Robot Projects for RobotBASIC
Volume I: The Fundamentals

Copyright February 2014 by John Blankenship
All rights reserved

Project 1: Controlling the Robot's Movements

This project, as with all projects in *Volume I*, assumes the reader has some basic knowledge of RobotBASIC. If you are totally new to RobotBASIC, it is suggested that you watch some of the many RobotBASIC YouTube videos available from www.RobotBASIC.org to give you a little help getting started. Those that desire more background information should consider *RobotBASIC Projects for Beginners* and *Robots in the Classroom*. These and other RobotBASIC books (as shown in Figure 1.1) are available from www.Amazon.com. Quantity discounts for schools are available on some books ordered directly from RobotBASIC.

Figure 1.1

Readers should not feel that they have to purchase additional books to use RobotBASIC, though. All of the Projects in this series will attempt provide *adequate* information for most readers. Due to the nature of these projects though, more advanced students may want to read some of our other books or research some topics on their own. If you find areas that are confusing for typical students, please let us know by emailing RobotBASIC@yahoo.com. Finally, RobotBASIC comes with an extensive HELP system that provides an enormous amount of information so use it if you ever need more details about a command than the text provides. You can access the HELP by pressing the **?** in the blue circle on the RobotBASIC tool bar.

The projects for RobotBASIC RB-9 robot are organized into Volumes that should be used in order. Each project within a Volume introduces new topics that will be needed in later projects. Likewise, all the projects in Volume II will assume that you have performed and understand all of the projects in Volume I.

Electronic versions of the projects from Volume I are now available for free download from the Education Tab at www.RobotBASIC.org. Some schools or even individual students may wish to have the convenience of printed versions of these projects. For that reason, standard print versions of each Volume will be made available from www.Amazon.com. The printed versions will have full code for some of the programs suggested for further study, so they can serve as a Teacher's Manual.

Starting RobotBASIC

To get started with RobotBASIC, visit www.RobotBASIC.org and visit the FREE PROGRAM DOWNLOAD tab. Scroll down slightly and click the link ZIPPED DEMOS & EXE to download a large zip file that includes RobotBASIC, the help file, and many demo programs. Save the file to a suitable place such as your DOCUMENTS folder and unzip the file so that you have access to RobotBASIC. Open the new folder and click on the RobotBASIC EXE to start the program. You may want to create a desktop shortcut to make starting RobotBASIC easier in the future.

When RobotBASIC executes the first time, it will ask you to review and accept the license which generally states that RobotBASIC is free, and that you can use it but not sell it to others. At that point you will see the RobotBASIC editor screen as shown in Figure 1.2.

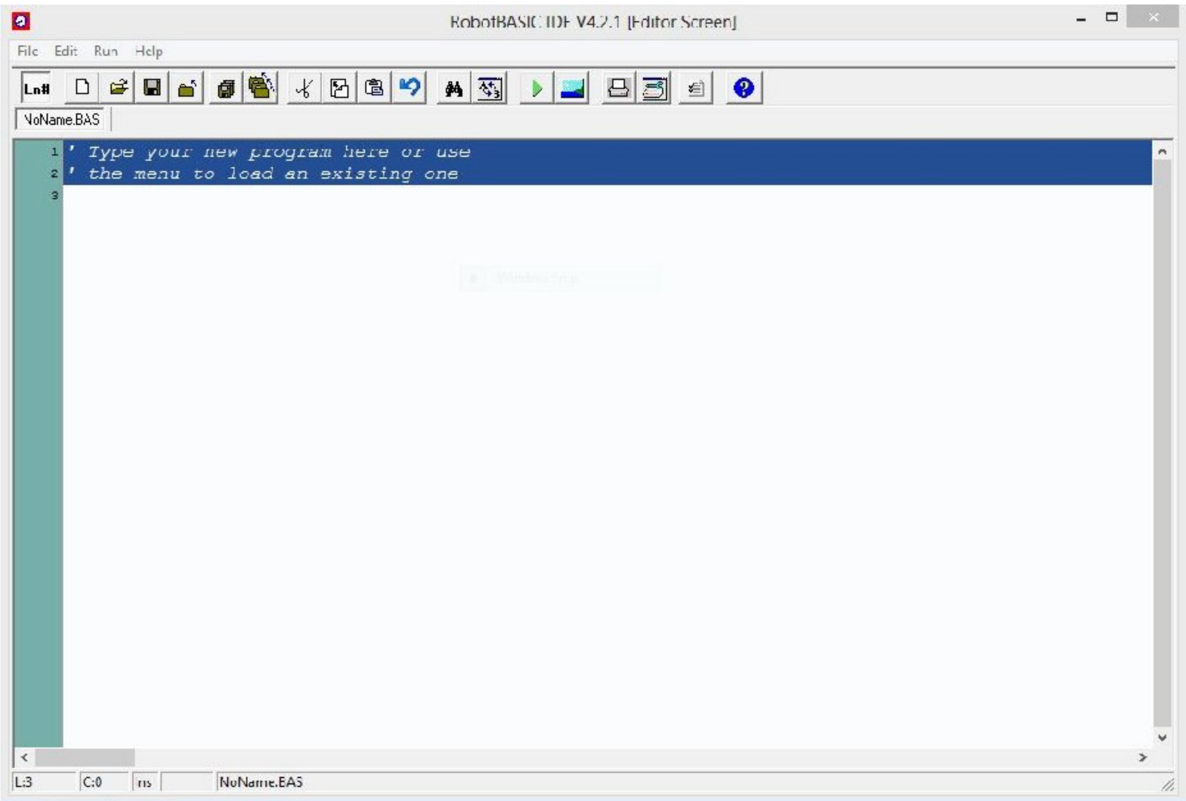

Figure 1.2: This is the startup editor screen for RobotBASIC

The tool bar at the top of the RobotBASIC editor window offers many shortcuts to items in the main menus. You can, for example, refer to the HELP file by pressing the **?** on the toolbar instead of choosing COMMAND SYNTAX from the HELP menu.

When you write a program you type it into the text area just as you would with any word processor or notepad. You can execute the program by using the menus or pressing the green triangle in the tool bar. For example, type in the short program in Figure 1.3 and then run it by clicking the green triangle.

```
x=5
y=10
a=x+y-3
print a
end
```

Figure 1.3: Type this program into the RobotBASIC editor.

When a computer program runs, it executes each statement in order unless control statements cause statements or sections of the program to be repeated or ignored. The program in Figure 1.3 will print 12 as expected. This printing, as does all output from

RobotBASIC programs occurs in a new output window (sometimes called the terminal window). Closing the output window returns you to the editor.

If you spell something wrong (**`prit`** instead of **`print`**, for example) RobotBASIC will issue an error. Closing the error window will return you to the editor and highlight the offending line so that you can examine it and fix the problem.

Variables, such as **`x`**, **`y`**, and **`a`**, in this example program are case sensitive – that is **`a`** is not the same as **`A`**. RobotBASIC *commands* (such as **`print`**) are not case sensitive, so **`print`** is the same as **`Print`**. You cannot have a variable named the same as command, but you could, for example, have a variable named **`MyPrint`** or **`print1`**.

You can save or retrieve (open) programs using the FILE menu or the toolbar icons, again much the same way as you would when using a word processor.

A Robot Simulator

One of the great features of RobotBASIC is its integrated Robot Simulator. The simulated robot is easy to use and provides sensory capabilities far beyond that of most educational robots – actually more than almost all robots currently on the market. In this project, you will learn how to control the basic movements of the simulated robot and then how to use the very same programs to control a real RobotBASIC robot.

The simulated robot can be initialized at a specific **`x,y`** position on RobotBASIC's output or terminal screen with the following statement.

```
rLocate 400,300
```

Notice that the command starts with the letter **`r`**. All the robot-related commands in RobotBASIC start with an **`r`**. Type the above command into RobotBASIC's program space (as a new program) and run the program by pressing the GREEN triangle near the top of the main RobotBASIC screen. When you do, you will see the simulated robot appear at the center of the output screen (which has dimensions of 800x600 pixels) as shown in Figure 1.4.

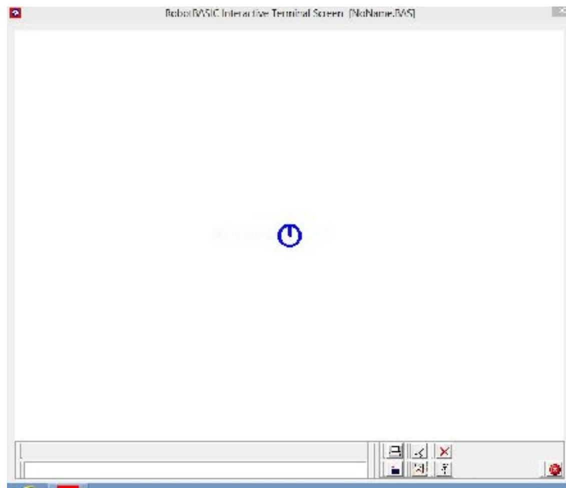

Figure 1.4: The robot has been initialized at the center of the output screen.

After the robot has been initialized, it can be moved around the screen as demonstrated by the program in Figure 1.5. The simulated robot is 40 pixels in diameter, so the **`rForward 120`** command will move it a distance equal to three times its diameter.

```
rLocate 400,300
rForward 120
rTurn 90
```

Figure 1.5: This program moves the robot a distance equal to three times its diameter, then turns it 90° to the right.

You can slow the robot down by adding the following command between the **`rLocate`** and the **`rForward`** commands. The number given is a delay so larger numbers will make the robot move slower.

```
rSpeed 5
```

Consider that the robot will move in a square if the last two commands in Figure 1.5 are repeated three more times. We could, of course, just type the commands in four times, but computer languages have flow-control structures that allow the creation of loops that can execute a series of commands multiple times. Figure 1.6, for example, will repeat the forward and turn commands four times, causing the program to move the robot in a square pattern.

```
rLocate 400,300
for n = 1 to 4
  rForward 120
  rTurn 90
next
```

Figure 1.6: A loop is used to repeat the movement commands four times, causing the robot to move in square.

Leaving a Trail

We can tell the robot to drop a pen and draw a line as it moves by adding the statements shown in Figure 1.7. The **`rInvisible`** command tells the robot to ignore objects of color **`GREEN`**. Without this command, the robot will see the line it is drawing as an object in the room. This will cause an error because the robot will think it has collided with an object. The **`LineWidth`** statement forces the line being drawn to be four pixels wide, making it easier to see. The **`rPen`** statement lowers pen, which automatically draws using the first color in the invisible list.

```
rLocate 400,300
rInvisible GREEN
LineWidth 4
rPen DOWN
for n = 1 to 4
  rForward 120
  rTurn 90
next
```

Figure 1.7: The program in Figure 1.6 can be modified so that the robot draws a line as it moves.

When the program in Figure 1.7 is run, it produces the screen shows in Figure 1.8. Notice that repeating the movement commands four times does move the robot in a square motion and returns it to its original starting position.

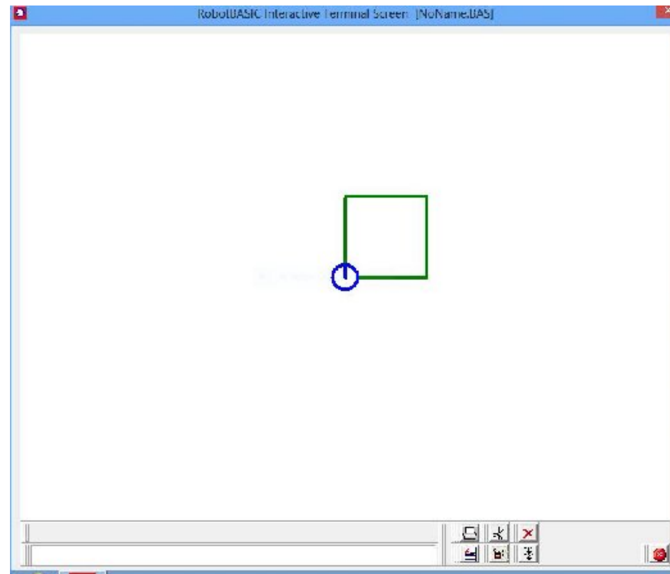

Figure 1.8: This screen is created by the program in Figure 1.7.

Organizing Your Program

The program in Figure 1.7 is still small, but it is never too early to think about organizing your programs so they are easier to read and understand as they get bigger. This can be done by placing functional pieces of your program in separate modules called *subroutines*. The subroutines can be executed with a **`gosub`** statement when needed, from the main program as shown in Figure 1.9. Notice that subroutine names must end with a colon when you are defining a new subroutine. Subroutine names, just like variables, are case sensitive. When a subroutine terminates by executing a **`return`** statement, the program continues execution with the line following the **`gosub`** that directed the flow to the subroutine.

Notice that the new program shown in Figure 1.9 is now made up of two main modules, one that initializes the robot and one that causes it to draw a square. Notice also that the names of the modules reflect their function. This not only makes the program easier to understand, it creates modules that are easier to re-used when needed. Notice the **`Main`** program simply executes each subroutine by calling them with a **`gosub`** statement. The name **`Main`** just helps you visualize where the program starts. You could use any name though, or even no name at all, as the program will start at the first executable statement found in the file.

```
Main:
  gosub InitializeRobot
  gosub DrawSquare
end

InitializeSimulator:
  rLocate 300,500
  rInvisible GREEN
  LineWidth 4
  rPen DOWN
return

DrawSquare:
  for n = 1 to 4
    rForward 120
    rTurn 90
  next
return
```

Figure 1.9: This program is a more organized version of the program in Figure 1.7.

Notice also how each module in Figure 1.9 is indented to make it easy to identify where the module begins and ends. Notice also that this indenting is also used with loops. Indenting is not required, but it will make your programs far easier to read, especially as they get larger. It may seem like a hassle to indent your code, but as your programs get larger you will quickly see how important it is. For that reason, you should start indenting every program you write, right from the beginning.

Real World Considerations

Recall from Figure 1.8, that the robot moved in a square pattern. In the real world a robot is not likely to have the precise movement demonstrated by the Figure. It might, for example, turn slightly more or less than 90° or move slightly more or less than the requested distance. This can happen, for example, because one motor is slightly faster than the other – perhaps because it has better bearings and thus less friction (although there are many reasons for this type of inaccuracy).

One way to make a real robot's movements more precise is have encoders on each motor that count pulses that occur as they wheel moves. These pulses allow the robot's computer to keep track of how far and how fast each motor moves. Advanced programs can use this information to dynamically change the speed of the wheels so they each move the same amount, at least as accurate as the amount of movement associated with each encoder pulse. If for example, a wheel produces 100 pulses per revolution, then the computer should be able to know the wheel's position within 1%. This limitation, plus

the fact that one wheel might slip slightly on the floor, means that a real robot will always have some error associated with its movement, even if it has wheel encoders.

We can make the simulated robot act more like a real robot by forcing it to generate some percentage of random error by altering the **`InitializeSimulator`** module as shown in Figure 1.10. Notice also, the comment explaining the purpose of the line. Any text following double slashes is considered a comment and is ignored by RobotBASIC. If you make this modification and run the program again, the movement will look something like that shown in Figure 1.11. Remember, the actual movement will be different every time the program is run, because the error generated is random (just like the movement of a real robot).

```
InitializeSimulator:
  rLocate 400,300
  rInvisible GREEN
  LineWidth 4
  rPen DOWN
  rSlip 15 // produces up to 15% random error
return
```

Figure 1.10: The `rSlip` command tells the simulated robot to create random error.

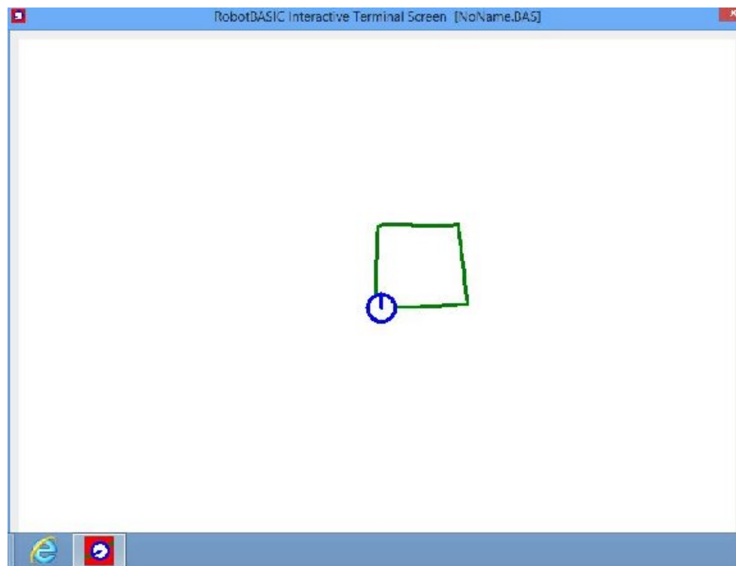

Figure 1.11: Now the simulated robot performs much more like a real robot.

In later projects, we will see how the robot can sense objects in the room and use that information to constantly correct its motion in order to achieve goals that we create for it, even if its movements are not precise. For example, a robot with an electronic compass might compare the direction it thinks it is facing to the compass reading and then make appropriate corrections. When sensors are used in this manner, the robot's normal movements do not have to be precise because errors are constantly corrected based on the sensory information.

Controlling a Real Robot

Now that we understand a little about the simulator, let's see how to make RobotBASIC control a real robot. This is really easy to do because RobotBASIC's RB-9 Robot, shown in Figure 1.12, has been designed to interface directly with the RobotBASIC programming language. Don't worry if you do not have a real robot that is compatible with RobotBASIC. You can learn the same principles just using the simulator. If you do not have a real robot, just skip forward to the topic **Suggestions for Study**.

If you have an RB-9 robot (or other RobotBASIC compatible robot) though, and initialize it instead of the simulator, then any program that controlled the simulator will be able to control the real robot causing it to perform the same or similar actions as the simulation. Remember though, real robots are never as precise, so general movements will not be exactly the same. We will see in later projects how this problem can be overcome. For now though, let's modify the program in Figure 1.9 so that it can control the real robot. The new version is shown in Figure 1.13.

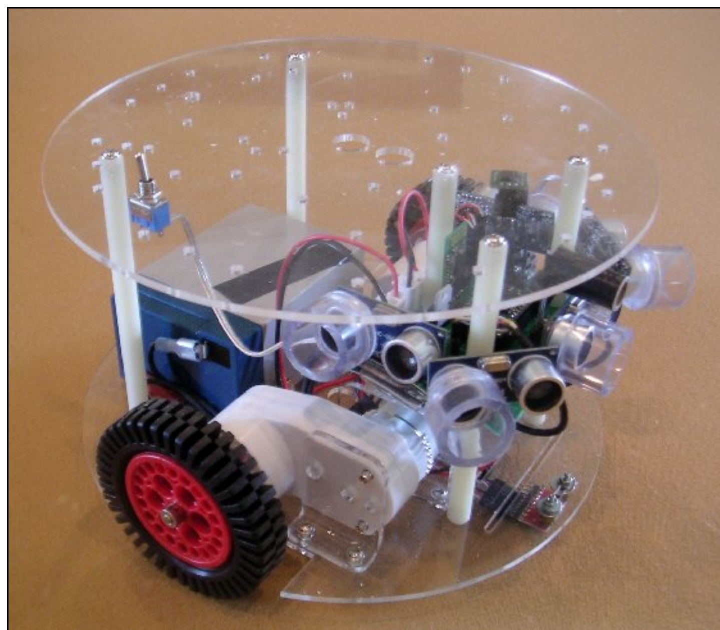

Figure 1.12: The RB-9 Robot has been designed to interface with RobotBASIC

```
#include "RB-9.bas"
gosub InitRROScommands

Real = TRUE // set to FALSE to use simulator
PortNum = 5 // set variable to your Bluetooth Port Number

Main:
  if Real
    gosub InitializeRealRobot
  else
    gosub InitializeSimulator
  endif
  gosub DrawSquare
end

InitializeSimulator:
  rLocate 400,300
  rInvisible GREEN
  LineWidth 4
  rPen DOWN
return

DrawSquare:
  for n = 1 to 4
    rForward 120
    rTurn 90
  next
return
```

Figure 1.13: This program can control either a real robot or the simulator.

Notice that the two subroutines in Figure 1.13 that do all the work did not have to be modified at all. Let's see what was changed. The first line in the program tells RobotBASIC to temporarily add some pre-written subroutines to your program. Including these subroutines makes them available to your program. These subroutines provide the technical details that make it easy for your programs control the RB-9 robot. All of these details are generally hidden from you so that you usually do not have to worry about them. Later projects will let you see the internal details of these routines so they can be modified, if necessary, for special situations. For now, let's just see how to use them.

Technical Information

The RB-9 robot has its own onboard computer. The robot's computer runs a special program called the RROS (the RobotBASIC Robot Operating System). The RROS handles many of the complicated details associated with controlling a real robot's hardware (motors, sensors, etc.). This is not unlike the Windows Operating System that makes it easy for your PC to utilize hardware such as its disk drive or mouse by handling all the details associated with communicating with and controlling such devices.

It is far more complicated to use a computer (or a robot) if it does not have an operating system, because the operating system lets you think about *what* you want to do instead of worrying about *how* to do it. For example, when you ask the robot to move with an **`rForward`** or **`rTurn`** command, RobotBASIC automatically sends your request to the robot over a wireless Bluetooth link. When the request is received by the robot, the RROS will control the robot's electronics to make the motors move the robot as requested. For example, assume that the request was to **`rTurn 90`**. In order to do this accurately, the RROS will count pulses generated by the robot's wheel motors and use that information to make each motor move amount that turns the robot approximately 90°.

The important point is that the robot does all this without you even having to know about it. In fact, the RROS does far more than just turn the motors on and off. Doing so would make the robot jerk when starting (like always pressing your car's accelerator peddle to the maximum when starting). To solve this problem, the RROS starts the motors moving at a slow speed and increases it slowly over time, then, when it is time for the robot to stop, the motors will automatically slow down before stopping instead of just slamming on the brakes.

The concept of a RROS is very powerful because it allows even beginning users to immediately do things that would normally require far more knowledge. The **`rForward 120`** statement is another example of this power. Remember it moved the simulated robot a distance equal to three times the robots diameter. When this statement is used with the real robot, the real robot will move a distance approximately equal to three times its own diameter.

Working with the RROS

One of the first things the modified program of Figure 1.13 does is to execute the subroutine **`InitRROScommands`**. This action initializes numerous variables that make it much easier for your programs to communicate with the RROS. Later projects will examine this initialization in detail, but for now just note what has to be done to make your programs control the real robot.

The next thing the program does is to set the variable **Real** equal to **TRUE** or **FALSE**. When it is **TRUE**, your program will control the real robot. Setting it to **FALSE** will cause the simulated robot to be used.

RobotBASIC will communicate with the real robot over a Bluetooth communication link. If the computer you are using has an internal Bluetooth transceiver then it can be used. If not, you will need a USB Bluetooth transceiver. Your computer's Bluetooth system (either internal or USB) will need to be paired with the Bluetooth hardware in the real robot. Generally, this only has to be done once. The paring code for the RobotBASIC robot is 1234. If you know how to perform the pairing, do so and note the port number used by Windows. If you do not know how to perform the pairing have someone in your IT department help or refer to the document *Pairing the Bluetooth Transceivers* available from www.RobotBASIC.org. Either way, just set the variable **PortNum** equal to the port your computer assigns to the Bluetooth pairing as shown in Figure 1.13.

Notice that the **Main** module in Figure 1.13 has been modified so that one of two initializing modules is executed based on the value of the variable **Real**. This decision is handled by an **IF** control-structure, which typically has two parts. The statements following the **IF** are executed when the if-condition is true, and the statements following the **ELSE** are executed when the if-condition is false. The **ENDIF** marks the end of the **if**-structure. In this example, when the value of **Real** is **TRUE,** the real robot is initialized with a subroutine that was included earlier in the program. Otherwise, the simulated robot is initialized just as it originally was. RobotBASIC offers several forms of the **IF** statement. Other forms will be used in later projects. Remember, you can always obtain more information about commands you are not familiar with from the HELP file.

Once either the simulator or the real robot is initialized, then any programming statements that you write will control the robot you selected. In this example, either the simulated or the real robot will attempt to move in a square pattern. Later projects will teach you how to make the robot do much more.

Limitations

No matter how good a simulation is, there will always be differences between the simulation and the real thing. As we proceed through future projects, such differences will be pointed out and options for solving the problems will be provided.

In this project the real robot should have no trouble mimicking the simulator's basic actions. There is one limitation, however, that needs to be observed. The simulator allows the **rForward** command to move any amount of pixels but when using the real robot the maximum value for a forward movement is around 250. However, it is recommended that you stay below 200 for most situations. As we proceed with more

complicated projects you will see this is not a significant limitation, because a real robot should never move very far without using its sensors to ensure that nothing is in its way. For most situations, the robot should only move forward 1 unit at a time before checking the sensory data. This assertion will become clearer in future projects.

Suggestions for Study

Experiment with various movements on the simulator and then let your program control the real robot as described earlier. Try making the robot move in a triangular pattern instead of a square. Try making it move in a circle. Hint: a hexagon or octagon will approximate a circle. The more sides the shape has, the closer the approximation will be. How can you control how big the triangle or circle is?