Robot Projects for RobotBASIC
Volume I: The Fundamentals

Project 8: Detecting Beacons

One of the most powerful features of the RobotBASIC Robots is their ability to detect beacons. A beacon detector can be an extremely useful navigation tool that can give your robot capabilities other robots only dream of. Our book *Robot Programmer's Bonanza* (available from Amazon.com) describes in detail how to navigate through a home or office environment using strategically placed beacons. This project will provide a brief introduction to beacons.

What is a Beacon?

In general, a beacon can be anything unique that a robot can locate and face. The simulated robot uses colors to represent its beacons. Such an idea may sound strange, but a robot with a camera could actually use small disks of unusual colors for beacons. The RB-9 Robot contains a special circuit capable of detecting infrared (IR) light pulsing at 15 different frequency patterns.

RobotBASIC offers beacon kits that make it easy to construct RobotBASIC compatible beacons as shown in Figure 8.1. These beacons can be placed at strategic locations to give your robot points of reference or to allow it to locate a particular place or object. The kits come in RED or BLUE. Red beacons can have identifying numbers from 1 to 8 and blue beacons from 8 to 15. Now that you know what a beacon is, let's see how to detect one.

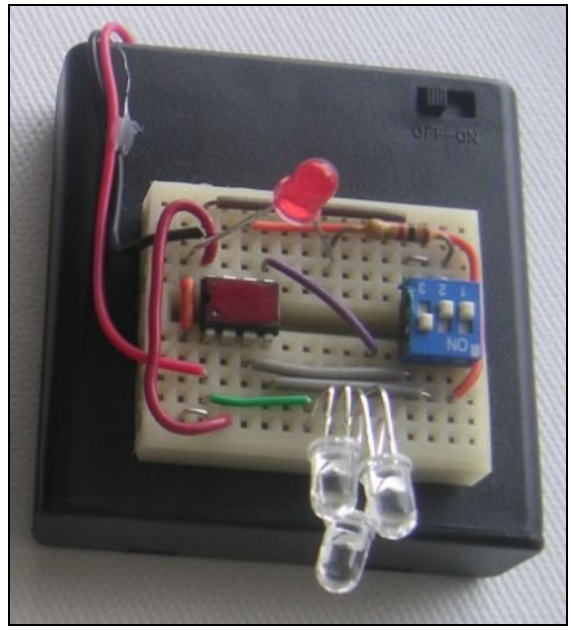

Figure 8.1: The beacon kit contains a preprogrammed micro-controller that makes it easy to create your own beacons.

Detecting a Beacon

You can use the function **rBeacon(param)** to see if a beacon is within range and directly in front of the robot. The value of **param** specifies the beacon number (or color number if using the simulator) that you wish to detect. For example, the following command will assign the variable **x** a value of **TRUE** if beacon 3 is seen and **FALSE** otherwise.

```
x = rBeacon(3)
```

Differences

When used with the simulator, the function **rBeacon** actually returns the distance to the beacon if one is seen. Since the distance is a non-zero value, it can be thought of as true. While it is possible to build real beacon sensors that can determine how far away a beacon is, the sensor used on the RB-9 robot does not have this capability. In order to ensure that all of our projects will work with both the simulation and the real robot, our programs will only check for true/false conditions.

There is another difference with the real robot's version of **rBeacon** compared to the simulator. If the parameter passed is zero (instead of a beacon number) then the real robot will return **true** if *any* beacon is seen (the simulator would only look for black, which is color 0 in RobotBASIC).

Future projects will show it is a valuable robotic behavior for the robot to turn to face a beacon. This could be done using standard RobotBASIC commands, but the RB-9 robot would have to rotate excruciatingly slow because of the communication delays between RobotBASIC and the RROS. To solve this dilemma, we added the following **rCommand**:

```
rCommand(FindBeacon, param)
```

When the above command is executed, the real robot will rotate on its own, looking for a beacon. Since everything is done by the RROS, there are no communication delays. Once a beacon is found the application program can use **rBeacon** to determine *which* beacon was found and either use it or move to find another beacon. The value of **param** will determine which direction the robot rotates. A value of 0 indicates left while 1 means right. There are two more **rCommands** that are useful during standard beacon operations. Let's look at them now.

Setting the Beacon Time

When you ask the robot to find a beacon, you do not want it to continue rotating forever if no beacon exists within range. Therefore, a time limit can be set that gives the robot a maximum time to look for a beacon. Generally, you should set the time limit to allow the robot to rotate at least 360º by changing the **param** in the following statement. The default value is 70.

```
rCommand(SetBeaconTime, param)
```

Muting the Beacon

The real robot has the option of creating a buzzing noise when the detector is pointing at a beacon – something that can be very helpful when troubleshooting beacon related behaviors. Normally this sound is disabled (muted), but you can enable it by changing **`param`** to **`TRUE`** or **`FALSE`** (or 1 or 0) in the following statement.

```
rCommand(MuteBeacon, param)
```

Even when muted, the sound transducer will produce a small tick each time a beacon is detected.

Using the Beacon Detector

In order to demonstrate how to use the beacon detector, we will write a program similar to the one used to introduce the compass in Project 7. For this example we will assume that the beacon for the simulator is a RED circle. The program should turn the robot to face the designated beacon and then move forward towards it until it detects an object (the beacon itself in this case, since RED has not been specified as invisible) within a specified distance.

The program in Figure 8.2 is structured very much like the compass example from Project 7 so it should be easy to examine and understand. When the simulation is used, the robot is created on the left side of the screen. Notice a subroutine creates the simulated beacon at a random position on the right half of the screen.

The sub-routine **`TurnToBeacon`** is passed the beacon number (or beacon color). Notice that both the simulation and the real robot turn left to look for the beacon. This means the robot will be pointed toward the RIGHT side of the beacon's beam when the beam is detected. More about this shortly.

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

bNum = 4   // RED is 4, change to the beacon number you
wish to use
Distance = 10 // 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
Number

Main:
  if not Real then gosub CreateBeacon
  gosub InitializeRobot
  call TurnToBeacon(bNum)
  call MoveToBeacon(bNum, Distance)
end
```

```
InitializeRobot:
  if Real
    gosub InitializeRealRobot
  else
    gosub InitializeSimulator
  endif
return

InitializeSimulator:
  // place the robot on the left side of the screen
  // at some RANDOM heading
  rLocate 200,300,random(360)
  // introduce error so the robot must correct as it moves
  rSlip 15
return

sub TurnToBeacon(a)
  if _Real
    while not(rBeacon(a)
      rCommand(FindBeacon,0)
    wend
  else
    while not rBeacon(a)
      rTurn -1
    wend
  endif
return

sub MoveToBeacon(a,d)
  while rRange(0)>d
    rForward 1
    if rBeacon(a)
      rTurn 1
    else
      rTurn -1
    endif
  wend
return

CreateBeacon:
  bx=400+Random(400)
  by=50+Random(500)
  circle bx-10,by-10,bx+10,by+10,RED,RED
return
```

Figure 2: This program turns the robot until it faces a specified beacon and then moves the robot forward until an object is encountered.

This process of finding the beacon is a little more complicated for the real robot than it is for the simulation. In this example, the real robot uses an **rCommand** to look for *any* beacon, and the process continues to do so *while* the standard **rBeacon** statement does not detect the *desired* beacon. This methodology would find the right beacon even if there were other beacons (of other colors or numbers) within the robot's environment.

The **MoveToBeacon** sub-routine will continue moving forward while the distance measured by the ranging sensor is greater than the distance specified at the beginning of the program. Within this loop, the robot will turn right if it sees the beacon and left if it does not. Notice that this is very similar to how the robot followed a line in Project 6. These choices make the robot stay on the right side of the beacon beam – which is where it stopped when the beacon was detected (as mentioned earlier)

If you run the program a few times using the simulator, you will see that occasionally the robot looses sight of the beacon and makes a circle trying to find it again. This happens because the robot turns too far left, so that it is on the left side of the line of sight to the beacon's beam. Once this happens, the robot will continue to turn left until it finds the beacon again. This problem can be solved if the simulation does not have as much random error, or if the beacon itself is larger. The important point is that the simulation allowed us to detect a potential problem even before we used the real robot.

Notice that the real beacon (Figure 8.1) has several IR LEDs to produce the beam of light the robot will try to follow. Using multiple LEDs effectively makes the beacon larger and easier to see from different positions in the room.

There is another problem. Notice that the simulated robot will occasionally collide with the beacon. This happens because the simulated ranging sensor only sees objects along a fairly straight line directly in front of the robot. Since the **MoveToBeacon** sub-routine keeps the robot at the edge of the beacon's signal it is possible for the robot to collide with the beacon before it is detected within the specified distance. As in the previous example, this problem can be greatly reduced if the random error is reduced. Another alternative is to use both the **rRange** and **rFeel** sensors to look for objects. Just replace the **while** statement in the **MoveToBeacon** sub-routine with the following statement.

```
while (rRange(0)>d) and (not rFeel())
```

When the above statement is used, the robot will stop if either of the two conditions is true. So it will stop if the ranging sensor sees something closer than the specified distance or it will stop if the **rFeel** sensors detect something. Notice that each of the two conditions are enclose with parenthesis and combined with an **and**. The projects in Volume II will explore situations like this in much more detail.

Suggestions for Study

Enter the program shown in Figure 8.2 and test it with both the simulation and the real robot. Try to verify the limitations discussed for the simulated robot and correct them as described in the text.

The range of the real beacon detector is based on the amount of power in the light beam generated by the beacon. Weak batteries in the beacon, for example, can greatly reduce how far away the robot can see it. Test your robot to see what its effective range is for detecting beacons.