inside the servo that controls the DC motor updates what it's doing every 20 ms, how many times per second does the DC motor get updated? Hint: This is a division problem.
2. Discuss how you would modify Figure 0.1 and Program Listing 1.2 if you wanted to test each of your servos using I/O line P15. Keep in mind that the board does not have sockets for this pin but it can be wired up using the wireless breadboard.

Exercises

1. What would happen if you change the low time in the **update** method? How would this effect the servos' operation? Draw a diagram similar to Figure 0.2 based on a pulse train using values of 1000 and 3000.
2. What is the necessary **update** method argument to make a 1.626 ms pulse?
3. What are the maximum and minimum pulse widths that can be generated using the PWM **update** command?

Projects

1. Program the J-Bot to move in several different patterns. Try the following:
 - (a) Identify a pair of **forward** values that make the J-Bot move slowly straight forward. Shoot for wheel speeds of 4 revolutions per minute (RPM). Some trial and error will be necessary to find the **forward** value to make each servo turn at this rate.
 - (b) Identify a pair of **backward** values that make the J-Bot move very slowly straight backward at the same speed. How do these values compare (or not compare) to those identified in 1 (a.)?
2. Make a graph of wheel speed as a function of pulse width for each servo. Use several pulse widths between 0.8 and

2.2 ms (**update** values between 10 and 350). Either count how many revolutions the wheel completes in a specified time (20 seconds or a minute), or see how much time it takes to complete 10 revolutions.

Your graph might look similar to Figure 2.5. This graph was generated by a Microsoft Excel spreadsheet using eight data points and the "Best Fit" option. Collect data points and make your own graphs, one for each servo. In general, when you plot more data points, you can expect your graph to be more reliable. However, there are lots of techniques for reducing the number of measurements. One example would be to take a few measurements to find the curved areas of the graph, then focus on taking many measurement in those areas while taking only a few measurements in the areas that are linear.

Note: Expect your graphs to look different from Figure 2.5 because it's for a different kind of servo from the one in your kit.

Use your graphs to predict the pulse widths required to make your J-Bot go straight forward or straight backward with less trial and error. Try this at a variety of speeds. The rotational speed of one servo will correspond with a certain PWM **update** value. Remember that the values you select will come from opposite sides of the center point.

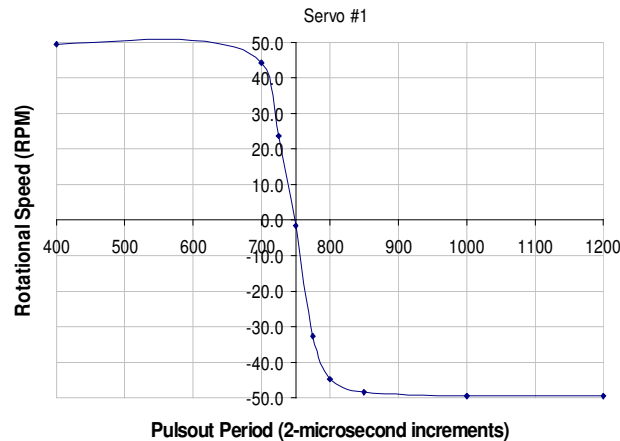


Figure 2.5: **pulsout period** vs. Rotational speed for servo.

Test the accuracy of the graphically determined predictions by programming them into your J-Bot and testing to see how straight it goes. By focusing on one servo for your corrections, you can calculate the percent error of your initial guess. So that error from the other servo is not a

factor, fine tune it using trial and error so that it rotates at the exact speed predicted by your graphs. Then, you can figure the percent error on the servo that you have not fine tuned using this equation:

$$\% \text{ error} = \frac{\text{exact duration} - \text{predicted duration}}{\text{exact duration}} \times 100\%$$

The exact **period** is what you will arrive at by adjusting a single servo using trial and error. The predicted **period** is the value from the graph for that servo. You can expect percent errors of between 5 and 25% depending on the resolution of your graph and other factors such as imperfections in wheel alignment and slight differences in wheel size.