Artificial Intelligence Branch

Computer science Dep

University of Technology 2023-2024

By

Prof. Dr. Alia Karim Abdulhassan

The BASIC Stamp Microcontroller

1.1 Microcontroller

A microcontroller is essentially an inexpensive single-chip computer. Single chip means the entire computer system lies within the confines of a sliver of silicon encapsulated inside the plastic housing of an integrated circuit. The microcontroller has features similar to those of a standard personal computer.

The microcontroller contains:
a CPU (central processing unit),
RAM (random access memory),
ROM (read-only memory),
I/O (input/output) lines,
serial and parallel ports,
timers,
and sometimes other built-in peripherals such as ana- log-to-digital (A/D) and digital-to-analog (D/A) converters.

The key feature, however, is the microcontroller's capability of uploading, storing, and running a program.

1.2 Why Use a Microcontroller?

Being inexpensive single-chip computers, microcontrollers are easy to embed into larger electronic circuit designs. Their ability to store and run unique programs makes them extremely versatile. For instance, one can program a microcontroller to make decisions and perform functions based on situations (I/O line logic) and events. The math and logic functions allow the microcontroller to mimic sophisticated logic and electronic circuits.

Programs can also make the microcontroller behave as a neural network and/or a fuzzy logic controller. Microcontrollers are incorporated in consumer electronics and are responsible for the "intelligence" in these smart electronic devices.

1.3 The Compiler

There are a number of compilers on the market that allow users to write programs (code) in different high-level languages. High-level language frees the programmer from wrestling with and controlling the microcontroller's registers when writing code and accessing the different aspects of the microcontroller's features and memory.

The compiler reads through the text file and creates (compiles) an equivalent machine code instruction listing (.hex file) of the program. The machine code (.hex file) is a list of hexadecimal numbers that represent the Basic program. The list of hexadecimal numbers (.hex file) is uploaded (programmed) into the microcontroller.

1.4 Firmware

Many writers use the term *firmware.* This word is used when software is embedded in a hardware device that can read and execute by the device but cannot be modified. So when our program (software) is embedded (uploaded) into the microcontroller, it may be referred to as firmware. Other phrases may include the term *firmware* instead of *software,* such as “upload the firmware” or “once the firmware has been installed into the device.”

1.5 Consumables

Consumables are the electronic components, the microcontroller chip itself, with a few support components to get the microcontroller up and running.

2. Boe-Bot’s Brain

The Boe-Bot and a close-up of its BASIC Stamp® 2 programmable microcontroller brain are shown in Figure 2.1. The BASIC Stamp 2 module is both powerful and easy to use, especially with a robot.

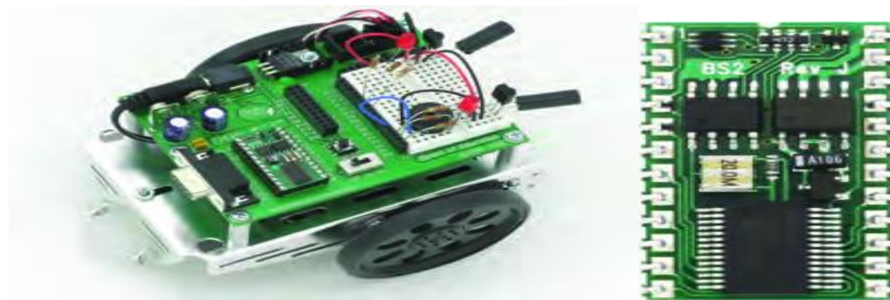


Figure 2-1 BASIC Stamp Module on a Boe-Bot Robot

Through writing simple programs that make the BASIC Stamp and your Boe-Bot do four essential robotic tasks:
1. Monitor sensors to detect the world around it
2. Make decisions based on what it senses
3. Control its motion (by operating the motors that make its wheels turn)
4. Exchange information with its Roboticist (that will be you!)

2.1 The PBASIC programming language

The programming language you will use to accomplish these tasks is called **PBASIC**, which stands for:
· Parallax - Company that invented and manufactures BASIC Stamp microcontrollers
· All-purpose - Powerful and useful for solving many different kinds of problems
· Symbolic - Using symbols (terms that resemble English word/phrases)
· Instruction - To tell a computer what to do ·
Code - In terms that the computer (and you) can understand

2.2 The BASIC Stamp Microcontroller

It's a programmable device that is designed into your digital wristwatch, cell phone, calculator, clock radio, etc. In these devices, the microcontroller has been programmed to sense when you press a button, make electronic beeping noises, and control the device's digital display.

They are also built into factory machinery, cars, submarines, and spaceships because they can be programmed to read sensors, make decisions, and orchestrate devices that control moving parts.

Student Guide is the recommended first text for beginners. It is full of examples of how to use microcontrollers, and how to make the BASIC Stamp the brain of your own microcontrolled inventions.
It's available for free download from www.parallax.com/go/WAM, and it's also included in the BASIC Stamp Editor Help as a PDF file.

2.3 THE SOFTWARE

The BASIC Stamp Editor (version 2.5 or higher) is the software you will use in most of the activities and projects in this text. You will use this software to write programs that the BASIC Stamp module will run. You can also use this software to display messages sent by the BASIC Stamp that help you understand what it senses.

2.4 Computer System Requirements

You will need a personal computer to run the BASIC Stamp Editor software. Your computer will need to have the following features:
· Microsoft Windows 2K/XP/Vista/7 or newer operating system
· An available serial or USB port
· Internet access and an Internet browser program

2.5 Running the BASIC Stamp Editor for the first time

ü If you see the BASIC Stamp Editor icon on your computer desktop, double-click it (Figure 2.2).
ü Or, click on your computer's Start menu, then choose All Programs 4 Parallax Inc 4 BASIC Stamp Editor 2.5 4 BASIC Stamp Editor 2.5.

Figure 2.2 BASIC Stamp Editor Desktop Icon *Double-click to launch the program.*

Questions

1. What device will be the brain of your Boe-Bot?

Solution:

A BASIC Stamp 2 microcontroller module.

2. When the BASIC Stamp sends a character to your PC/laptop, what type of numbers are used to send the message through the programming cable?

Solution:

Binary numbers, that is, 0's and 1's.

3. What is the name of the window that displays messages sent from the BASIC Stamp to your PC/laptop?

Solution:

The Debug Terminal.

4.Explain what the asterisk does in this command: **DEBUG DEC 7 * 11**

5. There is a problem with these two commands. When you run the code, the numbers they display are stuck together so that it looks like one large number instead of two small ones. Modify these two commands so that the answers appear on different lines in the Debug Terminal.

```
DEBUG DEC 7 * 11
DEBUG DEC 7 + 11
```

Solution:

The Debug Terminal would display: 18 E3. To fix the problem, add a carriage return using the **CR** control character and a comma.

```
DEBUG DEC 7 * 11
DEBUG CR, DEC 7 + 11
```

6. Use **DEBUG** to display the solution to the math problem: 1 + 2 + 3 + 4. 2. Save FirstProgramYourTurn.bs2 under another name. If you were to place the **DEBUG** command shown below on the line just before the **END** command in the program, what other lines could you delete and still have it work the same?

Solution:

is a program to display a solution to the math problem: 1+2+3+4.

```
'{$STAMP BS2}
'{$PBASIC 2.5}
DEBUG "What's 1+2+3+4?"
DEBUG CR, "The answer is: "
DEBUG DEC 1+2+3+4
END
```

7.Modify the copy of the program to test your hypothesis (your prediction of what will happen).

```
DEBUG "What's 7 X 11?", CR, "The answer is: ", DEC 7 *
11
```

<u>Solution:</u>

The last three **DEBUG** lines can be deleted. An additional **CR** is needed after the "Hello" message.

```
' {$STAMP BS2}
' {$PBASIC 2.5}
DEBUG "Hello, it's me, your BASIC Stamp!", CR
DEBUG "What's 7 X 11?", CR, "The answer is: ", DEC 7 *
11
END
```

The output from the Debug Terminal is:

```
Hello, it's me, your BASIC Stamp!
What's 7 X 11?
The answer is: 77
```

This output is the same as it was with the previous code. This is an example of using commas to output a lot of information, using only one **DEBUG** command with multiple elements in it.

Displaying Messages at Human Speeds

PAUSE command to tell the BASIC Stamp to wait for a while before executing the next command.

PAUSE *Duration*

PAUSE command
Duration argument, how long it should wait before moving on to the next command.

The units for the ***Duration*** argument are thousandths of a second (ms).
one second=1000.

PAUSE 1000

If you want to wait for twice as long, try: PAUSE 2000

Note:
A second “s.” :1 s, it means one second.
A millisecond “ms.”

PAUSE 1000 delays the program for 1000 ms, which is 1000/1000 of a second, which is one second, or 1 s.

Example: Show how the PAUSE command can be used to display messages at human speeds.

```
' {$STAMP BS2}
' {$PBASIC 2.5}
DEBUG "Start timer..."
PAUSE 1000
DEBUG CR, "One second elapsed..."
PAUSE 2000
DEBUG CR, "Three seconds elapsed..."
DEBUG CR, "Done."
END
```

The longest possible ***Duration*** argument is 65535. If you've got a minute to spare, try **PAUSE 60000**.

DO and LOOP :

```
DO
DEBUG "Hello!", CR
PAUSE 1000
LOOP
```

Example: Display a message once every second.

```
' {$STAMP BS2}
' {$PBASIC 2.5}
DO
DEBUG "Hello!", CR PAUSE 1000
LOOP
```

PULSOUT that can deliver high signals for precise amounts of time.

PULSOUT *Pin*, *Duration*

A microsecond is a millionth of a second. It's abbreviated µs.
For example, 8 microseconds is abbreviated 8 µs.

You can send a **HIGH** signal that turns the P13 LED on for 2 µs (that's two millionths of a second) by using this command:

```
PULSOUT 13, 1
```

This command would turn the LED on for 4 µs:

```
PULSOUT 13, 2
```

This command sends a high signal that you can actually view:

```
PULSOUT 13, 65000
```

How long does the LED circuit connected to P13 stay on when you send this pulse?

Let's figure it out. The time it stays on is 65000 times 2 µs.
That's:
Duration $65000 * 2\mu s = 65000\ 0.000002s = 0.13\ s$
which is still pretty fast, thirteen hundredths of a second.

Example:' Send a 0.13 second pulse to the LED circuit connected to P13 every 2 s.

```
' {$STAMP BS2}
' {$PBASIC 2.5}
DEBUG "Program Running!"
DO
PULSOUT 13, 65000
PAUSE 2000
LOOP
```

Viewing the Full Speed Servo Signal

Remember the servo signal is 100 times as fast as the program you just ran. First, let's try running the program ten times as fast.

That means divide all the ***Duration*** arguments (**PULSOUT** and **PAUSE**) by 10. ü Modify the program so that the commands look like this:

```
DO
PULSOUT 13, 6500
PULSOUT 12, 6500
PAUSE 200
LOOP
```

Run it and verify that it makes the LEDs blink ten times as fast. Modify the program so that the commands look like this:

```
DO
PULSOUT 13, 650
PULSOUT 12, 650
PAUSE 20
LOOP
```

Run the modified program and verify that it makes both LEDs about the same brightness.
Try substituting 850 in the ***Duration*** argument for the P13 **PULSOUT** command.

```
DO
PULSOUT 13, 850
PULSOUT 12, 650
PAUSE 20
LOOP
```

Run the modified program and verify that the P13 LED now appears slightly brighter than the P12 LED. You may have to cup your hands around the LEDs and peek inside to see the difference. They differ because the amount of time the P13 LED stays on is longer than the amount of time the P12 LED stays on.

Try substituting 750 in the ***Duration*** argument for both the **PULSOUT** commands.

```
DO
PULSOUT 13, 750
PULSOUT 12, 750
PAUSE 20
LOOP
```

Run the modified program and verify that the brightness of both LEDs is the same again. It may not be obvious, but the brightness level is between those given by ***Duration*** arguments of 650 and 850.

Run the servo connected to P13 at full speed counterclockwise and the servo connected to P12 at full speed clockwise.

```
' {$STAMP BS2}
' {$PBASIC 2.5}
DEBUG "Program Running!"
DO
PULSOUT 13, 850
PULSOUT 12, 650
PAUSE 20
LOOP
```

Run the servos for a certain amount of time

Let's say you want to run both servos, the P13 servo at a pulse width of 850 and the P12 servo at a pulse width of 650. Now, each time through the loop, it will take:

1.7ms – Servo connected to P13
1.3 ms – Servo connected to P12
20 ms – Pause duration
1.6 ms – Code overhead ---------

24.6 ms – Total

If you want to run the servos for a certain amount of time, you can calculate it like this:

Number of pulses = Time s / 0.0246 s = Time / 0.0246
Lets' say we want to run the servos for 3 seconds. That's:
Number of pulses = 3 / 0.0246 = 122

Now, you can use the value 122 in the ***EndValue*** of the **FOR…NEXT** loop, and it will look like this:

```
FOR counter = 1 TO 122
PULSOUT 13, 850
PULSOUT 12, 650
PAUSE 20
NEXT
```

Example:
Run both servos in opposite directions for three seconds, then reverse ' the direction of both servos and run another three seconds.

```
' {$STAMP BS2}
' {$PBASIC 2.5}
DEBUG "Program Running!"
counter VAR   Byte
FOR counter = 1 TO 122
PULSOUT 13, 850
PULSOUT 12, 650
```

```
PAUSE 20
NEXT
FOR counter = 1 TO 122
PULSOUT 13, 650
PULSOUT 12, 850
PAUSE 20
NEXT
END
```

Question

First, calculate the number of loops needed to get the servos to run for three seconds, for each combination of rotation. As given on page 65, the code overhead is 1.6 ms.

Four combinations (1,2,3,4).
Each combination: Determine **PULSOUT** ***Duration*** arguments:

1. Both counterclockwise: 12, 850 and 13, 850 2.
2. Both clockwise: 12, 650 and 13, 650
3. 12 CW and 13 CCW: 12, 850 and 13, 650
4. 12 CCW and 13 CW: 12, 650 and 13, 850

Each combination: Calculate how long it will take for one loop:
1. one loop = 1.7 + 1.7 + 20 ms + 1.6 = 25.0 ms = 0.025 s
2. one loop = 1.3 + 1.3 + 20 ms + 1.6 = 24.2 ms = 0.0242 s
3. one loop = 1.7 + 1.3 + 20 ms + 1.6 = 24.6 ms = 0.0246 s
4. one loop = 1.3 + 1.7 + 20 ms + 1.6 = 24.6 ms = 0.0246 s
Each combination: Calculate number of pulses needed for 3 s of running:
1. number of pulses = 3 s / 0.025 s = 120
2. number of pulses = 3 s / 0.0242 s = 123.9 = 124
3. number of pulses = 3 s / 0.0246 s = 121.9 = 122
4. number of pulses = 3 s / 0.0246 s = 121.9 = 122

Move servos through 4 clockwise/counterclockwise rotation ' combinations.

```
'{$STAMP BS2}
'{$PBASIC 2.5}
DEBUG "Program Running!"
counter        VAR     Word
FOR counter = 1 TO 120          ' Loop for three seconds
PULSOUT 13, 850               ' P13 servo counterclockwise
PULSOUT 12, 850               ' P12 servo counterclockwise PAUSE 20
NEXT

FOR counter = 1 TO 124          ' Loop for three seconds
PULSOUT 13, 650               ' P13 servo clockwise
PULSOUT 12, 650               ' P12 servo clockwise
PAUSE 20
NEXT

FOR counter = 1 TO 122          ' Loop for three seconds
PULSOUT 13, 650               ' P13 servo clockwise
PULSOUT 12, 850               ' P12 servo counterclockwise PAUSE 20
NEXT

FOR counter = 1 TO 122          ' Loop for three seconds
PULSOUT 13, 850               ' P13 servo counterclockwise
PULSOUT 12, 650               ' P12 servo clockwise PAUSE 20
NEXT
END
```

Boe-Bot Navigation

BASIC BOE-BOT MANEUVERS

Figure 4-1 shows your Boe-Bot's front, back, left, and right. When the Boe-Bot goes forward, in the picture, it would have to roll to the right edge of the page. Backward would be toward the left edge of the page. A left turn would be make the Boe-Bot ready to drive off the top of the page, and a right turn would have it facing the bottom of the page.

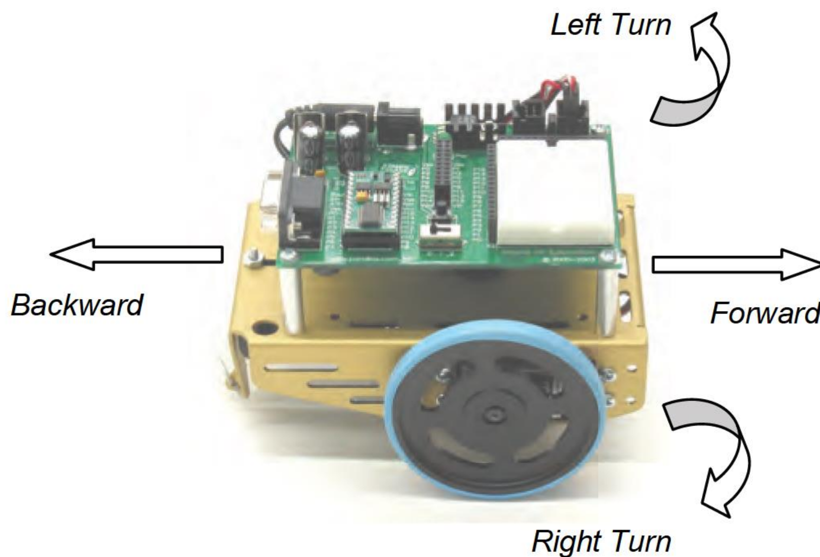


Figure 4-1
Your Boe-Bot and Driving Directions

<u>Boe-Bot go forward,</u>

The Boe-Bot's left wheel has to turn counterclockwise, but its right wheel has to turn clockwise.

e

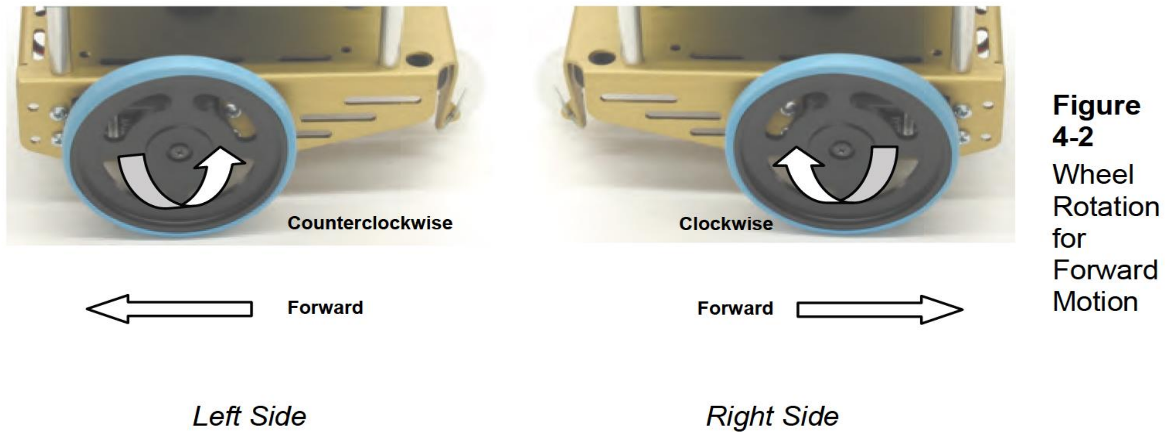


Figure 4-2 Wheel Rotation for Forward Motion

Definitions:

it generates a tone to signal the start of the program. It will be used in all programs that run the servos.

```
FREQOUT 4, 2000, 3000          ' Signal program start/reset.
```

Example:

This **FOR…NEXT** loop sends 122 sets of pulses to the servos, one each to P13 and P12, pausing for 20 ms after each set and then returning to the top of the loop.

```
FOR counter = 1 TO 122
PULSOUT 13, 850
PULSOUT 12, 650
PAUSE 20
NEXT
```

PULSOUT 13, 850 causes the left servo to rotate counterclockwise while **PULSOUT 12, 650** causes the right servo to rotate clockwise. Therefore, both wheels will be turning toward the front end of the Boe-Bot, causing it to drive forward. It takes about 3 seconds for the **FOR…NEXT** loop to execute 122 times, so the Boe-Bot drives forward for about 3 seconds.

example: Make the Boe-Bot roll forward for three seconds.

```
' {$STAMP BS2}
' {$PBASIC 2.5}
DEBUG "Program Running!"
counter       VAR     Word
FREQOUT 4, 2000, 3000                  ' Signal program start/reset.
FOR counter = 1 TO 122                 ' Run servos for 3 seconds.
PULSOUT 13, 850
PULSOUT 12, 650
PAUSE 20
NEXT
END
```

Q: Run the program and verify that it ran at half the time and covered half the distance.
Q: Try these steps over again, but this time, change the **FOR…NEXT** loop's ***EndValue*** to 244.

Example:
Modify your program with these **PULSOUT** commands:

```
PULSOUT 13, 780
PULSOUT 12, 720
```

Then Run the program, and verify that your Boe-Bot moves slower.

Moving Backward

These two **PULSOUT** commands can be used to make your Boe-Bot go backwards:
PULSOUT 13, 650
PULSOUT 12, 850

Rotate left

These two commands will make your Boe-Bot rotate in a left turn counterclockwise as you are looking at it from above):

PULSOUT 13, 650
PULSOUT 12, 650

Rotate right

These two commands will make your Boe-Bot rotate in a right turn (clockwise as you are looking at it from above):

PULSOUT 13, 850
PULSOUT 12, 850

Example : Move forward, left, right, then backward for testing and tuning. ' {$STAMP BS2}

```
' {$PBASIC 2.5}

DEBUG "Program Running!"
counter     VAR    Word

FREQOUT 4, 2000, 3000                ' Signal program start/reset.

FOR counter = 1 TO 64               ' Forward
PULSOUT 13, 850
PULSOUT 12, 650
PAUSE 20
NEXT

PAUSE 200

FOR counter = 1 TO 24               ' Rotate left - about 1/4 turn
PULSOUT 13, 650
PULSOUT 12, 650
PAUSE 20
NEXT
```

```
PAUSE 200
FOR counter = 1 TO 24                ' Rotate right - about 1/4 turn
PULSOUT 13, 850
PULSOUT 12, 850
PAUSE 20
NEXT

PAUSE 200
FOR counter = 1 TO 64                ' Backward
PULSOUT 13, 650
PULSOUT 12, 850
PAUSE 20
NEXT
END
```

Your Turn – Pivoting

You can make the Boe-Bot turn by pivoting around one wheel. The trick is to keep one wheel still while the other rotates.

1) keep the left wheel still and make the right wheel turn clockwise (forward), the Boe-Bot will pivot to the left.

```
PULSOUT 13, 750
PULSOUT 12, 650
```

2) If you want to pivot forward and to the right, simply stop the right wheel, and make the left wheel turn counterclockwise (forward).

```
PULSOUT 13, 850
PULSOUT 12, 750
```

3) These are the **PULSOUT** commands for pivoting backwards and to the right.

```
PULSOUT 13, 650
PULSOUT 12, 750
```

4) Finally, these are the **PULSOUT** commands for pivoting backwards and to the left.

```
PULSOUT 13, 750
PULSOUT 12, 850
```

Example: Make the Boe-Bot roll forward for ten seconds.

```
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"
counter     VAR   Word
FREQOUT 4, 2000, 3000              ' Signal program start/reset.
FOR counter = 1 TO 407             ' Number of pulses – run time.
PULSOUT 13, 850                    ' Left servo full speed ccw.
PULSOUT 12, 650                    ' Right servo full speed cw.
PAUSE 20
NEXT
END
```

Adjusting Servo Speed to Straighten the Boe-Bot's Path

If your Boe-Bot goes perfectly straight, try this example anyway. If you follow the instructions, it should adjust your Boe-Bot so that it curves slightly to the right. Let's say that the Boe-Bot turns slightly to the left. There are two ways to think about this problem: either the left wheel is turning too slowly, or the right wheel is turning too quickly. Since the Boe-Bot is already at full speed, speeding up the left wheel isn't going to be practical, but slowing down the right wheel should help remedy the situation.

Remember that servo speed is determined by the **PULSOUT** command's ***Duration*** argument. The closer the ***Duration*** is to 750, the slower the servo turns. This means you should change the 650 in the command **PULSOUT 12,650** to something a little closer to 750.

If the Boe-Bot is only just a little off course, maybe **PULSOUT 12,663** will do the trick. If the servos are severely mismatched, maybe it needs to be **PULSOUT 12,690**.

It will probably take several tries to get the right value. Let's say that your first guess is that **PULSOUT 12,663** will do the trick, but it turns out not to be enough because the Boe- Bot is still turning slightly to the left.

So try **PULSOUT 12,670**. Maybe that overcorrects, and it turns out that **PULSOUT 12,665** gets it exactly right. This is called an iterative process, meaning a process that takes repeated tries and refinements to get to the right value.

If your Boe-Bot curved to the right instead of the left, it means you need to slow down the left wheel by reducing the ***Duration*** of 850 in the **PULSOUT 13,850** command. Again, the closer this value gets to 750, the slower the servo will turn.

- Modify BoeBotForwardTenSeconds.bs2 so that it makes your Boe-Bot go straight forward.
- Use masking tape or a sticker to label each servo with the best **PULSOUT** values.
- If your Boe-Bot already travels straight forward, try the modifications just discussed to see the effect. It should cause the Boe-Bot to travel in a curve instead of a straight line.

You might find that there's an entirely different situation when you program your Boe- Bot to roll backward.

ü Modify BoeBotForwardTenSeconds.bs2 so that it makes the Boe-Bot roll backward for ten seconds. ü Repeat the test for straight line. ü Repeat the steps for correcting the **PULSOUT** command's ***Duration*** argument to straighten the Boe-Bot's backward travel.

Tuning the Turns

Software adjustments can also be made to get the Boe-Bot to turn to a desired angle, such as 90°. The amount of time the Boe-Bot spends rotating in place determines how far it turns. Because the **FOR…NEXT** loop controls run time, you can adjust the **FOR…NEXT** loop's ***EndValue*** argument to get very close to the turning angle you want.

Here's the left turn routine from ForwardLeftRightBackward.bs2:

```
FOR counter = 1 TO 24        ' Rotate left - about 1/4 turn PULSOUT 13, 650
PULSOUT 12, 650
PAUSE 20
NEXT
```

Let's say that the Boe-Bot turns just a bit more than 90° (1/4 of a full circle). Try **FOR counter = 1 TO 23**, or maybe even **FOR counter = 1 TO 22**. If it doesn't turn far enough, increase the run time of the rotation by increasing the **FOR…NEXT** loop's ***EndValue*** argument to whatever value it takes to complete the quarter turn.

If you find yourself with one value slightly overshooting 90° and the other slightly undershooting, try choosing the value that makes it turn a little too far, then slow down the servos slightly. In the case of the rotate left, both **PULSOUT** ***Duration*** arguments should be changed from 650 to something a little closer to 750. As with the straight line exercise, this will also be an iterative process.

Your Turn – 90° Turns

ü Modify ForwardLeftRightBackward.bs2 so that it makes precise 90° turns. ü Update ForwardLeftRightBackward.bs2 with the **PULSOUT** values that you determined for straight forward and backward travel. ü Update the label on each servo with a notation about the appropriate ***EndValue*** for a 90° turn.

Carpeting can cause navigation errors. If you are running your Boe-Bot on carpeting, don't expect perfect results! A carpet is a bit like a golf green—the way the carpet pile is inclined can affect the way your Boe-Bot travels, especially over long distances. For more precise maneuvers, use a smooth surface.

ACTIVITY #3: CALCULATING DISTANCES

In many robotics contests, more precise robot navigation lends itself to better scores. One popular entry level robotics contest is called dead reckoning. The entire goal of this contest is to make your robot go to one or more locations and then return to exactly where it started.

Data types

variableName* VAR *Size
You can declare four different sizes of variables in PBASIC:
Size – Stores
Bit – 0 to 1
Nib – 0 to 15
Byte – 0 to 255
Word – 0 to 65535,
or -32768 to + 32767

Example Program: Make the Boe-Bot roll forward for one second.

```
' {$STAMP BS2}
' {$PBASIC 2.5}
DEBUG "Program Running!" counter        VAR     Word FREQOUT 4, 2000, 3000
' Signal program start/reset.
FOR counter = 1 TO 41 PULSOUT 13, 850
PULSOUT 12, 650
PAUSE 20
NEXT
END
```

Example Circle: Boe-Bot navigates a circle of 1 unit diameter.

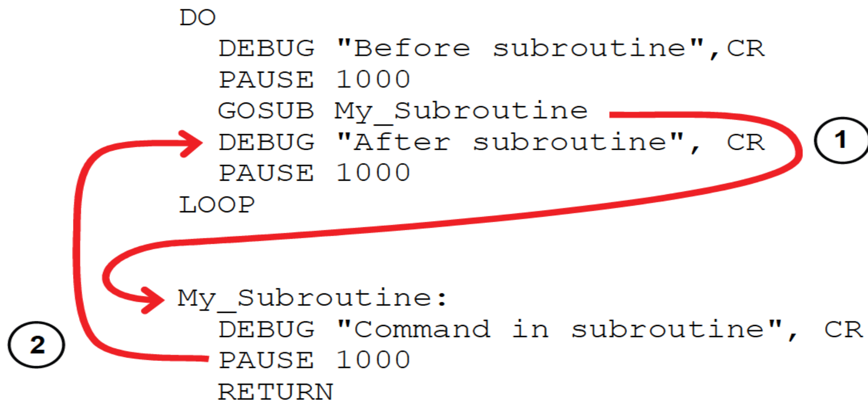
```
'{$STAMP BS2}
'{$PBASIC 2.5}
DEBUG "Program running!"
pulseCount     VAR     Word    ' Pulse count to servos

FREQOUT 4, 2000, 3000              ' Signal program start/reset.
DO
PULSOUT 13, 850                 ' Veer right PULSOUT 12, 716
PAUSE 20
LOOP
```

SUBROUTINES

GOSUB label // the program jumps to the label
RETURN // jumps back to the command immediately after the GOSUB command

Example:



Example This program demonstrates a simple subroutine call.

```
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Before subroutine", CR
PAUSE 1000
GOSUB My_Subroutine
DEBUG "After subroutine", CR
END

My_Subroutine:

DEBUG "Command in subroutine", CR
PAUSE 1000
RETURN
```

Example : This program demonstrates that a subroutine is a reusable block of commands.

```
' {$STAMP BS2}
' {$PBASIC 2.5}

DO
GOSUB High_Pitch
DEBUG "Back in main", CR
PAUSE 1000
GOSUB Low_Pitch
DEBUG "Back in main again", CR
```

```
PAUSE 1000
DEBUG "Repeat...", CR, CR

LOOP

High_Pitch:
DEBUG "High pitch", CR
FREQOUT 4, 2000, 3500
RETURN
Low_Pitch:
DEBUG "Low pitch", CR
FREQOUT 4, 2000, 2000
RETURN
```

Example: Make forward, left, right, and backward movements in reusable subroutines.

```
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"
counter     VAR    Word
FREQOUT 4, 2000, 3000                    ' Signal program start/reset.
GOSUB Forward
GOSUB Left
GOSUB Right
GOSUB Backward END

Forward:

FOR counter = 1 TO 64 PULSOUT 13, 850
PULSOUT 12, 650
PAUSE 20
NEXT
PAUSE 200
RETURN


Left:

FOR counter = 1 TO 24
PULSOUT 13, 650
PULSOUT 12, 650
PAUSE 20
NEXT
PAUSE 200
RETURN
```

Right:

```
FOR counter = 1 TO 24
PULSOUT 13, 850
PULSOUT 12, 850
PAUSE 20
NEXT
PAUSE 200
RETURN
```

Backward:

```
FOR counter = 1 TO 64
PULSOUT 13, 650
PULSOUT 12, 850
PAUSE 20
NEXT
RETURN
```

Exerscie:

P13	P12	Description	Behavior
850	650	Full Speed: P13 CCW, P12 CW	Forward
650	850	Full Speed: P13 CW, P12 CCW	Backward
850	850	Full Speed: P13 CCW, P12 CCW	Right rotate
650	650	Full Speed: P13 CW, P12 CW	Left rotate
750	850	P13 Stopped, P12 CCW Full speed	Pivot back left
650	750	P13 CW Full Speed, P12 Stopped	Pivot back right
750	750	P13 Stopped, P12 Stopped	Stopped
760	740	P13 CCW Slow, P12 CW Slow	Forward slow
770	730	P13 CCW Med, P12 CW Med	Forward medium
850	700	P13 CCW Full Speed, P12 CW Medium	Veer right
800	650	P13 CCW Medium, P12 CW Full Speed	Veer left