

# 11

## Introduction to Servomotor Programming

---

PulseOut

Pulse Widths

Robot Motion

---

### 11.1 Welcome to Servo Programming

In the previous chapters, we've done some cool things, but we've essentially treated our robot as a desktop computer. Now we are ready to attach the BX-24 to a robot body and make it **mobile** with the use of servomotors. In this chapter, you will learn what servos are, how they operate, and how modified servos can be used as drive motors for the Robodyssey Mouse.

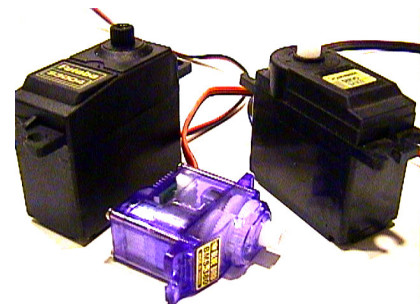
This chapter covers many important topics, so read it slowly and thoroughly, not only because it will help you to become a better programmer, but because it will help you avoid mistakes that could damage your robot or the BX-24! Unlike many robot kits on the market today, the BX-24 microcontroller and Robodyssey robots aren't toys. They aren't made for young children and they aren't made for pure entertainment. These are real computers and real robots, with real circuits, and real possibilities for success or failure. While this is a sobering prospect, it is also an exciting one – when you learn how to program a Robodyssey robot, you gain real knowledge in the fields of physics, robotics, and electrical and mechanical engineering.



Don't be afraid to make mistakes – we all make mistakes. Read this chapter with a clear head and pay close attention to the caution signs like the one shown on the left. These signs indicate important information, which may prevent serious mistakes. Let's begin!

### 11.2 What are Servos?

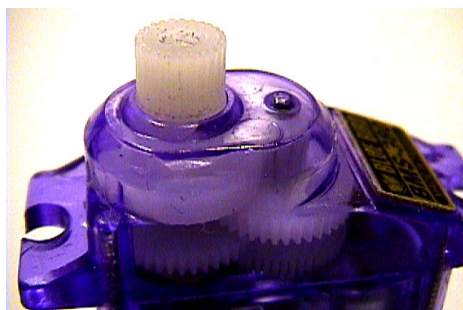
Before you start programming your servos, you should learn a little bit about them. Servos, like the ones shown in Figure 11.1, are electro-mechanical devices most commonly found in radio controlled (R/C) airplanes, cars, and boats. However, as robotics becomes more popular, servo sales for robotics purposes account for an increasingly large percentage of the servo market share.



**Figure 11.1.** Servos like these are commonly used by the R/C hobbyist and roboticist.

Servos are small mechanical devices whose sole purpose is to rotate a tiny shaft extending from the top of the servo housing, shown in Figure 11.2. Extending from the side of the servo is a thin cable comprised of three wires. Two of the wires are used to send power to the servo's motors and one wire is used to send commands from the BX-24 to the servo. (More on these wires later.)

R/C hobbyists use servos to steer model airplanes, cars, or boats. Roboticists use servos to drive the wheels of rovers and the legs of walking robots. Attach an infrared ranger to Robodyssey's Turret Mount (P/N: TurWSer) and it becomes a pivoting robotic head capable of scanning the horizon for obstructions. Servos can also be programmed to open and close the fingers of claw-like hands such as the Robodyssey Gripper (P/N: GKWS).<sup>1</sup> I've even used them as winches and shovels on my robots.<sup>2</sup> The possibilities are endless!



**Figure 11.2.** The servo's sole purpose is to rotate the shaft on top of the servo.

### 11.3 Building the Mouse

Lets put the topic of servos on hold, for now, and introduce – ta-da! – the Mouse robot!



Chances are you've already put together your Mouse, but in case you haven't, **I strongly suggest that you read and follow the Robodyssey assembly manual that came with your robot.** The following instructions are not intended to take the place of the Robodyssey manual!



Locate one of the Futaba servos and one of the wheels that came with your Mouse kit. Find the large Allen (hex) wrench that came with your kit. Install the left servo and wheel as shown in Figures 11.3 and 11.4. (All references to “left” and “right” are from the robot's point of view. That is, “left” refers to the robot's anatomical left.) Take care that you **do not spin the wheel** during this or any other procedure, as this may damage the servo. Hold the wheel firmly with one hand so that it does not spin, being careful not to over-tighten the bolts – hand-tight is fine!



**Figure 11.3.** The left servo is installed on the Mouse. “Left” refers to the Mouse's anatomical left.

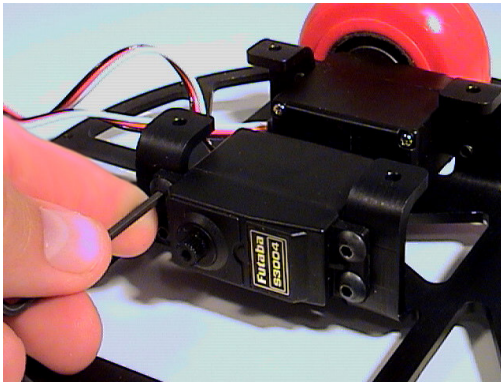


**Figure 11.4.** The left wheel is secured to the servo's shaft. Be sure to hold the wheel to prevent it from rotating as you tighten!

Repeat the process and attach the servo/wheel assembly for the Mouse's right-hand side as shown in Figures 11.5 and 11.6 (next page). Again, do **not** allow the wheels to spin, and do **not** over-tighten the axle bolts!

<sup>1</sup> You'll learn about the infrared ranger in *Chapter 16*, and the Turret Mount and Gripper in *Appendix G*.

<sup>2</sup> See Challenge Problems 22 and 23 at the end of this chapter.

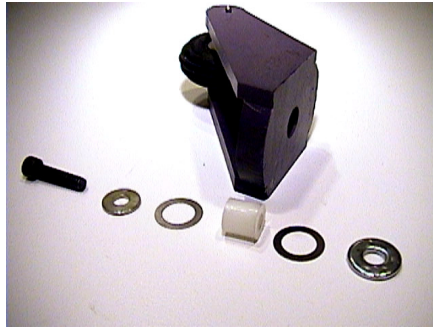


**Figure 11.5.** The right servo is installed on the Mouse.

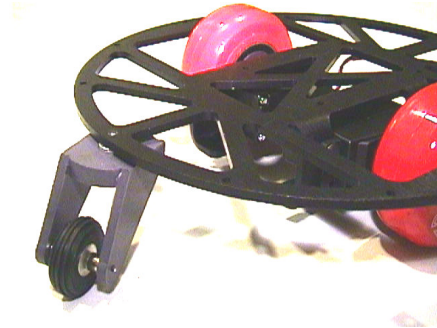


**Figure 11.6.** The right wheel is then secured.

Robodyssey gives you the option of attaching a tail wheel or a phenolic ball to the rear of the Mouse. The tail wheel shown in Figure 11.7 is best for use on rough terrain like carpet and asphalt. The phenolic ball shown in Figure 11.8 is best for use on hard, smooth surfaces like tabletops and tiled floors. Whatever option you choose, install the rear support on the end that is farther from the wheels. This way the robot will be properly balanced.



**Figure 11.7.** Use the tail wheel when the Mouse is on rough, uneven surfaces.



**Figure 11.8.** Use the phenolic ball when the Mouse is on hard, smooth surfaces.

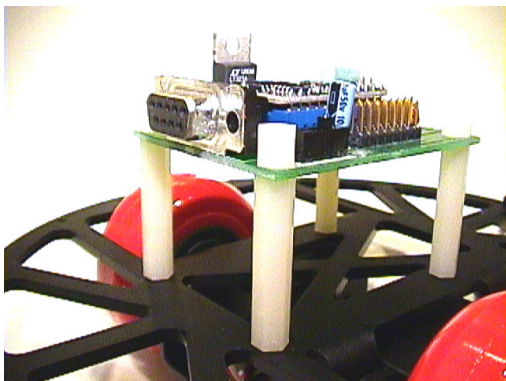
### Attach the RAMB

If you haven't done so already, attach the Robodyssey Advanced Mother Board (RAMB) and battery pack to the Mouse chassis.<sup>3</sup> Be sure to place the plastic insulators between the RAMB and the metal chassis of the Mouse. Robodyssey offers both short cylindrical spacers and 1-inch hexagonal nylon standoffs for this purpose. I prefer to use the tall standoffs (Figure 11.9) because it is easier to attach the serial cable when the Mouse is loaded

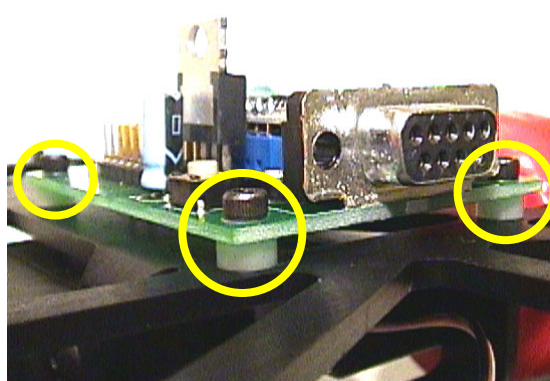
<sup>3</sup> At the time of this printing, Robodyssey had developed a new RAMB prototype. See my website ([www.basicxandrobotics.com](http://www.basicxandrobotics.com)) and Robodyssey's ([www.robodyssey.com](http://www.robodyssey.com)) for the latest updates and details.



down with sensors and other peripheral devices. If you choose to use the small spacers, secure the RAMB with metal screws as shown in Figure 11.10. If you use the hexagonal standoffs, you can secure the RAMB either with metal screws or other nylon standoffs, as shown in Figure 11.9.<sup>4</sup>



**Figure 11.9.** In this configuration, hexagonal nylon standoffs are used to raise the RAMB above the robot's main deck.



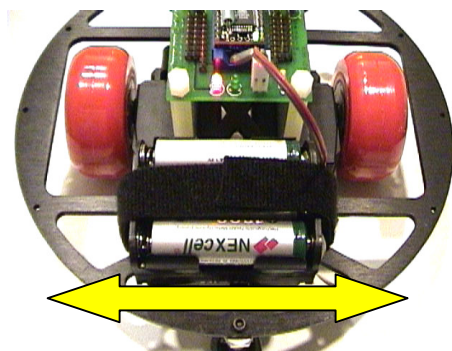
**Figure 11.10.** In this configuration, the RAMB is mounted on short cylindrical spacers and secured with metal screws.



Attach the battery pack to the Mouse with the Velcro strap. Note that the battery pack should be oriented from left-to-right – **not** front-to-back. **This is important!** If the batteries are oriented front-to-back, one of the battery springs may contact the rear screw, which holds the tail wheel or phenolic ball in place. If this happens, the entire Mouse body will be electrically charged. This is bad.



Do **not** use alkaline batteries to power the Mouse. Alkalines provide more power than do rechargeable batteries, and can destroy many small servomotors, including Blue Bird servos.



**Figure 11.11.** Attach the battery pack with the Velcro strap, orienting the batteries **left-to-right**.



Now that the Mouse is assembled, you are almost ready to connect the servos to the RAMB. First, however, it is critical that you understand a bit more about the function of the BX-24 and RAMB pins. **Don't attach any wires until you thoroughly understand the following sections on the BX-24 and RAMB pins, the color conventions of the wire, and the RAMB power!** Take your time reading these sections; if they don't make sense to you, read them again. You may want to bookmark these pages because they contain handy references and important tips.

<sup>4</sup> Take care that you do not over-tighten the nylon standoffs – the threaded ends break off easily.

## 11.4 BX-24 and RAMB Pins

Examine the underside of the BX-24 chip and you'll see 24 tiny pins, shown in Figure 11.12. These pins fit snugly into the 24-pin socket on the RAMB, allowing electrical signals to be sent to and from the BX-24. Certain pins are used for specific tasks. For example, pin 1 on the BX-24 is used to transmit data to the PC, and pin 2 is used to receive data from the PC.

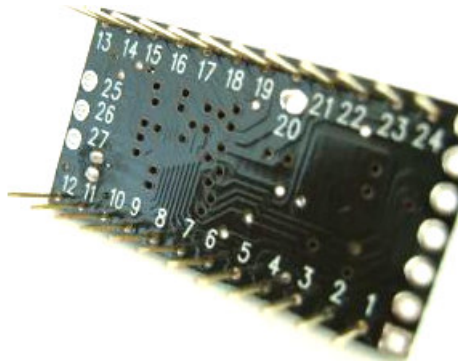


Figure 11.12. The 24 pins of the BX-24.

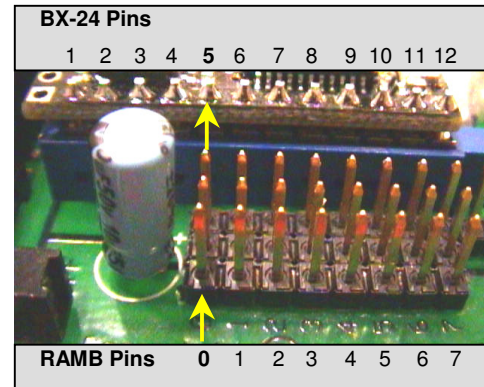


Figure 11.13. Pin row 0 on the RAMB corresponds to pin 5 on the BX-24.

When the BX-24 is secured in the RAMB socket as shown in Figure 11.13, **each pin** on the BX-24 corresponds to a **row of three pins** on the RAMB. Furthermore, notice that pin row 0 on the RAMB corresponds to pin 5 on the BX-24. (Pin row 1 on the RAMB corresponds to pin 6 on the BX-24, and so on.)

### Signal Pins

Each pin row on the RAMB consists of three individual pins. The pins closest to the BX-24 chip are known as the **signal pins** because they are used to carry electrical signals between the BX-24 and peripheral devices such as sensors, servos, and piezoelectric buzzers.<sup>5</sup> The signal pins are highlighted in Figure 11.14.

### Power Pins

The pins in the middle of each row are used to carry power to these peripheral devices, and are known as the **power pins**. Servomotors, for example, need electrical power to spin their shafts. When the servos are plugged into the RAMB, they conveniently draw their power from the middle pin. The power pins are highlighted in Figure 11.15.

### Ground Pins

The pins farthest from the BX-24 chip (closest to the edge of the RAMB) are known as the **ground pins** because they are connected to the electrical ground of the RAMB circuitry. When a sensor or servo is plugged into the RAMB, the ground pins provide these peripheral devices with the necessary electrical ground. The ground pins are highlighted in Figure 11.16.

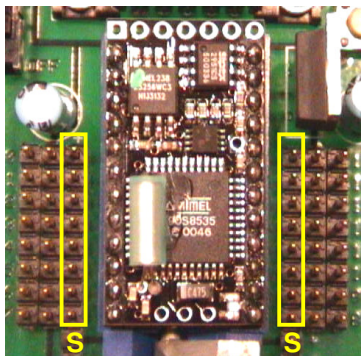


Figure 11.14. The signal pins are those closest to the BX-24 chip.

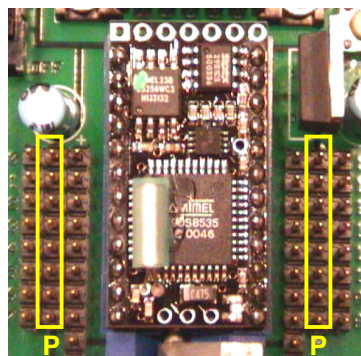


Figure 11.15. The power pins are located in the middle of the pin row.

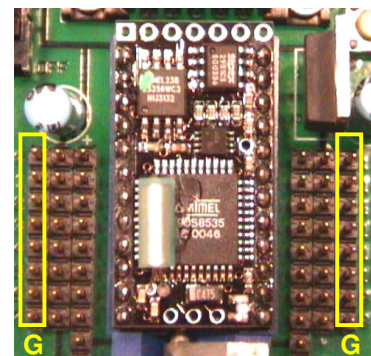


Figure 11.16. The ground pins are those farthest from the BX-24 chip.

<sup>5</sup> See Appendix D and Challenge Problem 9 for more on the use of piezo buzzers.



As I mentioned at the beginning of this chapter, your robot is not a toy and you must take care not to damage it. One certain way to **seriously damage** the RAMB, BX-24, and batteries is to allow a metal conductor to touch simultaneously a power pin and a ground pin. This is known as an electrical “**short**” because the path taken by the electrical current has been shortened and the load resistance has been bypassed. When a short occurs, the current from the batteries flows so rapidly that it can cause the wires and circuitry to become super-heated, damaging the motherboard, microcontroller, and batteries.<sup>6</sup>

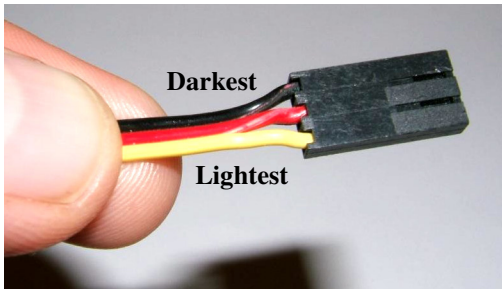
So take care that screwdrivers, Allen wrenches, bolts, wire, and other pieces of metal do not accidentally touch the power and ground pins simultaneously!



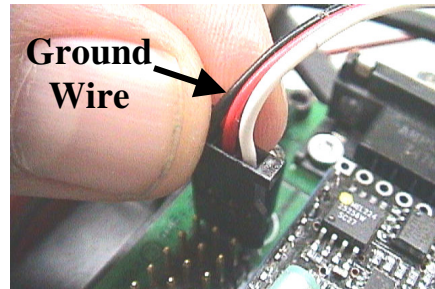
**CAUTION! NEVER** short a power pin to a ground pin! Doing so will produce a **huge electric current** that may damage the microcontroller, RAMB, and batteries.

## 11.5 Color Conventions

It is a common convention in electronics to wire electrical components with wires of distinguishable colors. Usually, sensors and servos are wired from *light-to-dark* – the wire of the lightest color carries the signal, the darkest wire is the ground wire, and the middle wire carries the power.<sup>7</sup> For example, the cable shown in Figure 11.17 has the Yellow-Red-Black color scheme. The yellow wire is the lightest, so it must be connected to one of the **signal pins** of the RAMB. The middle red wire must be connected to one of the RAMB’s **power pins**. The black wire is the darkest, so it must be connected to one of the RAMB **ground pins**.



**Figure 11.17.** These servo wires have the color scheme Yellow-Red-Black.



**Figure 11.18.** This servo is plugged into pin 0 on the RAMB. Note the white and black wires are connected to the signal and ground pins, respectively.

In Figure 11.18, the color scheme is a little different, but the *light-to-dark* rule still applies. Here, the cable’s colors are White-Red-Black. Therefore, the white, red, and black wires are connected to the signal, power, and ground pins, respectively.

When connecting any wires to the RAMB, remember these two rules:



**Lightest color wire = Signal = Closest to BX-24**

**Darkest color wire = Ground = Farthest from BX-24**

<sup>6</sup> I once attended a conference where one of the guest speakers had inadvertently placed a 9-volt battery in his pants pocket with his car keys. One of the keys happened to touch both of the battery leads. After a second or two, the key became so hot that it began to burn a hole in the man’s pants. In the middle of his speech – in front of the entire audience – he cried out in alarm, and ripped off his pants. He took care of the problem, put his pants back on, and finished his speech. True story.

<sup>7</sup> All Robodyssey components follow this convention.

## 11.6 RAMB Power

The RAMB gets all its power from the battery pack of six rechargeable Ni-MH batteries. Since each battery is rated at 1.2 volts, six fully charged batteries wired in series provides a maximum voltage of 7.2V. (Never use alkaline batteries to power your Mouse, especially when you are using servos. Alkaline batteries provide 1.5V, so six of them provide 9.0V to the RAMB. This much voltage is likely to destroy many kinds of servos.) As we've just discussed, peripheral devices plugged into the RAMB get their power from the RAMB power pin. Some devices can handle large voltages between 5V and 7.2V, while other devices work better with a smaller, steady voltage of exactly 5V. The RAMB offers *two options* for powering the sensors, servos, and other peripheral devices – **regulated** power and **unregulated** power. Half of the RAMB power pins provide unregulated battery power, and the other half provide regulated power.<sup>8</sup>

### Unregulated Voltage

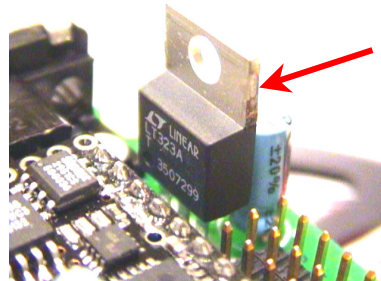
The RAMB provides *unregulated voltage* to its middle power **pins 0 through 7**. Here the word *unregulated* implies that the voltage (and power) to these pins is not held constant.<sup>9</sup> This unregulated voltage comes directly from the battery pack. Thus, when the batteries are fully charged, the voltage to these pins is maximized at 7.2V. Over time, the batteries will drain, and the voltage to these pins will drop accordingly. The unregulated power pins offer the most power, so we will connect our mid-sized servos (for example, the Futaba S3004) to these pins (0-7) on the RAMB. (Never plug sensors or small servomotors into the unregulated side of the RAMB.<sup>10</sup>)

### Regulated Voltage

Devices, such as infrared range finders, voltage dividers, sensors, and small servomotors (such as the Blue Bird BMS-380) require low or constant voltage to function properly.<sup>11</sup> Power **pins 8 through 15** on the RAMB are wired to provide such a *regulated voltage*. Here, the term *regulated* means controlled or constant and the voltage on these pins will always be about 5V.<sup>12</sup> Of course, this doesn't happen by magic. An electronic component known as a **voltage regulator** (see Figure 11.19) is used to reduce a portion of the battery pack voltage to a constant 5V. This regulated 5V source is then sent to power pins 8-15 on the RAMB. The best way to remember which pins have regulated voltages and which have unregulated voltages is by looking at the RAMB itself. Note the location of the voltage regulator shown in Figure 11.20. The voltage regulator is positioned on the same side of the motherboard as the regulated voltage pins, so it is easy to remember.



**Figure 11.19.** Here is a voltage regulator before it is mounted to the RAMB.



**Figure 11.20.** The voltage regulator is mounted on the same side of the RAMB as the regulated power pins.



It is possible to power a larger servo (e.g., Futaba's S3004) with the regulated power pins, but use caution. If these large servos stall, they will consume so much current that the voltage regulator could seriously overheat. Using the regulated power should be okay if there is not a heavy load on the servos.

<sup>8</sup> From now on, I will refer to the three pins of each pin row on the RAMB as a single pin. For example, "Plug the servo into pin 0 on the RAMB."

<sup>9</sup> Electrical power consumed by an electronic device can be calculated by the formula,  $P = V^2/R$ , where  $V$  is the input voltage and  $R$  is the resistance of the device. Often the terms *power* and *voltage* are interchanged if the resistance of the device is constant.

<sup>10</sup> See *Appendix E* for more specs on the various servomotors sold by Robodyssey.

<sup>11</sup> See *Chapter 16* for information regarding the infrared ranger and *Chapter 17* for a discussion on voltage dividers.

<sup>12</sup> If the battery voltage drops below 5V, the voltage on these pins will likewise drop below 5V.



## 11.7 Powering Up the Servos

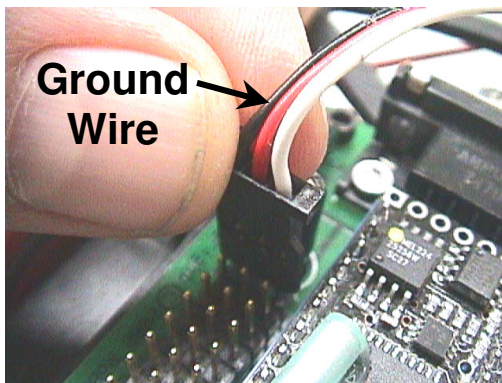
### Connect the Servo Cables to the RAMB



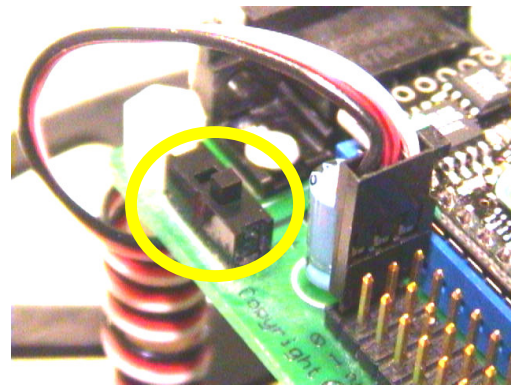
You are finally ready to connect the servos to the motherboard. However, before you do so, **make sure that you disconnect the power to the RAMB**. That is, unplug the battery pack from the RAMB if it is connected. Then, if you connect the servo incorrectly, nothing will be damaged.



First, connect the **left servo** cable to pin 0 on the RAMB, as shown in Figure 11.21. Remember that “left” refers to the robot’s left. Make sure that lightest-colored wire is plugged into the signal pin (closest to the BX-24) and the darkest-colored wire is plugged into the ground pin (farthest from the BX-24). Note that we are taking advantage of the unregulated voltage to power our mid-size servo. Recall that pin 0 on the RAMB corresponds to pin 5 on the BX-24 microcontroller. Take some time to review the pin-numbering scheme, because there will be frequent references to pins on the RAMB and pins on the BX-24 throughout this chapter.



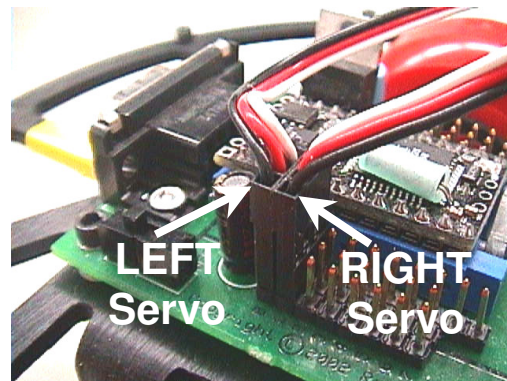
**Figure 11.21.** The left servo cable is connected to pin 0 on the RAMB (BX-24 pin 5). Note the darkest wire (in this case the black wire) is the ground wire and is positioned farthest from the BX-24.



**Figure 11.22.** Notice the power switch to the RAMB is in the off position while the cable is being connected.



Repeat this procedure for the **right servo** cable, connecting its cable to **pin 1** on the RAMB (BX-24 pin 6). **Double-check** to ensure its lightest-color wire is positioned closest to the BX-24 and the darkest-color wire is positioned farthest from the BX-24, as shown in Figure 11.23.



**Figure 11.23.** The left servo cable is connected to pin 0 on the RAMB (BX-24 pin 5). The right servo is connected to pin 1 on the RAMB (BX-24 pin 6). Note the lightest color wire of both cables is positioned closest to the BX-24, and the darkest wire is farthest from the BX-24.

When you are certain that the servo cables have been connected properly, connect the batteries to the motherboard and turn it on. The robot may give a quick jerk, but this is normal.<sup>13</sup> If your servos aren’t smoking, you probably did everything right. Now you are ready to get your Mouse moving.

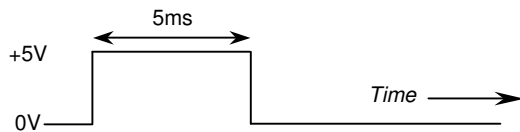
<sup>13</sup> If you are using this robot in the classroom, be aware that another student may have left his or her program on the BX-24. If necessary, be prepared to stop execution of that program.



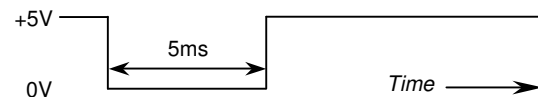
## 11.8 Pulse Widths – The Servo’s Heartbeat

Each servo has a built-in processor that responds to electrical pulses sent to it. The BX-24 creates an electrical pulse by sending voltage to one of its pins for a very specific amount of time. The microcontroller cannot control how *much* voltage is sent – it simply turns the voltage on or off. When the voltage is on, the BX-24 outputs +5V. When the voltage is off, 0V is output. The BX-24 can turn the output voltage on and off rapidly, thereby creating **pulses** of *high* and *low* voltages.

The duration of these pulses is known as the **pulse width**. The longer the voltage is applied, the larger the pulse width. The pulse width is measured in seconds, but we often use milliseconds (ms) to describe them because the pulse duration can be very short. Recall from your introductory science classes that 1s is equivalent to 1000ms, therefore 1ms = 0.001s. For example, Figure 11.24 shows a pulse width of 5ms. **In our code, however, we always enter pulse widths in units of seconds**, so we would enter 0.005s in this case.



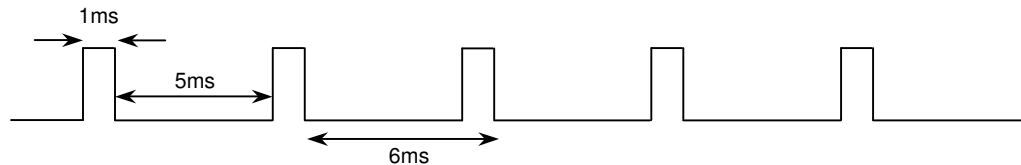
**Figure 11.24.** A *high* pulse whose pulse width (duration) is 5ms (0.005s).



**Figure 11.25.** A *low* pulse whose pulse width is also 5ms in length.

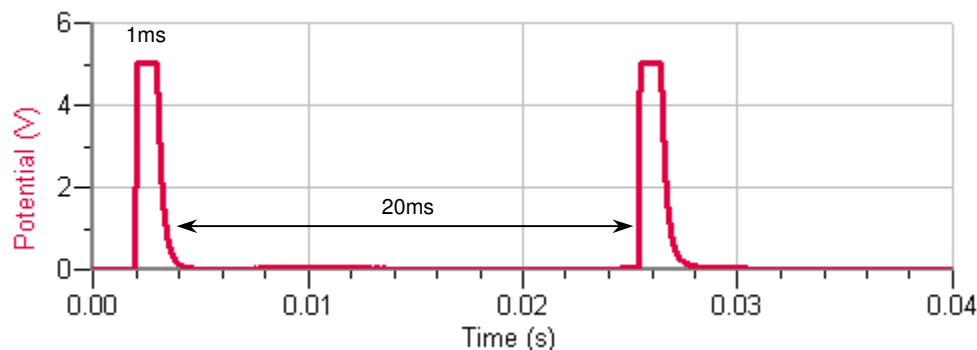
The BX-24 can send either a *high* pulse (as shown in Figure 11.24) or a *low* pulse (as shown in Figure 11.25). Notice in high pulse mode, the voltage is normally 0V and pulses high when commanded. Conversely, in low pulse mode, the output is normally +5V and goes low when commanded.

With Do-Loops and For-To-Next loops, the BX-24 can produce a series of pulses at a regular rate. This repetitive series of pulses is known as a **pulse train** and can be produced by inserting a fixed time delay between the pulses. For example, Figure 11.26 shows a train of pulses, each with a 1ms pulse width, separated by a 5ms delay. The time between successive pulses is known as the **period**, which, in this case, is 6ms. It is easy to see why this output is often called a *square wave*.



**Figure 11.26.** A train of 1ms pulses. Since there is a 5ms delay between pulses, the pulse period is 6ms.

Figure 11.27 shows the actual BX-24 output voltage of a train of 1ms (0.001s) pulses with a 20ms (0.02s) delay between pulses. In reality, the pulses are not perfect square waves, as the above figures would lead us to believe.



**Figure 11.27.** A train of 0.001s-pulses with a 0.020s delay between pulses as measured by an oscilloscope.

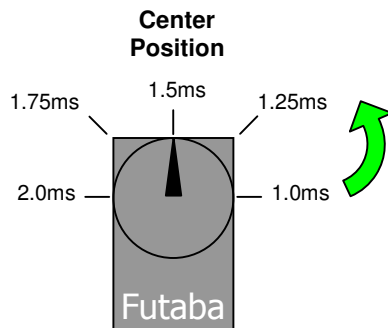
## 11.9 Servo Operation

So, how does the BX-24 control the servo? You have probably guessed that it does so with electrical pulses. The pulses are sent from the BX-24 to a control board within the servo itself. The servo's control board interprets the pulses and rotates its shaft either clockwise or counterclockwise based on the pulse widths. Servos will not respond to just any pulse width; rather, they are limited to a well-defined range of pulse widths. Most servos have a minimum pulse width limit around 0.0010s (1.0ms) and a maximum limit around 0.0020s (2.0ms), although the actual minimum and maximum pulse widths will vary slightly between the various servo brands. Sending a pulse whose pulse width is outside this range may **damage** the servo's control board. Repeatedly sending these bad pulses will almost certainly **destroy** the servo.

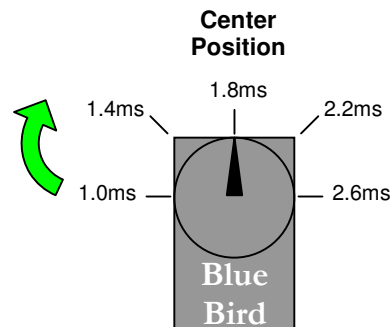


### Pulse Widths = Positions

Servos interpret pulse widths as **positions**. Each position along the arc traced out by the rotating shaft has a corresponding pulse width. When we send a pulse to the servo, the control board calculates which way the shaft should rotate in order to reach the corresponding position. There are two ways that servo manufacturers can wire their servos: positions increasing **clockwise**, and positions increasing **counterclockwise**. The positions and corresponding pulse widths for Futaba and Blue Bird brand servos are shown in Figures 11.28 and 11.29. All servos have a **center position**, sometimes called the *neutral position*. Both servos below are in the **center position** as indicated by the dark pointers. Observe that the Futaba center position corresponds to 1.5ms and the Blue Bird's center is at the 1.8ms-position.<sup>14</sup>



**Figure 11.28.** The pulse width positions of a Futaba S3004 servo increase counterclockwise.



**Figure 11.29.** The pulse width positions of a Blue Bird BMS-380 servo increase clockwise.

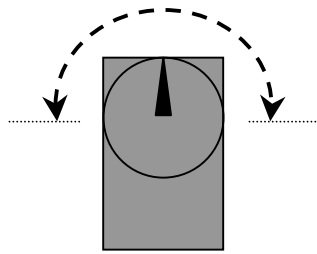
One popular servo model, Futaba's S3004, rotates counterclockwise with increasing pulse widths, while another servo, the Blue Bird BMS-380, rotates clockwise with increasing pulse widths. This difference is significant to note. If your servos were included as part of Robodyssey's Mouse kit then you probably have the Futaba S3004. Robodyssey does sell Blue Bird servos as well, but I assume that most readers own the Futaba servo, so I will focus my discussion on the S3004. Rest assured, the following discussion is valid for **all servo models** – you simply must keep in mind which servo you are using and program it accordingly. For a detailed comparison of a number of popular servo models, see *Appendix E: Servo Comparisons and Physical Limitations*.

### Modified Versus Unmodified Servos

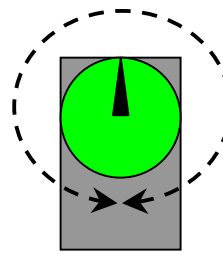
There is another important difference among servo brands: servos can be either **modified** or **unmodified**, and we must know which kind we are dealing with before we can program them. *Unmodified servos* are those that come straight off the hobby shop shelves and have never been “tampered” with. All you radio-control (R/C) hobby-types are certainly familiar with the unmodified servos, for they are used to control your airplanes, cars, or boats. *Modified servos* have been, well, modified – albeit by an expert and with good reason. Modified servos have been altered electrically and mechanically, making them capable of spinning *indefinitely* in either direction. This handy modification means that modified servos can act as drive motors for robots – drive motors that are easily controlled by the BX-24! Both modified and unmodified servos play an important role in robotics, as we are about to find out.



<sup>14</sup> These are approximate values, because center positions can vary greatly from servo to servo.



**Figure 11.30.** An **unmodified** servo can rotate only through 180°.



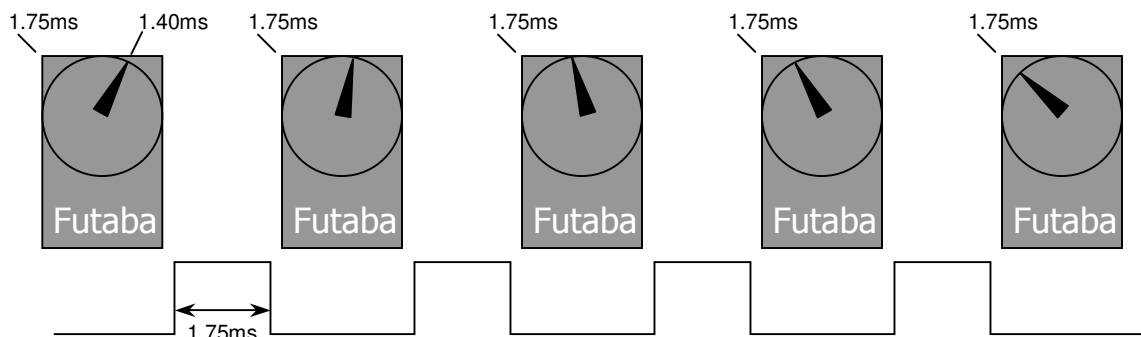
**Figure 11.31.** A **modified** servo is capable of full rotation in either direction. For our purposes, modified servos are illustrated in green.

You **cannot** tell the difference between modified and unmodified servos simply by looking at them.<sup>15</sup> If you purchased them from a hobby shop then you probably have **unmodified** servos. If you purchased a Robodyssey Mouse kit, the servos included with the robot have been **modified** for continuous rotation. As shown in Figures 11.30 and 11.31, an unmodified servo is designed to rotate only through an arc of roughly 180°, while the modified servo is able to rotate over 360° in either direction. Modified and unmodified servos respond differently to pulse width inputs. Let's look at the response of unmodified servos.

### Unmodified Servo Control

When an **unmodified servo** detects a well-defined pulse, the servo shaft will begin rotating toward the position that corresponds to that pulse width. This is shown in Figure 11.32. Notice that the servo can move only a small distance with each pulse, so it may require several pulses to get the servo to actually reach the desired position.<sup>16</sup> If the shaft's current position is sufficiently **near** the desired final position, one pulse may be all that is necessary to move the required distance. If the shaft is already **at** the desired final position, the shaft will not rotate at all. (It is important to note that *the shaft rotates faster when it is far away from the desired position*. This fact will be significant when we cover advanced servo operations in *Chapter 13*!)

Let's look at an example: A Futaba S3004 servo is initially situated at the 1.40ms-position when it receives a train of 1.75ms-pulses, as shown in Figure 11.32. The servo's control board determines that the shaft must rotate **counterclockwise** to reach the 1.75ms-position, and furthermore, it determines that the *initial* rotation speed should be **high** because the 1.40ms-position is far away from the 1.75ms-position. Since the first pulse does not move the shaft to the desired position, the process is repeated and the shaft continues to rotate counterclockwise with **decreasing speed**. After a few pulses, the shaft reaches the desired 1.75ms-position.



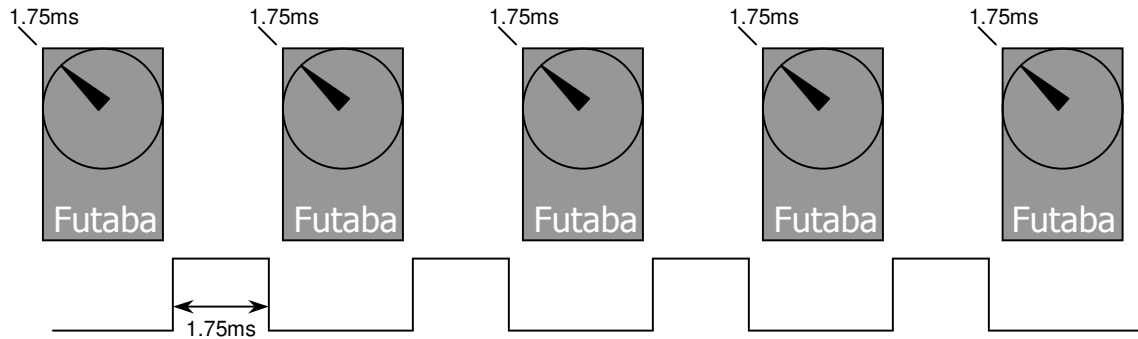
**Figure 11.32.** This unmodified Futaba S3004 servo is originally at the 1.40ms-position when it receives a train of 1.75ms-pulses. The servo's control board interprets the pulses and causes the shaft to rotate (counterclockwise) toward the 1.75ms-position. After a few pulses, the shaft is in the desired 1.75ms-position.

<sup>15</sup> In my figures, you **can** tell which are modified servos simply by looking because I've given them green shafts.

<sup>16</sup> I have found that a train of 17 individual pulses is sufficient to rotate *my* servos 180°; servo models are slightly different and may require more or fewer pulses to get the job done.

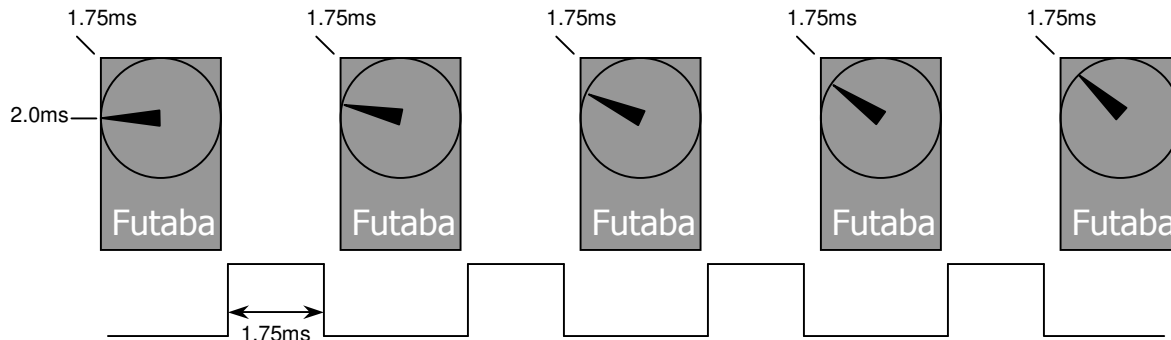


If additional 1.75ms-pulses are sent to the servo, they are ignored since the shaft is already in the 1.75ms-position. This is shown in Figure 11.33, below.



**Figure 11.33.** Additional 1.75ms-pulses are sent to the unmodified servo but the control board ignores them since the servo is already at the 1.75ms-position.

Figure 11.34 below shows another train of 1.75ms-pulses being sent to the Futaba servo, but this time the shaft will rotate **clockwise** since the desired 1.75ms-position is to the **right** of the original 2.0ms-position. In addition, the shaft will spin at a relatively **slow speed** since the 1.75ms-position is near the 2.0ms-position.



**Figure 11.34.** In this example, the unmodified Futaba servo rotates clockwise to reach the 1.75ms-mark.

Follow these general rules when programming a **Futaba S3004 unmodified servo**:

- A train of pulses whose pulse widths are approximately **1.0ms** will always rotate the shaft **fully clockwise** to its right-most position.
- A train of pulses whose pulse widths are approximately **1.5ms** will rotate the shaft to its **center position**.
- A train of pulses whose pulse widths are approximately **2.0ms** will always rotate the shaft **fully counterclockwise** to its left-most position.

For more details on the Futaba S3004 and other servos, consult *Appendix E: Servo Comparisons and Physical Limitations* for handy data tables and comparisons.



If you would like to write code for an unmodified servo, I strongly urge you to read *Chapter 13* and *Appendices E and G*. If you are not careful, unmodified servos can be easily damaged.

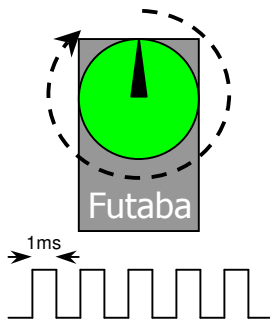
## Modified Servo Control

A **modified servo** behaves exactly like an unmodified servo, with two major exceptions:

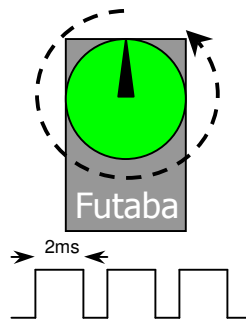
1. A modified servo is capable of spinning indefinitely either clockwise or counterclockwise.
2. A modified servo receiving a train of pulses will come to *rest* only if the pulse widths correspond to the servo's *center position*.

Modified servos are able to spin indefinitely because the servo's feedback mechanism has been removed with a handy bit of rewiring. This means the servo never knows the actual position of its shaft. Instead, it is tricked into thinking that the shaft is *always* in the **center position**. Therefore, any pulse whose pulse width does **not** correspond to the servo's center position will cause the servo to rotate either clockwise or counterclockwise. Moreover, the greater the difference between that pulse width and the center position, the faster the shaft will rotate – but that's for another chapter.

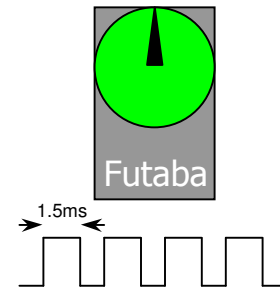
If a stream of pulses with **1.0ms** pulse widths is sent to a modified Futaba S3004 servo, the shaft will spin **clockwise** and will continue spinning clockwise until the pulses stop. Here's why: the modified Futaba servo always thinks its shaft is at the 1.5ms center position, so when it receives a pulse of 1.0ms it rotates the shaft clockwise, as shown in Figure 11.35. When the next 1.0ms-pulse is received, the servo still believes it is at the 1.5ms center position and continues to rotate the shaft clockwise. The servo will **never** reach its desired 1.0ms-position, and its shaft will continuously rotate clockwise as long as the stream of 1.0ms pulses is being sent to the servo.



**Figure 11.35.** A train of **1ms** pulses will rotate a modified servo **clockwise**.



**Figure 11.36.** A train of **2ms** pulses will rotate a modified servo **counterclockwise**.



**Figure 11.37.** A train of **1.5ms** pulses will cause a modified servo to remain stationary in the center.

Now consider a stream of **2.0ms** pulses sent to a modified Futaba servo. In this case, the shaft will spin **counterclockwise** until the pulses cease, as shown in Figure 11.36. So, while an unmodified servo will rotate to a particular position and hold that position, the modified servo is tricked into thinking that the shaft is still at the center position. Roboticians are clever folks, aren't they?

Finally, if a stream of **1.5ms** pulses is sent to a modified Futaba servo (Figure 11.37), the servo will ignore the pulse train because it thinks it is already at the 1.5ms-position.<sup>17</sup> In this case, the servo will **not rotate**.

Follow these general rules when programming a **Futaba S3004 modified servo**:


- Any pulse **less than** approximately **1.5ms** will rotate the shaft **clockwise**.
- A train of pulses whose pulse widths are approximately **1.5ms** will not rotate the servo as the servo always assumes it is at its **center position**, regardless of its actual physical position. (Thus, the 1.5ms pulse train will cause the shaft to remain stationary.)
- Any pulse **greater than** approximately **1.5ms** will rotate the shaft **counterclockwise**.

<sup>17</sup> The 1.5ms center position is an approximation, and could vary by as much as  $\pm 0.25\text{ms}$ .

Obviously, the servos we will use to drive our robot must be *modified* servos, since we need them to continually spin the robot's wheels. So, are you ready to take your Mouse for a test drive?

## 11.10 Let's Do The Hokey-Pokey!

### Put Your LEFT Foot In ...

 It's finally time to write some code and get the Mouse moving. Create a project entitled "MouseMove" and place it in a folder named "Mouse Move". Make sure the left and right Futaba servos are properly connected to the RAMB and then enter the following:

```
Public Sub Main()  
    Call PulseOut(5, 0.0020, 1)    ' Left servo counterclockwise  
End Sub
```



**Double-check that the middle number, (0.0020) was entered correctly. This is very important; otherwise, your servo could be damaged!** Once you are sure the number is correct, compile and download this code to the BX-24 and see what happens. Don't blink – you may miss it! If everything was installed and programmed correctly, the Mouse's **left wheel** should have rotated a few degrees **counterclockwise**. That is, the left wheel of the Mouse should have moved one step **forward**.<sup>18</sup> Nothing to write home about yet, but it's a start.

(If you received the **Compile Error** message shown at the right, check to make sure that you actually did include the right parenthesis in your code. Chances are your right parenthesis is sitting there, as pretty as you please. If so, this probably means you were *lazy* and did not include the **Call** keyword in your code! Put it in now and recompile. The error message should not appear again.)

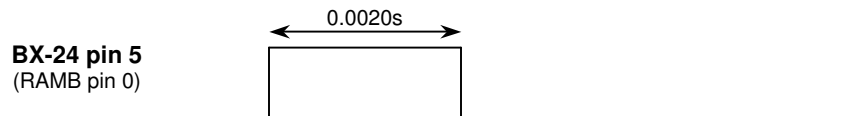


Let's make some sense of the **PulseOut** procedure. The syntax for this procedure is as follows:

```
Call PulseOut (Pin, PulseWidth, State)
```

where *Pin* is the pin number we wish to address, *PulseWidth* is the duration of the electrical pulse to be sent to the *Pin*, and *State* specifies whether the pulse is high or low. The data type for *Pin* is a Byte. *PulseWidth* is a Single data type that can have values that range from about 1.085μs (0.000001085s) to 71.1ms (0.0711s).<sup>19</sup> The *State* argument has the Byte data type and can be either high (1) or low (0).<sup>20</sup>

The line **Call PulseOut(5, 0.0020, 1)** procedure tells the BX-24 to send out one high pulse with a 0.0020s (2.0ms) pulse width to the left servo via pin 5 on the BX-24 (pin 0 on the RAMB). Since we are using modified servos, the 0.0020s pulse rotated our modified Futaba servo counterclockwise, making the left wheel move forward. The pulse generated by our code is illustrated in Figure 11.38.



**Figure 11.38.** A diagram of the electrical pulse generated by **Call PulseOut(5, 0.0020, 1)**. A high pulse, whose duration is 0.0020s (2.0ms), is sent to pin 5 on the BX-24 (pin 0 on the RAMB).

<sup>18</sup> If you are using servos made by a company other than Futaba, your Mouse may move backwards rather than forwards. Keep reading to see how to correct your code.

<sup>19</sup> Just because the BX-24 can accept this wide range of pulse widths, don't assume that your servos can!

<sup>20</sup> A high pulse sends a +5V pulse out of the signal pin, and a low pulse sends 0V pulse.



### ... Put Your LEFT Foot Out ...

Now add the following lines of code to your program:

```
Delay(0.5)                                ' 500ms delay
Call PulseOut(5, 0.0010, 1)               ' Left servo clockwise
```

Run the program and you should observe the Mouse's left wheel move forward one "step", pause for half a second, and then move backwards one "step". The line that reads `Call PulseOut(5, 0.0020, 1)` will make the left servo turn counterclockwise a small amount, and the line `Call PulseOut(5, 0.0010, 1)` will make the left servo turn clockwise approximately by the same amount. The net result is that the Mouse should end up where it started. The only difference in these two lines of code is the *PulseWidth* argument – 0.0020 and 0.0010; the value 0.0020 makes the Futaba servo rotate counterclockwise and 0.0010 causes the servo to rotate clockwise.<sup>21</sup>



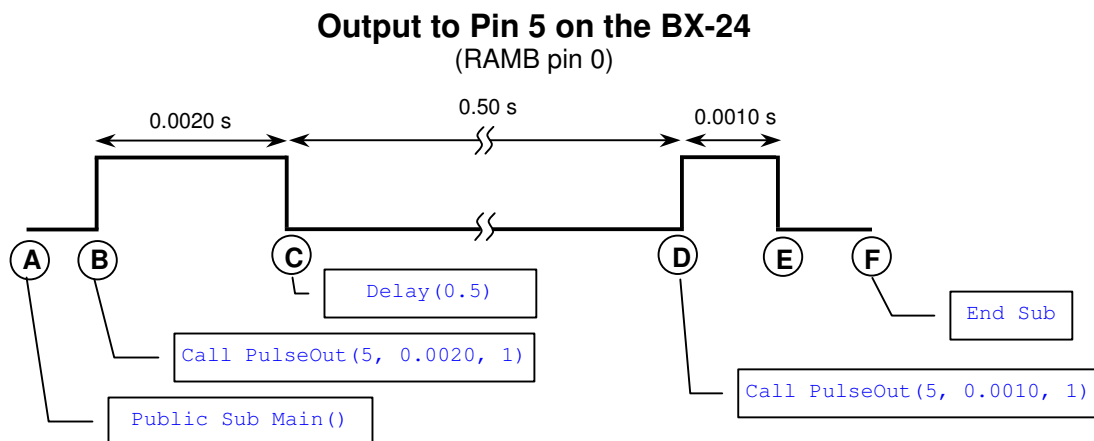
Recall that servos must receive pulse widths only within a well-defined range. Most servos, including the Futaba S3004 that are shipped with the Robodyssey Mouse, have a minimum pulse width-limit around 0.0010s (1.0ms) and a maximum limit around 0.0020s (2.0ms). Sending the servo a pulse width outside this range may damage its control board, so check your code carefully to ensure that the pulse widths you are sending out are within the proper range.



Servo manufacturers suggest placing a small **delay** of at least 0.02s (20ms) between pulses to prevent overdriving the servo! The delay of 0.5s (500ms) in our program is more than sufficient.)

### The Pulse of Our Program

Examine Figure 11.39, which will analyze what your program is doing. The program begins execution at point **A**, with the line `Public Sub Main()`. Very quickly the program reaches point **B** and the line of code that reads `Call PulseOut(5, 0.0020, 1)`, which sends a high pulse to the servo connected to pin 5 on the BX-24. The pulse width is 0.0020s, so the left wheel turns one step counterclockwise at this instant, causing the left wheel to move forward. At point **C**, pin 5 no longer outputs a pulse since the pulse duration has elapsed. The next line of code, `Delay(0.5)`, begins at point **C** and will delay any processing done by the BX-24 for 0.5s (500ms). The final line of code, `Call PulseOut(5, 0.0010, 1)`, sends another high pulse to pin 5 at point **D**. The pulse width here is 0.0010s, so the left wheel rotates one step clockwise, sending the left wheel backward. A brief instant after the termination of the final pulse at point **E**, the program terminates at point **F**, when the line `End Sub` is reached.



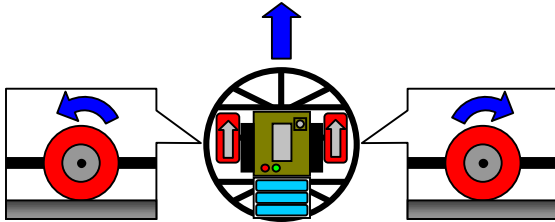
**Figure 11.39.** Two high pulses, whose durations are 2.0ms and 1.0ms, are sent to pin 5 on the BX-24 (pin 0 on the RAMB). A delay of 500ms separates the two pulses. The first pulse turns the Mouse's left wheel counterclockwise (forward) and the second pulse turns the Mouse's left wheel clockwise (reverse).

<sup>21</sup> This may be reversed for other servos.

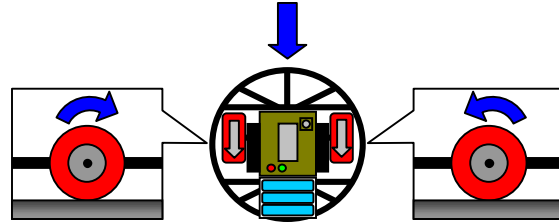
### Put Your RIGHT Foot In, Put Your RIGHT Foot Out...

All of this can be repeated using the **right servo**, allowing both wheels to get into the act. Let's modify our existing code to make **both** the left and right wheels take one step backward and one step forward.

To make the Mouse move **forward**, the *left* wheel must move *counterclockwise*, as we've already seen. The opposite is true for the right wheel: in order for the *right* wheel to move the Mouse forward, it must turn *clockwise* as shown in Figure 11.40. This is flip-flopped when we want the Mouse to move **backwards**: by moving the left wheel clockwise and the right wheel counterclockwise, the Mouse will be propelled backwards as shown in Figure 11.41.



**Figure 11.40.** In order for the Mouse to move forward, the left wheel must rotate counterclockwise while the right wheel rotates clockwise.



**Figure 11.41.** In order for the Mouse to move in reverse, the left wheel rotates clockwise while the right wheel rotates counterclockwise.

We can make this happen with code by sending a clockwise pulse to the right wheel immediately after we make the left wheel rotate counterclockwise. This will move the Mouse one step forward. Then, after the left wheel rotates clockwise, we can turn the right wheel counterclockwise, moving the Mouse backwards.

Our code should now look like this:

```
Public Sub Main()
    ' Mouse moves forwards
    Call PulseOut(5, 0.0020, 1)      ' Left servo counterclockwise
    Call PulseOut(6, 0.0010, 1)      ' Right servo clockwise

    Delay(0.5)                       ' 500ms delay

    ' Mouse moves backwards
    Call PulseOut(5, 0.0010, 1)      ' Left servo clockwise
    Call PulseOut(6, 0.0020, 1)      ' Right servo counterclockwise
End Sub
```

Do you see what's going on? The left wheel needs a pulse of 0.0010s and the right wheel needs a pulse of 0.0020s to make the Mouse move backwards because pulses of 1.0ms and 2.0ms make the Futaba servos turn clockwise and counterclockwise, respectively. You should convince yourself that the Mouse does, in fact, move in reverse when the left wheel rotates clockwise and the right wheel rotates counterclockwise. When thinking about moving and turning the Mouse, always keep in mind:

- ✓ *A pulse width whose duration is less than 1.5ms will rotate Futaba servos clockwise.*
- ✓ *A pulse width whose duration is greater than 1.5ms will rotate Futaba servos counterclockwise.*

### Add Some **CONST**ants

It can get confusing having to remember the proper pin numbers and pulse widths when mobilizing your robot, but we can make robot programming much easier for ourselves if we incorporate some constants in our program. At the beginning of the **Main** program, create a constant to represent the BX-24 pin number that controls the left servo. Name the constant **LeftServo** and assign it the value 5, since we connected our left servo to BX-24 pin 5 (RAMB pin 0). Recall that the data type for the *Pin* argument of the **PulseOut** procedure is a Byte, so we must define our constant accordingly.<sup>22</sup>

<sup>22</sup> Revisit *Chapter 5: Variables, Constants, and Data Types* to refresh your memory about using **Const**.

However, why stop here when we can create another constant that will further simplify our program? We should create a constant named `Left_Forward` and give it a value of 0.0020. We can then replace the line that reads `Call PulseOut(5, 0.0020, 1)` with `Call PulseOut(LeftServo, Left_Forward, 1)`, which does exactly the same thing. (Using these constants makes programming the Mouse almost too easy!) To me, it is certainly easier to read and understand the line `PulseOut(LeftServo, Left_Forward, 1)` than `PulseOut(5, 0.0020, 1)`.



To make it complete, create and define four more constants that can be used to propel the Mouse forwards and backwards. Specifically, name them `LeftServo`, `Left_Reverse`, `Right_Forward`, and `Right_Reverse`, and set them to the appropriate values. Be certain that the values for the direction constants are between the minimum servo pulse width value of 0.0010s and the maximum value of 0.0020! Otherwise, you could damage your servos.

The following code shows how I defined and used the new constants. Running it will produce the same robot motion as before, but now the code is more versatile and much easier to read.

```
Public Sub Main()
    ' Servo Pin Constants
    Const LeftServo as Byte = 5           ' Pin #0 on RAMB
    Const RightServo as Byte = 6          ' Pin #1 on RAMB

    ' Servo Direction Constants (Pulse widths)
    Const Left_Forward as Single = 0.0020 ' Counterclockwise rotation
    Const Left_Reverse as Single = 0.0010 ' Clockwise rotation
    Const Right_Forward as Single = 0.0010 ' Clockwise rotation
    Const Right_Reverse as Single = 0.0020 ' Counterclockwise rotation

    ' Mouse moves forwards
    Call PulseOut(LeftServo, Left_Forward, 1)
    Call PulseOut(RightServo, Right_Forward, 1)

    Call Delay(0.5)                       ' 500ms delay

    ' Mouse move backwards
    Call PulseOut(LeftServo, Left_Reverse, 1)
    Call PulseOut(RightServo, Right_Reverse, 1)
End Sub
```

## 11.11 Do the Cha-Cha!

Now that the Mouse can take baby steps, let's spice it up a bit and do the Cha-Cha! That is, let's make the Mouse take ten steps forward and five steps back. I know, I know. This isn't the way the Cha-Cha is *traditionally* done, but then again mice don't *traditionally* dance the Cha-Cha!

We *could* copy and paste the lines that make the Mouse take all of these steps, but why bother when we have For-To-Next loops at our disposal? Add a **Cha-Cha** section to our existing program, but first separate your existing code from this new stuff below with a two-second delay.

```
' ***** Do the Cha-Cha *****
Call Delay(2.0)                       ' Separation delay

Dim i as Integer
Const NumF as Integer = 10
Const NumB as Integer = 5
```



```

' Move forward
For i = 1 to NumF
    Call PulseOut(LeftServo, Left_Forward, 1)
    Call PulseOut(RightServo, Right_Forward, 1)
    Call Delay (0.25)
Next

' Move backward
For i = 1 to NumB
    Call PulseOut(LeftServo, Left_Reverse, 1)
    Call PulseOut(RightServo, Right_Reverse, 1)
    Call Delay (0.25)
Next

```

I created the constants, `NumF` and `NumB`, to represent the number of forward and backward steps. Both were incorporated in For-To-Next loops with a counter, `i`. The creation of these constants was perhaps unnecessary. We could have run our loops with numeric literals 10 and 5. However, programming with constants makes for efficient code and it is a good habit to get into. Finally, the four `PulseOut` commands and quarter-second delays leave us with one swingin' Mouse.

If you would like to see it “dance” again, simply press the reset button on the RAMB! As an exercise at the end of this chapter, modify the code to make the Mouse repeat this dance any number of times. Create your own dance steps by varying the values of `NumF`, `NumB`, and the delay. Experiment and have some fun.

### Do the Cha-Cha Faster!

Rodents are supposed to be fast, so let's speed up the Mouse's dance by reducing the delay time between each step. Try replacing `Delay (0.25)` with `Delay (0.1)`. Run the program and you'll see that the dance is now faster, but still quite choppy.



Let's make the dance go *as fast as possible* and see what happens. The manufacturer of our servos recommends a minimum delay time of 0.020s (20ms) between consecutive pulses to any one servo. This gives the servo ample time to fully rotate. Therefore, using `Delay(0.02)` is the smallest delay the servomotors can handle, pulsing the wheels about 50 times per second.

### Speed Is a Virtue

As explained above, the Mouse's speed can be controlled by varying the delay times between servo pulses. We can alter the speed of the robot more readily with a `Speed` variable and a few constants. It makes sense to define three speeds for our Cha-Cha application: `Fast`, `Medium`, and `Slow`. Let's arbitrarily pick delay times of 0.02s, 0.10s, and 0.50s for `Fast`, `Medium`, and `Slow`, respectively. Add these lines to the beginning of your application, below the program's other constants:

```

' The Mouse's speed is controlled by these delay times (in seconds):
Const Fast as Single = 0.02           ' 50 steps per second (fastest)
Const Medium as Single = 0.10         ' 10 steps per second
Const Slow as Single = 0.50           ' 2 steps per second
Dim Speed as Single                   ' Use to adjust delay interval

```

If we wish to have the Mouse move at the fastest possible speed, all we have to do is define the `Speed` variable as `Fast` as shown here:

```
Speed = Fast
```

You can put this line anywhere in the program – just make sure that it comes before you need to use it. To use the `Speed` variable, simply change all the delay calls in your program as follows:

```
Call Delay(Speed)           ' Controls speed
```

Do you see how `Speed` is used within the `Delay` procedure? Nifty, huh? Now it is easy to alter the Mouse's speed by simply altering the `Speed` variable! Of course, you could incorporate the desired speed value directly into the `Delay` call:

```
Call Delay(Fast)           ' Short delay = fast speed
```

Feel free to alter the values of the constants, but remember that the smallest possible delay (i.e., the Mouse's fastest speed) is 0.02s (20ms)! In other words, `Fast` is as fast as is safe.

The following Self Test shows how you can alter your program to make the Mouse move eight steps forward at `Fast` speed and eight steps backward at `Slow` speed. Try doing this yourself before looking at my solution.

#### Self Test # 1. Spring Ahead, Fall Back

**Problem:** Alter the Cha-Cha part of your current project so that the Mouse will move eight steps forward at `Fast` speed and eight steps backward at `Slow` speed. Take note of the distances traveled in each leg of the journey

**Solution:** This is an almost trivial solution, but you will need to carefully alter the necessary constant and variable values. (I've indicated these in bold.)

```
Const NumF as Integer = 8
Const NumB as Integer = 8

Speed = Fast
' Move forward
For i = 1 to NumF
    Call PulseOut(LeftServo, Left_Forward, 1)
    Call PulseOut(RightServo, Right_Forward, 1)
    Call Delay (Speed)
Next

Speed = Slow
' Move backward
For i = 1 to NumB
    Call PulseOut(LeftServo, Left_Reverse, 1)
    Call PulseOut(RightServo, Right_Reverse, 1)
    Call Delay (Speed)
Next
```

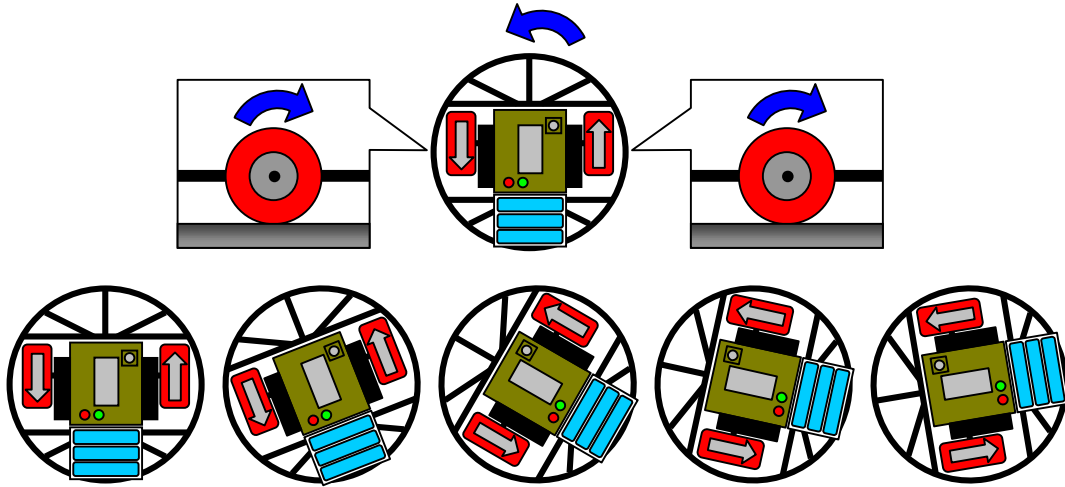
Compare the distance traveled by the Mouse in the forward and backward directions. Running the program, I find that my Mouse moves forward about one inch and backwards about three. This difference is due to the `Speed` value (that is, the delay time between pulses). At `Fast` speed, new `PulseOut` commands are received by the servo while it is still carrying out the previous command and complete rotations are not possible.

## 11.12 To Everything Turn, Turn, Turn

Going forwards and backwards is great, but it wouldn't be much of a robot if we couldn't make it turn. There are many ways to do this, and in this section we will use `For-To-Next` loops to make the Mouse turn in circles. To do so is actually quite easy; perhaps you've already figured it out. Here are a few ways that it can be done.

### Turning on a Dime – Literally

To make the **quickest and tightest turn** possible, we must simultaneously *counter-rotate* each of the Mouse's wheels. This has the effect of pivoting the Mouse to the left about its center – literally turning it on a dime. Therefore, to make the Mouse **rotate to the left**, we must reverse the left wheel while the right wheel is rotating forward. In other words, for sharp left turns, both wheels must turn clockwise, as shown in Figure 11.42. To make the Mouse **rotate to the right**, reverse the right wheel and drive the left wheel forward; for sharp right turns, both wheels must rotate counterclockwise.



**Figure 11.42.** To make the Mouse turn left, as if pivoting about its center, rotate both wheels clockwise. That is, turn the left wheel backwards and the right wheel forwards.

To make the Mouse spin ten “steps” to the left, try this bit of code. (I added another two-second delay in my program to separate this new code from the old code. What can I say – I’m a creature of habit.)

```
' ***** Center Rotation *****
Call Delay(2.0)                               ' Separation delay
Speed = Slow                                  ' Choose a speed

' Turn ten steps to the left
For i = 1 to 10
    Call PulseOut(LeftServo, Left_Reverse, 1)
    Call PulseOut(RightServo, Right_Forward, 1)
    Delay (Speed)
Next
```

I arbitrarily chose to use the `Slow` speed for this exercise – the delay time is 0.50s (500ms). After 10 pulses, how far did your Mouse rotate? Mine rotates approximately 90° with fresh batteries and 70° with run-down batteries. (Changing the `Speed` variable will affect how far ten pulses will rotate the robot – the greater the speed, the more pulses are required.)

To turn to the right, simply reverse the process by rotating the left wheel forward and the right wheel backward:

```
' Turn ten steps to the right
For i = 1 to 10
    Call PulseOut(LeftServo, Left_Forward, 1)
    Call PulseOut(RightServo, Right_Reverse, 1)
    Delay (Speed)
Next
```



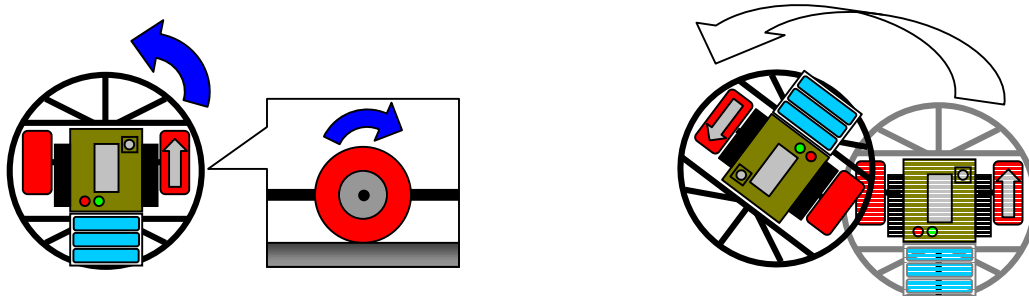
### The Wheels on the Mouse Go ‘Round and ‘Round ...

One question worth asking is, “How many pulses does it take to turn my Mouse a total of 360°?” I wish I could say that all servos behave identically, but I can’t. Even if they did, my Mouse would move in a different way from yours due to differences in battery power, smoothness of the tail wheel, and friction forces within the servo and between the wheels and tabletop. Even our delay times play an important role. For example, *my* Mouse, with fully charged batteries on a smooth surface, is capable of pivoting 360° about its center in 22 steps with a delay time of 0.5s (*Speed = Slow*) and in 57 steps with a delay time of 0.02s (*Speed = Fast*).

Take the time to work Challenge Problem #1 at the end of this chapter and fill in the worksheet documenting the number of pulses required to rotate your Mouse 360°. It will come in handy as a reference as you complete this book or when you decide to compete in a local robot competition!

### Sidewinder Turning

You need not always turn your Mouse by rotating it about its center – you can pivot it about one of its wheels. To do this, simply keep one wheel stationary and make the other rotate. For instance, to make a **wide turn** to the left, rotate the right wheel clockwise and leave the left wheel alone, as shown in Figure 11.43.



**Figure 11.43.** Wide turns can be made by rotating one wheel while keeping the other stationary. Here the Mouse is pivoting about its left wheel by rotating the right servo clockwise.

This type of turning action reminds me of the motion of the sidewinder snake. To see this sidewinder-turning in action, perform the following Self Test, which will make the Mouse take three large strides.

#### Self Test # 2. The Side-Winding Mouse

**Problem:** Make the Mouse move left-to-right by alternately rotating the left and right servos forward. If done properly, this motion will resemble that of a sidewinder snake. Make three of these large lumbering steps as quickly as possible.

**Solution:** The Mouse’s first step is a half-circle turn pivoting 180° about the right wheel. When the first turn is completed, rotate the Mouse 180° about the left wheel. Repeat this process of rotating about the right wheel and then about the left two more times.

To make the Mouse turn as quickly as possible, use the *Fast* speed (with a delay of 0.02s between pulses to the servo). At this rate, I know that *my* Mouse requires 70 pulses to complete one 180° “step”. Furthermore, I know that the Mouse is to make six such steps, or three pairs of right-left strides. To do this most efficiently, we should use *nested For-To-Next loops*, using one counter to keep track of the three of big strides and another counter to count the individual pulses required to make each step.

Check it out!

*Solution continued on the next page...*

**Self Test # 2. The Side-Winding Mouse** *(Continued from previous page)***Solution:** *(Continued)*

```

' ***** Side-Winder Turning *****
Call Delay(2.0)                                ' Separation delay

Dim StrideCounter as Integer                    ' Our stride counter
Const NumOfStrides as Integer = 3              ' Easy to change!
Const PulsesPerStep as Integer = 70            ' 70 pulses = 180 deg.

Speed = Fast

For StrideCounter = 1 to NumOfStrides
    ' Pivot about the RIGHT wheel:
    For i = 1 to PulsesPerStep
        Call PulseOut(LeftServo, Left_Forward, 1)
        Call Delay(Speed)
    Next

    ' Pivot about the LEFT wheel:
    For i = 1 to PulsesPerStep
        Call PulseOut(RightServo, Right_Forward, 1)
        Call Delay(Speed)
    Next
Next
Next

```

To run this program you will need about 2.5 feet of desk or floor space, so clear away an area before running the code!

Both center-pivot and sidewinder turning have advantages and disadvantages. Because it requires fewer pulses, center-pivot turning can be done **more rapidly**. On the other hand, sidewinder turning is **more precise** since the Mouse will turn fewer degrees per pulse. Of course, center-pivot turning requires virtually no additional room to turn around, while sidewinder turning needs a considerable amount of space.

### 11.13 It Keeps Going, and Going, and Going, ...

There is no reason that our Mouse ever has to stop moving (unless the batteries run down or the robot collides with an immovable object!). Therefore, we will finish this chapter with a rather simple program that will allow the Mouse to move forward indefinitely.

Why don't **you** try to figure out how to do it, and then examine the following Self Test to see how I did it?

**Self Test # 3. Forward – Ho!**

**Problem:** Make the Mouse move forward indefinitely at Medium speed.

**Solution:** Using a For-To-Next loop is fine if you wish to make a pre-determined number of steps like we did with the Cha-Cha, but now we want the Mouse to run indefinitely. How should we **do** this? With a **Do-Loop**, of course!

*Solution continued on the next page...*

**Self Test # 3. Forward – Ho!** (Continued from previous page)**Solution:** (Continued)

```

' ***** Forever Forward! *****
Call Delay(2.0)                ' Separation delay
Speed = Medium

Do
  Call PulseOut(LeftServo, Left_Forward, 1)
  Call PulseOut(RightServo, Right_Forward, 1)
  Call Delay (Speed)
Loop

```

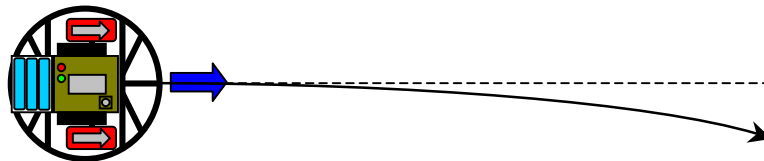
Since we've programmed the Mouse to never stop, be prepared to **manually stop** the program's execution with the Stop button on the Downloader or with the power switch on the RAMB. Otherwise, your Mouse may run off the table and crash to the floor! Challenge Problem #4 at the end of this chapter asks you to modify this program so the LEDs blink as the Mouse moves. How does this affect the Mouse's speed?

**11.14 Troubleshooting and Final Comments****Wheel Wobbles**

Run the code from Self Test #3, but alter it so that it runs at **Fast** speed. Carefully examine the Mouse's wheels as they rotate. If one wheel wobbles, make sure that the axle screw, which attaches the wheel to the servo shaft, is tight – but do **not** over tighten! If the wobble continues, remove the wheel, rotate it a few degrees and reattach it to the shaft. Repeat this procedure until the wobble stops or is lessened.

**Straight Shooter**

Program your Mouse to move forward; take note of its motion. The Mouse **should** move in a perfectly straight line, but some robots will veer in a slight arc as shown in Figure 11.44.



**Figure 11.44.** Sometimes a robot will veer from the desired straight-line path and travel in a slight arc.

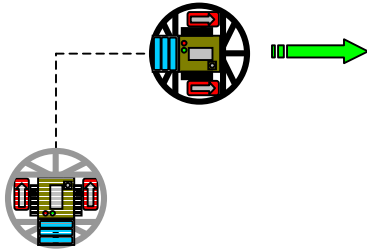
If you experience this problem, there are several things to check that may correct the problem:

- ☐ If you are using the rear tail wheel, replace it with the phenolic ball.
- ☐ Make sure that you are sending the correct pulse widths to your servos – the **minimum** pulse width should be 0.0010s and the **maximum** should be 0.0020s.
- ☐ Run the Mouse at **Fast** speed.
- ☐ Place the Mouse on different surfaces – sometimes the wheels will slip on hard or slick surfaces.
- ☐ Recharge the batteries. All robots behave more predictably when they operate at full power.
- ☐ Make sure your servos are firmly attached to the Mouse chassis.
- ☐ Try calling the **PulseOut** commands in reverse order. If you are pulsing out the right servo first, try pulsing to the left one first.

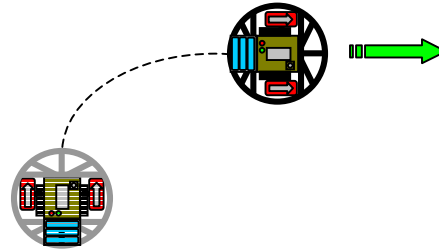
If these suggestions don't make your Mouse move in a straight line, don't fret; I will show you how to tweak the robot's movement in *Chapter 13: Advanced Servo Operations*.

### Smoothing Out the Curves

My students often want their Mouse to turn in smooth arcs rather than with sharp 90° turns, as shown in the figures below. I tell them this is certainly possible to do, and in fact, it is covered in *Chapter 13: Advanced Servo Operations*. I also point out that all robot programmers start off turning their robots with sharp turns.<sup>23</sup>



**Figure 11.45.** It is easy to make a robot turn in sharp 90° angles.



**Figure 11.46.** Soon we will learn how to make the robot turn in smooth arcs.

Personally, I think that it is important for novice programmers to practice the rudimentary basics before making their robot behave more gracefully. With that in mind, try the following Challenge Problems, and put to use the skills you learned in the previous chapters. I strongly suggest that you pick a few problems that interest you and sink your teeth into them.

Happy problem solving!

## Challenge Problems



**Remember: always keep your pulse widths between the minimum and maximum allowed values!** That is, between **0.0010s** and **0.0020s**. Use the constants, `Left_Forward`, `Right_Forward`, etc, to ensure you don't enter a wrong number. Also, **always use a delay of at least 0.02s between successive pulses.**

- 11-1.** Determine how many pulses it takes to turn your Mouse 360° about its center, and fill in the worksheet below. Vary the delay time between pulses and the direction of rotation. You should simulate actual conditions and remove the serial cable before performing the test.

### Number of Pulses to Rotate the Mouse 360°

**Pulse Widths:** Maximum:  Minimum:

#### Turning Left

#### Turning Right

Delay time:	<input type="text" value="0.02s"/>	# of Pulses:	<input type="text"/>	# of Pulses:	<input type="text"/>
Delay time:	<input type="text" value="0.1s"/>	# of Pulses:	<input type="text"/>	# of Pulses:	<input type="text"/>
Delay time:	<input type="text" value="0.5s"/>	# of Pulses:	<input type="text"/>	# of Pulses:	<input type="text"/>
Delay time:	<input type="text"/>	# of Pulses:	<input type="text"/>	# of Pulses:	<input type="text"/>
Delay time:	<input type="text"/>	# of Pulses:	<input type="text"/>	# of Pulses:	<input type="text"/>

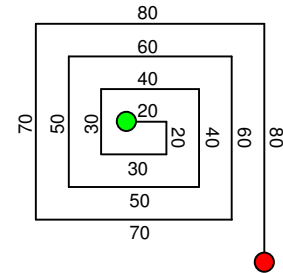
<sup>23</sup> To make the turns more consistent, you may wish to put a 0.25s or so delay between any forward movements and your turns. This small delay will give the Mouse time to coast to a stop before turning.

- 
- A diagram of a step function. The function is represented by a black line that starts at a green dot at the bottom left, moves horizontally to the right, then vertically up, then horizontally to the right, then vertically down, and finally horizontally to the right. A red dot is placed at the end of the final horizontal segment.



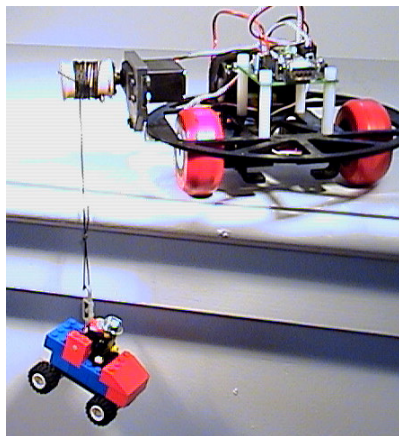
**11-19.** Have the Mouse navigate a maze. (You will need to use a ruler to measure the paths and preprogram the solution into the BX-24; we are not asking it to solve the maze autonomously – yet!) Have a competition with other robots and use a stopwatch to keep score: one point is added for each second the Mouse is in the maze. Points are added when the robot touches a wall. The robot with the fewest number of points wins.

**11-20.** Have the Mouse trace-out a spiraling square path. First, have the Mouse trace-out two sides of the square 20 “steps” long. Then have it trace two more sides, but this time have it move 30 steps per side, and so on, until it traces two 80-step sides as shown to the right. Do not use the **brute force** method here. Rather, nest a few For-To-Next loops inside a Do-Loop to solve this! Set the speed to **Fast** when the Mouse is moving forward, and **Slow** when turning. Create a variable named **Side** and initialize it to 20. After two sides have been traced out, increase **Side** by 10 steps. Continue until **Side** > 80.

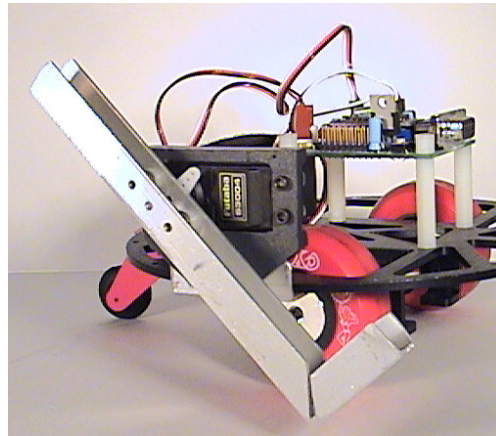


**11-21.** Attach a writing implement (such as a marker) to the Mouse. Then place the robot on a large piece of paper and have it draw designs, shapes, and even words. Get other robots involved and create a masterpiece! A large, portable white board works great for this task.

**11-22.** Create a robotic winch by mounting a small dowel or empty thread spool to a modified servo as shown in Figure 11.47. Attach one end of a light string (nylon fly fishing line works great) to the spool and the other to a magnet or hook. Use the winch to raise and lower objects between the tabletop and the floor.



**Figure 11.47.** A homemade winch made from an old wooden spool and string.



**Figure 11.48.** A homemade arm made from a piece of scrap aluminum U-channel.

**11-23.** Mount an **unmodified** servo with an attached arm to the Mouse chassis, as shown in Figure 11.48. Program the arm to either knock objects out of the way or flip them over. Be careful not to place too much weight on the end of the arm since the servos can produce only so much torque. (See *Appendix E: Servo Comparisons and Physical Limitations* for servo torque limits.)



Before doing any work with unmodified servos, be sure to study *Appendix E* and *Appendix G* for important precautions and helpful hints!