RobotBASIC Projects For The Lego NXT

Robot Programming for Beginners

John Blankenship & Samuel Mishal

ISBN-13: 978-1450558570 ISBN-10: 1450558577

Contents AT A Glance

Table Of Contents

Preface

Not long ago, if you wanted to build a robot you needed a degree in electronics. Nowadays, many companies provide motor controllers, sensors, and even kits that make it easy for anyone to experiment with hobby robotics. No one though, makes experimenting with hardware any easier than Lego.

Lego's NXT system allows you to snap together a robot base without tools of any kind. They provide a variety of self-contained, modular sensors and motors that can be interfaced with the NXT computer by simply connecting them with plug-in cables.

The problem with the NXT Robot though is software. While the visual programming language that ships with the system is supposed to be easy-to-use for beginners, many find it far from intuitive. Unless the tasks you are attempting are rudimentary and uncomplicated you may find the NXT's programming system difficult to comprehend. Even many of the after-market languages available for the NXT have cryptic syntax that can frustrate a new user.

One solution to these problems is RobotBASIC. Its easy-to-use English-like syntax makes programming easy to grasp, even for beginners. We provide a library of routines that allow you to control the NXT without downloading *anything* to the robot itself. RobotBASIC controls the NXT's motors and reads sensory data by talking directly to the NXT computer using Lego's wireless protocol. With our system, you program totally on the PC and when your program is ready, just run it and watch the robot respond.

Our Lego Library even has optional (but recommended) predefined constants and conversions that make programming even easier for beginners. You can command the robot to move at a FAST rate, for example, instead of setting the motor speed to an obscure number like 74. As another example of simplicity, our library returns distances measured with the range sensor in inches rather than milliseconds.

We also provide a Lego Simulation Library that allows your NXT programs to operate with the RobotBASIC simulator, letting you experiment even when the Lego hardware is not available.

The Simulation Library is extremely valuable for schools because a robotics curriculum can be implemented with only a single robot per classroom or perhaps even one per school. Every student can work with their own simulated robot both at home and in the classroom and when someone gets their program working, just plugging in a USB Bluetooth adapter will instantly allow their program to control the real NXT.

This system makes programming easier to understand because the user can concentrate on concepts rather than cryptic syntax or an unintuitive graphical interface.

Finally, RobotBASIC is a powerful, full-featured robot-control language, so after you have learned all you can from the NXT you can still use the RobotBASIC skills you learn from this book when you move on to other hardware technologies with more options and capabilities.

Chapter 1

The Lego NXT

The Lego NXT system comes with three gear-head motors and numerous sensors that can be used, along with a multitude of plastic snap-together structural parts, to build a robot. Lego's experiments often create different robots based on the sensors needed at the time. We felt it made more sense to build *one* robot that had all the sensors needed for our experiments.

Our Robot

The robot used throughout this book is shown in Figure 1.1. It uses two motors to move the robot, each motor moving one of the main drive wheels. When both wheels turn the same direction, the robot will move forward or backward. When they move in opposite directions, the robot will rotate left or right.

The Sensors

The robot supports a variety of sensors. A bumper sensor is mounted at the front of the robot to allow it to detect objects in its way. A sound sensor allows it to react to hand claps or other sudden noises.

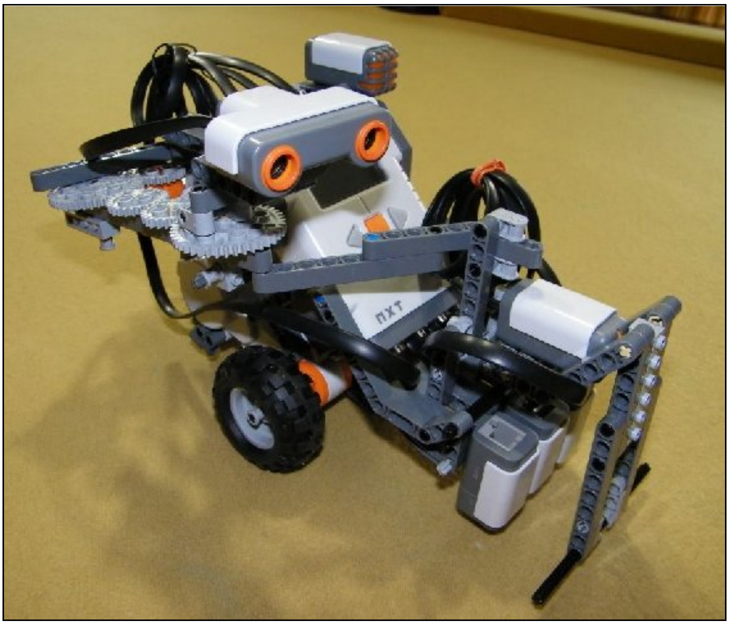


Figure 1.1: This NXT robot will be used throughout the book.

An ultrasonic ranging sensor measures the distance to objects. This sensor is mounted on a motor controlled turret so that objects can be detected in front of the robot or off to the left side. When facing forward, the ranging sensor can be used to avoid objects blocking the robot's path. When it is faced to the side, it can be used to make the robot remain close to an object so as to navigate around it by following its contour.

Three line sensors are mounted near the front of the robot. Three are used in our robot in order to demonstrate a variety of line following methodologies. If your robot only has one line sensor, don't worry, because projects are provided for that configuration too.

Physical Construction

If you have experimented with a Lego NXT system you will notice that the main body of our robot is basically constructed like one shown in their construction guide. If you have played with any of the Lego parts, you probably will have no trouble modifying Lego's original design so that it can support the additional sensors needed for our projects. Many of our modifications (such as the bumper sensor assembly and the rotating rear wheel) are similar to those shown in various projects in the Lego documentation. If you need more help, Appendix A provides several photos of our robot in various stages of completion. Your robot does not have to be exactly like ours. The important thing is that your robot support all the sensors *you* wish to experiment with.

As you proceed with your construction, there are a few things to keep in mind. The line sensors should be mounted at the front of the robot so that they hang about a quarter of an inch above the ground. The turret motor that moves the range sensor on our model is mounted on the rear of the robot to maintain proper balance. Without the rear weight, the robot can tip forward pushing the line sensors against the ground causing faulty readings.

The rotating turret that positions the ranging sensor works well, but it is far from sophisticated. A simple lever attached to one of the gears acts as a stop at both ends of the turret's rotational travel. To move to either of the two possible positions, we simply turn on the motor briefly (the on-time was arrived at experimentally) in the proper direction after which the motor coasts toward its destination.

The lever easily stops the turret's movement at each end of its travel since the motor is no longer under power. A series of gears (you could also use some form of belt-assembly) allows the drive motor to be mounted on the rear of the robot (for weight balancing, as mentioned earlier) while the ranging sensor is mounted on the front half of the robot. The range sensor needs to be at the front of the robot so it can properly navigate around objects (Chapter 6).

The Lego NXT computer, often referred to as the *brick*, can only control three motors and four sensors simultaneously. Our robot has only three motors so no problems there, but it has six sensors (sound, bumper, range, and three line sensors) all permanently mounted on the robot. Since none of the experiments in this book need more than four sensors, you will only connect the ones needed for the experiment at hand.

Now that you know what our robot will look like, let's move on to the next chapter and find out how to control its motors.

Chapter 2

Controlling the Motors

Normally when you experiment with a Lego robot, you must download your programs to the NXT's computer. In this text, we will use a different approach. Fortunately, Lego provides links in their system software to allow their robot to be controlled over a Bluetooth wireless line using *direct commands*. This feature allows us to control the NXT robot without downloading any programs. In fact, there will be no need to program the NXT computer at all.

Lego Documentation

Lego makes available extensive documentation and technical information on how to utilize their direct commands, but you won't need to know any of these details thanks to our system. We used Lego's documentation to create a library of routines that make it easy for you to control the robot using simple RobotBASIC commands.

We realize though, that some readers may want more information on how the direct commands work, so we provide the full source code for **LegoLibrary.bas** in

Appendix B. We strongly suggest though, that you work through the entire book before you even look at Appendix B. Remember, you don't need *any* of the material in Appendix B to understand any of the projects in this book, so ignore it for now and concentrate on learning the principles of how to program the robot.

RobotBASIC's Direct Command Library

In order to make controlling the robot as easy as possible, we have isolated all the technical complexities in a set of library routines. When the library is ***included*** in your RobotBASIC programs, you will have available numerous commands that allow you to control motors and read sensory data. Later chapters will explain how to utilize the sensors to create intelligent behaviors. For now, let's see how to control the Lego motors using the library routines. Look at the program shown in Figure 2.1.

```
#include "LegoLibrary.bas"
BluetoothPort = 34
call LegoInit(BluetoothPort)
call LegoDriveMotors(FAST,FAST)
call Wait(3000)
call LegoDriveMotors(STOP,STOP)
end
```

Figure 2.1: This simple program demonstrates how to control two of the Lego NXT's motors.

A Motor Control Demo Program

The program in Figure 2.1 demonstrates how to control the robot's motors, but it also shows how easy it is to use the library routines. ***This text assumes you have at least a <u>little</u> programming experience with RobotBASIC***. We will provide nearly everything you need to know, but if you have never programmed with RobotBASIC before, we suggest you go through ***Appendix E*** and perhaps the ***PDF Tutorial*** on the home page at **www.RobotBASIC.com.** You can also use the RobotBASIC **HELP File** to obtain more information on commands used in this book. If you have never programmed before in any language, you should consider reading one of our beginner's books such as ***Robots in the Classroom*** or ***RobotBASIC Projects for Beginner's***, both of which are available through our web site.

RobotBASIC has **two** types of modular structures called *subroutines*. The first of these is the **`GOSUB`**-style subroutine typicaly used in most dialects of the BASIC language. RobotBASIC also has a more advanced, **`CALL`**able function-style subroutine that allows for parameter passing and local variables. This book will make extensive use of both of these types of subroutines and it is important that you understand how to use both types. See Appendix F for more information.

The first line in the program of Figure 2.1 causes the **`LegoLibrary`** to be *temporarily* added to a program when you run it, and then erased when the program terminates. The library file must be in the same directory as your RB program, or you can specify a path with the file name in the **`#include`** statement.

When the library is included, all the *routines* in the library file will be available to your program just as if you had written them yourself. In this program we will *call* three different library routines to demonstrate how easy this process is. Calling these routines is how we tell them to execute. We can pass information to a routine by listing it between the parentheses following the name of the routine. When the routine finishes its job, it terminates and execution continues with the code following the line with the **`call`** statement.

The names of the library routines (as well as all variables used in RobotBASIC) are case sensitive. Misspelling a name or using the wrong capitalization will prevent RobotBASIC from finding it. This will cause an error indicating that the name of the desired function is incorrect. The commands in RobotBASIC (things like **`call`**, **`#include`**, and so forth) are NOT case sensitive.

LegoInit( )

The first **`call`** is to a routine called **`LegoInit()`**. It is assumed you have linked your Lego NXT system to your PC using a Bluetooth adapter. If you are not familiar with how to do this, refer to Appendix C for more information.

When you **`call`** the **`LegoInit()`** routine, you must pass it the port number assigned to your connection. Again, refer to Appendix C if you need help with this topic.

On our machine, the Bluetooth connection for the Lego robot was 34. You could place the 34 directly in the call to **`LegoInit()`**, but we will use a variable as shown in Figure 2.1. The reason for using a variable will become clear later in the book. The **`call`** to **`LegoInit()`** automatically establishes communication with the Lego robot and initializes the library so it can be used in your program. Calls to the library routines will cause errors if the library has not been properly initialized.

LegoDriveMotors( )

The next program line calls the **`LegoDriveMotors()`** routine and passes it two predefined variables. In this case, we are asking both the left and right motors to turn at the **`FAST`** speed. You can also command each motor to move **`SLOW`** and **`STOP`**. Actually, the library lets you specify numerically the exact speeds you want, but you should use the predefined values for everything in this book, because they will allow you to use the simulation library, but more on that later.

The first parameter passed to **`LegoDriveMotors()`** controls the motor connected to Port 1, which on our robot is the left motor. The second number controls port 2 which should be the right-hand motor.

Once the motors are turned on, the program delays for 3000 ms (3 seconds) by calling a library routine called **`Wait`**. Finally, another call to **`LegoDriveMotors()`** turns all motors off.

RobotBASIC has a **`delay`** command but it should *not* be used with the programs in this book. The **`Wait`** routine shown in Figure 2.1 provides the same delay, but it also provides added functionalities needed by other library routines.

When you run the program in Figure 2.1, the robot should move forward for three seconds then stop. If the program does not work properly, start by checking to see if your motors are connected to the correct ports. If there is a problem with the Bluetooth connection, RobotBASIC will issue an error saying the port is not available.

Once you get the program running, try replacing both the **`FAST`** parameters with **`SLOW`** to make the robot move forward, but at a slower speed. If a parameter is negative, the associated motor will turn in reverse. Two negative numbers will make the robot go backwards. If one number is negative and the other positive, the robot should spin in place, because one wheel is moving forward and the other is moving backward.

Can you guess what would happen if you issued this command:

```
call LegoDriveMotors(SLOW, FAST)
```

Since the right motor will move faster than the left, the robot will move forward, but in a slow turn to the left.

Look at the program in Figure 2.2. It turns the motors on in a particular way then waits a specified number of milliseconds. When the program is run, the robot should move forward about half the length of the robot, then make a left turn of about 90 degrees, then move forward again. If the robot is not making a proper right turn, try adjusting the **`Wait`** time. A bigger wait time will make the robot turn more, a smaller delay will turn the robot less.

```
#include "LegoLibrary.bas"
BluetoothPort = 34
call LegoInit(BluetoothPort)
call LegoDriveMotors(SLOW, SLOW)
call Wait(1500)
call LegoDriveMotors(-SLOW, SLOW)
call Wait(1100)
call LegoDriveMotors(SLOW, SLOW)
call Wait(1500)
call LegoDriveMotors(STOP, STOP)
end
```

Figure 2.2: This program moves the robot through a specific sequence.

Open-Loop Control

When you control a robot as demonstrated in Figure 2.2, it is called *open-loop* control. This simply means that you tell the robot what to do, but you never get any feedback to let you know if your commands were actually executed successfully. In the chapters that follow, we will begin to learn how information derived from sensors can influence how our robot behaves. The ultimate goal for this text is to help you learn to use this sensory data to implement *closed-loop* feedback systems so that your robot can react to its environment.

More Library Routines

Actions such as moving forward about half the length of the robot and turning 90° come in handy, as we will see in future chapters. Figure 2.3 shows how we could build two routines that perform these actions.

The low-level routines in the Lego Library are function-style **call**able routines because they need parameters to be passed to them. The routines in Figure 2.3 are standard **gosub**-style routines. Appendix F provides information on these two types of subroutines.

The first routine in Figure 2.3 is called **LegoAdvance**. It turns the motors on so as to move the robot forward then waits a predefined time period before turning the motors off. If the **AdvanceTime** is calibrated properly, the robot should move about ½ its length.

```
LegoAdvance:
  call LegoDriveMotors(SLOW, SLOW)
  call Wait(AdvanceTime)
  call LegoDriveMotors(STOP, STOP)
return
//------------------------------------
LegoFaceLeft:
  call LegoDriveMotors(-SLOW, SLOW)
  call Wait(LeftTurnTime)
  call LegoDriveMotors(STOP, STOP)
return
//------------------------------------
LegoHalt:
  call LegoDriveMotors(STOP, STOP)
return
```

Figure 2.3: These routines, when properly calibrated, make the robot move half its length and turn right 90º.

The second routine in Figure 2.3 is **LegoFaceLeft**. It is similar to the **LegoAdvance** routine except that it turns the motors on in opposite directions, making the robot rotate around its center. If the variable **LeftTurnTime** is initialized to an appropriate value, the robot should turn approximately 90º.

The last routine in Figure 2.3 halts the robot by making a call to the **`LegoDriveMotors()`** routine to stop both motors.

These new routines are easier to use (especially for beginners) than the callable routines used in Figure 2.2 because the new routines do not require you to specify any parameters. For that reason, these routines are also included in the **`LegoLibrary`**. Look at the program in Figure 2.4. It uses the new routines to perform the same basic actions as the program in Figure 2.2. Notice that you **`GOSUB`** to these routines instead of **`CALL`**ing them.

```
#include "LegoLibrary.bas"
BluetoothPort = 34
call LegoInit(BluetoothPort)
gosub LegoAdvance
gosub LegoFaceLeft
gosub LegoAdvance
gosub LegoHalt
end
```

Figure 2.4: This program performs the same actions as Figure 2.2.

Another advantage to using the new routines is that it is easier to see what the program is doing. When you see the statement **`gosub LegoFaceLeft`** you can guess it will turn the robot to the left even if you don’t know how to program. For that reason, we will use routines like these throughout the book.

Additional Routines

In order to make it as easy as possible for beginners, we have provided many routines similar to the ones shown in Figure 2.3. A list of routines that make it easy to control the movement of the Lego robot is shown in Figure 2.5. The figure also explains what each routine does, but, as mentioned earlier, the name of the routine is usually sufficient.

The easy rights and lefts move the robot forward with a gentle turn in the direction indicated. Hard rights and lefts turn quickly with only a little forward movement. These actions were chosen because they can serve as the basis for the more complex behaviors developed in later chapters. When you write programs to control the Lego robot you can freely mix any of the Library routines to accomplish your goal. Just be sure to use the correct statement (**`gosub`** or **`call`**) to execute your chosen routines.

ROUTINE NAME	ACTION PERFORMED
LegoAdvance	Forward ½ length of robot
LegoRetreat	Backward ½ length of robot
LegoFaceRight	Right turn 90º
LegoFaceLeft	Left turn 90º
LegoHardRight	Quick turn to the right
LegoEasyRight	Slow turn to the right
LegoHardLeft	Quick turn to the left
LegoEasyLeft	Slow turn to the left
LegoHalt	Stop the robot

Figure 2.5: These library routines make it easy to control the robot's movement.

Calibrating the Routines

These new routines should move your robot as previously described, but no two motors are alike due to differences in friction, efficiency, and other factors. For that reason you should not expect the robot to move exactly half its distance when it advances, nor should you expect it to make perfect right-angle turns. This is typical when an open-loop system is used to control a robot.

The whole purpose of this book is to show you how to use information obtained from the sensors mounted on the robot to make your robot move appropriately and even intelligently. So, at this point, don't worry if your robot is not doing exactly what you think it should do. As long as the movements are reasonable (perhaps a 10-15% error) then we will be able to use the library to create accurate closed-loop behaviors. If your robot's movements are off by large amounts, refer to Appendix D for information on how to calibrate the libraries for your particular robot. Remember, unless your robot has significant errors when it moves, you do not need to perform any calibration at all.

The Lego Simulator

In addition to the **LegoLibrary**, we also have provided a **LegoSimulationLibrary**. The simulation library has all the same functions, but they control the RobotBASIC simulated robot instead of the real Lego robot. To see the simulation in action, use either the program in Figure 2.2 or 2.4, but change the first line in the program to:

```
#include "LegoSimulationLibrary.bas"
```

After you have made the change, run the program. You will see a small circular robot appear on the screen. It will move forward about half its length and then turn to the left and move forward again, just like the real Lego robot did.

If you run the program several times, and watch closely, you will see that the simulated robot does not always move or turn exactly the same amount. It has a slight amount of random error associated with everything it does, just like a real robot. Let’s look at an example to demonstrate this point.

Examining the New Movements

Use the program in Figure 2.6 to see how the robot reacts to the new library routines shown in Figure 2.5. Learn to control your robot’s movements by modifying the program with different commands. Compare the turning radius with both hard and easy turns to the face right and left commands. With each variation, change the **`#include`** statement to use the standard **`LegoSimulationLibrary`** to see how closely the real robot responds like the simulation.

```
#include "LegoLibrary.bas"
BluetoothPort = 34
call LegoInit(BluetoothPort)
gosub LegoEasyRight
call Wait(4000)
gosub LegoHalt
end
```

Figure 2.6: Use the program to experiment with the new movement routines.

Demonstrating Errors

Look at the program in Figure 2.7. The **`for`** loop causes the robot to move forward twice then face right, four times in a row. Think about this for a moment or perhaps even try it yourself. Stand in an open room, move forward two steps, and turn right. Now do this three more times. If you took exactly the same size steps and turned right exactly 90º, then you would have moved in a square pattern and returned to your original position.

Remember though, our robot's movements are not perfect. Enter the program in Figure 2.7 and run it to see the simulated robot try to move in a square pattern. You will notice that the simulated robot has an error in its movement. The error is realistic – run the program several times and you will see that the robot moves differently each time. If you change the first line of the program to use the standard **`LegoLibrary`**, you will see the real robot move in a similar manner.

The lines in Figure 2.8 were drawn because of the **`call`** to the library routine **`LegoPen(DOWN)`** in Figure 2.7, which causes the robot to leave a trail as it moves (as if a pen mounted at the center of the robot was lowered so it can draw as the robot moves). You can substitute **`UP`** with **`DOWN`** to raise the pen if you wish to stop leaving a trail.

```
#include "LegoSimulationLibrary.bas"
BluetoothPort = 34
call LegoInit(BluetoothPort)
call LegoPen(DOWN)
for i=1 to 4
  gosub LegoAdvance
  gosub LegoAdvance
  gosub LegoFaceRight
next
gosub LegoHalt
end
```

Figure 2.7: This program moves the robot in a square pattern.

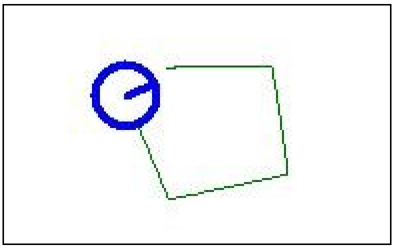


Figure 2.8: The simulated robot has error in its movement, just like a real robot.

You can see from Figure 2.8 that the robot is indeed *trying* to move in a square motion. It is important to realize that even if you could calibrate the library *perfectly*, the robot will still not move in a square because there is always some error added to its actions. We will learn how to overcome this error in future chapters by creating programs that allow our robot to adjust its movements based on its environment. If you are new to robot programming this may sound difficult or even impossible, but it is actually far easier than you might imagine. Roll up your sleeves and move on to the next chapter. The adventure is about to begin.

Chapter 3

The Line Sensor

This chapter introduces many principles and techniques needed in later chapters, so you should study it carefully before moving on.

Recall the exercise in Chapter 2 where you were asked to pretend you were a robot and perform two actions (move forward, turn right), four times in a row. If done perfectly, your movements would describe a square. If you tried it though you know you don't end up exactly where you started and you are facing at least a slightly different direction.

Feedback

If you try these actions with your eyes open you can move in a square easily, because the feedback from your eyes allows you to constantly monitor your motion and continually correct any mistakes. This is a powerful concept and one we should apply to our robot in nearly all situations.

An Algorithm for Following a Line

In order to demonstrate this principle we will develop an *algorithm* (a step-by-step plan) for our robot to follow. Coming up with an appropriate algorithm is not always easy so let's examine how it can be done.

For this example, we will assume that our robot has a single line sensor mounted near its front bumper. The sensor generates a value of 1 (meaning it is over a black line) or 0 (meaning it sees only white space). These sensors work by emitting light towards the ground and then using a photo-detector to determine if the light is reflected back. A light colored surface reflects the light and a dark surface does not. An easy way to create our line is to draw it with a black felt-tip marker on a white poster board.

The easiest way for you to develop an algorithm is to place yourself in the position of the robot. In this example, the robot is being asked to follow a line but it only has one "eye" and can only see a tiny little area below the sensor. To give you the same limitation as the robot, cut a small hole in a 3 by 5 card and lay it on the line as shown in Figure 3.1.

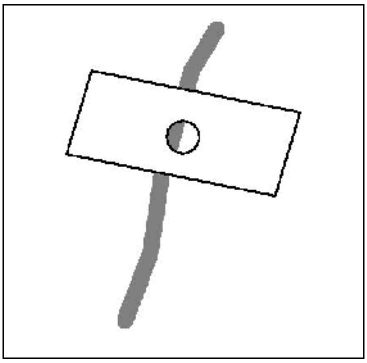


Figure 3.1: The hole in the card lets you see the line the way the robot does.

In order to make this experiment even more like the robot, get a friend to help you. One of you will be the robot program that decides what to do, and the other person will be the sensor and the motors of the robot – we will call that person the hardware-person.

Assume a starting position where the line can be seen through the hole. The hardware-person says "I see the line" and the program person must decide what to do. For instance she may say “Move forward” or “Turn right” or other similar actions. After the hardware-person moves the card in the manner requested, he acts like the sensor and reports whether the line can be seen or not. Using this new information, the program-person has to decide on the new action.

If you think about this for a moment, it should be obvious that the program-person only has two actions that can be ordered; one action if a line is seen and another action if a line is not seen. Get a 3 by 5 card and another person to help and see if you can figure out what actions should be taken when the line is seen and not seen. Remember, the program-person must sit well away from the line so she cannot see it. The only information she has is what is reported by the hardware-person. Please try this exercise before reading the answer below.

The Answer

The answer may be easier than you think. Basically, if we see the line we want to move away from it, always in the same direction (let’s say we move forward and right). If we don’t see the line, we move back toward it (perhaps forward and left). We can’t just move left and right, because that would mean the robot would just sit in one spot and rotate back and forth. The idea behind this solution is that the robot is constantly moving to the right away from the line if it sees it, and back to the left when it can’t. Using our two-person experiment, see if this simple algorithm works using a real line and the 3 by 5 card. Once you see that it does work, we are ready to convert this idea into a program.

Converting the Algorithm into a Program

Figure 3.2 shows the code that implements the algorithm described above. The main **`while`**-loop does all the work. It uses a call to the function **`LegoLineSensor()`** to read the status of the sensor. There are two parameter passed to the subroutine. The 2 indicates which Lego Port the sensor is plugged into. The variable **a** is passed so that the routine can place the answer in it. After the call, the variable **a** will be a 1 if a line is seen or 0 otherwise. A multi-line **if**-statement checks the value of **a** and if the line is seen, the library routine **LegoEasyRight** is used to move the robot forward with a drift to the right. If the line is not seen (the **ELSE** block), the robot moves forward with a drift to the left. The loop causes this action to be repeated, thus implementing our algorithm. Notice that both **call** and **gosub** statements are used depending on the type of library routine being executed.

```
#include "LegoSimulationLibrary.bas"
BluetoothPort = 34
gosub DrawLine
call LegoInit(BluetoothPort)
call LegoLineInit(2)
while true
  call LegoLineSensor(2,a)
  if a=1
    gosub LegoEasyRight
  else
    gosub LegoEasyLeft
  endif
wend
end
```

Figure 3.2: This program allows the robot to follow a line using one line sensor.

Initialization

There are three actions that are needed before we are ready to actually execute the loop that implements our algorithm. First, we need to draw a line. In order to make it easy to experiment with our simulation, we included a **DrawLine** subroutine in the library. Simply call it to create the environment shown in Figure 3.3. The **DrawLine()** function only works when the Simulation library is used. A call to this function is simply ignored by the standard library. Of course, when you use the real robot, you will have to provide a real line.

It is also necessary to initialize the **LegoLibrary** just as we have in previous programs. Notice that since we are using a line sensor, we also have to initialize it with a call to **`LegoLineInit()`**. We must tell the line-sensor initialization module which port on the Brick the sensor is connected to. In this example, we used port 2.

Notice that the program in Figure 3.2 includes the **LegoSimulationLibrary** which causes the algorithm to be carried out with the simulated robot. When the program is run, Figure 3.2 shows the path that the robot takes as it tries to follow the line.

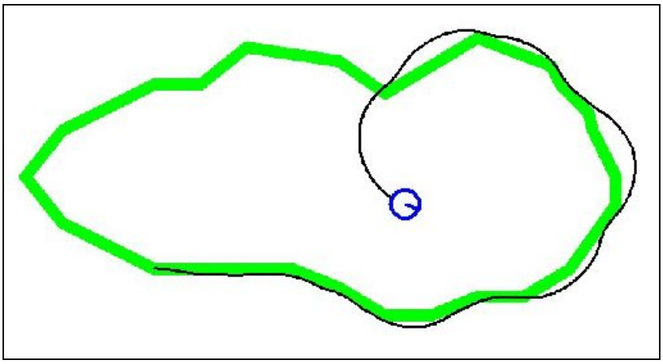


Figure 3.3: The library contains a function to draw a line.

If the line turns too sharply, the robot cannot stay with it and strays from the line, but eventually makes its way back. If the robot ever crosses to the left side of the line (as it does in Figure 3.3) it continues to turn left trying to find the line again – just as the program directs it to do. Of course this action shows a flaw in our algorithm. The robot's behavior is fine as long as the line it is expected to follow does not curve too quickly, but it fails when the line makes a sharp turn.

When you run the program, your robot might lose the line at a totally different spot. This is true because the random error (to create realism) introduced in our library will make every run of the program slightly different.

On the real robot you can use any ports you wish for the various sensors. On the simulation the port numbers used for line sensors will control which of the three simulated line sensors is used. The middle sensor is Port 2, the sensor on the right is Port 1, and Port 3 activates the left sensor.

Possible Solutions

One possible solution to our robot loosing the line is to make the line wider, or even create a solid object and let the robot follow the edge of the object rather than thinking of it as a line. If you think about it, this is actually what our algorithm is doing. When the robot sees the object it moves away and when it doesn't it moves back towards it.

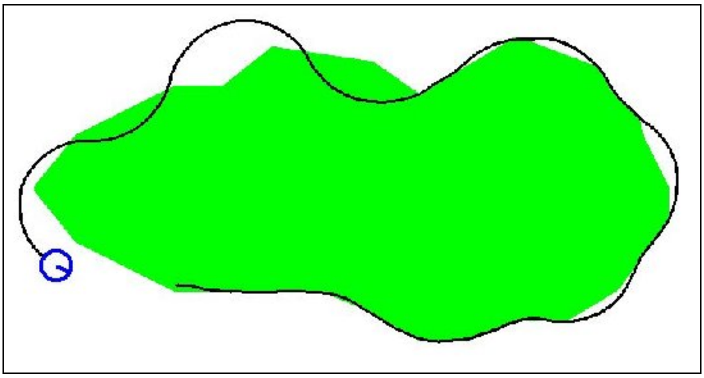


Figure 3.4: The robot can hug close to the edges of a solid object.

Figure 3.4 shows the robots behavior if the object is made solid. Now, when the robot ventures too far left, it still sees the object and knows it should turn to the right. The program now works better, but the robot exhibits a loopy behavior on sharp turns.

Other Solutions

Another possible solution is to make the robot turn quicker instead of those slow lazy drifts to the right and left. This can be done by changing the code as shown in Figure 3.5. The lines that were changed are shown in bold. The robot is made to turn more quickly by stopping the wheel on the turn side instead of just slowing it down.

With the changes shown in Figure 3.5, the robot follows the line almost perfectly, as shown in Figure 3.6. There is an unfortunate side effect though. The quick turns made by the robot make the robot oscillate erratically back and forth as it follows the line.

```
#include "LegoSimulationLibrary.bas"
BluetoothPort = 34
gosub DrawLine
call LegoInit(BluetoothPort)
call LegoLineInit(2)
while true
  call LegoLineSensor(2,a)
  if a=1
    gosub LegoHardRight
  else
    gosub LegoHardLeft
  endif
wend
end
```

Figure 3.5: The robot can handle sharp turns if it turns more quickly itself, but now there is an erratic movement.

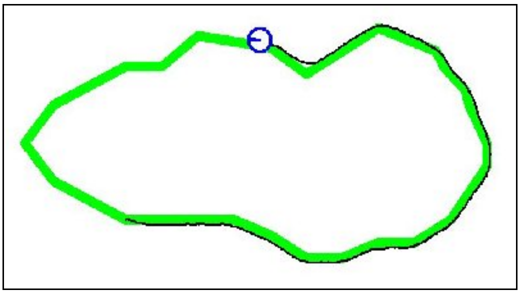


Figure 3.6: This is the path created by the program in Figure 3.5.

Running these programs and studying the robot's behavior can give you a better understanding of our algorithm, which will be valuable as we strive to improve on it in later chapters. It can also help you understand the deficiencies such as the wobbly movement associated with using fast turns. There is an important aspect of this discussion that needs to be emphasized. We have learned many powerful ideas and concepts about a robot following a line, and yet we have not utilized a real robot at all.

Simulations are used in many industries (not just robotics) to solve problems and explore solutions without the time and expense of using real hardware. Once solutions are found, they can then be applied to real-world situations. Depending on how accurate the simulations are, the final programs may have to be modified to fine tune the final behaviors, but that is far easier than developing the entire algorithm on real-world hardware.

Moving to the Real-World

The next step is to see how closely our simulator emulates the real world. Try running the programs of Figure 3.2 and 3.5 but change the first line to include the standard **LegoLibrary.bas** instead of the simulation one. Of course, if you are going to use the real robot, you will need a real line for the robot to follow as shown in Figure 3.7. Notice we have (starting at the bottom of the photo) a curvy line, a nearly straight line, and a circular line similar to what was used with the simulation. The lines were drawn with a black felt tip marker on white poster board. To keep the real line proportional to the size of the simulated line (compared to the spacing of the simulated sensors) the real line should be about one and three-quarter inches wide. If the spacing for the real line and the simulated line are not similar, you cannot expect the two robots to react in the same manner.

The program shown in Figure 3.2 should only follow a fairly straight line without losing it. The program in Figure

3.5 will follow all the lines shown, and it does so with the erratic wobble predicted by the simulation.

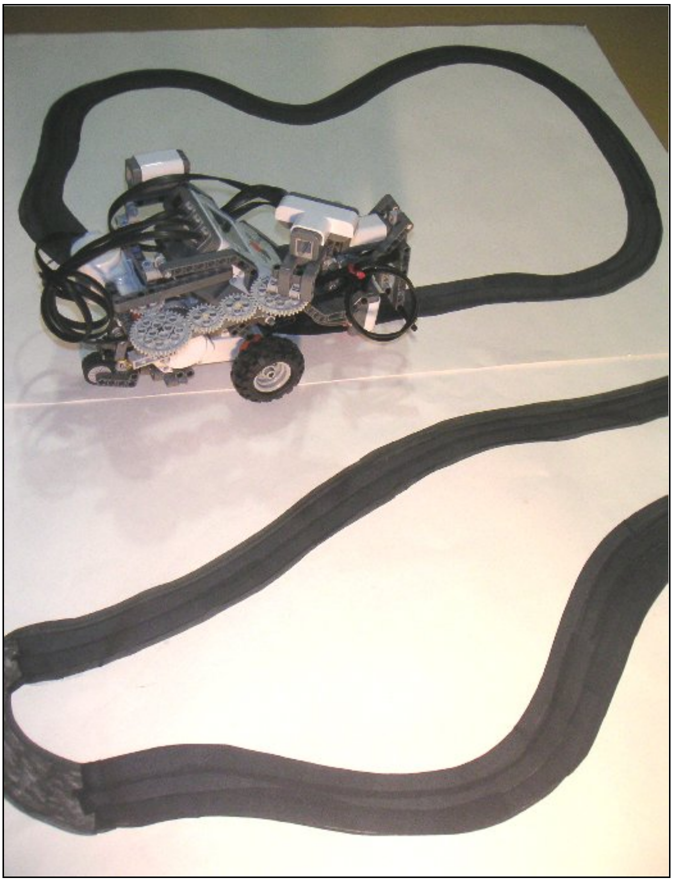


Figure 3.7: You must create lines for the real robot to follow.

Reducing the Speed

Another option for improving the robot's ability to follow the line is to reduce its speed. This will reduce the wobble effect demonstrated by the program in Figure 3.5 and the slower movement will help prevent the program in Figure 3.2 from losing the line when it curves too quickly.

Appendix C provides information on customizing the libraries, thus allowing you to control the speed of the robot as well as many other parameters. Unless your robot is not following the line properly, there is no reason to introduce confusing options but it is valuable to know that such options exist.

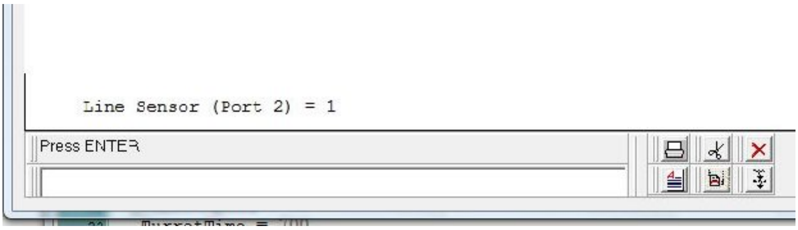
A Debugging Option

When a program is not producing the expected results, it becomes necessary to debug your code. For those readers not familiar with debugging techniques, we recommend studying the RobotBASIC HELP file as well as our introductory programming books (mentioned in Chapter 1).

In addition to all of RobotBASIC's standard debugging tools, we have added a special feature to the Lego Libraries. Just add the following line immediately after you call **`LegoInit()`** to enable debugging.

```
   LegoDebug = TRUE
```

This will cause your program to *pause* every time *any* sensor function is called and wait for you to press the ENTER key. In our program, for example, the program will stop when **`LegoLineSensor()`** is called. At that point, the robot (either the real robot or the simulation, depending on which Library you are using) will automatically stop moving and the RobotBASIC terminal screen will display something similar to Figure 3.8.



```
Line Sensor (Port 2) = 1
```

Figure 3.8: Both Lego Libraries have this special debugging feature.

As you can see from the figure, the sensor being interrogated will be displayed, as well as its current reading. In this example, we know that the robot is currently seeing a line because the line sensor has a value of 1. Knowing that, you should be able to predict what action the program should take. When you press the ENTER key, the robot will make its next move and stop again when a sensor function is called. If the robot does not move as expected you have one additional clue as to what might be wrong with your algorithm or your implementation of it.

This debugging feature can help you (or students) understand how an algorithm works (especially nice for teachers). It can also provide insights on how to improve on an algorithm you are trying to develop because it helps you see specific sensor conditions that are not being handled properly. We will examine these principles more in later chapters.

Summary

This chapter introduced many principles you will need in future chapters. Our next step is to learn how to utilize the data from the sound, bumper, and range sensors. Once you have a good grasp of how sensors can be used, we will create programs that utilize multiple sensors to give the robot more intelligent behaviors.