Robot Projects for RobotBASIC
Volume I: The Fundamentals

Project 7: The Compass

This project will introduce the robot's compass, which allows the robot to determine its orientation in a room or stay on a specified course direction. The heading supplied by the compass will be a value from 0 to 359 (degrees) with 0 generally representing due north.

Once the robot is initialized, you can read the compass angle using a statement like this.

```
dir = rCompass()
```

to find the current direction your robot is heading. You could use the compass to make your robot face east (90º) using the following code fragment. The **<>** is the syntax for *not equal*.

```
while rCompass() <> 90
  rTurn 1
wend
```

The above principle will work fine for the simulator, but if a real robot is rotating at even a moderate speed though, it will likely overshoot the intended destination. A better approach might be to have the loop end if the robot's angle is within a few degrees of the desired angle. The root of this problem is the fact that the compass can only be read every 100ms or so because of the time needed for communication between RobotBASIC and the real robot.

To solve these problems, and many others for the real robot, a special **`rCommand`** has been provided that tells the RROS to move the real robot to a specific heading specified by the variable **`angle`** in the following statement.

```
rCommand(TurnToHalfAngle,angle/2)
```

The parameters used with **`rCommand`** are 8-bit numbers (which simply means the can only have values from 0 to 255). Since the value of **`Parameter2`** cannot exceed 255, it must be divided by 2 in the above example. Having to divide the desired heading by 2 does reduce the resolution slightly, but the result is still more accurate and faster than you could obtain using only standard RobotBASIC commands.

Calibrating the Compass

The real robot's compass will be affected by magnetic fields and metal objects in its vicinity so, for best results, you should calibrate your compass periodically, and always calibrate it before using it in a new environment or if the readings are not accurate. The following code fragment demonstrates how to calibrate the compass. Notice how the **rCommand** is used to request the calibration action of the robot. Future projects will introduce many more **rCommand**s to alter or improve how the robot performs.

```
SetTimeOut(40000)// sets time out to 40 seconds
rCommand(CalibrateCompass,0)
SetTimeOut(5000)
```

When the **rCommand** is executed, the RB-9's internal programming will slowly rotate it for about 30 seconds while the compass is automatically calibrated. Normally, RobotBASIC expects a quick response from the remote robot, so the first line above is necessary to prevent a Timeout Error. The last line resets the timeout period to the default period of 5 seconds.

Real vs. Simulation

One of the important points of the above discussion is that the code needed to point the robot in a desired direction is different for the simulation and the real robot. While this is not a typical situation, it does happen, and we need a way to handle it properly.

The program in Figure 7.1 shows one way to solve this problem. It also demonstrates a simple use for the compass. In this case, either the real or simulated robot, will turn to the **angle** specified, and move forward a short distance automatically correcting itself to maintain the original specified heading.

Another advantage of the program in Figure 7.1 is that we now have a subject complicated enough to start learning more sophisticated ways of organizing a program. Study the description of this program carefully as it provides many new ideas you will need for future projects.

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

Angle = 45     // change to the angle you wish to use
Distance = 100 // change to the distance you wish to use
Real = TRUE // set to FALSE to use simulator
PortNum = 5 // set this variable to your Bluetooth Port
```

```
Main:
  gosub InitializeRobot
  call TurnTo(Angle)
  call ForwardAtHeading(Angle, Distance)
end

InitializeRobot:
  if Real
    gosub InitializeRealRobot
  else
    gosub InitializeSimulator
  endif
return

InitializeSimulator:
  // place the robot in the center of the screen
  // at some RANDOM heading
  rLocate 400,300,random(360)
  // introduce error so the robot must correct as it moves
  rSlip 15
return

sub TurnTo(a)
  if _Real
    rCommand(TurnToHalfAngle,a/2)
  else
    while rCompass()<>a
      rTurn 1
    wend
  endif
return

sub ForwardAtHeading(a,d)
  for n = 1 to d
    rForward 1
    call CompareCompass(a,dir)
    rTurn dir
  next
return


sub CompareCompass(a, &ans)
  b = rCompass()
  if b>a
```

```
    ans = 1
  elseif b<a
    ans = -1
  else
    ans = 0
  endif
  if abs(a-b)<180 then ans = -ans
return
```

Figure 7.1: This program turns the robot to a specified heading and maintains that heading as it moves the robot forward for a specified distance.

The program in Figure 7.1 begins, as many of our programs have, by including the **RB-9.bas** file and initializing the RROS commands. It then sets up the parameters that will be used in our main program.

The **Main** program is composed entirely of subroutines, making it easy to see exactly what this program does. Most versions of the BASIC language only have standard subroutines that are executed with a **gosub** statement. RobotBASIC also has a more advanced style of subroutine that is executed with a **call** statement, as shown in Figure 7.1. The subroutine itself is defined with the keyword **sub** as shown in the Figure. There are many advantages to this type of subroutine. To make it easier to talk about these routines, we will refer to standard **gosub**-style routines as subroutines and the new style as a sub-routine since they are created with a **sub** statement.

The first advantage is that all the variables used *inside* a sub-routine are *local* to that routine, that is they are held in a separate table from the standard *global* variables that are common to the main program and all standard subroutines. This means you can have a variable named **x**, for example in your main program, and a totally different variable named **x** inside a sub-routine. This means you can write sub-routines without worrying about conflicts with the sub-routine's variables and those used elsewhere in the program. This idea will be discussed in more detail shortly.

A second advantage is that you can pass information to a sub-routine using a list of parameters enclosed in parenthesis. Figure 7.2 shows some examples that demonstrate the basic principles of how a sub-routine works. When the program is run, it produces the output shown in Figure 7.3.

```
x = 2
a = 30
call Add(x, 3, answer)
print "Answer = ",answer
print "----------------"
call Add(2*x, a, answer)
print "Answer = ",answer
end

sub Add(a,b,&c)
   c = a+b
   print "Local a = ",a
   print "Global a = ",_a
return
```

Figure 7.2: Example of how to use a sub-routine.

Figure 7.2 shows a sub-routine called **Add**. The **call** statement will pass it two values which will be stored in the local variables **a** and **b**. The first variable in the **call** statement will be passed to the first variable in the **sub**-list, etc. These values are added together and stored in the variable **c**. Notice though, that the variable **c** in the **sub**-list is preceded by an **&** in the parameter list causing the variable **c** to be treated much differently than **a** and **b**. Lets examine the first portion of the program to make this difference clear.

```
Local a = 2
Global a = 30
Answer = 5
----------------
Local a = 4
Global a = 30
Answer = 34
```

Figure 7.3: This output is produced by the program in Figure 2.

The program starts by creating two global variables, **x** and **a**, and initializes them to 2 and 30. The first time **Add** is called, it is passed the *value* of **x** and the number **3**. The current value of **x** is stored in the local variable **a**, and the number 2 in the variable **b** because these variables are linked based on their position in the parameter list. These values are added and then stored in the variable **c**, but since **c** was preceded by an **&**, it is actually stored in the main program variable **answer**. What actually happens, is that when a variable is preceded by an **&**, it is actually passed the *name* of the variable, not its value. This allows the sub-routine to actually change the value of the *original* variable, not a new local variable. A sub-routine can actually access any global variable by

preceding it with an underscore (_) character. This feature is also demonstrated by Figure 7.2 where the sub-routine prints the value of both the local and global variables named **a**. Notice that the program prints an answer of 5 because it was passed 2 and 3. Notice also that it prints the local value of **a** as 2 while the global value is 30.

The second time **Add** is called, the values of 2 times **x** and **a** are passed. In this case, the local value of **a** is 4 and the global variable is still 30. The answer is the sum of 4 and 30 or 34 as shown in Figure 7.3. Write a few of your own programs to experiment with this idea until you feel you understand the principles involved. When you are ready, proceed with the discussion of Figure 7.1.

The main program in Figure 7.1 uses a subroutine to initialize either the real or simulated robot just as we have done in previous projects. Next, the sub-routine **TurnTo** is called and passed the value of **Angle**. This will cause the robot (either real or simulated) to face the specified direction. Next, the sub-routine **ForwardAtHeading** is called and passed the value of **Angle** as well as the value of **Distance**. This purpose of this routine is to move the robot forward a specified distance while maintaining the specified angle. Using sub-routines in this manner gives a little elegance to the way the program is organized because it makes it easier for the reader to understand what is happening.

You can, for example, see exactly what the program is doing by just looking at the small main program. Part of the reason this portion of the program is so easy to understand, is because it only deals with WHAT should happen. The details of HOW it happens are contained in the routines that do all the real work. This is not unlike how a large corporation is run. At the top, someone in management makes decisions on what needs to be done. Those orders are passed down to workers who actually perform the tasks necessary to complete the goals set by management. This structure allows each level to perform their job without having to worry about other aspects of the business. In a similar manner, this structure allows programmers to concentrate on specific parts of the program without having to deal with everything at once.

Now that the main program has shown us what the program is doing, lets look at the sub-routines to see how the work is actually accomplished. In the sub-routine **TurnTo** the global variable **Real** is used to decide how to actually turn the robot. Notice how an underscore is used to allow **Real** to be accessed. If the real robot is being controlled the **rCommand** mentioned earlier requests that the robot to turn to the desired angle. If the simulation is used, a **while**-loop turns the simulated robot, also as mentioned earlier.

As programs start to get complicated, as this one does, it is important to inform those new to programming that there is NO SINGLE RIGHT ANSWER when writing a program. In this version of **TurnTo**, for example, the value of **Real** was accessed using the underscore. It would have been just as correct to pass the value of **Real** as a parameter

or to even use subroutines instead of sub-routines. All of these approaches can get the job done and all should be considered as acceptable solutions. As you become more experienced at programming you will find styles and methods that you prefer. It is worth mentioning though, that employers will often dictate specific styles for company programs because they want uniformity throughout the company in order to makes it more efficient for everyone involved in a project to share ideas.

The sub-routine **`ForwardAtHeading`** is passed an angle and a distance. A loop moves the robot forward an amount based on the distance. The principle for maintaining the desired heading is easy to understand. If the robot is right of the desired angle, it should turn left. If it is left of the desired angle it should turn right. Notice that the basis for these actions is very similar to how we made the robot follow a line in Project 6. In this case though, we have a slight problem. The problem is that the current heading (the compass reading) and the desired reading cannot just be compared because even though comparing works most of the time, there are situations that fail. Let's look at some examples to make this clear.

If the compass reading is greater than the desired angle, in this example, you would normally want the robot to turn left. For instance, if the compass reads 50 and the desired angle is 45, then the robot obviously needs to turn to the left. But what if the compass reads 358 and the desired angle is 3. In this case the compass reading is also greater than the desired angle, but the robot is now 5° *left* of where it should be, and needs to turn right.

With a little examination, we can determine that *normally* the robot should turn left when the compass reading is greater than desired angle and right when it is less than the desired angle. If the difference between the compass reading and the desired angle is greater than 180 though, the corrective action should be reversed. This principle is important to understand when we want to compare two angles, but the details of how this is handled should not have to be considered by the **`ForwardAtHeading`** routine when it does its work. For that reason, we have created a sub-routine called **`CompareCompass`** that does all the work for us and generates a -1 to indicate the robot should turn left and a 1 if it should turn right. Again, this is like letting workers in a large corporation make decisions about how to perform their work so that their supervisor does not have to be bothered with those details.

Some programmers might choose to use the value of **`dir`** to control how the robot should turn with code like this.

```
    if dir = -1 then rTurn –1
    if dir = 1 then rTurn 1
```

While this works fine, a more experienced programmer will perform this action more efficiently with the single line below (as in Figure 7.1).

```
    rTurn dir
```

Notice this is only possible because of the values used to indicate whether to turn left and right. If the values provided by **`CompareCompass`** were 1 and 2, then the first method would have been required. Both methods work, so neither is wrong. But, obviously one method could be considered more efficient than the other. Even though there will always be multiple ways to program a solution, an experienced programmer will learn to utilize methods like this to decrease the size and increase the speed of their programs.

Study the principles discussed in this project carefully to ensure you understand them. Don't worry if some things seem hard to grasp. Future projects will revisit these ideas and the repetition will improve your understanding over time.

Suggestions for Study

Enter the program of Figure 7.1 and verify that it will control both the real and simulated robot as expected.

Notice that the simulated robot always turns the same direction in order to face the specified heading. Modify the program so the simulated robot will turn either left or right based on which direction is closest to the intended destination. Hint: Use the **`CompareCompass`** routine to determine which way to turn.