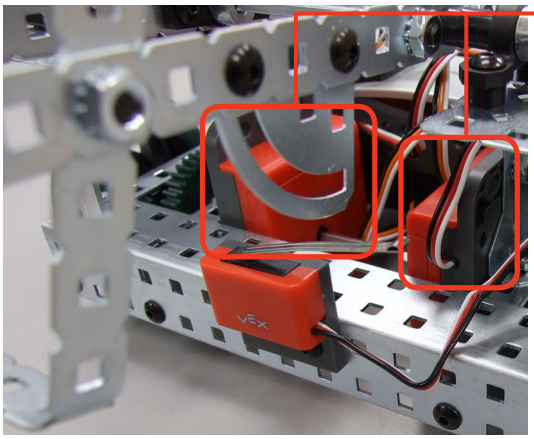
Autonomy/Encoders Forward for Distance

In this lesson, you will learn how to use an Encoder to more accurately control the distance that the robot will travel.

1. Your robot should have one encoder hooked up to the wheels on each side of the robot for this lesson. Consult the building instructions for Squarebot 3.0 if you have not already built or upgraded your model.

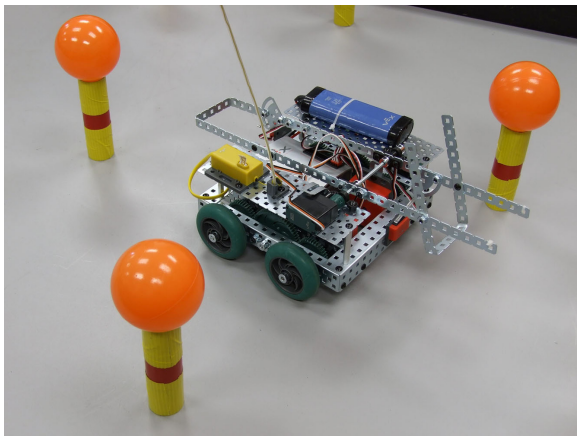


1. Attach Encoders
Upgrade your robot to Squarebot 3.0 or a similar design equipped with one encoder on each wheel.

> ***Important Note:***
>
> This unit is designed to work with the **original Encoders**, but will work with the newer **Quadrature Encoders**. To see which version you are using and learn their key differences, view the **Quadrature Encoders Reference Page** found in both the **Autonomy Section** and the **Product Index.**

In this unit, you will learn to use the encoders to control the **distance** the robot moves, and to **self-correct** its steering to counteract the random drift that occurs during movement. This will eventually serve two purposes. First, it will provide you with a new form of **Operator Assist** that uses the sensors on the robot to drive straighter than a human operator might be capable of doing.

Second, it will begin building a base of behaviors that the robot can use on its own during the **Autonomous Period** that is part of the revised challenge. For 20 seconds at the start of the match, your robot will have a chance to run and score points completely on its own.



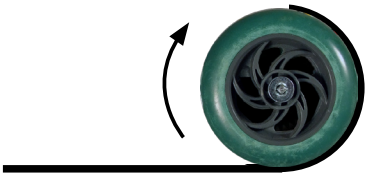
This is similar to what you did initially with the Labyrinth challenge, but a good deal more difficult. This task is much more sensitive to exact positioning. The wide halls of the Labyrinth were forgiving, but picking up mines leaves much less room for error.

Autonomy/Encoders Forward for Distance *(cont.)*

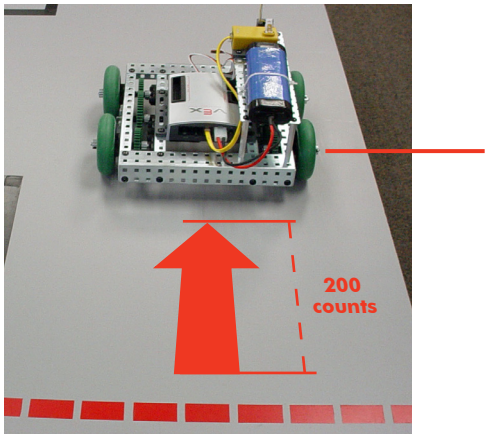
Encoder

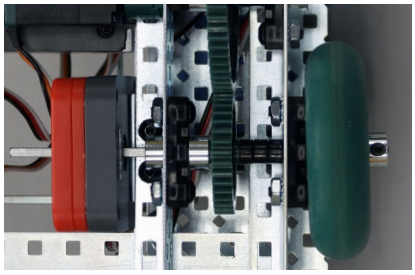
The **Encoder** is a sensor that measures rotation. On the Squarebot 3.0, encoders are attached to the wheels of the robot. The encoders measure how far the wheels have turned, and therefore, how far the robot has moved.

The turning of the wheel is what propels the robot forward. Because the outer surface of the tire generates a large amount of friction, the wheel pushes against the ground as it turns, driving the robot forward. The distance the robot covers moving forward is equal to the length of the tire tread that spins by.



This enables us to make reasonably accurate predictions about exactly how far the robot will move after every full turn of the wheels. The encoder measures in 4-degree increments, and so will register 90 **counts** for a full 360-degree rotation. For this lesson, the important part is simply that the number of encoder counts is directly linked to the amount that the wheels turn, and therefore, to the distance that the robot travels.

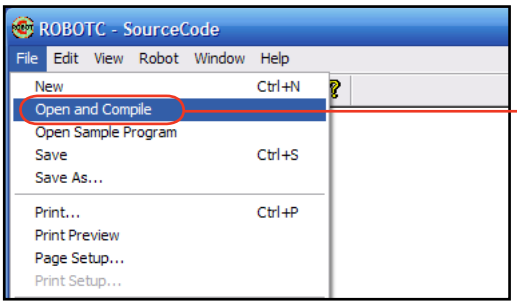




Being able to directly measure the turning of the wheels allows us to dictate the motion of the robot in terms of **distances** rather than simply defining a period of time that the robot should move. This is a much more reliable method, because it is unaffected by factors like speed or battery power. Using the encoder-based distance to determine movement, therefore, will result in much better consistency of movement. This is exactly what your robot needs to do its job well, with or without human supervision!

Autonomy/Encoders Forward for Distance *(cont.)*

2. Create a new ROBOTC program.



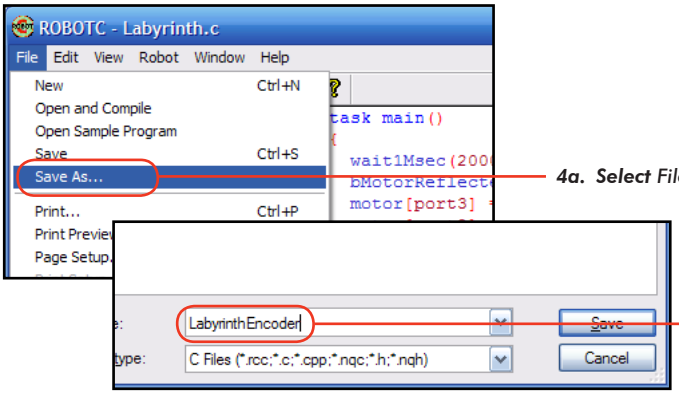
2. Create a new program
Select *File* > *New*

3. Add the following code to your program.

```
1   task main()
2   {
3       wait1Msec(2000);
4       bMotorReflected[port2] = 1;
5   }
```

3. Add this code
Fill in the basic parts of the new program: **`task main`**, a **`wait1Msec`** command to delay starting for 2 seconds, and a **`bMotorReflected`** command to reflect the motor on port 2.

4. Save your program as "LabyrinthEncoder".

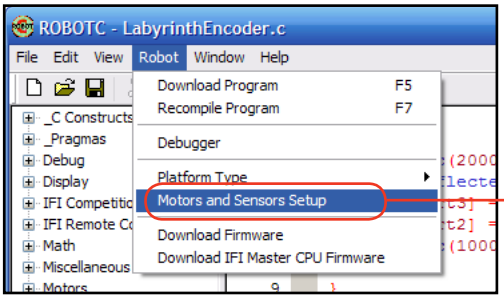


***4a. Select** File > Save As*

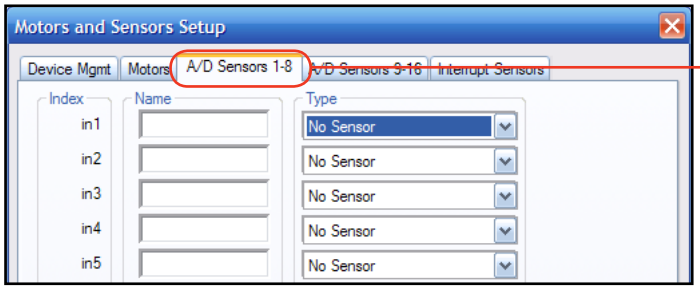
4b. Save As "LabyrinthEncoder"
Save your program in the usual folder, with the name "LabyrinthEncoder".

Autonomy/Encoders Forward for Distance *(cont.)*

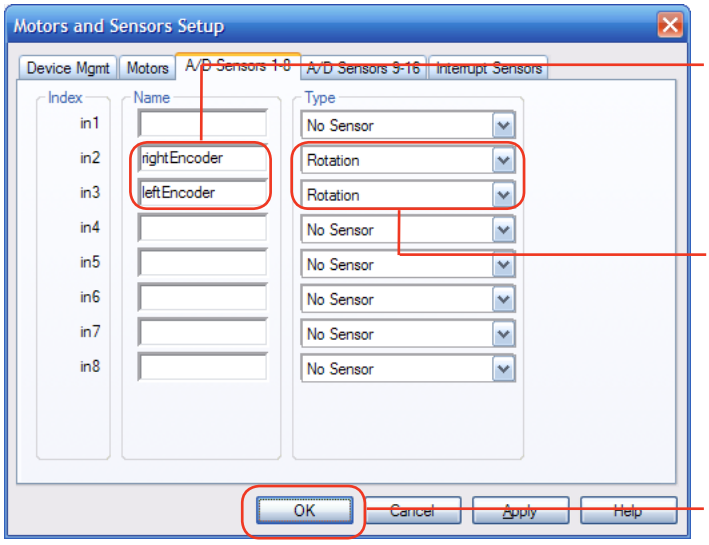
5. Configure the two new Encoder sensors using the ***Motors and Sensors Setup*** menu. Conveniently, Squarebot 3.0 is designed so that the Encoder port numbers match the motors they are associated with. The encoder attached to the Port 2 motor is on Analog/Digital (A/D) Sensor Port 2. The encoder on A/D Sensor Port 3 goes with the motor on Motor Port 3.



5a. Open the Motors and Sensors Setup Menu
Go to *Robot* > *Motors and Sensors Setup*.



5b. Select "A/D Sensors 1-8"
Open the tab marked "A/D Sensors 1-8".



5c. "Name" the sensors
Assign the name "leftEncoder" to the sensor in port "in3" (A/D input 3). Name the "in2" (A/D input 2) sensor "rightEncoder".

5d. Set the "Type" of both sensors
Identify the sensor attached to both in2 and in3 as "Rotation" sensors (Encoders sense rotation).

5e. Click OK
Confirm the new sensor configuration.

Autonomy/Encoders Forward for Distance *(cont.)*

Checkpoint

The encoders are now configured. While our eventual goal is to bring improved movement capabilities to the Minefield challenge, we will start first with an easier problem: the **Labyrinth** challenge from the Movement unit. In the original Labyrinth, the robot was programmed to travel down corridors and go around corners, relying on **`wait1Msec`** timing commands to control the duration of each movement. We will now revisit this problem, using encoders to control movement distance instead of timing.

This version of the program will rely on the encoders to tell the robot when it is time to start and stop the motors. The motors should continue running until the encoders say the robot has gone far enough. This is equivalent to "move while the encoders have moved fewer than the needed number of counts."

We will use a **`while`** loop that runs the motors **while** the encoder count remains **less than** the target number. This number can be calculated, but for now, we'll make an educated guess of 500 counts (about five and a half wheel rotations). Furthermore, because the wheels should be going about the same speed (we'll double-check this assumption later), we only need to watch one of the two encoders to see how far the robot has gone.

---

6. Add a the first movement to run for a certain number of Encoder counts. Remember that encoders must always be reset before use.

```
Auto  const tSensors rightEncoder = (tSensors) in2;
Auto  const tSensors leftEncoder  = (tSensors) in3;
```

```
1   task main()
2   {
3        wait1Msec(2000);
4        bMotorReflected[port2] = 1;
5        SensorValue[leftEncoder] = 0;
6        while(SensorValue[leftEncoder] < 500)
7        {
8           motor[port3] = 63;
9           motor[port2] = 63;
10       }
11  }
```

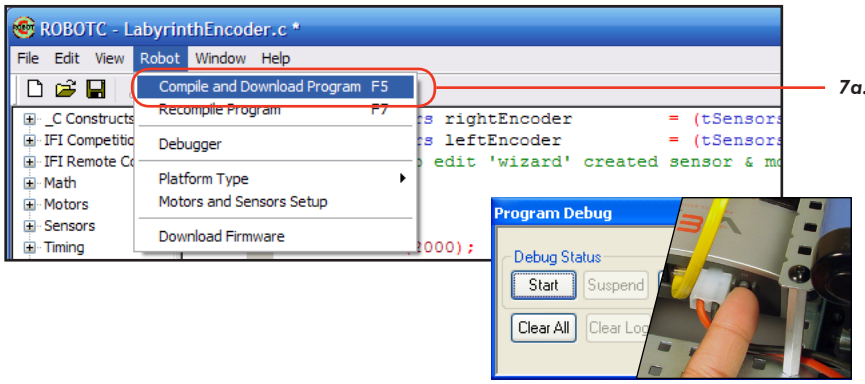
6a. Add this code
The encoder must be reset before use. Its value can be directly accessed and modified by setting the value of **`SensorValue[leftEncoder]`** just like you would set the value of a variable.

6b. Add this code
The first movement should now be based around a **`while`** loop. The loop continues running the movement commands while the number of encoder counts remains lower than the needed value of 500.

When the encoder counts reach 500, the loop ends and the robot will move on to the next lines of code. (That is, the robot will stop because the next line is the end of the program).

Autonomy/Encoders Forward for Distance *(cont.)*

7. Compile, download and run the program.



7a. Compile and download
Make sure that your robot is turned on and that the robot is plugged into your computer with the USB cable. From the menu, select ***Robot*** > ***Compile and Download Program***.

7b. Run the Program

Checkpoint

Your robot should begin executing the first behavior two seconds after it is turned on. The robot will start moving — and keep moving! If you pick it up and let it run for a while, you will notice that it eventually stops, but nowhere near the right distance. The guess we used for the number of encoder counts was off. 500 is far too many counts for the distance that the robot needs to move.

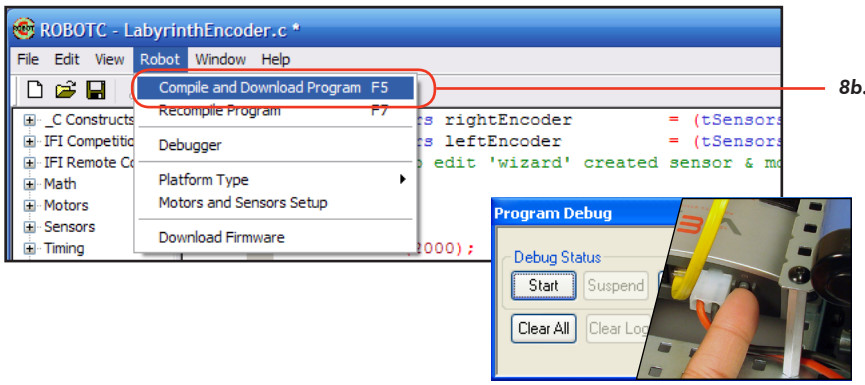
Repeat step 8 below to change the number of encoder counts that the robot will run for, until you have found a number that works best for your robot and Labyrinth.

8. Adjust the number of rotations in your program to a more appropriate number.

```
Auto  const tSensors rightEncoder = (tSensors) in2;
Auto  const tSensors leftEncoder  = (tSensors) in3;

 1    task main()
 2    {
 3         wait1Msec(2000);
 4         bMotorReflected[port2] = 1;
 5         SensorValue[leftEncoder] = 0;
 6         while(SensorValue[leftEncoder] < 240)
 7         {
 8            motor[port3] = 63;
 9            motor[port2] = 63;
10         }
11    }
```

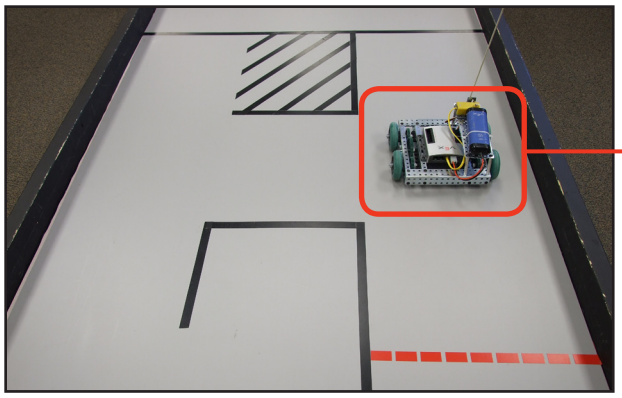
8a. Modify this code
Change the number of encoder counts for which the moving forward loop will continue running.

Autonomy/Encoders Forward for Distance *(cont.)*



8b. Compile and download
Make sure that your robot is turned on and that the robot is plugged into your computer with the USB cable. From the menu, select ***Robot*** > ***Compile and Download Program***.

8c. Run the Program



8d. Test robot
See whether your robot now goes the correct distance.

8e. Repeat
Continue adjusting the number of encoder counts until your robot stops in the right spot.

8. Add the next movement behavior in the Labyrinth.

```
Auto  const tSensors rightEncoder = (tSensors) in2;
Auto  const tSensors leftEncoder  = (tSensors) in3;

 1    task main()
 2    {
 3         wait1Msec(2000);
 4         bMotorReflected[port2] = 1;
 5         SensorValue[leftEncoder] = 0;
 6         while(SensorValue[leftEncoder] < 240)
 7         {
 8            motor[port3] = 63;
 9            motor[port2] = 63;
10         }
11         SensorValue[leftEncoder] = 0;
12         while(SensorValue[leftEncoder] < 70)
13         {
14            motor[port3] = -63;
15            motor[port2] = 63;
16         }
17    }
```

8. Add this code
Convert the turn just like you did the forward movement. You must also reset the encoder count again – the reset and the while loop are part of the same behavior and will generally stick together.

Note that the Encoder counts upward whenever the wheel turns, regardless of the direction, so even a backward-turning wheel will run the encoder count up rather than down.

The value 70 is an educated guess for the amount of wheel rotation required for a 90 degree turn.

Autonomy/Encoders Forward for Distance *(cont.)*

9. Repeat the reset-then-move pattern for all remaining units.

```
Auto  const tSensors rightEncoder = (tSensors) in2;
Auto  const tSensors leftEncoder  = (tSensors) in3;

 1    task main()
 2    {
 3         wait1Msec(2000);
 4         bMotorReflected[port2] = 1;
 5         SensorValue[leftEncoder] = 0;
 6         while(SensorValue[leftEncoder] < 240)
 7         {
 8            motor[port3] = 63;
 9            motor[port2] = 63;
10         }
11         SensorValue[leftEncoder] = 0;
12         while(SensorValue[leftEncoder] < 70)
13         {
14            motor[port3] = -63;
15            motor[port2] = 63;
16         }
17         SensorValue[leftEncoder] = 0;
18         while(SensorValue[leftEncoder] < 270)
19         {
20            motor[port3] = 63;
21            motor[port2] = 63;
22         }
23         SensorValue[leftEncoder] = 0;
24         while(SensorValue[leftEncoder] < 92)
25         {
26            motor[port3] = 63;
27            motor[port2] = -63;
28         }
29         SensorValue[leftEncoder] = 0;
30         while(SensorValue[leftEncoder] < 180)
31         {
32            motor[port3] = 63;
33            motor[port2] = 63;
34         }
35         SensorValue[leftEncoder] = 0;
36         while(SensorValue[leftEncoder] < 93)
37         {
38            motor[port3] = 63;
39            motor[port2] = -63;
40         }
41         SensorValue[leftEncoder] = 0;
42         while(SensorValue[leftEncoder] < 140)
43         {
44            motor[port3] = 63;
45            motor[port2] = 63;
46         }
47    }
```

Test As You Program

Write, download, and test each behavior one step at a time. This allows you to locate errors quickly and easily, since you will always know which section of code was being worked on when the problem first occurred.

9a. ***Add this code***
Add a forward movement for the second straight "hallway" on the board.

9b. ***Add this code***
Add an encoder-based right turn.

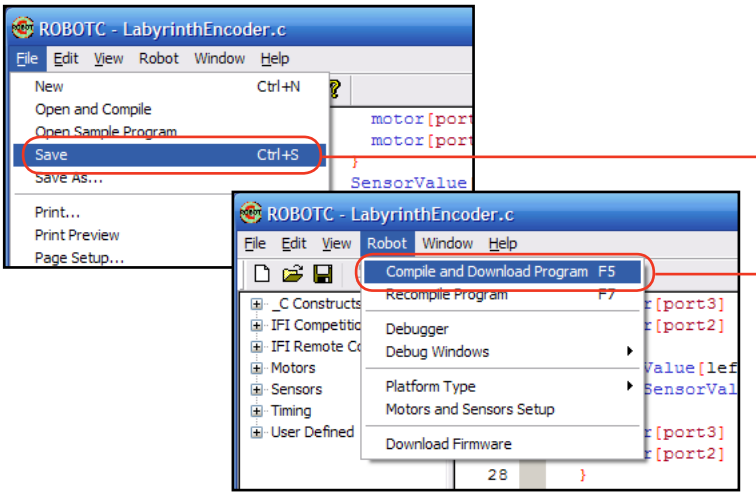
9c. ***Add this code***
Add a forward movement for the third straight "hallway" on the board.

9d. ***Modify this code***
Add another encoder-based right turn. It should be the same as the first one in 8b above.

9e. ***Add this code***
Add a forward movement for the final straight leg into the goal area.

Autonomy/Encoders Forward for Distance *(cont.)*

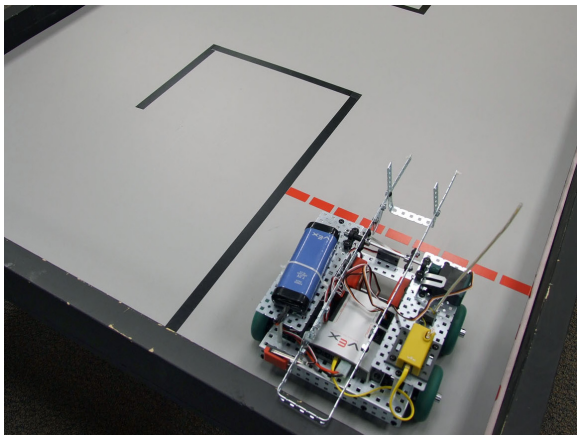
10. Save and download this program.



10a. Save your program
Select *File* > *Save*.

10b. Compile and download
Make sure that your robot is turned on and that the robot is plugged into your computer with the USB cable. From the menu, select *Robot* > *Compile and Download Program*.

11. Run the program. Make any final adjustments to the encoder values.



Checkpoint

The robot now works the way it appeared to work before. What did we gain? Supposedly, we have reduced the sensitivity of the robot to variations in speed. Is this true? In the past, changing the speed of the robot also required us to change the timing of its movements. If the robot is really better able to cope with speed changes, then we need to find out what happens when we change the motor powers.

Autonomy/Encoders Forward for Distance *(cont.)*

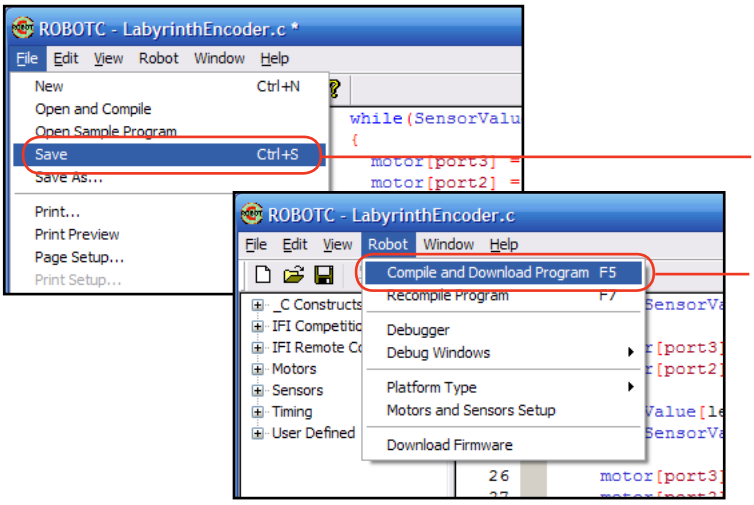
12. Change the speed at which the robot moves and run the course again.

```
Auto  const tSensors rightEncoder = (tSensors) in2;
Auto  const tSensors leftEncoder  = (tSensors) in3;

 1    task main()
 2    {
 3         wait1Msec(2000);
 4         bMotorReflected[port2] = 1;
 5         SensorValue[leftEncoder] = 0;
 6         while(SensorValue[leftEncoder] < 240)
 7         {
 8           motor[port3] = 96;
 9           motor[port2] = 96;
10         }
11         SensorValue[leftEncoder] = 0;
12         while(SensorValue[leftEncoder] < 70)
13         {
14           motor[port3] = -96;
15           motor[port2] = 96;
16         }
17         SensorValue[leftEncoder] = 0;
18         while(SensorValue[leftEncoder] < 270)
19         {
20           motor[port3] = 96;
21           motor[port2] = 96;
22         }
23         SensorValue[leftEncoder] = 0;
24         while(SensorValue[leftEncoder] < 92)
25         {
26           motor[port3] = 96;
27           motor[port2] = -96;
28         }
29         SensorValue[leftEncoder] = 0;
30         while(SensorValue[leftEncoder] < 180)
31         {
32           motor[port3] = 96;
33           motor[port2] = 96;
34         }
35         SensorValue[leftEncoder] = 0;
36         while(SensorValue[leftEncoder] < 93)
37         {
38           motor[port3] = 96;
39           motor[port2] = -96;
40         }
41         SensorValue[leftEncoder] = 0;
42         while(SensorValue[leftEncoder] < 140)
43         {
44           motor[port3] = 96;
45           motor[port2] = 96;
46         }
47    }
```

12. *Modify this code*
Change the motor power level in all the movements to 96 (or -96), about 3/4ths of full power.

Autonomy/Encoders Forward for Distance *(cont.)*

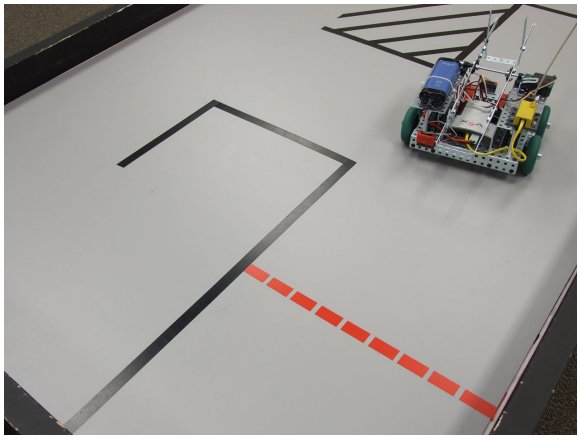
13. Save and download this program.



13a. Save your program
Select *File* > *Save*.

13b. Compile and download
Make sure that your robot is turned on and that the robot is plugged into your computer with the USB cable. From the menu, select *Robot* > *Compile and Download Program*.

14. Run the program again and observe the behavior of the robot.



14. Run the program and observe the robot
Although the robot is moving at half speed, it is still able to navigate the labyrinth!

End of Section

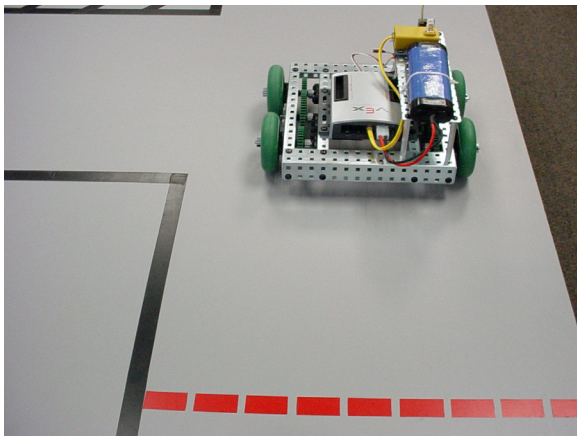
By using the encoders to track the movement of your robot in terms of wheel rotations rather than making a risky assumption about its speed over time, you have increased its ability to perform the Labyrinth behavior consistently. Variations in battery power and other mechanical factors should now have a much more limited impact on the performance of your robot. In fact, even making a major change in motor power in the program did not change the distance that the robot moved.

This is a good start to our experience with encoders, but surely we can do something more powerful with them.

Autonomy/Encoders Automated Straightening

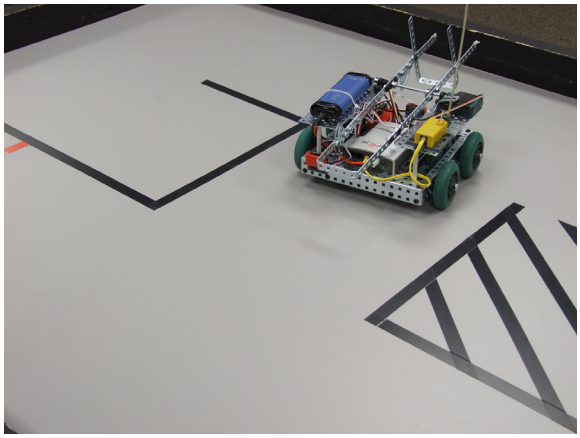
In this lesson, you will learn how to use a pair of encoders to straighten out the movement of the robot.

Long ago, in the "Manual Straightening" lesson, you hand-tweaked the power values of two motor commands to make your robot move straighter. Even so, the results were not always spectacular. There are simply too many unknown and changing quantities to be able to compensate for them all ahead of time. However, we can correct problems when they do happen if we can tell that they are happening.



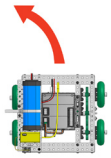
We can tell whether the robot is going straight by seeing if the wheels are **together**. To move in a straight line, the wheels of the robot need to stay together as they move. **If** they move at different speeds or in different directions, the robot turns. **If** both wheels have traveled the same distance at the same point in time, the robot is most likely moving straight. However, if the wheels have not traveled the same distance at the same point in time, the robot is turning toward one side.

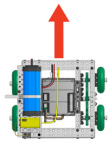
In fact, by looking at the two distances, we can even tell which way the robot is turning. The wheel that has gone farther is ahead, and the wheel that has not gone as far is behind. The body of the robot will be tilted toward the side that is behind. Effectively, the robot turns toward the wheel that has gone the shorter distance.

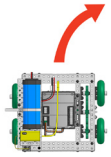


Autonomy/Encoders Automatic Straightening *(cont.)*

Relative encoder values	Robot path	Corrective Action

left < right	 Drifting left	Slight right turn
		Example:
		left motor = 63;
		right motor = 50;

left == right	 Straight	No correction
		Example:
		left motor = 63;
		right motor = 63;

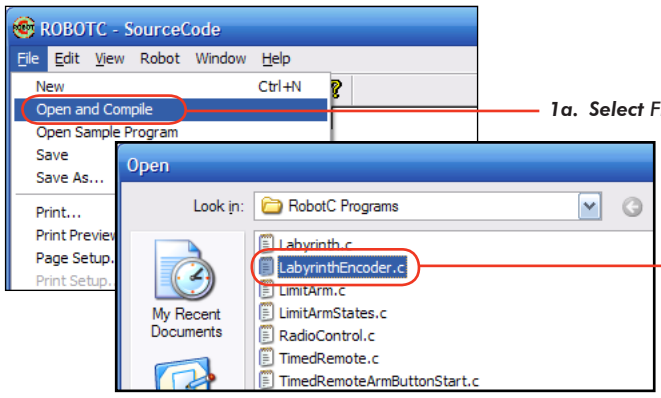
left > right	 Drifting right	Slight left turn
		Example:
		left motor = 50;
		right motor = 63;

Encoders and Paths

By comparing the values of the encoders on the left and right wheels of the robot, you can get a general idea of what path the robot is following (left, right, or straight). Correcting the robot toward straight-line motion is as simple as telling it to adjust its motor powers to push it in the direction opposite to the way it drifts. These comparisons and corresponding corrections can be implemented into the program using three simple if-statements embedded in each "moving forward" while loop, one for each possible path (left, right, or straight).

If your robot continually monitors and corrects its path, it should be able to drive straight even with the motor variation that plagued earlier methods. Now, we will program it.

1. Open the "LabyrinthEncoder" program from the previous lesson.



***1a. Select** File > Open and Compile*

1b. Open the "LabyrinthEncoder" program
Find "LabyrinthEncoder" in the directory where you normally save your programs, and double-click to open it (or select it and click ***Open***).

Autonomy/Encoders Automatic Straightening *(cont.)*

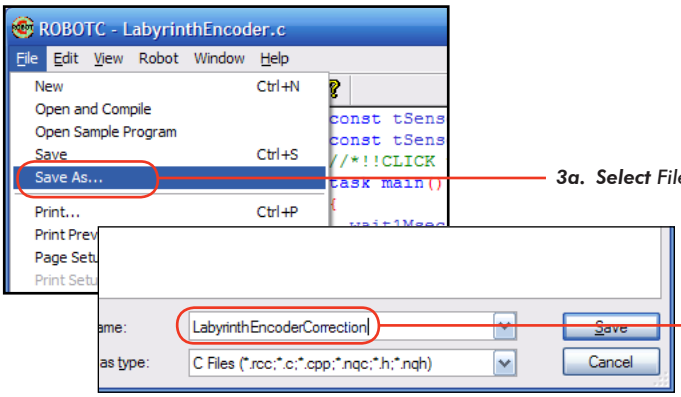
2. Review your "LabyrinthEncoder" program and identify the code within each "moving forward" while loop. These sections of the program will be modified using the self-straightening code.

```
Auto  const tSensors rightEncoder = (tSensors) in2;
Auto  const tSensors leftEncoder  = (tSensors) in3;

 1    task main()
 2    {
 3         wait1Msec(2000);
 4         bMotorReflected[port2] = 1;
 5         SensorValue[leftEncoder] = 0;
 6         while(SensorValue[leftEncoder] < 240)
 7         {
 8           motor[port3] = 96;
 9           motor[port2] = 96;
10         }
11         SensorValue[leftEncoder] = 0;
12         while(SensorValue[leftEncoder] < 70)
13         {
14           motor[port3] = -96;
15           motor[port2] = 96;
16         }
17         SensorValue[leftEncoder] = 0;
18         while(SensorValue[leftEncoder] < 270)
19         {
20           motor[port3] = 96;
21           motor[port2] = 96;
22         }
23         SensorValue[leftEncoder] = 0;
24         while(SensorValue[leftEncoder] < 92)
25         {
26           motor[port3] = 96;
27           motor[port2] = -96;
28         }
29         SensorValue[leftEncoder] = 0;
30         while(SensorValue[leftEncoder] < 180)
31         {
32           motor[port3] = 96;
33           motor[port2] = 96;
34         }
35         SensorValue[leftEncoder] = 0;
36         while(SensorValue[leftEncoder] < 93)
37         {
38           motor[port3] = 96;
39           motor[port2] = -96;
40         }
41         SensorValue[leftEncoder] = 0;
42         while(SensorValue[leftEncoder] < 140)
43         {
44           motor[port3] = 96;
45           motor[port2] = 96;
46         }
47    }
```

2. *Identify this code*
The "moving forward" sections of code will have similar or identical motor power level values. This is the code that will be replaced using the three if-statements.

Autonomy/Encoders Automatic Straightening *(cont.)*

3. Save this program as "LabyrinthEncoderCorrection" before making changes.



***3a. Select** File > Save As*

3b. Save As "LabyrinthEncoderCorrection"
Save your program in the usual folder, with the name "LabyrinthEncoderCorrection".

4. Since both encoders are now being used, make sure both are reset before use. Then, change the movement commands so that the correct *adjusted* movements are chosen based on the relative values of the encoders.

```
Auto  const tSensors rightEncoder = (tSensors) in2;
Auto  const tSensors leftEncoder  = (tSensors) in3;

 1    task main()
 2    {
 3      wait1Msec(2000);
 4      bMotorReflected[port2] = 1;
 5      SensorValue[leftEncoder] = 0;
 6      SensorValue[rightEncoder] = 0;
 7      while(SensorValue[leftEncoder] < 240)
 8      {
 9        if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
10        {
11          motor[port3] = 50;
12          motor[port2] = 63;
13        }
14        if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
15        {
16          motor[port3] = 63;
17          motor[port2] = 50;
18        }
19        if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
20        {
21          motor[port3] = 63;
22          motor[port2] = 63;
23        }
24      }
25      SensorValue[leftEncoder] = 0;
26      while(SensorValue[leftEncoder] < 70)
27      {
28          motor[port3] = -96;
29          motor[port2] = 96;
```

4a. Add this code
Reset the second encoder before use.

4b. Modify this code
Replace the "moving forward" sections of code with a set of three if-statements that determine which *corrected* movement is most appropriate based on the current encoder values.

Autonomy/Encoders Automatic Straightening *(cont.)*

4.1 (Optional) Comment out later behaviors to allow the first segment to run alone.

```
23                  motor[port2] = 63;
24              }
25          }
26  /*      SensorValue[leftEncoder] = 0;
27          while(SensorValue[leftEncoder] < 70)
28          {
29              motor[port3] = -96;
30              motor[port2] = 96;
31          }
            ...
57          while(SensorValue[leftEncoder] < 140)
58          {
59              motor[port3] = 96;
60              motor[port2] = 96;
61          }
62  */
63  }
```

3.1 Use /* Multiline Comments*/ to comment out other behaviors
Take advantage of the fact that the ROBOTC ignores any text **`/* between multiline comment markers */`** by temporarily disguising the other behaviors as comments so they will not run as commands in the program.

Commenting Out Code

```
23
24  /*      SensorValue[leftEncoder] = 0;
25          while(SensorValue[leftEncoder] < 70)
26          {
27              motor[port3] = -96;
28              motor[port2] = 96;
29          }
30  */
31
```

When ROBOTC compiles your program, it ignores all text between the multiline comment markers (**`/*`** and **`*/`**).

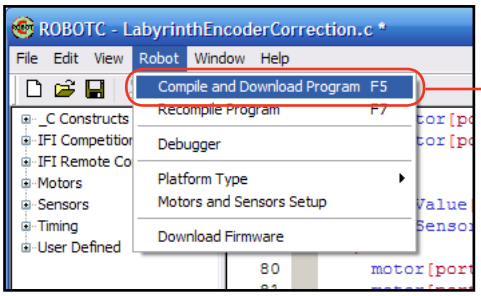
Placing comment markers around lines of code "disguises" them as comments rather than commands. This is a great way to temporarily disable parts of your program so you can test others.

Simply delete the multiline comment markers when you're ready to compile and run your entire program.

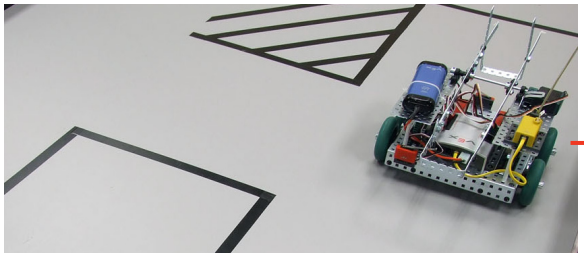
Disabling code in this way is called "commenting out" the code. Comment text is displayed in green in ROBOTC.

Autonomy/Encoders Automatic Straightening *(cont.)*

5. Test the first movement of your robot.



5a. Compile and download
Make sure that your robot is turned on and that the robot is plugged into your computer with the USB cable. From the menu, select ***Robot*** > ***Compile and Download Program***.



5b. Run your robot
Your robot now moves in a straighter line.

Checkpoint

The first movement of your robot (the one that was changed) is now very straight. If you would like to see the effect more clearly, try running the behavior for a longer distance and comparing it to the regular, non-correcting behavior.

Your Automatic Straightening behavior is a great improvement over the simple moving-forward behavior that you used back when you first worked on this challenge. Complete the upgrade by changing the remaining moving-forward behaviors in the Labyrinth to moving-straight behaviors as well.

Autonomy/Encoders Automatic Straightening *(cont.)*

6. Change the second, third, and fourth moving-forward behaviors to use the same self-correction code as the first. Remember to reset both encoders between behaviors.

```
28          motor[port3] = -96;
29          motor[port2] = 96;
30      }
31      SensorValue[leftEncoder] = 0;
32      while(SensorValue[leftEncoder] < 270)
33      {
34          if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
35          {
36            motor[port3] = 50;
37            motor[port2] = 63;
38          }
39          if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
40          {
41            motor[port3] = 63;
42            motor[port2] = 50;
43          }
44          if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
45          {
46            motor[port3] = 63;
47            motor[port2] = 63;
48          }
49      }
50      SensorValue[leftEncoder] = 0;
51      while(SensorValue[leftEncoder] < 92)
52      {
53          motor[port3] = 96;
54          motor[port2] = -96;
55      }
56      SensorValue[leftEncoder] = 0;
57      while(SensorValue[leftEncoder] < 180)
58      {
59          if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
60          {
61            motor[port3] = 50;
62            motor[port2] = 63;
63          }
64          if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
65          {
66            motor[port3] = 63;
67            motor[port2] = 50;
68          }
69          if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
70          {
71            motor[port3] = 63;
72            motor[port2] = 63;
73          }
74      }
```

6a. Modify this code Replace the pre-set motor commands in each "moving forward" section with the self-correcting code.

Autonomy/Encoders Automatic Straightening *(cont.)*

```
75      SensorValue[leftEncoder] = 0;
76      while(SensorValue[leftEncoder] < 93)
77      {
78          motor[port3] = 96;
79          motor[port2] = -96;
80      }
81      SensorValue[leftEncoder] = 0;
82      while(SensorValue[leftEncoder] < 140)
83      {
84          if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
85          {
86             motor[port3] = 50;
87             motor[port2] = 63;
88          }
89          if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
90          {
91             motor[port3] = 63;
92             motor[port2] = 50;
93          }
94          if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
95          {
96             motor[port3] = 63;
97             motor[port2] = 63;
98          }
99      }
100   }
```

6b. *Modify this code*
Replace the pre-set motor commands in each "moving forward" section with the self-correcting code.

```
28          motor[port3] = -96;
29          motor[port2] = 96;
30      }
31      SensorValue[leftEncoder] = 0;
32      SensorValue[rightEncoder] = 0;
33      while(SensorValue[leftEncoder] < 270)
34      {
35          if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
36          {
```

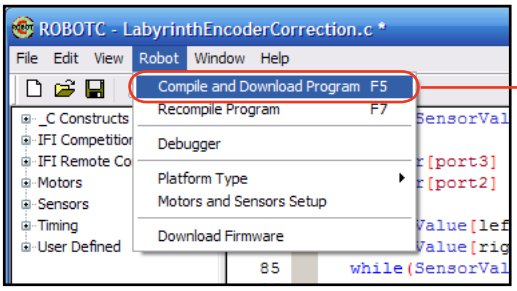
```
54          motor[port3] = 96;
55          motor[port2] = -96;
56      }
57      SensorValue[leftEncoder] = 0;
58      SensorValue[rightEncoder] = 0;
59      while(SensorValue[leftEncoder] < 180)
60      {
61          if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
62          {
```

```
28          motor[port3] = -96;
29          motor[port2] = 96;
30      }
31      SensorValue[leftEncoder] = 0;
32      SensorValue[rightEncoder] = 0;
33      while(SensorValue[leftEncoder] < 270)
34      {
35          if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
36          {
```

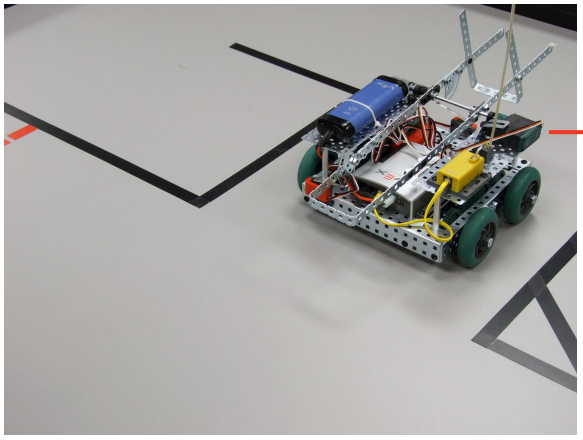
6c. *Modify this code*
Add a command to reset the second encoder between each pair of behaviors.

Autonomy/Encoders Automatic Straightening *(cont.)*

7. Compile, download, and run your program.



7a. Compile and download
Make sure that your robot is turned on and that the robot is plugged into your computer with the USB cable. From the menu, select ***Robot*** > ***Compile and Download Program***.



7b. Run your program
Run the program with the new, self-correcting code. Does the robot act differently now?

End of Section

The robot completes its task – especially the moving-straight portions – very reliably. So reliably, in fact, that it might even be practical to run try making them a little faster. What would it take to speed things up a little?

Assuming that you wanted every moving-forward behavior to use the motor powers 127 and 100 instead of 63 and 50, what would you have to change in the program?

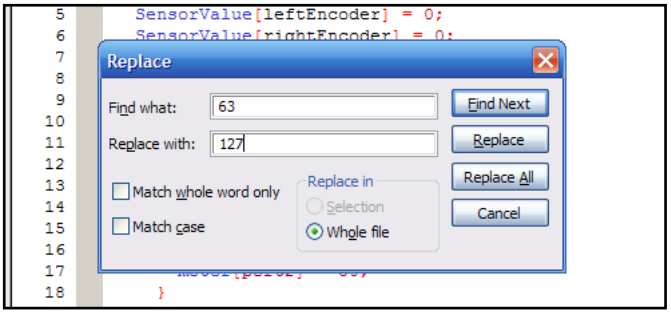
Autonomy/Encoders Automatic Straightening *(cont.)*

```
6       SensorValue[rightEncoder] = 0;
7       while(SensorValue[leftEncoder] < 270)
8       {
9           if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
10          {
11            motor[port3] = 100;
12            motor[port2] = 127;
13          }
14          if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
15          {
16            motor[port3] = 127;
17            motor[port2] = 100;
18          }
19          if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
20          {
21            motor[port3] = 127;
22            motor[port2] = 127;
23          }
```

Values that should change
All of the commands that set the motor power level to 63 or 50 must be changed to set the motor power level to 127 or 100. The remaining "move foward" behaviors should change as well.

Those are quite a few changes to make. If you had to go back and change all those values by hand, it could take a while, and the chances of accidentally missing one of the numbers is high. On the whole, it is a bit risky, and very inconvenient.

What about having the computer do the changes for us?



The *Edit* > *Find and Replace* feature in ROBOTC looks promising. However, it is too powerful for what we need. The number 63 (which we want to replace) appears in the "turn" behaviors as well as in the "move forward" behaviors. Since we do not want the robot to move any faster in the turns than it already does, we cannot change this value in the "turn" behaviors!

Really, it seems like this whole ordeal is unnecessary. The command for setting the motor power inside all four of these behaviors is exactly the same. Why do we have to make the change four different times in four different places when they're all exactly the same behavior?

Recognizing that all four of these chunks of code represent the **same behavior** is key. In the next lesson, you will learn how a control structure called a **function** can help by letting you define the moving-straight behavior as a custom command in the language of ROBOTC!

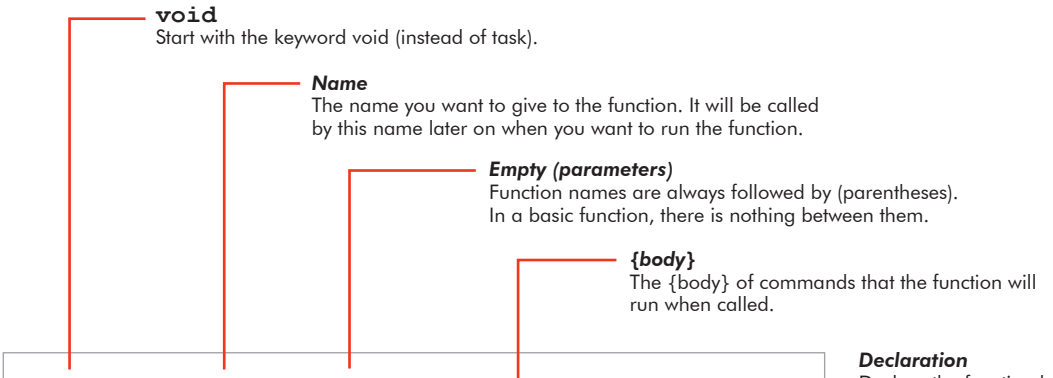
Autonomy/Encoders Behaviors and Functions

In this lesson, you will learn how to use functions to encapsulate the behaviors of your robot. Functions makes it easier for you to reuse and modify behaviors.

Functions in ROBOTC allow you to write a piece of code once, give that piece of code (usually a simple behavior) a name, and use it in multiple places. In effect, functions let you create your own commands in the language of ROBOTC!

How functions work

At a high level, a function must be created (or **declared**) before it can be used. To declare a function, you must give it a name and write the code that it executes when it is run. Once it is declared, the function can be called anywhere in the program by simply using its name as a command. For example, if your function was named `moveStraight()`, you would simply type `moveStraight();` as a command in your program.



Declaration
Declare the function before the main task starts. Similar to task main, it has a {body} of code that contains the commands that it will run.

```
task main()
{

  ... other code ...

  functionName();

  ... other code ...

}
```

Call
Call the function when you want it to run, by simply using its name as a command. Make sure to include the (parentheses) and a semicolon; afterwards.

The program will behave as if the entire {body} of the function were dropped into the code at the place where its name was used.

Autonomy/Encoders Behaviors and Functions *(cont.)*

Since we identified moving straight as a repeated behavior at the end of the last lesson, it is an excellent candidate for being made into a function. Such a function may look like this:

```
void moveStraight()
{
  if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
  {
     motor[port3] = 50;
     motor[port2] = 63;
  }
  if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
  {
     motor[port3] = 63;
     motor[port2] = 50;
  }
  if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
  {
     motor[port3] = 63;
     motor[port2] = 63;
  }
}
```

Declaration (at top)
Declares a function named **`moveStraight()`**, which checks the motor encoder and chooses the appropriate compensating power levels for each.

```
while(SensorValue[leftEncoder] < 270)
{
  moveStraight();
}
```

Call (inside `task main`)
At the appropriate point in **`main`**, the **`movestraight()`** function is called, and runs its {body} at that point in the code.

```
while(SensorValue[leftEncoder] < 270)
{
  if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
  {
     motor[port3] = 50;
     motor[port2] = 63;
  }
  if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
  {
     motor[port3] = 63;
     motor[port2] = 50;
  }
  if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
  {
     motor[port3] = 63;
     motor[port2] = 63;
  }
}
```

Result
The result is the same as if the following code were run at the calling point in **`main`**.

Autonomy/Encoders Behaviors and Functions *(cont.)*

1. Create a function named **`moveStraight()`**.

```
Auto  const tSensors rightEncoder = (tSensors) in2;
Auto  const tSensors leftEncoder  = (tSensors) in3;

 1    void moveStraight()
 2    {
 3
 4    }
 5
 6    task main()
 7    {
 8      wait1Msec(2000);
 9      bMotorReflected[port2] = 1;
10      SensorValue[leftEncoder] = 0;
11      SensorValue[rightEncoder] = 0;
```

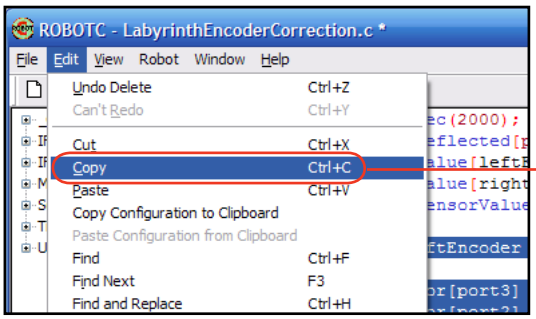
1a. Add this code
Near the top of your program, where the numbered lines start (just below the "Auto" lines), declare your function starting with the word **`void`**, followed by the function name and parentheses.

1b. Add this code
Create the {body} for the function, but leave it empty for now.

2. Copy the commands for the self-correcting movement behavior into the {body} of the new function.

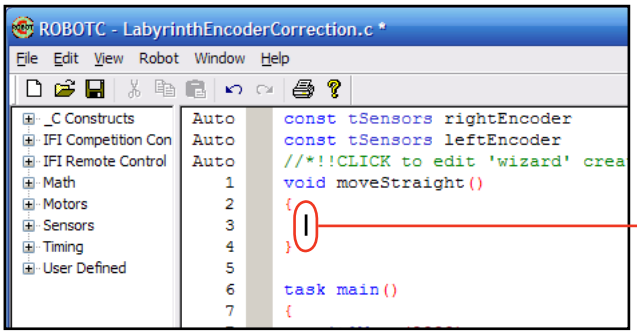
```
10      SensorValue[leftEncoder] = 0;
11      SensorValue[rightEncoder] = 0;
12      while(SensorValue[leftEncoder] < 240)
13      {
14        if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
15        {
16          motor[port3] = 50;
17          motor[port2] = 63;
18        }
19        if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
20        {
21          motor[port3] = 63;
22          motor[port2] = 50;
23        }
24        if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
25        {
26          motor[port3] = 63;
27          motor[port2] = 63;
28        }
29      }
30      SensorValue[leftEncoder] = 0;
31      while(SensorValue[leftEncoder] < 70)
```

2a. Highlight code
Find one instance of the group of **`if`**-statements that make up the straightening behavior, and highlight it.

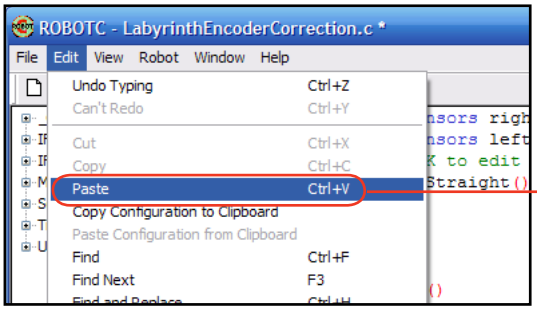


2b. Copy this code
Select *Edit* > *Copy* to copy the highlighted code onto the clipboard.

Autonomy/Encoders Behaviors and Functions *(cont.)*



2c. Place cursor here
Place your cursor inside the empty {body} of the **`moveStraight()`** function.



2d. Paste the copied code into the body of the function
Select *Edit* > *Paste* to put the copied code into the program at the spot where the cursor is.

Checkpoint

You have now created a function named **`moveStraight()`**, which contains all the necessary commands to make your robot run straight using the self-correction strategy from the previous lesson. In fact, because you just copied the commands straight into its {body}, **`moveStraight()`** literally is the *same code* as in the previous lesson!

3. Replace the first occurrence of the behavior in the main program with a call to **`moveStraight()`**.

```
24    SensorValue[leftEncoder] = 0;
25    SensorValue[rightEncoder] = 0;
26    while(SensorValue[leftEncoder] < 240)
27    {
28       moveStraight();
29    }
30    SensorValue[leftEncoder] = 0;
31    while(SensorValue[leftEncoder] < 70)
```

3. Modify this code
Delete the highlighted code, and replace it with a single call to **`moveStraight()`**. Since the {body} of **`moveStraight()`** is exactly the same as the lines that are being replaced, and calling **`moveStraight()`** will essentially place the code from its {body} here when run, the end result should be the same. However, it will take up only one line in the main function instead of a dozen.

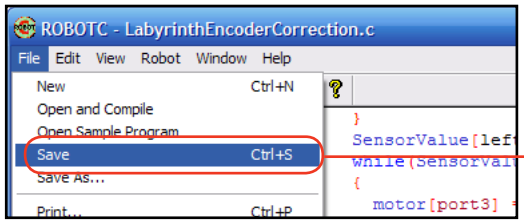
Autonomy/Encoders Behaviors and Functions *(cont.)*

4. Replace the other occurrences of the moving-straight behavior.

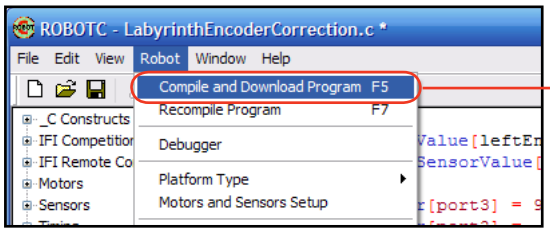
```
36      SensorValue[leftEncoder] = 0;
37      SensorValue[rightEncoder] = 0;
38      while(SensorValue[leftEncoder] < 270)
39      {
40        moveStraight();
41      }
42      SensorValue[leftEncoder] = 0;
43      while(SensorValue[leftEncoder] < 92)

48      SensorValue[leftEncoder] = 0;
49      SensorValue[rightEncoder] = 0;
50      while(SensorValue[leftEncoder] < 180)
51      {
52        moveStraight();
53      }
54      SensorValue[leftEncoder] = 0;
55      while(SensorValue[leftEncoder] < 93)

60      SensorValue[leftEncoder] = 0;
61      SensorValue[rightEncoder] = 0;
62      while(SensorValue[leftEncoder] < 180)
63      {
64        moveStraight();
65      }
66    }
```

4. Modify this code
Replace the remaining three occurrences of the moving-straight behavior with the **`moveStraight()`** function call, just like you did the first time.

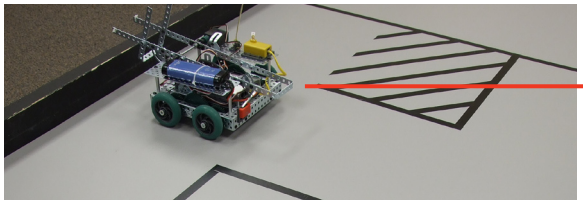
5. Save, download, and run your program.



5a. Save your program
Select *File* > *Save*.



5b. Compile and download
Make sure that your robot is turned on and that the robot is plugged into your computer with the USB cable. From the menu, select *Robot* > *Compile and Download Program*.



5c. Run your program

Autonomy/Encoders Behaviors and Functions *(cont.)*

Checkpoint

Your robot runs exactly the same as it did before. But, compare the code between the two versions. The version that uses functions is much shorter and easier to read.

```
Auto const tSensors rightEncoder = (tSensors) in2;
Auto const tSensors leftEncoder  = (tSensors) in3;
```

```
1  void moveStraight()
2  {
3     if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
4     {
5           motor[port3] = 50;
6           motor[port2] = 63;
7     }
8     if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
9     {
10          motor[port3] = 63;
11          motor[port2] = 50;
12    }
13    if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
14    {
15          motor[port3] = 63;
16          motor[port2] = 63;
17    }
18 }
19
20 task main()
21 {
22    wait1Msec(2000);
23    bMotorReflected[port2] = 1;
24    SensorValue[leftEncoder] = 0;
25    SensorValue[rightEncoder] = 0;
26    while(SensorValue[leftEncoder] < 240)
27    {
28          moveStraight();
29    }
30    SensorValue[leftEncoder] = 0;
31    while(SensorValue[leftEncoder] < 70)
32    {
33          motor[port3] = -96;
34          motor[port2] = 96;
35    }
36    SensorValue[leftEncoder] = 0;
37    SensorValue[rightEncoder] = 0;
38    while(SensorValue[leftEncoder] < 270)
39    {
40          moveStraight();
41    }
42    SensorValue[leftEncoder] = 0;
43    while(SensorValue[leftEncoder] < 92)
44    {
45          motor[port3] = 96;
46          motor[port2] = -96;
47    }
48    SensorValue[leftEncoder] = 0;
49    SensorValue[rightEncoder] = 0;
50    while(SensorValue[leftEncoder] < 180)
51    {
52          moveStraight();
53    }
54    SensorValue[leftEncoder] = 0;
55    while(SensorValue[leftEncoder] < 93)
56    {
57          motor[port3] = 96;
58          motor[port2] = -96;
59    }
60    SensorValue[leftEncoder] = 0;
61    SensorValue[rightEncoder] = 0;
62    while(SensorValue[leftEncoder] < 140)
63    {
64          moveStraight();
65    }
66 }
```

```
Auto const tSensors rightEncoder = (tSensors) in2;
Auto const tSensors leftEncoder  = (tSensors) in3;
```

```
1  task main()
2  {
3     wait1Msec(2000);
4     bMotorReflected[port2] = 1;
5     SensorValue[rightEncoder] = 0;
6     SensorValue[leftEncoder] = 0;
7     while(SensorValue[leftEncoder] < 240)
8     {
9           if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
10          {
11                motor[port3] = 50;
12                motor[port2] = 63;
13          }
14          if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
15          {
16                motor[port3] = 63;
17                motor[port2] = 50;
18          }
19          if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
20          {
21                motor[port3] = 63;
22                motor[port2] = 63;
23          }
24    }
25    SensorValue[leftEncoder] = 0;
26    while(SensorValue[leftEncoder] < 70)
27    {
28          motor[port3] = -96;
29          motor[port2] = 96;
30    }
31    SensorValue[rightEncoder] = 0;
32    SensorValue[leftEncoder] = 0;
33    while(SensorValue[leftEncoder] < 270)
34    {
35          if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
36          {
37                motor[port3] = 50;
38                motor[port2] = 63;
39          }
40          if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
41          {
42                motor[port3] = 63;
43                motor[port2] = 50;
44          }
45          if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
46          {
47                motor[port3] = 63;
48                motor[port2] = 63;
49          }
50    }
51    SensorValue[leftEncoder] = 0;
52    while(SensorValue[leftEncoder] < 92)
53    {
54          motor[port3] = -96;
55          motor[port2] = 96;
56    }
57    SensorValue[rightEncoder] = 0;
58    SensorValue[leftEncoder] = 0;
59    while(SensorValue[leftEncoder] < 180)
60    {
61          if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
62          {
63                motor[port3] = 50;
64                motor[port2] = 63;
65          }
66          if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
67          {
68                motor[port3] = 63;
69                motor[port2] = 50;
70          }
71          if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
72          {
73                motor[port3] = 63;
74                motor[port2] = 63;
75          }
76    }
77    SensorValue[leftEncoder] = 0;
78    while(SensorValue[leftEncoder] < 93)
79    {
80          motor[port3] = -96;
81          motor[port2] = 96;
82    }
83    SensorValue[rightEncoder] = 0;
84    SensorValue[leftEncoder] = 0;
85    while(SensorValue[leftEncoder] < 140)
86    {
87          if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
88          {
89                motor[port3] = 50;
90                motor[port2] = 63;
91          }
92          if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
93          {
94                motor[port3] = 63;
95                motor[port2] = 50;
96          }
97          if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
98          {
99                motor[port3] = 63;
100               motor[port2] = 63;
101         }
102   }
103 }
```

Autonomy/Encoders Behaviors and Functions *(cont.)*

As you can see, the use of **functions** to encapsulate the behavior has saved considerable space in the program. Another huge benefit is to the **readability** of the program. The **`task main`** code is much easier to read, having fewer nested layers. A simple descriptive name, **`moveStraight()`**, takes the place of a complex-looking piece of code. You can already begin to see additional places where functions could lead to convenient simplifications (such as **`leftTurn()`** or **`forward240degrees()`**).

However, first things first. We must return to the task that led us this way in the first place: speeding up the forward movement. Without the need for an all-inclusive find and replace, we can now change the behavior itself simply by modifying the code in the function!

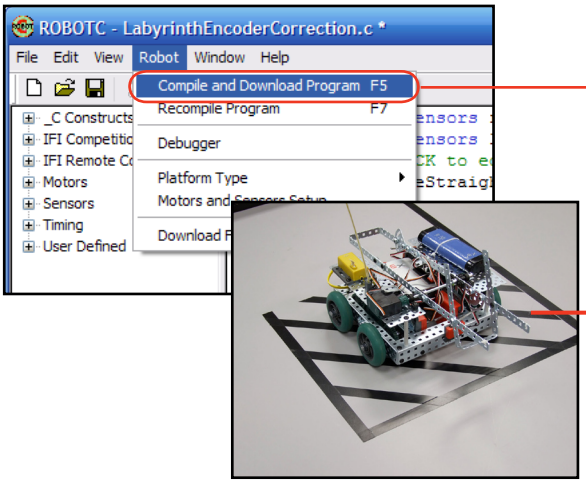
---

6. Change the motor powers in the function to 127 and 100, in order to speed up the robot as planned.

```
1   void moveStraight()
2   {
3     if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
4     {
5         motor[port3] = 100;
6         motor[port2] = 127;
7     }
8     if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
9     {
10        motor[port3] = 127;
11        motor[port2] = 100;
12    }
13    if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
14    {
15        motor[port3] = 127;
16        motor[port2] = 127;
17    }
18  }
```

6. Modify this code
Change the "higher" power levels of 63 to a full 127, and the "lower" power levels of 50 to 100.

7. Download and run your program.



7a. Compile and download
Make sure your robot is turned on and that the robot is plugged into your computer with the USB cable. From the menu, select ***Robot*** > ***Compile and Download Program***.

7b. Run your program

Autonomy/Encoders Behaviors and Functions *(cont.)*

End of Section

The behavior of the robot is updated on all four segments of straight movement, even though the behavior was modified in only one place: the function **declaration**. Since all function **calls** to `moveStraight()` refer back to the **declaration**, any changes in the declaration {body} take effect everywhere the function is used.

Alternately, you can think of the program as jumping to the function in each case, running the code, and jumping back afterwards. These two points of view are completely **equivalent**, so you can use whichever one you prefer when you think about functions.

The use of **functions** to encapsulate **behaviors** in your program has many benefits:

- It saves space in your program
- It lets you reuse code
- Your code reflects the nature of behaviors as separate actions
- Changes to your code only need to be made in one place

In the last lesson of this chapter, we'll finally bring the Encoder into the competition program.

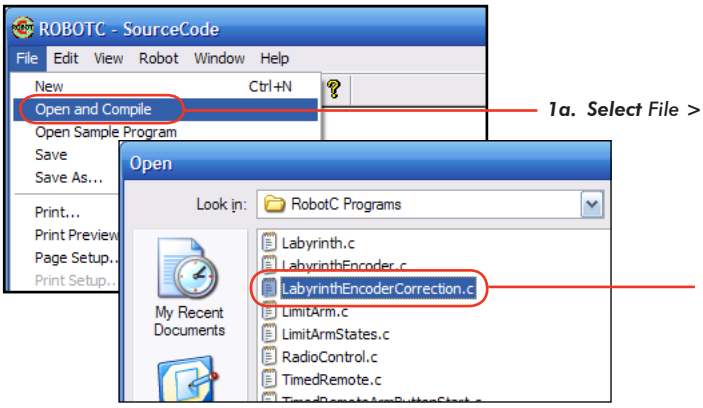
Autonomy/Encoders Straight Button

In this lesson, you will learn how to move a function into another program, and how a sensor-assisted autonomous behavior can work inside an otherwise operator-controlled program.

At the beginning of this chapter, we said that the self-correcting straight movement behavior would have applications in both Fully Autonomous and Operator Assist modes. Here, we will finalize the Operator Assist version, giving the human operator access to the self-correcting straight movement at the touch of a button. The **autonomous behavior** of the robot can now be an integral part of the overall strategy. The robot is able to perform the **straightening** task better than the human. The human can direct the robot when to start and stop it.

We will start by copying the **`moveStraight()`** behavior from the previous program into the competition program. Then, we will **bind** it to be activated when one of the unused buttons on the back of the Transmitter is pressed. The bottom button on Channel 5 (the lower left-hand Transmitter button) is not being used, so we can use that button to activate the moving straight behavior.

1. Open the "LabyrinthEncoderCorrection" program.



***1a. Select** File > Open and Compile*

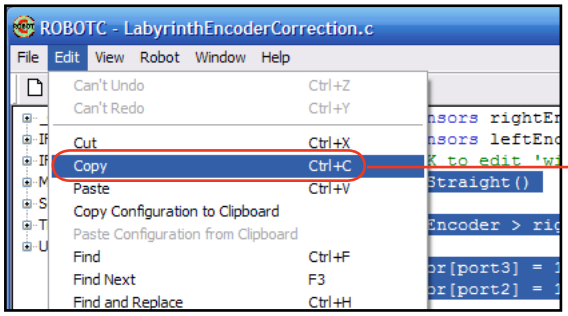
1b. Open the "LabyrinthEncoderCorrection" program
Find "LabyrinthEncoderCorrection" in the directory where you normally save your programs, and double-click to open it (or select it and click ***Open***).

2. Copy the **`moveStraight()`** function.

```
1   void moveStraight()
2   {
3       if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
4       {
5         motor[port3] = 100;
6         motor[port2] = 127;
7       }
8       if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
9       {
10        motor[port3] = 127;
11        motor[port2] = 100;
12      }
13      if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
14      {
15        motor[port3] = 127;
16        motor[port2] = 127;
17      }
18    }
19  }
```

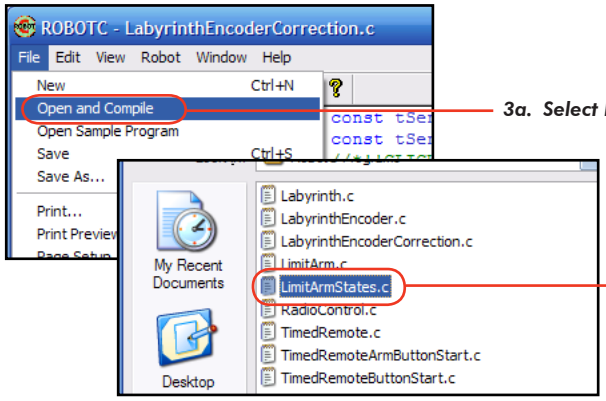
2a. Highlight this code
Find the **`void moveStraight()`** function in your program and highlight it.

Autonomy/Encoders Straight Button *(cont.)*



2b. Copy the highlighted code
Select *Edit* > *Copy* to copy the highlighted code onto the clipboard.

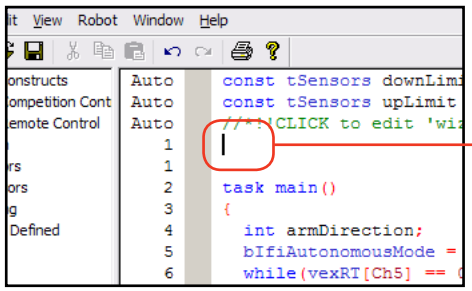
3. Open the "LimitArmStates" program from the Touch Sensors section.



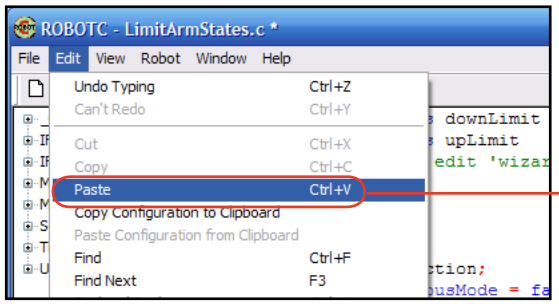
***3a. Select** File > Open and Compile*

3b. Open the "LimitArmStates" program
Find "LimitArmStates" in the directory where you normally save your programs, and double-click to open it (or select it and click *Open*).

4. Paste the moveStraight() function.



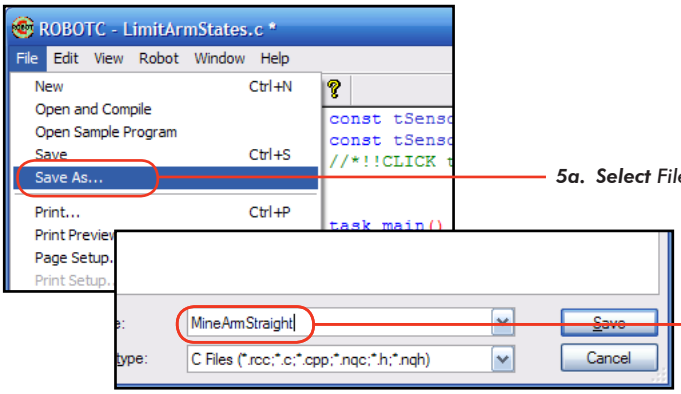
4a. Place the cursor here
Place your cursor on the line between `task main` and the lines of code that begin with `Auto`.



4b. Paste the copied code into the program
The `moveStraight()` function is now declared and available for use in this program.

Autonomy/Encoders Straight Button *(cont.)*

5. Save the edited program as "MineArmStraight".



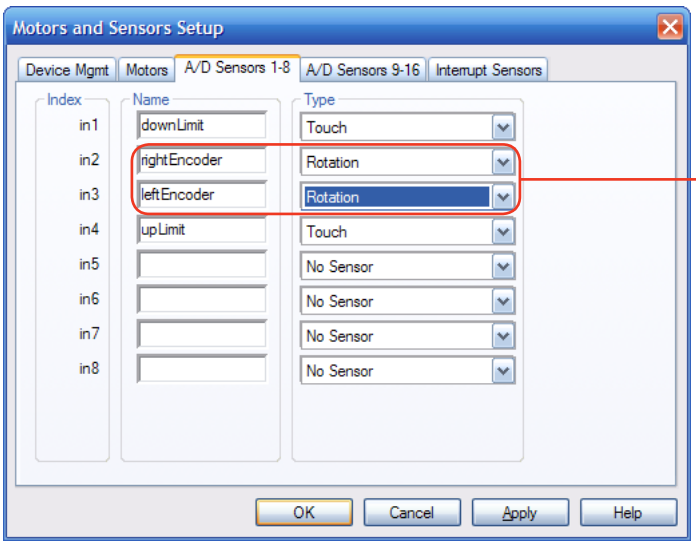
***5a. Select** File > Save As*

5b. Save As "MineArmStraight"
Save your program in the usual folder, with the name "MineArmStraight".

6. In the section where other buttons are handled during driving, add another **`if`**-statement to check whether the bottom button on Channel 5 is being pressed.

```
1            armDirection = 1;
2          }
3          if(vexRT[Ch6] == -127)
4          {
5            armDirection = -1;
6          }
7          if(vexRT[Ch5] == -127)
8          {
9            moveStraight();
10         }
11       }
12
13       motor[port3] = 0;
14       motor[port2] = 0;
```

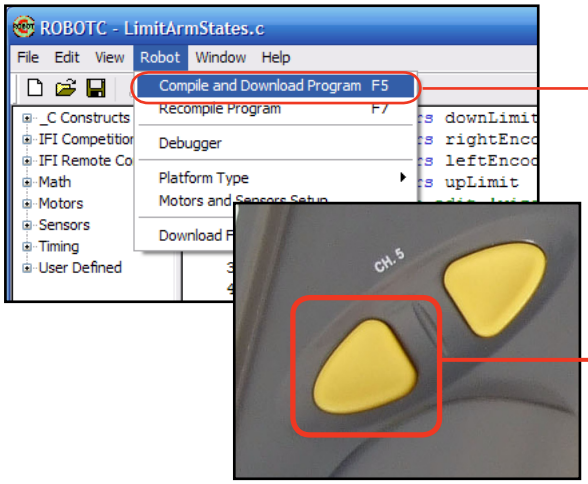
6a. Add this code
Check to see whether the bottom left-hand button is pressed. If it is, run the **`moveStraight()`** behavior.



6b. Set the Encoders
This is a different program so we don't have the encoders set up. Choose *Robot* > *Motors and Sensors Setup* and click on the *A/D Sensors 1-8* tab. Then, set the *leftEncoder* and *rightEncoder* to be of type *Rotation*.

Autonomy/Encoders Straight Button *(cont.)*

7. Download and run your program.



7a. Compile and download
Make sure that your robot is turned on and that the robot is plugged into your computer with the USB cable. From the menu, select *Robot* > *Compile and Download Program*.

7b. Run and test
Run your program, and press the lower left-hand button on the Transmitter.

Does the robot execute the **`moveStraight`** behavior correctly?

Checkpoint

The behavior activates, but only haltingly. The motor commands in the **function** are interfering with the **radio control** motor commands (or vice versa)! Radio control should only happen if the autonomous behavior is not pressed.

8. Move the Radio Control motor commands so that they are only run if the button is not pressed. This way the operator implicitly disables the sticks while the robot is moving autonomously.

```
32          armDirection = 0;
33          ClearTimer(T1);
34          while(time10[T1] < 12000)
35          {
36            motor[port3] = vexRT[Ch3];
37            motor[port2] = vexRT[Ch2];
38
39            if(vexRT[Ch6] == 127)
40            {
```

8a. Delete this code
This behavior will be moved down to line 80 below. Highlight these lines of code and choose *Edit* > *Delete* to cut them from here.

```
74            if(vexRT[Ch5] == -127)
75            {
76               moveStraight();
77            }
78            else
79            {
80               motor[port3] = vexRT[Ch3];
81               motor[port2] = vexRT[Ch2];
82            }
83          }
```

8b. Add this code
Add an else{} block to the moving straight **`if`**-statement. Paste or retype the radio control commands from above inside the {body} of the else block.

Autonomy/Encoders Straight Button *(cont.)*

Checkpoint

The robot will now move straight when the button is pushed. However, this will only work if the encoder values are zero (0) or close to zero to begin with. If you have **driven** the robot for a bit and made some turns, they may actually be hundreds of counts apart. The robot, however, will perceive this as unwanted drift, and will try to correct the situation. You must reset the encoders to avoid this.

You cannot reset the encoders every time, however, or the robot will keep losing track of the drift that it *is* supposed to correct for. It would make more sense if the encoders were reset each time the autonomous moving straight behavior is **engaged**, so that the behavior always starts with the encoder values set to zero.

Remember from the **Variables and States** lesson in the **Operator Assist** chapter that Variables allow you to store different kinds of information. In this case, we will use a variable to store information about whether the robot needs to reset its encoders. We will create and use the boolean (true/false) variable **`isMovingStraight`** to store one of two values: **`true`** if the robot is currently engaged in a moving-straight behavior, and **`false`** if it is not. The encoder resets when the robot *starts* moving straight. That is, the encoder resets when the value of the variable changes from **`false`** to **`true`**.

---

9. Create the boolean variable **`isMovingStraight`**. It will be used to keep track of when you should reset the encoders. (That is, only when you are trying to use the function **`moveStraight()`**.)

```
21  task main()
22  {
23    int armDirection;
24    bool isMovingStraight;
25    bIfiAutonomousMode = false;
26    while(vexRT[Ch5] == 0)
27    {
28
29    }
30
31    bMotorReflected[port2] = 1;
32
33    armDirection = 0;
34    isMovingStraight = false;
35    ClearTimer(T1);
```

9a. Add this code
Add a boolean variable named **`isMovingStraight`** underneath your other variable.

9b. Add this code
Set the value of **`isMovingStraight`** to **`false`** before the **`while`** loop begins.

```
75        if(vexRT[Ch5] == -127)
76        {
77          if(isMovingStraight == false)
78          {
79                  SensorValue[leftEncoder] = 0;
80                  SensorValue[rightEncoder] = 0;
81                  isMovingStraight = true;
82          }
83          else
84          {
85                  moveStraight();
86          }
87        }
88        else
89        {
90          isMovingStraight = false;
91          motor[port3] = vexRT[Ch3];
92          motor[port2] = vexRT[Ch2];
93        }
```

9c. Modify this code
Add an **`if-else`** statement to the section of your code where you check if the "move Straight" button is pushed. In the **`if`** block, reset the encoders.

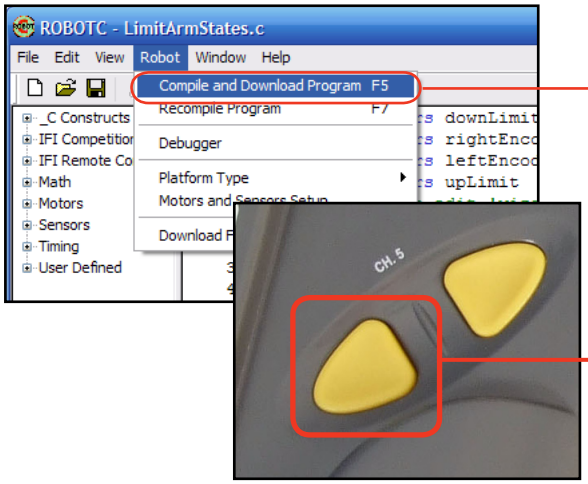
Also, you'll want to set the **`isMovingStraight`** variable to **`true`** to show that you want to move straight.

Finally, place the **`moveStraight()`** function inside the **`else`** block.

9d. Add this code
Set **`isMovingStraight`** to **`false`** inside the outer **`else`** block.

Autonomy/Encoders Straight Button *(cont.)*

10. Download and run your program.



10a. Compile and download
Make sure that your robot is turned on and that the robot is plugged into your computer with the USB cable. From the menu, select *Robot* > *Compile and Download Program*.

10b. Run and test
Run your program, and press the lower left-hand button on the Transmitter.

What did you observe about the **`moveStraight`** behavior this time?

Checkpoint

The robot now behaves as desired, though the speed of movement from the Labyrinth is not well suited for this new task.

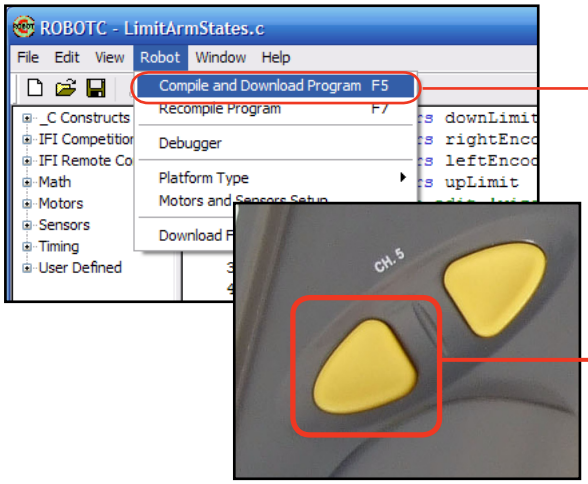
11. Adjust the speed of the autonomous behavior to be more appropriate for the delicate task of mine removal.

```
3       if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
4       {
5          motor[port3] = 50;
6          motor[port2] = 63;
7       }
8       if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
9       {
10         motor[port3] = 63;
11         motor[port2] = 50;
12      }
13      if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
14      {
15         motor[port3] = 63;
16         motor[port2] = 63;
17      }
```

11a. Modify this code
Slow down the movement of the robot during its straight behavior. Change the value of the motor power from 127 to 63 and from 100 to 50.

Autonomy/Encoders Straight Button *(cont.)*

12. Download and run your program.



12a. Compile and download
Make sure that your robot is turned on and that the robot is plugged into your computer with the USB cable. From the menu, select *Robot* > *Compile and Download Program*.

12b. Run and test
Run your program, and press the lower left-hand button on the Transmitter.

Did the robot execute the `moveStraight` behavior at a more appropriate speed this time?

End of Section

Your robot is now equipped with a highly capable combination of **manual and autonomous control**. The human operator has access to direct motor control, as well as sensor-assisted arm and driving controls. Sensors have given the robot the ability to compete for nearly triple points during the Autonomous Period, and augmented the capabilities of your team during the Remote Control Period.

There are still more sensors available for the Vex robot, but this is an excellent place to pause and work on refining your strategy with the current set. You have access to variables, sensors, autonomous, and operator-assist commands. Step back and see if there are any other behaviors you can automate or improve. You have all the basics at your disposal now.