

Robodyssey Mouse

Assembly Instructions

Version 1.03

Robodyssey Systems, LLC.
20 Quimby Avenue
Trenton, New Jersey 08610

Phone/Fax: 609-585-8535
Web: www.robodyssey.com
Email: info@robodyssey.com

Copyright 2005 Robodyssey Systems, LLC. All rights reserved.
Distributed by Robodyssey Systems, LLC, Trenton, New Jersey, www.robodyssey.com
Basic Express™ and BasicX™ are trademarks of NetMedia, Inc.
Basic Stamp™ is a trademark of Parallax, Inc.
BasicX™ and the BX-24™ are registered trademarks of NetMedia, Inc
Microsoft™ is a registered trademark in the United States
Microsoft™ Excel™ is a registered trademark of Microsoft Corporation
Microsoft™ Visual Basic™ is a registered trademark of Microsoft Corporation
Windows™ is a registered trademark of the Microsoft Corporation
Futaba™ is a registered trademark of Futaba Electronics
All trademarked names referenced herein are the property of their respective holders.

Index

Topic	Page
Introduction	4
Kit Contents	5
Assembly Instructions	6
Elevated Motherboard Option	9
Tail Wheel Option	10
 Basic Mouse	
Mouse Basic Option	12
Kit Contents	12
 Robodyssey Motherboard	
Introduction	17
Connecting it all	18
 BasicX Software	
Starting BasicX	20
“Hello Word”	22
Sensing Your World	24
Moving Through the World	31
Putting it all Together	35
Arguments	36
 Robodyssey Mouse Programming Guide	
Constants	40
Forward Movement	41
Turns and Reverse	41
Analog Sensors	43
Putting it all together	43
Fun Extras	45
 Appendix A	
BasicX Language	48

Introduction

The Robodyssey Mouse is a robust and versatile wheeled robotics platform developed for educators, hobbyists, and professionals. Originally developed for the classroom, the all-aluminum construction provides a lightweight yet durable robot that will last for years to come.

Just a sampling of the Mouse's features include:



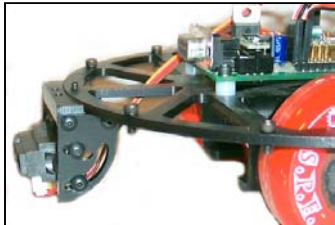
Anodized Laser Cut Aluminum Chassis

The Mouse body is made from anodized aluminum, which provides the best combination of durability, weight, and appearance. No rusting, corroding, or broken parts to deal with.



Futaba S3004 Ball Bearing Servos

A tough robot requires a set of tough servos, and these high-quality Futaba servos certainly fit the bill. The servos operate on power packs from 6V to 9V, and provide plenty of power to get the Mouse moving.



Numerous Tapped Mounts

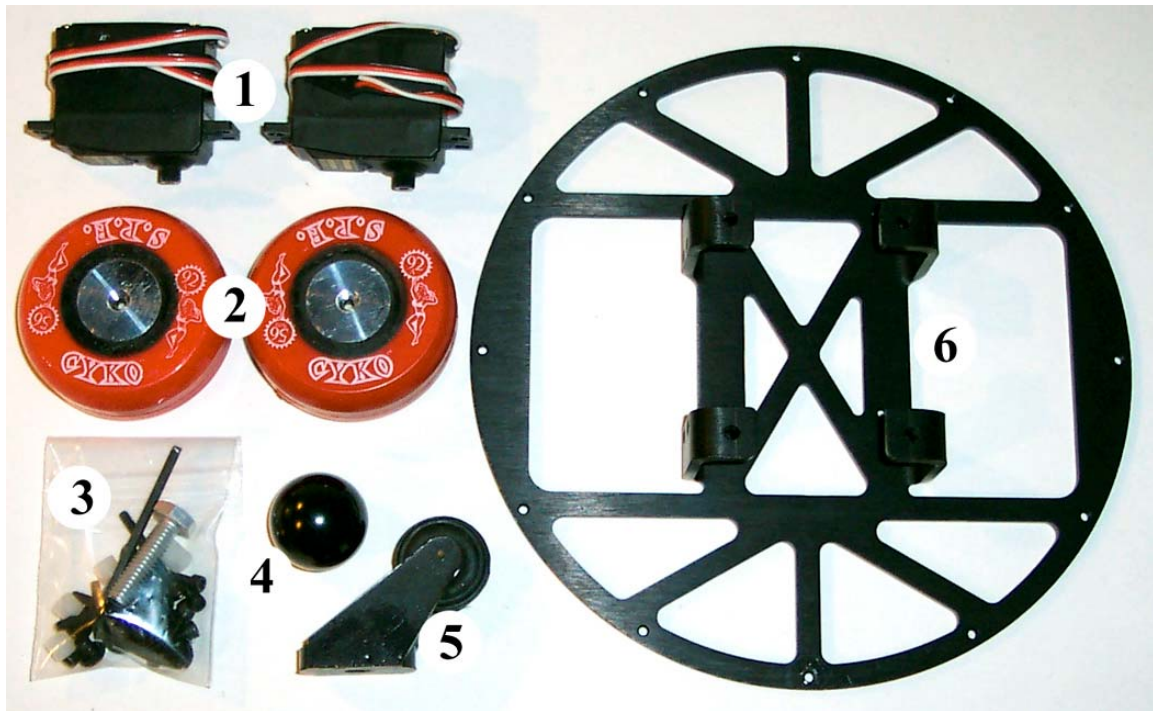
Need to mount some extra sensors? The Mouse body has numerous tapped mounting positions for your components. Servos, motherboard and sensors are securely fastened to the body, and won't come loose.



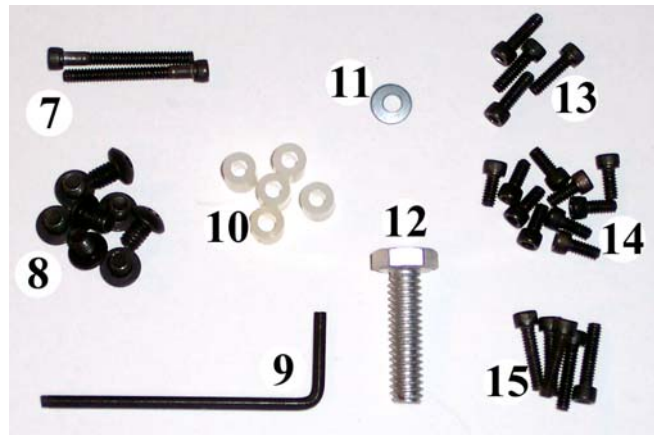
Polyurethane Skate Wheels

Wheels designed to handle the most aggressive of skateboarders are perfect for the Robodyssey Mouse. Never worry about breaking a wheel, or slipping on smooth surfaces.

Mouse Kit Contents



- | | |
|--------------------|-----------------|
| 1 Servos (2) | 4 Tail Ball |
| 2 Skate Wheels (2) | 5 Tail Wheel |
| 3 Small Parts Bag | 6 Mouse Chassis |



- | | |
|----------------------|----------------------|
| 7 Axle Screws (2) | 11 Tail Wheel Washer |
| 8 Servo Screws (8) | 12 Tail Ball Shaft |
| 9 Allen Wrench | 13 Medium Screws (4) |
| 10 Nylon Spacers (5) | 14 Small Screws (10) |
| 15 Long Screws (5) | |

Assembly Instructions

Assembly of the Mouse kit will take you approximately 35 minutes. Be sure to look up the proper part for each assembly step, indicated by the number next to the part name which corresponds to the pictures in “Mouse Kit Contents.”

You will only need the Small Allen Wrench provided in your kit to assemble your Mouse.

Servo Attachment



Figure 1.

Attach the Servos (1) to the Chassis (6) using four Servo Screws (8) to attach each servo. Use the Small Allen Wrench (9) that came with your Mouse kit. In Figure 1, pay careful attention to the positioning of each servo. Notice how A, the servo gear, is positioned closely to B, a tapped hole in the chassis. This positioning provides the best balance for the Mouse, and you will see why later on.



Figure 2.

Attach both servos in the same manner, oriented as shown in Figure 2 (both servo gears in the same direction).

Wheel Attachment



Use the Axle Screws (7) to attach the wheels to the servos, as shown in Figure 3. Do not over tighten the Axle Screws!

Figure 3.



With both wheels attached, your Mouse will resemble Figure 4.

Figure 4.

Position a Long Screw (15) in the threaded hole located in the center back of the chassis. Screw the Long Screw (15) through the top of the mouse chassis all the way into the body.



Figure 5.

Screw the aluminum Tail Ball Shaft (12) onto the Long Screw on the back of the Mouse Chassis.



Figure 6.

Firmly screw the Tail Ball (4) over the Tail Ball Shaft (12) as shown in Figure 7.



Figure 7.

The completed Bare Bones Mouse!



Figure 8.

Elevated Motherboard Option

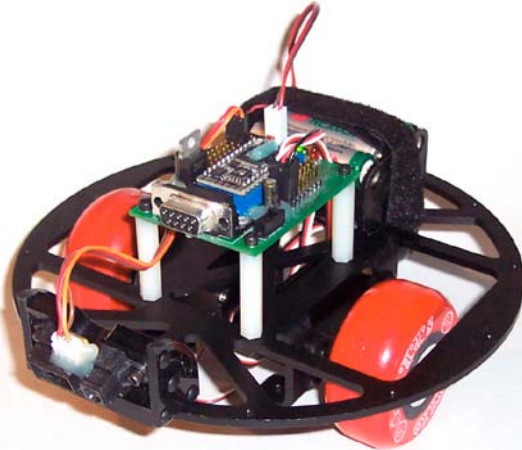


Figure 9.

Use the 1 ¼ inch nylon standoffs to elevate the Motherboard from the chassis. This format allows for increased area of vision for the Sharp Sensor. Elevated platforms may also make it easier to connect the programming cable depending on sensor configuration.

Tail Wheel Option

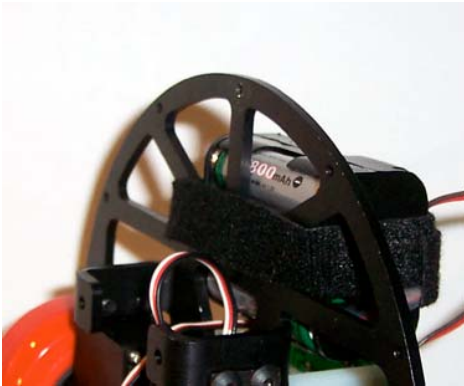


Figure 10.

Remove the Tail Ball and Shaft as shown in Figure 10.



Figure 11.

Insert the Tail Wheel Washer (11) and a Nylon Spacer (10) over a Long Screw (15) as shown in Figure 11.

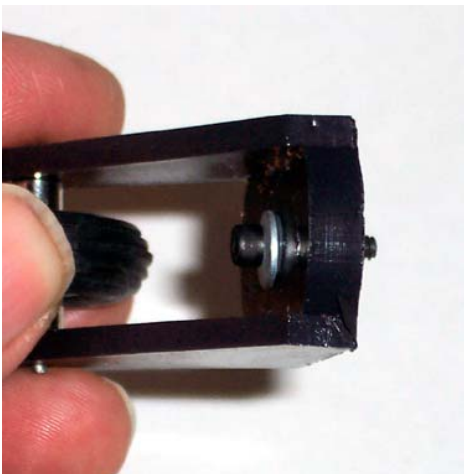


Figure 12.

Insert the assembly shown in Figure 11 into the Tail Wheel (5) as shown in Figure 12.



Using the Allen Wrench (9), tighten the Tail Wheel assembly onto the bottom of the Mouse chassis.

Figure 13

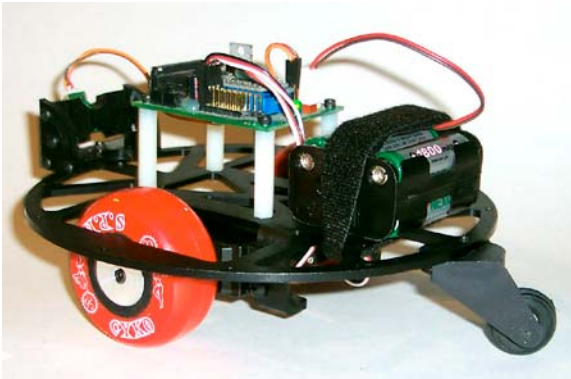


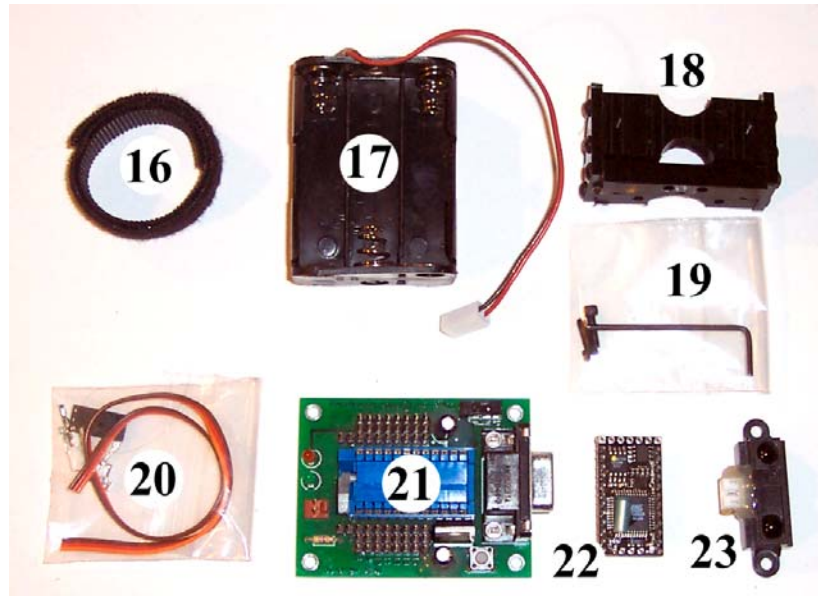
Figure 14 shows a completed Mouse with Tail Wheel and standoffs.

Figure 14

Mouse Basic Option

The following components may have come with your kit, depending on your order configuration.

Mouse Basic Kit Contents



16 Velcro Strap

17 Battery Compartment

18 Adjustable Bracket

19 Wrench and Sensor Screws

20 Sensor Cable Kit

21 Robodyssey Motherboard

22 BX-24

23 Sharp Sensor

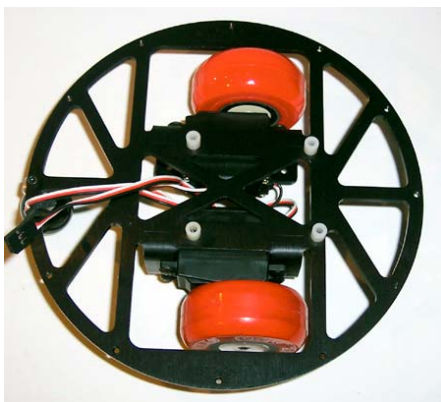


Figure 15

Place the four Motherboard Spacers (10) over the tapped holes in the center of the Mouse chassis as shown in Figure 15.



Figure 16.

Using four Medium Screws (13), attach the Robodyssey Advanced Motherboard as shown in Figure 16.

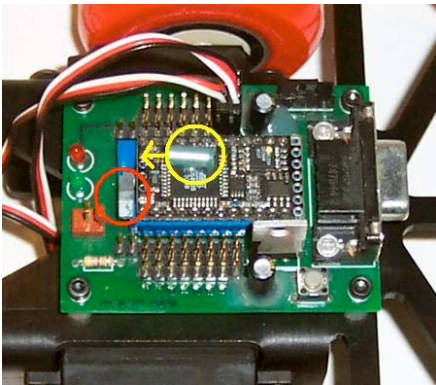


Figure 17.

Release the latch (circled in red) to place the BX-24(22) into the socket on the Motherboard (21).

Plug the wire connected to the Left Servo into the pins labeled 0. Plug the wire connected to the Right Servo into the pins labeled 1. You may choose other pins to connect the servos too, and simply change the software. Our examples will assume the servos are connected to pins 0 and 1.

It is imperative that the lightest colored wire is closest to the BX-24 (22) as shown in Figure 17.

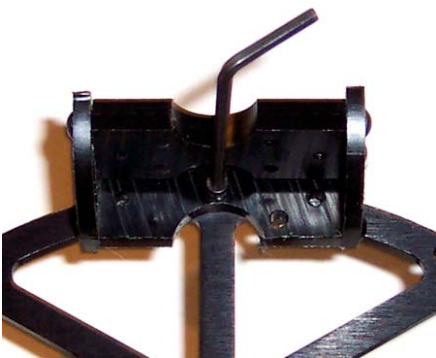


Figure 18.

Using a Short Screw (14) connect the Adjustable Bracket (18) to the Mouse Chassis (6) as Shown in Figure 18.

Using two medium screws, attach the Sharp Sensor to the front of the Adjustable Bracket (18) as shown.



Figure 19.

Connect the assembled Sensor Cable (20) to the Sharp Sensor (23) as shown. Proceed to Figure 21 for a close up of the wire connections.

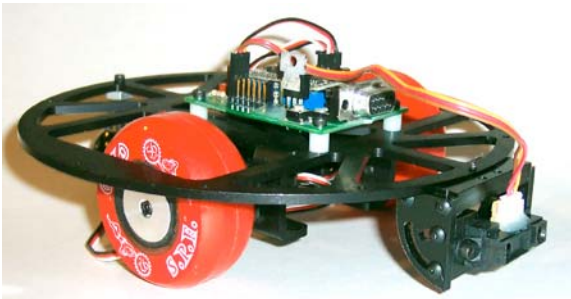


Figure 20.

This close-up shown the preferred arrangement of connections for the Basic Mouse. The pins for each function are selected in software and can be placed in a number of different locations. Our examples in our manuals and well as our web site will have the left servo connected to pin set 0, the right servo connected to pin set 1, and the Sharp Sensor connected to pin set 8.

The light colored wire on both sensors and servos should always be on the innermost pin.

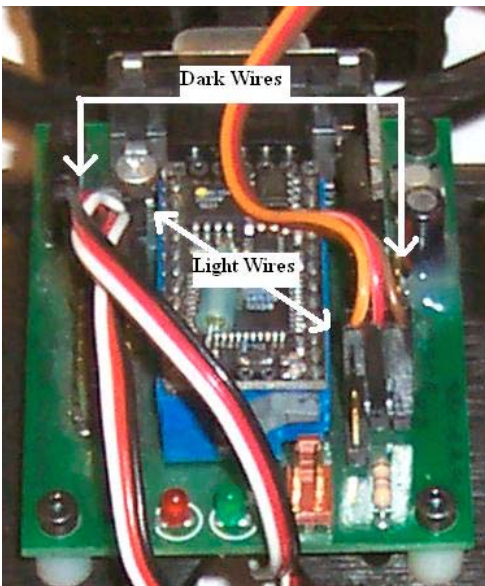
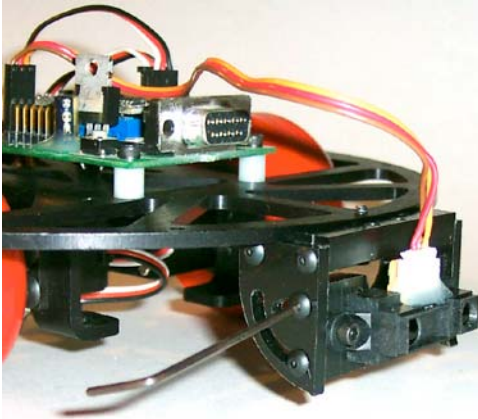
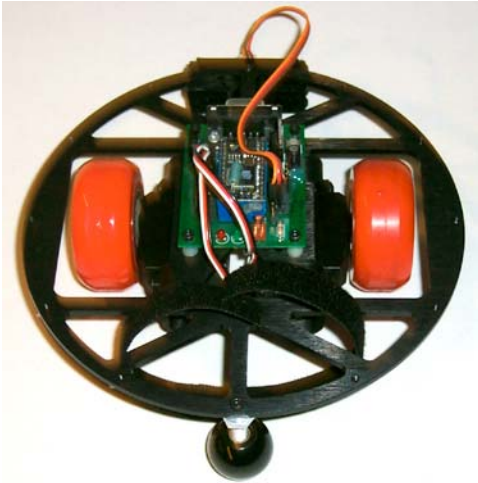


Figure 21.



Adjustment of the Sensor angle is achieved by slacking the socket head screws on the side of the Adjustable Bracket(18).

Figure 22.



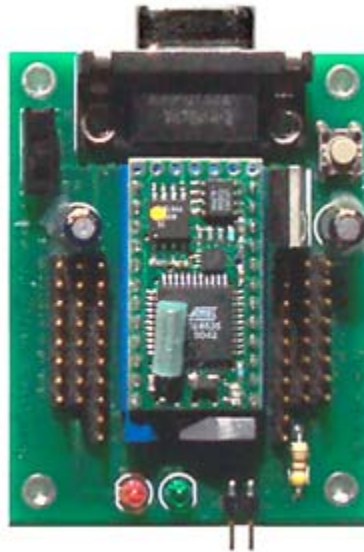
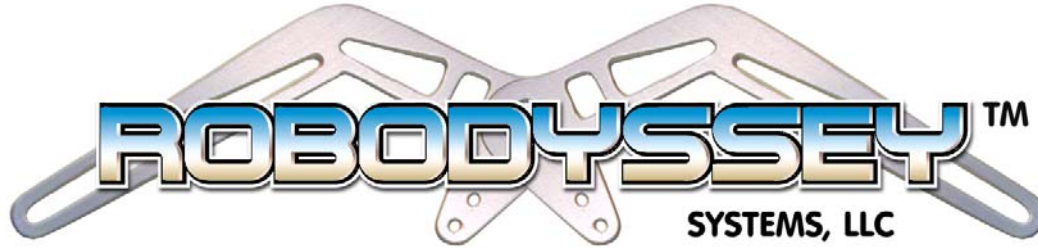
Weave the Velcro strap through the Mouse chassis as Shown in Figure 23.

Figure 23.



The Completed Basic Mouse .

Figure 24.



Robodyyssey Motherboard

Programming guide for the BX-24 microprocessor

Version 1.1

Robodyyssey Systems, LLC.
20 Quimby Avenue
Trenton, New Jersey 08610

Phone/Fax: 609-585-8535
Web: www.robodyssey.com
Email: info@robodyssey.com

Introduction:

This manual will give you step by step instructions on how to use the BasicX programming language to design and implement novel robotic applications. BasicX is subset-compatible with Microsoft's popular Visual Basic language, but you do not need Visual Basic to use BasicX. One of the nice features of BasicX is its ability to use the same code to run both on your PC as well as your robot. There are some differences obviously, but you can develop and debug your algorithms in Visual Basic and then use them in your own robotic creations.

The Robodyssey Advanced Motherboard is a carrier board for the NetMedia BasicX and Parallax BasicStamp series of 24 pin microcontrollers. The Motherboard's features include:

-24 Pin Zero Insertion Force Socket

Preserve the delicate connector pins on your microcontroller with a ZIF socket.

-Convenient Signal & Power Pins

Each I/O Pin has adjacent positive and negative terminals for convenient 3 pin connections. Servo connectors fit perfectly.

-5V Regulated Power

Your microcontroller and half of the I/O power pins are connected to a 5V voltage regulator. The remaining half of the pins are connected directly to your power source.

-Serial DB9 Connector

Quickly upload new programs, or debug your code through the serial port with this convenient connector.

-On/Off & Reset Switches

Convenient power buttons avoid "pulling the plug" to power-down or reset.

-Battery and Power Indicators

Worried about frying your sensitive microcontroller by reversing the power connections? The Motherboard features two LED's, one indicates proper battery polarity, while the other indicates microcontroller operation.

-Noise Reducing Capacitors

Avoid signal noise in your projects with filter capacitors on the Motherboard.

Making it do something:

The goal of this manual is to create a simple application that gets the Robodyssey Advanced Mother Board (RAMB) up and running as quickly and as easily as possible. We will start with simply hooking it up on the bench and establishing communications between you PC and the RAMB.

Before we begin, lets familiarize ourselves with the RAMB and some of the basic needed equipment. If you already have your RAMB mounted, disconnect a servo linkage from a leg or wheel so we can just see the servo move without moving the robot.

Connecting it all

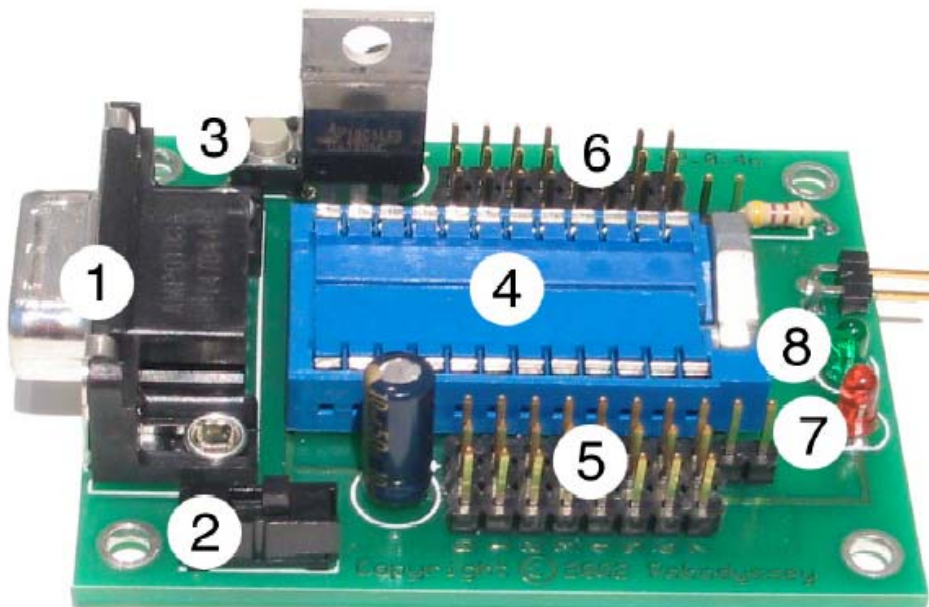


Figure 1

- | | |
|-------------------------------|--|
| 1 DB9 Serial Connector | 5 Vin I/O Pins |
| 2 Power Switch | 6 5V Regulated I/O Pins |
| 3 Reset Switch | 7 Microcontroller Operation LED |
| 4 ZIF Socket | 8 Correct Battery Polarity LED |

Figure 1 Shows the RAMB without a BX-24 installed.

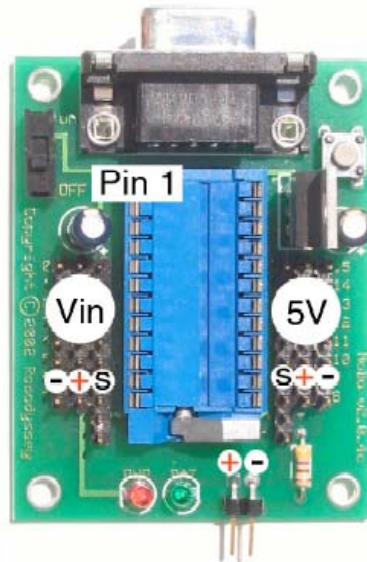


Figure 2

Figure 2 is worthy of some explaining. See the S+- on both sides of the board. They stand for “Signal”, “Positive” and “Ground”. Signal is the pin that connects to the processor stamp placed into the blue socket in the middle. Positive is the pin that provides external power to your devices. Ground is the pin that sucks in all the processed power used by your devices. Think of the “Positive” and “Ground” pins as the battery for your devices. The “Signal” is what tells those devices what to do. Another important point about the RAMB is it is divided between regulated and unregulated supply voltages. See the tall thing with the metal plate and a hole in the upper right. That thing is a voltage regulator. The voltage regulator takes a higher voltage and fixes it to a particular voltage, in this case 5 volts. All the “Positive” pins on the right side get a fixed 5 volts. All the “Positive” pins on the left side get whatever voltage is coming in from the batteries.

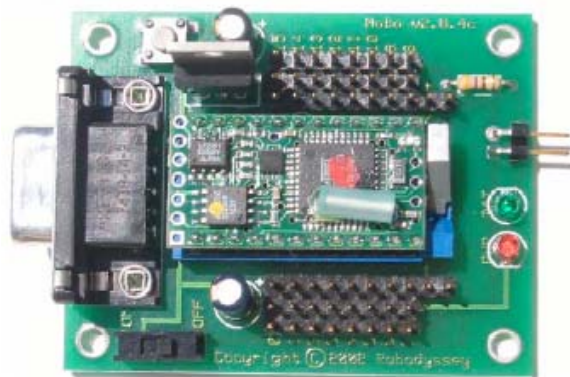


Figure 3

Figure 3 shows the RAMB with a BX-24 correctly installed.

Starting BasicX

Locate your BasicX icon or launch from the Start menu. You should see a screen like this if you are using BasicX 2.0 or 2.1.

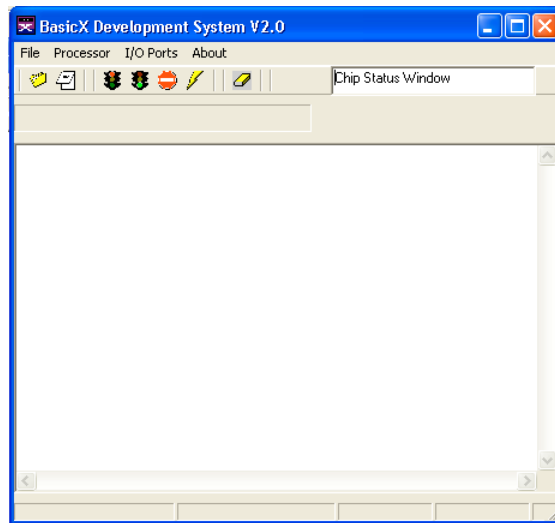


Figure 4

First, Lets do a little setting up house. If your using the Netmedia BX-24 and RAMB these should be default values, but, it never hurts to check. The first in our short series of things to check is the processor selection. Click on the “Processor” selection at the top of the BasicX Development System window. You should have a pop-up that looks like this:

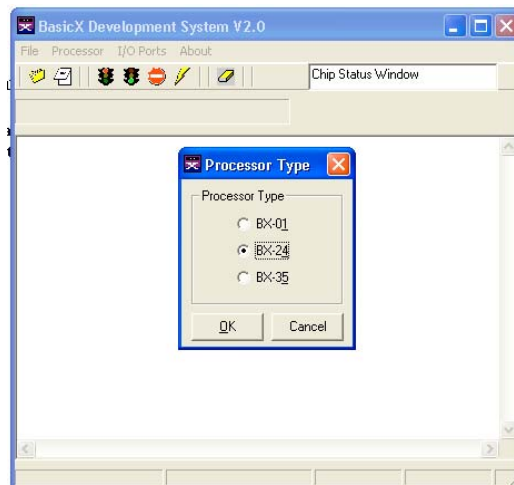


Figure 5

Make sure the BX-24 is selected, then click on OK.
Next, Click on the “I/O Ports” selection at the top of the BasicX Development System window. You should now have a window that looks like this

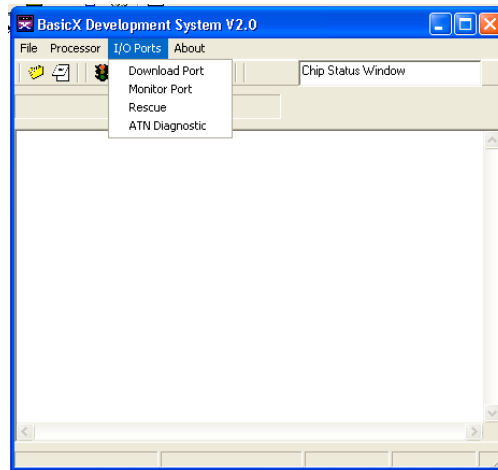


Figure 6

Select “Download Port” and another pop-up looking like this will appear.

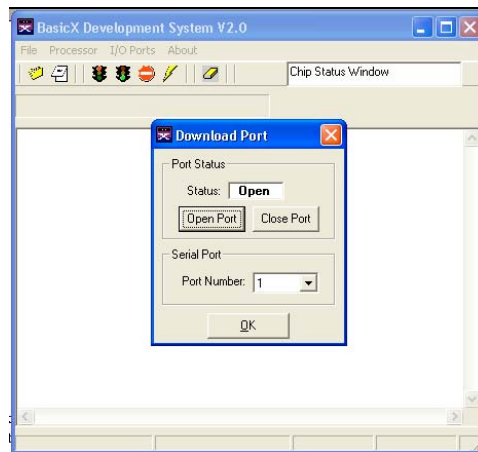


Figure 7

Select the Serial Port Number from the list then Click on the “Port Open” button. If there are no errors, the “Status” window will switch from a red “Closed” to a white “Open” as seen above. Finally, click “OK” to exit the pop-up. Next we need to launch an editor window in which to type our code. Click on the editor icon on the top of the BasicX Development System window.

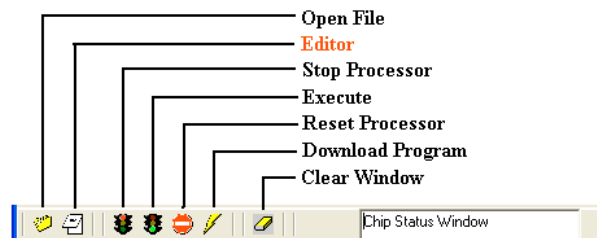


Figure 8

A new window called “BasicX Editor“ should appear.

Hello World!

Select “File” then “New Project” from the menu bar or simply click on the “New Project” icon. This will open a file window that will allow you to name the program and associated files.

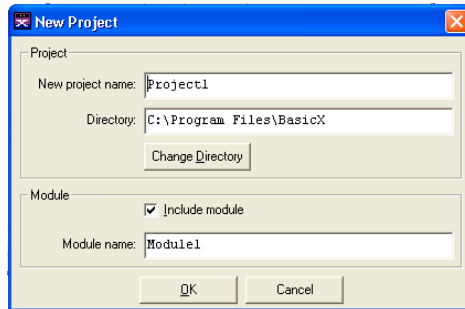


Figure 9

For this first project let’s just leave this the way it is and move on. Click the “OK” button to close this window. You now should have edit space that looks like this.

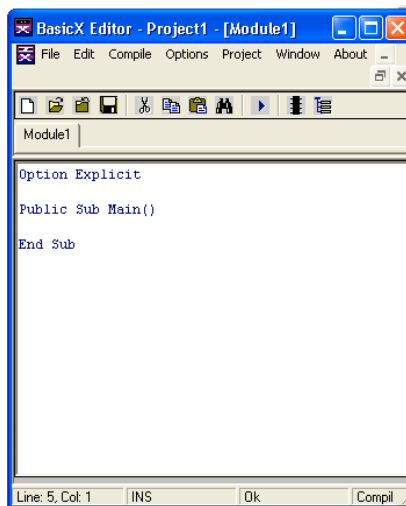


Figure 10

Now type between the lines “Public Sub Main()” and “End Sub” the below listed code:

```
Public Sub Main()  
Do  
    Debug.print “Hello World”  
    Sleep(1.0)  
Loop  
End Sub
```

Let’s try running it now that you have typed in this simple code. Under the “Compile” heading in the menu bar select “Compile and Run” as seen below.

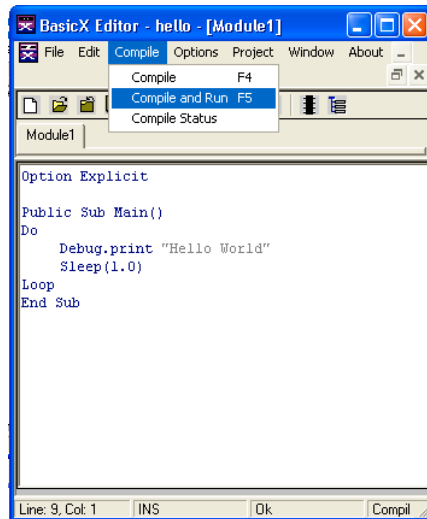


Figure 11

Did the BasicX Development System window pop back up??
 Did it start typing “Hello World” over and over again every second in the window??
 If it did it should look something like this:

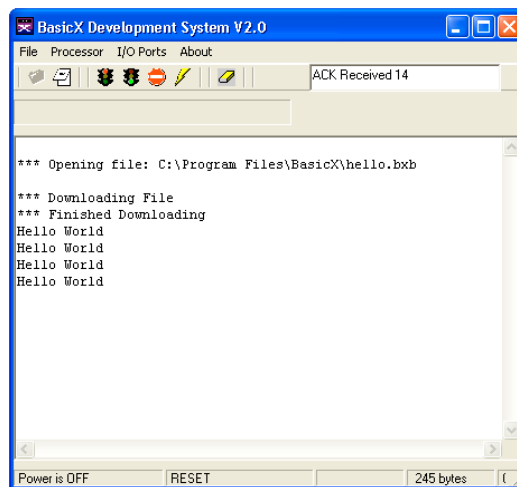


Figure 12

Congratulations! Your first program is running...and will continue to do so until you run out of batteries, turn off the power or stop it with a software selection. Let’s stop it by terminating the processor. From the BasicX Development System window select the “Stop Processor” button from the tool bar.

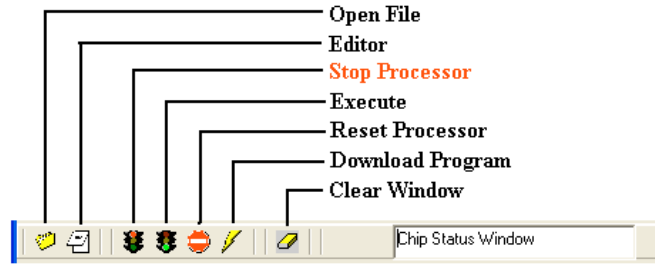


Figure 13

If it did not work you probably have a simple problem or two. The most likely is you have the wrong serial port selected. Try selecting another port as shown in figure 7. It is always a good idea to turn off and on the RMB to re-establish connection with your PC.

Sensing Your World

All right, lets keep moving toward our goal of becoming robot programmers! In this section we are going to explore connection and application of the Sharp GP2D12 IR Range Finder. Before we jump right in and start typing code, let's talk a bit about how the Sharp IR ranger works. The device emits a pulsed infrared beam. If an object is within range and in line with the IR beam, reflected light forms an image on a linear CCD array in the receiver. Triangulation is then used to determine range. The Sharp IR ranger is able to continuously measure the distance to an object. The device generates an analog voltage that is a function of range.

If we look at the chart below, we see that the voltage drops off as distance from the Sharp IR ranger increases. We also see the practical range of operation is from around 10cm to about 80cm. This voltage can be measured by an analog-to-digital (ADC) input line on a BasicX system.

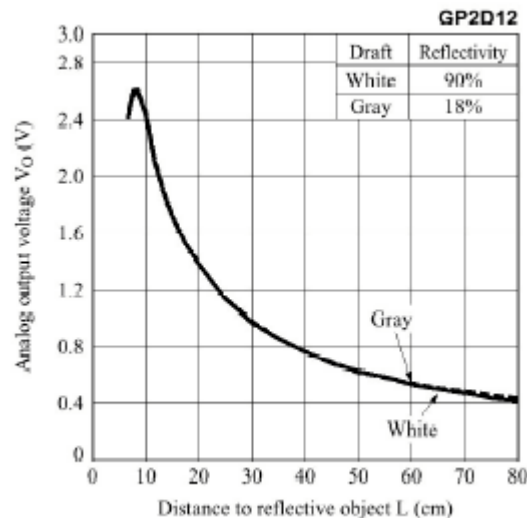


Figure 14

Well, Now that we know all this stuff about Sharp IR rangers, let's see what we can do with it.

OK, let's go back to our BasicX Editor window. This time we will name and save this program with a it more completeness. Click on "File" then "New Project" or just click on the "New Project" icon in the BasicX Editor window. That familiar New Project window should pop up as in figure 9.

Type in the "New Project Name" space something like:

IRSensor

And in the "Module Name" space type:

IR

As shown below:

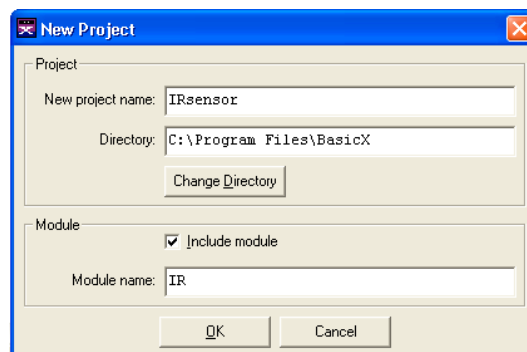


Figure 15

Click "OK" and a new blank screen should appear as in figure 10.

If you haven't already, plug in your Sharp IR ranger. For this example I plugged mine into the 3 pins labeled "8" on the RAMB. See the image below:



Figure 16

Pay very close attention to the plug arrangement. It is imperative that the ground wire be closest to the edge of the RAMB. Before we move on, I feel I should explain one more thing. If you look closely at your BX-24 stamp in the RAMB you will see it has 24 pins, 12 on each side. Naming convention in electronics has pin 1 on the upper left side of the stamp, with pins 2-24 being named counter-clockwise around the stamp. This puts pin 13 in the lower right and pin 24 in the upper right. Pins 1-4 and pins 21-24 are pretty much used for stamp operation and best not messed with by the novice programmer. Because of this, pin 1 on the RAMB corresponds with pin 5 on the BX-24. I have to bring this up since we need to tell the software what pin we are using for our devices. Do not use the pin numbers from the RAMB but rather use them from the BX-24 its self. Below, I have included a handy chart to help explain this issue.

RAMB Stamp		Stamp		RAMB	
Pin #	Pin #	Pin #	Pin #	Pin #	Pin #
N/A	1	24	N/A	N/A	N/A
N/A	2	23	N/A	N/A	N/A
N/A	3	22	N/A	N/A	N/A
N/A	4	21	N/A	N/A	N/A
0	5	20	15	15	15
1	6	19	14	14	14
2	7	18	13	13	13
3	8	17	12	12	12
4	9	16	11	11	11
5	10	15	10	10	10
6	11	14	9	9	9
7	12	13	8	8	8

OK, lets code!

Type into the designated space:

```

Public Sub Main()

Do
Debug.print "Value: " ;CStr(GetADC (13))
Sleep(1.0)
Loop

End Sub

```

As shown below:

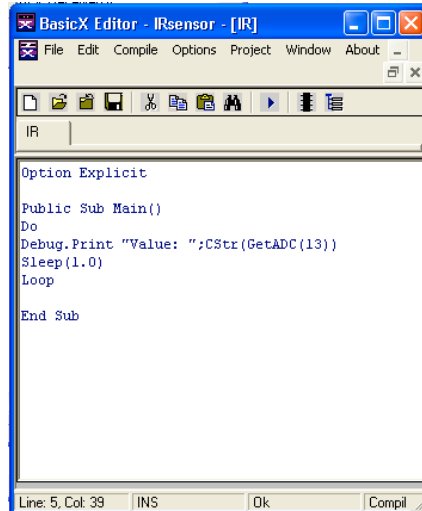


Figure 17

Before we start this program, let's discuss what it means.

Do // Loop

The “Do (some code stuff) Loop” is just that. A command that tells the processor to do something, then after it does, it hits the “Loop” and loops back to the “Do”.

Debug.Print

From the BX-24 manual we have:

“Debug.Print sends one or more strings out the Com1 serial port at 19 200 baud. Multiple string parameters must be separated by semicolons.

A carriage-return/linefeed pair is automatically appended unless an optional semicolon terminates the line. An empty Debug.Print outputs a carriage return/linefeed only.

Debug.Print automatically sets up Com1 for output. OpenCom is not needed.”

Which basically means, “Print whatever is between the quotes or whatever value you get on your computer screen”.

CStr

This tells the BX-24 to convert a value to a string. A string is a group of letters or numbers.

GetADC(13)

From the BX-24 manual we have:

“GetADC returns 10-bit analog voltage. The returned value is nondimensional, with a range of 0.0 to 1.0. For 5 V systems, the range corresponds to 0.0 V to 5.0 V, with a resolution of about 4.89 mV (5 / 1023).

ADC pin numbers depend on the system:

BX-24 ADC pins: 13 to 20

BX-35 ADC pins: 33 to 40

Note that GetADC automatically configures the pin for analog input. You don’t need a separate call to configure the pin to input mode.”

Now, what this means is, the BX-24 will take whatever voltage that is coming in (in this case on pin 13) and convert it to a value that the processor can use.

Sleep(1.0)

From the BX-24 manual we have:

“Suspends the current task for approximately the specified time interval. At the end of SleepInterval, the task will become ready again. How soon the task actually resumes execution depends on how busy the system is with other tasks.

A sleep of 0.0 is a useful way to allow other tasks to execute, while allowing immediate resumption if no other tasks are eligible to run.”

What this basically means is: “Processor, take a break”. In our case, that break was about (1.0) 1 second.

OK, let it rip! Hit the “Execute” button or under the “Compile” heading in the menu bar select “Compile and Run”.

Is it running??

Are you getting numbers appearing down the screen?

Try holding your hand in front of the Sharp IR ranger at a couple of different heights.

You should have something like this:

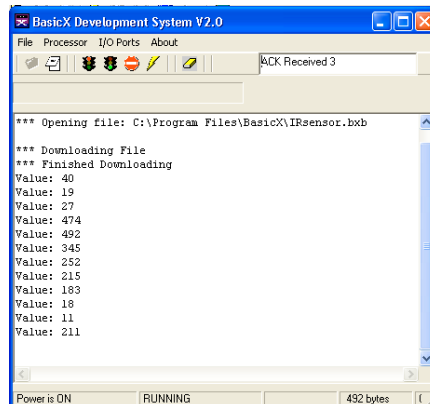


Figure 18

See how the value changes as your hand moves closer and away from the sensor?
Go ahead and stop the processor as you did in figure 13.

Lets do one more program with the sensor to explain another concept in robot programming. Now we are going to use the value generated to cause a response in the processor. The goal is to have code that will print "To Close", "To Far" or "Just Right" on your computer as your hand moves closer to or further from the sensor. This code example will be handy for several reasons:

- 1) We will introduce variable.
- 2) We will introduce the use of comments.
- 3) We will introduce subroutines.
- 4) We will introduce the If // Then statement.
- 5) This code is very handy for programming a robot navigating space and edges.

```

Option Explicit
Dim value as Integer          ' Variable for IR data
'*****
Public Sub Main()
Do
    Call SensorIn            ' Main program loop
Loop
End Sub
'*****
' I like to use stars to separate my subroutines

Sub SensorIn ()
Do
    value = GetADC(13)      ' Name of the subroutine
    If value > 450 Then     ' Beginning of the loop
        Call Toclose      ' Get IR and stick it in "value"
        Sleep (1.0)       ' Voltage goes up if close
    End If                 ' Go to the subroutine for to close action
    If value < 150 Then     ' Take a 1 second break
        Call Tofar        ' Close this If statement
        Sleep (1.0)       ' Voltage goes down if far
    End If                 ' Go to the subroutine for to far action
    Sleep (0.5)            ' Take a 1 second break
    Debug.Print "Just Right" ' Close the If statement
                            ' Take a 1/2 second break
Loop
End Sub
' If it was between 151 and 449 print "Just Right"
' End of the loop
'*****

Sub Tofar ()
Do
    Debug.Print "To Far"   ' Name of the subroutine
    value = GetADC(13)     ' Beginning of the loop
    If value > 150 Then    ' If it was between over 450 print "Just Right"
        Call SensorIn     ' Check to see what it still is
        Sleep (1.0)       ' If has changed go back to the beginning
    End If                 ' Go back to the sensor in routine
                            ' Take a 1 second break
                            ' Close the IF statement

```

```

        Sleep (0.5)                ' Take a 1 second break
Loop                                ' End of the loop
End Sub

*****
Sub Toclose ()                    ' Name of the subroutine
Do                                ' Beginning of the loop
    Debug.Print "To Close"        ' If it less than 150 then print "To Close"
    value = GetADC(13)            ' Check to see if it has moved
        If value < 450 Then      ' If it has moved go back to the beginning
            Call SensorIn       ' Go back to the sensor in routine
            Sleep (1.0)         ' Take a 1 second break
        End If                  ' Close the If statement
    Sleep (0.5)                  ' Take a 1/2 second break
Loop                              ' End of the loop
End Sub

```

Wow! Did you get all that typing? Now let me explain what it all means. Let's start with the variable. In this example, we needed a way to easily handle data. We could have asked for a fresh number every time, but that is not always the best thing to do especially if we want to do more than one thing with the data. The best way to deal with it is to declare a variable. This is done at the very beginning of the code just below "Option Explicit". In this case we used the term "Dim value as Integer". What that is saying is:

"Hey processor, I declare the term "value" as an integer".

Now, if I say "value = 4" then, the computer treats value as a 4. If I say "value = value - 1", I am really saying "value = 4 - 1" or "value = 3". Now, the number for "value" is 3.

Next on our list is comments. Commenting code is very important as the code gets longer and longer. By putting a single quote mark (') in front of a line of text, the processor does not use that information. Commenting is very useful in making sure you remember what you are doing. Believe me, In a short time you will likely forget how or why you did something unless you make comments.

Well, how about those subroutines? Where did they come from? Subroutines are like little programs within your program. They are incredibly useful in streamlining your code. All you have to do is start one with the phrase "Sub" then any name you choose and a pair of brackets (). Now, you can type your code like, walk strait, chew gum, or pat your head. When your done with the code just close up the subroutine with the "End Sub" command. The beauty of this is every time you want your creation to "walk strait" just call your subroutine. Type it once and call on it as often as you please.

If I understood this "If Then" stuff then I would be a better programmer. End If. "If Then" statements are the easiest to use and almost always the most useful. The biggest error in using them is forgetting to end them. For every "If", there is an "End If". If I am close to the edge then back up "End If". If I am in too bright a light then find shade "End If". If I am bored then chew gum "End If". I bet now you also see how useful those

subroutines were. JUST DON'T FORGET TO END THE "IF" WITH AN "END IF". I always do.

Moving Though The World

Special Notice to Wheeled Robot Owners:

If you own a Mouse or other robot that has servos modified for continuous rotation, this sample code will behave differently. Instead of moving to a fixed position, your servo will rotate at different speeds and directions. On some occasions, the servo may not move at all, if the pulse width is set for 0.0015.

Finally it is time to talk about servo control. As you may have guessed, I am first going to talk about servos and how they work. Servos are nifty little electromechanical devices that first appeared in the hobby trade for use in radio controlled cars, boats and planes. Now, they are the system of choice for a full range of robotic applications. Let's start by looking at how a servo works. From the BX-24 manual we find:

"Servos require electrical power and a position signal to operate. Servos typically require 4.8 VDC to 6.5 VDC power. Current requirements are highly variable. You may need to refer to the servo manufacturer's specifications for more precise voltage and current requirements.

The position signal usually consists of a 1 ms to 2 ms high-going pulse repeated at a 50 Hz refresh rate. An example of this signal and the servo's resulting movements can be seen below:

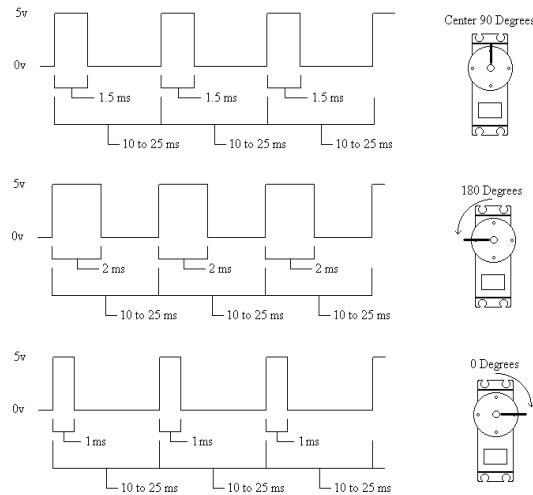


Figure 19

The pulse width range can vary depending on the make and model of servo. Some servos have a larger movement range and may require pulses as small as 0.5 ms or as large as 2.2 ms to achieve full range of movement. But be careful not to exceed a safe range --

depending on the servo, if a pulse is too long or too short, you may drive the servo against an internal stop, stall the motor and damage gears.

The optimum pulse rate can vary somewhat. If your refresh rate is either too slow or too fast, the servo may produce a loud buzzing sound. If this happens, try turning the output horn -- if the noise gets louder, the refresh rate is too slow. Otherwise it's too fast."

OK, so what does that mean? Tap your finger on the table once every second. That action has a few steps. Your finger goes down, contact with the table, your finger goes up and you wait until the next cycle. Now, let's break our finger tapping into 10 equal time units $1/10^{\text{th}}$ of a second each. I am going to say that bringing your finger down and bringing your finger up take $1/10^{\text{th}}$ of a second each. That leaves us with $8/10^{\text{th}}$ of a second. Only the time we wait and the time we have our finger on the table are left to play with. The longer we have our finger on the table, the shorter we wait to start over. So, if I hold my finger down for $2/10^{\text{th}}$ of a second, I have to wait $6/10^{\text{th}}$ of a second until we start again. If I hold my finger down for $5/10^{\text{th}}$ of a second, I only have to wait $3/10^{\text{th}}$ of a second until we start again. The only thing the servo really cares about is how long you hold your finger to the table. Hold it down just a tiny bit, the servo is over at its furthest point clockwise. Hold it down longer, the servo moves to a point counterclockwise. Now, to a servo, what you really need to do is tap your finger 50 times a second holding your finger to the table between $1/1000^{\text{th}}$ and $2/1000^{\text{th}}$ of a second!

So how do we use this information to program a servo? We really just need to set up 2 different timers. One telling us how long to pulse and another telling us how often we should do it. Fortunately, most of the hard work has been done by BasicX. Let's see what the manual has to say about "PulseOut".

Syntax (The way you say it)

Call PulseOut(Pin, PulseWidth, State)

Arguments (Stuff you can change about it)

Item	Type	Direction	Description
<i>Pin</i>	Byte	Input	Pin number
<i>PulseWidth</i>	Single	Input	Time interval. Units are in seconds, range is about 1.085 μs to 71.1 ms.
<i>State</i>	Byte	Input	Specifies either high (1) or low (0) pulse

Description

PulseOut sends a logic high or logic low pulse from any available I/O pin. The procedure waits until the pulse has been sent before returning.

The resolution of PulseOut is 8 / 7 372 800 seconds (about 1.085 μ s).

Note -- PulseOut can be used solely as a means of generating a delay -- that is, without affecting physical I/O pins. This is done by using pin 0 as the pin parameter. Pin 0 is treated as a dummy pin.

Check it out! Looks like all we need to do is tell the processor what pin and for how long to pulse. Piece of cake! That last Argument, State, is just in case you want to suck in power instead of push it out. We want to push it out, so we will always have a “1” in that spot.

OK, let’s do some code! For this first example, I just want to make the servo move to about it’s center position. All we need is a “PulseOut” command for how long the pulse will last and a delay until we start again. Nothing to it!

First we should make sure your connected correctly. I connected the servo to the pin labeled 11 on the RAMB as seen below. Be sure to have the ground wire closest to the edge of the board. You can connect to any available pins, just be sure to recognize which pin it is.



Figure 20

Go ahead and open or create a new file. Call it what you like and get to the edit window. Now, type in this code:

```
Sub Main()  
Do  
    Call PulseOut(16, 0.0015, 1)    ' Generate a 1.5 ms pulse.  
    Call Delay(0.02)                ' Produce a pulse rate of around 50 Hz.  
Loop  
End Sub
```

Well, don’t just sit there. Execute it!

Did it work? Did your motor move to the center position? Well that was a flash of excitement. But now you're bored. Let's move on and see if we can get it doing something more rhythmic. Stop your program and type this into your edit window:

```
Option Explicit
Const ServoPulses as Byte = 30           'Number of pulses it takes to reach a position
Public Sub Main()
Do
    Call Walk
Loop
End Sub
*****
Sub Walk()
Dim x as Byte
    For x = 1 to ServoPulses             ' Do this for ServoPulses number of times
        Call PulseOut(16, 0.002, 1)     ' Generate a 20 ms pulse.
        Call Delay(0.02)                 ' Produce a pulse rate of around 50 Hz.
    Next
    For x = 1 to ServoPulses             ' Do this for ServoPulses number of times
        Call PulseOut(16, 0.001, 1)     ' Generate a 10 ms pulse.
        Call Delay(0.02)                 ' Produce a pulse rate of around 50 Hz.
    Next
End Sub
```

Cool huh? You should be getting interested now! Just imagine if there was a leg hooked to that servo! As you can see, I threw in a couple of new things here. The first is “Const” which is short for constant. In other words, not variable. I declared that “ServoPulses” would be constant and unchanging. I also called it a “Byte”.

What does that mean?

Once again referring to the BasicX Manual we find this table.

Data types

<i>Type</i>	<i>Storage</i>	<i>Range</i>
<i>Boolean</i>	<i>8 bits</i>	<i>True .. False</i>
<i>Byte</i>	<i>8 bits</i>	<i>0 .. 255</i>
<i>Integer</i>	<i>16 bits</i>	<i>-32 768 .. 32 767</i>
<i>Long</i>	<i>32 bits</i>	<i>-2 147 483 648 .. 2 147 483 647</i>
<i>Single</i>	<i>32 bits</i>	<i>-3.402 823 E+38 .. 3.402 823 E+38</i>
<i>String</i>	<i>Varies</i>	<i>0 to 64 characters</i>
<i>BoundedString</i>	<i>Varies</i>	<i>0 to 64 characters</i>

By calling my constant a Byte I limit the amount of memory needed to hold the information.

Another new introduction is the “For // Next” statement. For // Next statements are usually used to run a specific number of loops. In our case we had it count up from 1 to ServoPulses so that the servo had sufficient time to move.

Putting it All Together

Let's do one more demonstration before we start programming a robot. We're going to make an IR sensing servo controller. We should be able to position a servo based on our hand position. Open up a new Edit window and enter this:

```
Option Explicit
Dim value as Integer           ' Variable for IR data
Const ServoPulses as Byte = 20 ' Number of pulses it takes to reach a position
Const PinServoUses as Byte = 16 ' Connecting pin for the servo
Const ShortPulse as Single = 0.001 ' Replaces the pulse time in Pulseout
Const MidPulse as Single = 0.0015 ' Replaces the pulse time in Pulseout
Const LongPulse as Single = 0.002 ' Replaces the pulse time in Pulseout
*****
Public Sub Main()
Do
    Call SensorIn           ' Main program loop
Loop
End Sub
*****
Sub SensorIn ()
Dim x as Byte
Do
    value = GetADC(13)      ' Beginning of the loop
    If value > 450 Then     ' Get IR and stick it in "value"
        Call Toclose       ' Voltage goes up if close
    End If                 ' Go to the subroutine for to close action
    If value < 150 Then     ' Close this If statement
        Call Tofar         ' Voltage goes down if far
    End If                 ' Go to the subroutine for to far action
    Sleep (0.2)            ' Close the If statement
    Debug.Print "Just Right" ' Take a 0.2 second break
    For x = 1 to ServoPulses ' If it was between 151 and 449 print "Just Right"
        Call PulseOut(PinServoUses, MidPulse, 1) ' Do this for ServoPulses number of times
        Call Delay(0.02) ' Generate a 15 ms pulse.
    Next                   ' Produce a pulse rate of around 50 Hz.
Loop
End Sub
*****
Sub Tofar ()
Dim x as Byte
Do
    Debug.Print "To Far"   ' Name of the subroutine
    For x = 1 to ServoPulses ' Beginning of the loop
        Call PulseOut(PinServoUses, ShortPulse, 1) ' If it was between over 450 print "Just Right"
        Call Delay(0.02) ' Do this for ServoPulses number of times
    Next                   ' Generate a 10 ms pulse.
    value = GetADC(13)     ' Produce a pulse rate of around 50 Hz.
    If value > 150 Then    ' Check to see what it still is
        Call SensorIn     ' If has changed go back to the beginning
    End If                ' Go back to the sensor in routine
Loop                     ' Close the IF statement
```

```

        Sleep (0.2)                ' Take a 0.2 second break
Loop                                ' End of the loop
End Sub

*****
Sub Toclose ()                    ' Name of the subroutine
Dim x as Byte
Do                                ' Beginning of the loop
    Debug.Print "To Close"        ' If it less than 150 then print "To Close"
    For x = 1 to ServoPulses      ' Do this for ServoPulses number of times
        Call PulseOut(PinServoUses, LongPulse, 1) ' Generate a 20 ms pulse.
        Call Delay(0.02)         ' Produce a pulse rate of around 50 Hz.
    Next
    value = GetADC(13)           ' Check to see if it has moved
    If value < 450 Then          ' If it has moved go back to the beginning
        Call SensorIn           ' Go back to the sensor in routine
    End If                       ' Close the If statement
    Sleep (0.2)                 ' Take a 0.2 second break
Loop                              ' End of the loop
End Sub

```

Ok, did you do it? Did it Work? You should have had the motor moving to 3 different positions, while on the computer, you see your position printed as you moved your hand. For those of you just copying and pasting I did change a couple of things. Notice that I dropped a couple of sleep commands and shortened the “ServoPulses”. I did these to speed up the program a bit. Most importantly, I added some more declare statements. Now when you need to “tweak” your robot you only need to change the value at the beginning of your code. See how they replace the actual numbers within the brackets of “PulseOut”. Handy huh?

So, You want to have an Argument?

Well I do! But not the kind you’re thinking of. The last thing I think you need before you move to your robot is some kind of understanding of Arguments. Have you been wondering about the parenthesis at the end of each Subroutine? Well, that’s where we start this argument. Let’s look first at what Bill Gates has to say about them in his Visual Basic Manual:

Passing Arguments to Procedures

Usually the code in a procedure needs some information about the state of the program to do its job. This information consists of variables passed to the procedure when it is called. When a variable is passed to a procedure, it is called an argument.

Argument Data Types

The arguments for procedures you write have the Variant data type by default. However, you can declare other data types for arguments. For example, the following function accepts a string and an integer:

```

Function WhatsForLunch(WeekDay As String, Hour _
As Integer) As String
    ' Returns a lunch menu based on the day and time.

```

```

If WeekDay = "Friday" then
    WhatsForLunch = "Fish"
Else
    WhatsForLunch = "Chicken"
End If
If Hour > 4 Then WhatsForLunch = "Too late"
End Function

```

For More Information Details on Visual Basic data types are presented earlier in this chapter. You can also see the Language Reference for specific data types.

Passing Arguments By Value

Only a copy of a variable is passed when an argument is passed by value. If the procedure changes the value, the change affects only the copy and not the variable itself. Use the ByVal keyword to indicate an argument passed by value. For example:

```

Sub PostAccounts(ByVal intAcctNum as Integer)
.
. ' Place statements here.
.
End Sub

```

There, didn't that help! Not!

For lack of a better description, I think of Arguments as the hands that hold the data. If I put an Argument in my subroutine, I am giving it strict instructions on how to pass data and how to treat it.

Let's look at a really simple example of how this works. Type this into an edit box:

Option Explicit

```

*****
Public Sub Main()
Do
    Call DoIt (5)           ' Notice the 5 in the parenthesis
    Debug.Print "I did it" ' Print when back from sub
Loop

End Sub
*****
Sub DoIt (ByVal NumTimes as Byte) ' I have declared "NumTimes
' So now the stuff in the parenthesis gets passed directly to the subroutine
Dim i as Byte                  ' Need a local variable
For i = 1 to NumTimes          ' Count from 1 to Numtime
    Debug.Print "OK"          ' Print it
    Sleep(1.0)                 ' Take a break
Next                            ' Do it until Numtime is up

End Sub

```

Cool huh! See, by having arguments in your subroutines the data just automatically flowed through your program. Let me now show you that the argument types can be mixed up a bit. Type this into an edit window:

Option Explicit

```
*****
Public Sub Main()
Do
    Call DoIt (2,3)           ' Notice the 2 and the 3 in the parenthesis
    Debug.Print "I did it"   ' Print when back from sub
Loop

End Sub
*****
Sub DoIt (ByVal NumTimes as Byte, ByVal Onemore as Integer) ' I have declared "NumTimes
    ' and Onemore, each as a different type.
    ' So now the stuff in the parenthesis gets passed directly to the subroutine
    Dim i as Byte           ' Need a local variable, notice Byte
    For i = 1 to NumTimes   ' Count from 1 to Numtime
        Debug.Print "OK"    ' Print it
        Sleep(1.0)         ' Take a break
    Next                   ' Do it until Numtime is up

    Dim x as Integer        ' Need a local variable, notice Integer
    For x = 1 to Onemore    ' Count from 1 to Onemore
        Debug.Print "OK, I get it!" ' Print it
        Sleep(1.0)         ' Take a break
    Next

End Sub
```

Well check it out! See how those variables passed right through. It may not seem like much now, but wait till you start programming something like the repetitive steps of a robot!

Wow, I think we are ready to start programming a robot. Turn next to the manual describing your robot and lets start!

Robodyssey Mouse

Programming Guide (BasicX)

Version 1.02

Getting Started

To get started programming your new Mouse, make sure you have the following items available:

1. Fully Assembled Robodyssey Mouse with Robodyssey Advanced Motherboard
2. BasicX microcontroller installed in Robodyssey Advanced Motherboard
3. Power Pack
4. 9 Pin Male - Female Serial Cable
5. BasicX Software (Available on Robodyssey CD or from www.basicx.com)

Common Constants

Although you could certainly program your Mouse anyway you like, we've standardized on a common setup of constants to make pin configurations and software sharing a bit easier.

Be sure to connect your servos to your Robodyssey Motherboard as described in your Assembly Instructions. Here's a quick description of how to connect your servos.



Figure 1 - Proper Servo Connections

```
Option Explicit
Const LeftServo As Byte = 5
Const RightServo As Byte = 6
Const LeftServoReverse As Single = 0.001
Const LeftServoForward As Single = 0.002
Const RightServoReverse As Single = 0.002
Const RightServoForward As Single = 0.001
Public Sub Main()
End Sub
```

Figure 2.

As shown in Figure 1, connect the left servo to Pin 0 on the Robodyssey Advanced Motherboard, and the right servo to Pin 1.

Now we can declare two constants, LeftServo and RightServo, as well as constants for the forward and reverse pulse timings for each servo. See Figure 2 for the code.

The pin numbering is slightly different between the Robodyssey Advanced Motherboard and the BasicX software. The Motherboard I/O pins are numbered starting from zero, while in the BasicX software the pins are referenced by the physical locations on the chip. Therefore, Pin 0 on the Motherboard corresponds to Pin 5 in the BasicX software.

To make sure your physical connections and your software are in sync, always add five to the Motherboard pin number to get the correct number to use in your software.

Forward Movement

Execute the following program to move the Mouse a distance specified by the number of pulses declared by the variable x.

```
*****
'   Going Forward for x
*****
Option Explicit

Const LeftServo As Byte = 5           ' Pin 0 on the RAMB
Const RightServo As Byte = 6         ' Pin 1 on the RAMB
Const LeftServoReverse As Single = 0.001 ' Make slightly larger to go slower ie 0.0012
Const LeftServoForward As Single = 0.002 ' Make slightly smaller to slower ie 0.0018
Const RightServoReverse As Single = 0.002 ' For modified servos, 0.0015 is "center" or stop
Const RightServoForward As Single = 0.001 ' Center usually falls between 0.00135 and 0.00165

Dim x as Integer

*****
*** Start of Main Loop *****
Public Sub Main()

Delay(3.0)           ' 3 second delay
x = 100             ' Make x larger to go further
Do While x > 1     ' Robot stops when x = 1
  x = x - 1        ' Subtract 1 from x every iteration
  Debug.Print Cstr(x) ' print to screen if interested
  Call PulseOut(LeftServo, LeftServoForward, 1) ' Pulse Left servo 0.002
  Call PulseOut(RightServo, RightServoForward, 1) ' Pulse Right servo 0.001
  Delay(0.02)      ' Required delay between servo pulses
Loop
End Sub
```

Right Turns, Left Turns & Reverse

Using the Mouse's two wheel differential drive, it's easy to make zero-radius turns just by sending the servos opposite direction. This simple program tells the mouse to move according to the specified by x. A larger value for x will cause the robot to move further or turn longer. This program is fun for "Dead Reckoning" maze challenges. Challenge yourself to program your robot to navigate to the refrigerator and back!

```
*****
'   Dead Reckoning
*****
Option Explicit

Const LeftServo As Byte = 5           ' Pin 0 on the RAMB
Const RightServo As Byte = 6         ' Pin 1 on the RAMB
Const LeftServoReverse As Single = 0.001 ' Make slightly larger to go slower ie 0.0012
Const LeftServoForward As Single = 0.002 ' Make slightly smaller to slower ie 0.0018
Const RightServoReverse As Single = 0.002 ' For modified servos, 0.0015 is "center" or stop
Const RightServoForward As Single = 0.001 ' Center usually falls between 0.00135 and 0.00165

Dim x as Integer
```

```

*****
*** Start of Main Loop *****
' Simply copy and paste a x value and a direction to create your own
' set of directions.
Public Sub Main()

Delay(3.0)           ' Wait 3 seconds to start

x = 100              ' Change this value to change distance traveled
  Call Forward

x = 15               ' Change this value to change time spinning
  Call TurnRight

x = 100              ' Change this value to change distance traveled
  Call Forward

x = 15               ' Change this value to change time spinning
  Call TurnLeft

x = 100              ' Change this value to change distance traveled
  Call Forward

x = 15               ' Change this value to change time spinning
  Call TurnRight

x = 100              ' Change this value to change distance traveled
  Call Backup

End Sub
*****
*** Forward *****
Sub Forward()
Do While x > 1
  x = x - 1
  Call PulseOut(LeftServo, LeftServoForward, 1)
  Call PulseOut(RightServo, RightServoForward, 1)
  Delay(0.02)
Loop
End Sub
*****
*** TurnRight *****
Sub TurnRight()
Do While x > 1
  x = x - 1
  Call PulseOut(LeftServo, LeftServoForward, 1)
  Call PulseOut(RightServo, RightServoReverse, 1)
  Delay(0.02)
Loop
End Sub
*****
*** TurnLeft *****
Sub TurnLeft()
Do While x > 1
  x = x - 1
  Call PulseOut(LeftServo, LeftServoReverse, 1)
  Call PulseOut(RightServo, RightServoForward, 1)
  Delay(0.02)
Loop
End Sub
*****

```

```

'**** Backup *****
Sub Backup()
Do While x > 1
  x = x - 1
  Call PulseOut(LeftServo, LeftServoReverse, 1)
  Call PulseOut(RightServo, RightServoReverse, 1)
  Delay(0.02)
Loop
End Sub

```

Analog Sensors

The BasicX microcontroller has 8 10-bit Analog to Digital Converters (ADC) that allow voltage measurements from 0V to 5V. The ADC's are available on Pins 8 - 15 on the Motherboard, or pins 13 - 20 in the BasicX software.

To connect a Sharp GP2D12 Infrared Range Finder, follow the Wire Assembly Instructions as described in Figure 21 of your Mouse Assembly Manual.

The Sharp GP2D12 outputs an analog voltage inversely proportional to the distance of the measured object. Basically, the greater the voltage measured from the sensor, the closer the object and vice versa.

This simple program displays the value on your screen that the Sharp sensor detects.

```

*****
' A to D display
*****
Option Explicit

Const FrontIRPin As Byte = 13          'Front Sensor connected to pin 8 of the RAMB

Public Sub Main()
*****
Do
Debug.Print CStr(GetADC(FrontIRPin))   ' Display value on your PC
Delay(1.0)                             ' Wait 1 second to repeat
Loop
*****
End Sub

```

Putting it all Together

This is where we put it all together. In this simple program, the robot is using the Sharp sensor to detect both an edge and an obstacle. The two If-Then statements make the decision based on the value acquired by the sharp sensor.

```

*****
'   Basic Avoidance
*****

Option Explicit

Const LeftServo As Byte = 5           ' Pin 0 on the RAMB
Const RightServo As Byte = 6          ' Pin 1 on the RAMB
Const LeftServoReverse As Single = 0.001 ' Make slightly larger to go slower ie 0.0012
Const LeftServoForward As Single = 0.002 ' Make slightly smaller to slower ie 0.0018
Const RightServoReverse As Single = 0.002 ' For modified servos, 0.0015 is "center" or stop
Const RightServoForward As Single = 0.001 ' Center usually falls between 0.00135 and 0.00165

Const FrontIRPin As Byte = 13         ' Front Sensor
Dim x as Integer                      ' Place to hold IR values
Dim Value as Integer

*****
*** Start of Main Loop *****
Public Sub Main()

Delay(3.0)                             ' Give you a chance to let go
Do
    Call PulseOut(RightServo, RightServoForward, 1) ' These two pulseouts will cause
    Call PulseOut(LeftServo, LeftServoForward, 1)   ' the robot to move forward
    Call CheckSensors()                             ' Jump down to check the IR Sensor
    Delay(0.02)                                     ' Standard Servo Delay
Loop

End Sub
*****
**** Check Sensors *****
Sub CheckSensors()

' This routine is look for an obstacle.

Value = GetADC(FrontIRPin)              ' Get the Sharp Sensor reading

    If Value > 350 Then                  ' Make the 350 smaller if you want the robot to respond further away
        Call Backup                      ' Make the robot back up
        Call TurnLeft                   ' Make the robot turn left
    End If

' This is looking for the edge of a table. The smaller the number, the further it must be able to see

    If Value < 80 Then                  ' If the drop is short, then make the number bigger
        Call Backup                      ' Make the robot back up
        Call TurnRight                   ' Make the robot Turn Right
    End If

End Sub
*****
**** TurnRight *****
Sub TurnRight()
x = 10                                  ' Make bigger to turn longer
Do While x > 1
    x = x - 1
    Call PulseOut(LeftServo, LeftServoForward, 1)
    Call PulseOut(RightServo, RightServoReverse, 1)
    Delay(0.02)                          ' Standard Servo Delay
Loop
End Sub
*****

```

```

'**** TurnLeft *****
Sub TurnLeft()
x = 10 ' Make bigger to turn longer
Do While x > 1
  x = x - 1
  Call PulseOut(LeftServo, LeftServoReverse, 1)
  Call PulseOut(RightServo, RightServoForward, 1)
  Delay(0.02) ' Standard Servo Delay
Loop
End Sub
'*****
'**** Backup *****
Sub Backup()
x = 10 ' Make bigger to back up longer
Do While x > 1
  x = x - 1
  Call PulseOut(LeftServo, LeftServoReverse, 1)
  Call PulseOut(RightServo, RightServoReverse, 1)
  Delay(0.02) ' Standard Servo Delay
Loop
End Sub

```

Fun Extras

This simple little program has the mouse spinning round and round till something gets in front of the sensor. Then, the Mouse proceeds to chase after it!

```

'*****
' Follow Me by Chris Odom
'*****
Option Explicit
Public Const Left_Forward as Single = 0.002
Public Const Right_Forward as Single = 0.001
Public Const Left_Reverse as Single = 0.001
Public Const Right_Reverse as Single = 0.002
Public Const LeftServo as Byte = 5
Public Const RightServo as Byte = 6

'*****
'*** Start of Main Loop *****
Public Sub Main()

Delay(3.0)
Do
  Call PulseOut(RightServo, Right_Forward, 1)
  Call PulseOut(leftServo, Left_Forward, 1)
  Call CheckSensors()
  Delay(0.02)
Loop

End Sub
'*****
'**** Check Sensors *****
Sub CheckSensors()

Do While GetADC(13) < 250 ' Change this Value from 250
  Debug.Print "I don't see anything"
  Call PulseOut(RightServo, Right_Forward, 1)
  Call PulseOut(leftServo, Left_Reverse, 1)
  Delay(0.05)
Loop

```

```

Do While GetADC(13) > 400           ' Change this Value from 400
  Debug.Print " I got it"
Loop

  Debug.Print "IR Reading: " & CStr(GetADC(13))
End Sub

```

This program shows the range of speed the wheels are able to achieve with different pulseouts. The program will slowly change the pulseout time and display that value on your PC. It is best to have the Mouse on a platform so that the wheels do not touch the ground.

```

*****
'      Wheel Adjust
*****
Option Explicit
Dim Left as Single
Dim Right as Single
Public Const LeftServo as Byte = 5
Public Const RightServo as Byte = 6
Dim x as Integer

*****
*** Start of Main Loop *****
Public Sub Main()

Do
Debug.Print "Left Wheel"
Delay(3.0)
Left = 0.0023
Do While Left > 0.001
  Left = Left - 0.00005
  Debug.Print Cstr(Left)
  Call CheckLeft()
Loop
Debug.Print "Left Wheel done"
Delay(3.0)

Debug.Print "Right Wheel"
Delay(3.0)
Right = 0.0023
Do While Right > 0.001
  Right = Right - 0.00005
  Debug.Print Cstr(Right)
  Call CheckRight()
Loop
Debug.Print "Right Wheel done"
Delay(3.0)
Loop
End Sub

```

```
'*****  
'*** Check Left Wheel *****  
Sub CheckLeft()  
  
x = 100  
Do While x > 1  
  x = x - 1  
  Call PulseOut(LeftServo, Left, 1)  
  Delay(0.02)  
Loop  
  
End Sub  
'*****  
'*** Check Right Wheel *****  
Sub CheckRight()  
  
x = 100  
Do While x > 1  
  x = x - 1  
  Call PulseOut(RightServo, Right, 1)  
  Delay(0.02)  
Loop  
  
End Sub
```

Appendix 1

BasicX Language

The BasicX Language we will be using is very similar to Microsoft's Visual Basic. For a detailed System Library, please see your Robodyssey CD or our web site for the BasicX documentation. Below you will find a quick reference for the available data types, as well as a listing of the System Library.

Type	Storage	Range
Boolean	8 bits	True .. False
Byte	8 bits	0 .. 255
Integer	16 bits	-32 768 .. 32 767
Long	32 bits	-2 147 483 648 .. 2 147 483 647
Single	32 bits	-3.402 823 E+38 .. 3.402 823 E+38
String	Varies	0 to 64 characters

Math functions

Abs	Absolute value
ACos	Arc cosine
ASin	Arc sine
Atn	Arc tangent
Cos	Cosine
Exp	Raises <i>e</i> to a specified power
Exp 10	Raises 10 to a specified power
Fix	Truncates a floating point value
Log	Natural log
Log10	Log base 10
Pow	Raises an operand to a given power
Sin	Sine
Sqr	Square root
Tan	Tangent

String functions

Asc	Returns the ASCII code of a character
Chr	Converts a numeric value to a character
LCase	Converts string to lower case
Len	Returns the length of a string
Mid	Copies a substring
Trim	Trims leading and trailing blanks from string
UCase	Converts string to upper case

Memory-related functions

BlockMove	Copies a block of data in RAM
FlipBits	Generates mirror image of bit pattern
GetBit	Reads a single bit from a variable
GetEEPROM	Reads data from EEPROM
MemAddress	Returns the address of a variable or array
MemAddressU	Returns the address of a variable or array
PersistentPeek	Reads a byte from EEPROM
PersistentPoke	Writes a byte to EEPROM
PutBit	Writes a single bit to a variable
PutEEPROM	Writes data to EEPROM
RAMpeek	Reads a byte from RAM
RAMpoke	Writes a byte to RAM
SerialNumber	Returns the BasicX version number

Queues

GetQueue	Reads data from a queue
OpenQueue	Defines an array as a queue
PeekQueue	Looks at queue data without removing any data
PutQueue	Writes data to a queue
PutQueueStr	Writes a string to a queue
StatusQueue	Determines if a queue has data available for reading

Real time clock

GetDate	Returns the date
GetDayOfWeek	Returns the day of week
GetTime	Returns the time of day
GetTimestamp	Returns the date and time of day
PutDate	Sets the date
PutTime	Sets the time of day
PutTimestamp	Sets the date, day of week and time of day
Timer	Returns floating point seconds since midnight

Type conversions

CBool	Convert Byte to Boolean
CByte	Convert to Byte
CInt	Convert to Integer
CLng	Convert to Long
CSng	Convert to floating point (single)
CStr	Convert to string
CuInt	Convert to UnsignedInteger
CuLng	Convert to UnsignedLong
FixB	Truncates a floating point value, converts to Byte
FixI	Truncates a floating point value, converts to Integer
FixL	Truncates a floating point value, converts to Long
FixUI	Truncates a floating point value, converts to UnsignedInteger
FixUL	Truncates a floating point value, converts to UnsignedLong

Tasking

CallTask	Starts a task
CPUsleep	Puts the processor in various low-power modes
Delay	Pauses task and allows other tasks to run
DelayUntilClockTick	Pauses task until the next tick of the real time clock
FirstTime	Determines whether the program has ever been run since download
LockTask	Locks the task and discourages other tasks from running
OpenWatchdog	Starts the watchdog timer
ResetProcessor	Resets and reboots the processor
Semaphore	Coordinates the sharing of data between tasks
Sleep	Pauses task and allows other tasks to run
TaskIsLocked	Determine whether a task is locked
UnlockTask	Unlocks a task
WaitForInterrupt	Allows a task to respond to a hardware interrupt
Watchdog	Resets the watchdog timer

