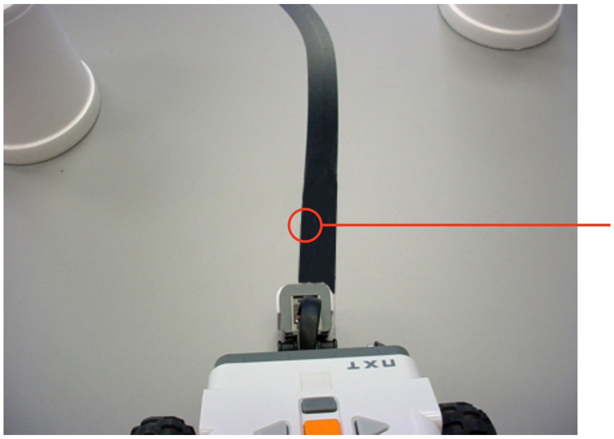
Line Tracking **Basic Lesson**

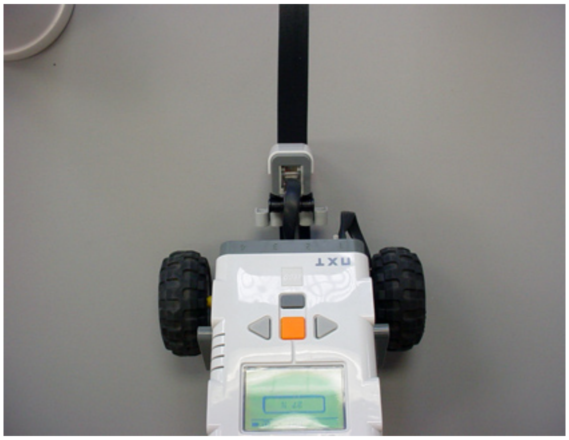
Now that you're familiar with a few of the key NXT sensors, let's do something a little more interesting with them. This lesson will show you how to use the Light Sensor to track a line.

The trick to getting the robot to move along the line is to always aim toward the *edge* of the line. For this example, we'll use the left edge.



Track the left side
The Light Sensor will be positioned and programmed to track the left side of the black line.

Put yourself in the robot's position. If the only dark surface is the line, then seeing dark means you are on top of it, and the edge would be to your left. So you move toward it by going forward and left by performing a Swing Turn.



Light Sensor sees dark
The robot is over the dark surface. The left edge of the line must be to the robot's left.

Swing turn left
Therefore, turn left toward the edge of the line.

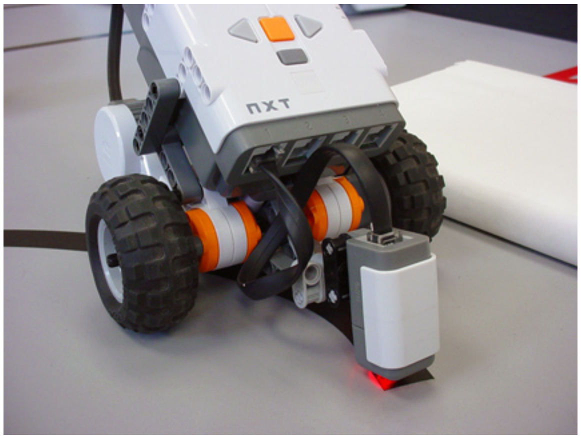
Line Tracking **Basic** (cont.)

The only time we should see Light is when we've driven off the line to the left. If we need to get to the left edge, it's always a right turn to get back to line. Make the forward-right turn as long as you're seeing Light, and eventually, you're back to seeing Dark!

Light Sensor sees light
The robot is now over the light surface. The left edge of the line must be to the robot's right.

Swing turn right
Therefore, turn right toward the edge of the line.

Put those two behaviors in a loop, and you will see that the robot will bounce back and forth between the light and dark areas. The robot will eventually bobble its way down the line.

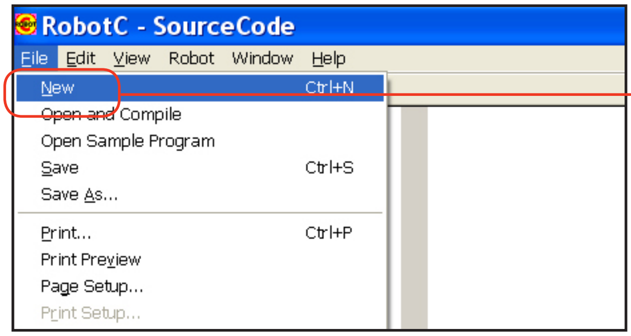


Track the line:
The robot will perform the line track behavior to the end of the line

Line Tracking **Basic** (cont.)

In this lesson you will learn how to use the light sensor to follow a line, using behaviors similar to the Wait for Dark (and Wait for Light) behaviors you have already worked with.

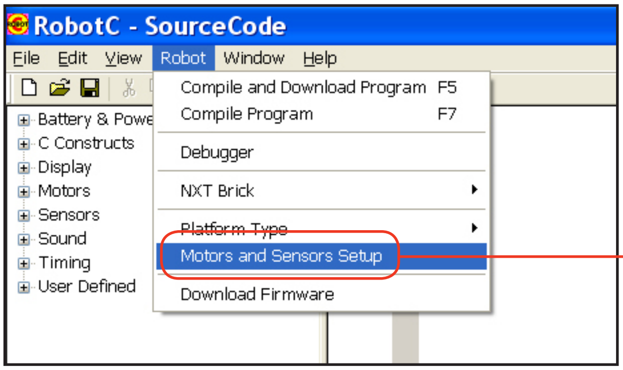
1. Start with a new, clean program.



1. Create new program
Select File > New to create a blank new program.

---

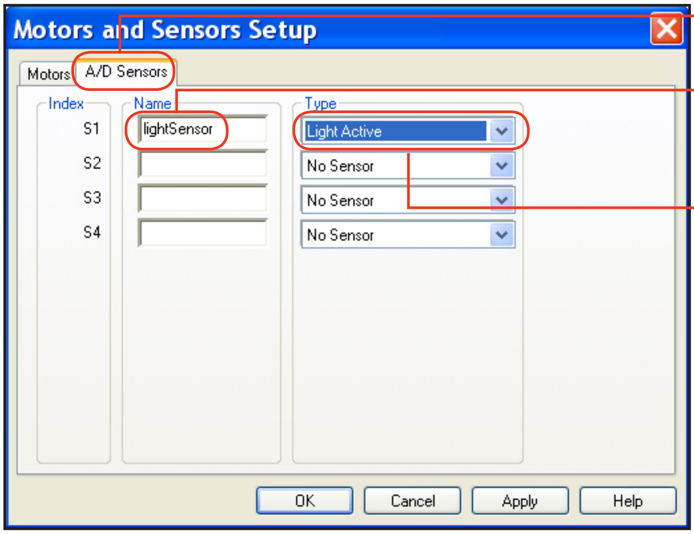
2. The first step is to configure the Light Sensor. Go to the Motors and Sensors Setup menu. Click "Robot" then choose the "Motors and Sensors Setup".



2. Open "Motors and Sensors Setup"
Select Robot > Motors and Sensors Setup to open the Motors and Sensors Setup menu.

Line Tracking **Basic** (cont.)

3. Configure an Active Light Sensor named "lightSensor" on Port1.



Motors and Sensors Setup

Index	Name	Type
S1	lightSensor	Light Active
S2		No Sensor
S3		No Sensor
S4		No Sensor

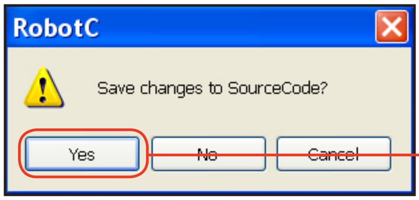
3a. Open A/D Sensors Tab
Click the A/D Sensors tab

3b. Name the sensor
Name the Light Sensor on port S1 "lightSensor".

3c. Set Sensor Type
Identify the Sensor Type as a "Light Active" sensor.

---

4. Press OK, and you will be prompted to save the changes you have just made. Press Yes to save.



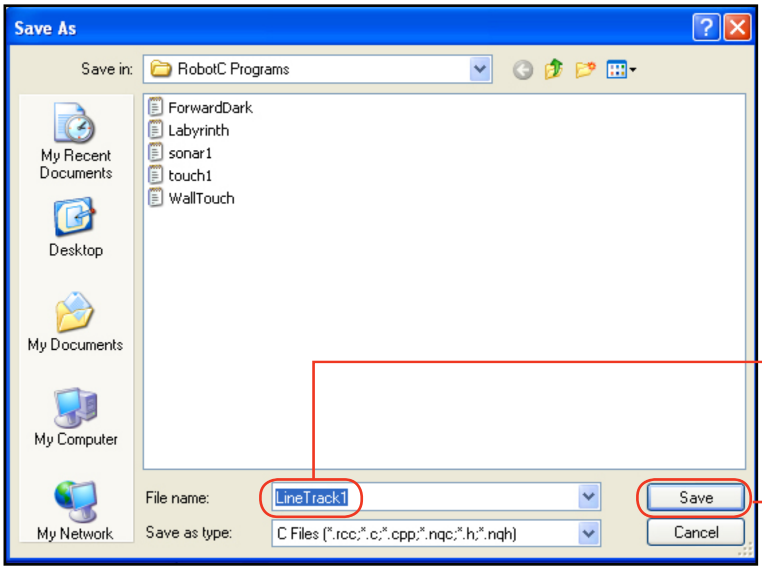
RobotC

Save changes to SourceCode?

4. Select "Yes"
Save your program when prompted.

---

5. Save this program as "LineTrack1".



Save As

Save in: RobotC Programs

ForwardDark
Labyrinth
sonar1
touch1
WallTouch

File name: LineTrack1
Save as type: C Files (*.rcc;*.c;*.cpp;*.nqc;*.h;*.nqh)

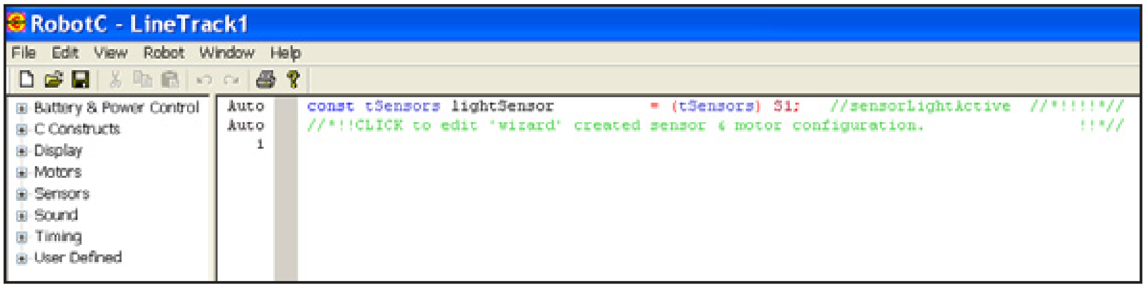
5a. Name the program
Give this program the name "LineTrack1".

5b. Save the program
Press Save to save the program with the new name.

Line Tracking **Basic** (cont.)

Checkpoint

Your program should look like the one below. The Light Sensor is configured, and we can now start with the rest of the code.



```
const tSensors lightSensor          = (tSensors) S1;   //sensorLightActive  //*!!!!*//
//*!!CLICK to edit 'wizard' created sensor & motor configuration.              !!*//
```

---

6. Let's start by putting the "easy" stuff in first: task main, parentheses, and curly braces.

```
task main()
{


}
```

6. Add this code
These lines form the main body of the program, as they do in every ROBOTC program.

---

7. Recall that in order to seek the left edge of the line, the robot must go forward-left for as long as it sees dark, until it reaches the light area. Similar to the Forward Until Dark behavior you wrote earlier, this uses a `while()` loop that runs "while" the `SensorValue` of the `lightSensor` is less than the threshold (which you must calculate as before).

```
task main()
{

    while(SensorValue(lightSensor) < 45)
    {

        motor[motorC] = 0;
        motor[motorB] = 80;

    }

}
```

7a. Add this code
This `while()` loop functions like the Forward Until Dark behavior you wrote earlier.

It will run the code inside the braces as long as the `SensorValue` of the `lightSensor` is less than the threshold value of `45`.

7b. Add this code
Instead of moving forward like Forward Until Dark, the robot should turn forward-left.

Left motor stationary, with right motor at 80% creates this motion.

Line Tracking **Basic** (cont.)

8. The robot has presumably driven off the line, and must now turn back toward it. The robot must turn forward-right as long is it continues to see the light table surface (i.e. until it sees the dark line again).

```
task main()
{

    while(SensorValue(lightSensor) < 45)
    {

        motor[motorC] = 0;
        motor[motorB] = 80;

    }

    while(SensorValue(lightSensor) >= 45)
    {

        motor[motorC] = 80;
        motor[motorB] = 0;

    }

}
```

8a. Add this code
This `while()` loop is very similar to the one above it, except that it will run the code inside it while the light sensor sees light, rather than dark.

8b. Add this code
This turns the robot forward-right by running the left motor at 80% while holding the right motor stationary.

Checkpoint
The code currently handles only one "bounce" off and back onto the line.
However, to track a line, the robot must repeat these two operations over and over again.
This will be accomplished using another `while()` loop, set to repeat forever. "Forever" will be achieved in a somewhat creative way...

Discussing Concepts Using Pseudocode

Often when discussing programs and robot behaviors, it is useful for programmers to use language that is a mixture of English and code. This hybrid language is called "pseudocode" and allows programmers to discuss programming concepts in a natural way. Pseudocode is not a formal language, and therefore there is no one "right" way to do it, but it often involves simplifications to aid in discussion.

(continued on next page...)

Line Tracking **Basic** (cont.)

9. Create a `while()` loop *around* your existing code. Position the curly braces so that both of the other while loop behaviors are inside this new while loop. For this new while loop's condition, enter "1==1", or "one is equal to one".

```
task main()
{

   while(1==1)
   {

      while(SensorValue(lightSensor) < 45)
      {

           motor[motorC] = 0;
           motor[motorB] = 80;

      }

      while(SensorValue(lightSensor) >= 45)
      {

           motor[motorC] = 80;
           motor[motorB] = 0;

      }

   }

}
```

9. Add this code
The new `while()` loop goes *around* most of the existing code, so that it will repeat those behaviors over and over.

The loop will run as long as "1==1", or "one is equal to one". This is always true, hence the loop will run forever.

Discussing Concepts Using Pseudocode (cont.)

The program on this page might look like this in pseudocode:

```
repeat forever
{
  while(the light sensor sees dark)
    {
      turn forward-left;
    }
  while(the light sensor sees light)
    {
     turn forward-right;
    }
}
```

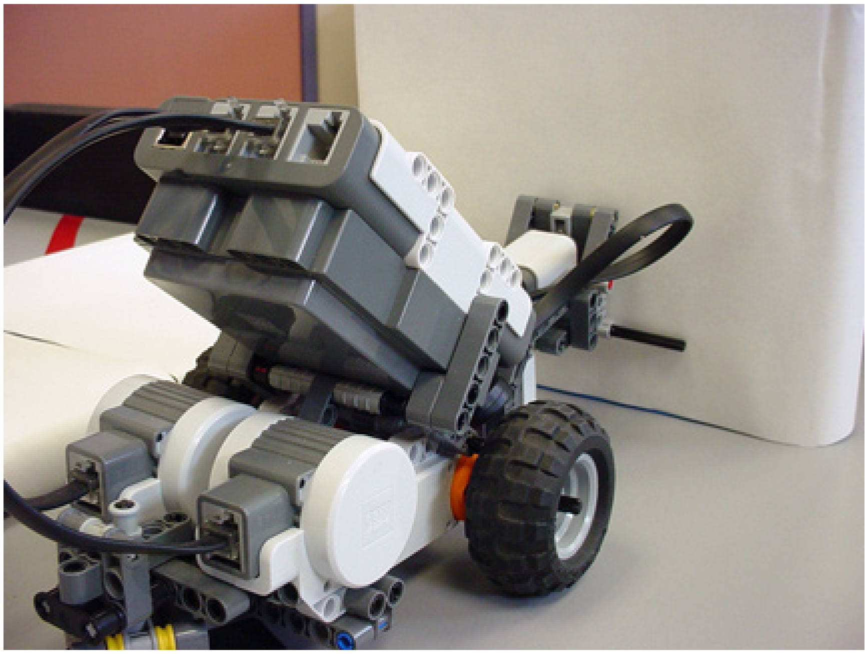
Line Tracking **Basic** (cont.)

End of Section

Now that your program is complete, check to see if it works. Save your program, and then download it to the robot and run. If you see that your robot is moving off the line in one direction, it means that your threshold is set wrong. The robot thinks it's seeing dark even on light, or light even on dark, and it's just waiting to see the other, which probably won't happen if the values are wrong. If, however, you see your robot bouncing back and forth, moving down the line, then your robot is working correctly, and it's time to move on to the next lesson.

Line Tracking Better Lesson

In the previous lesson we learned the basics of how to use the light sensor to follow a line. That version of the line tracker runs forever, and cannot be stopped except by manually stopping the program. To be more useful, the robot should be able to start and stop the line tracking behavior on cue. For example, the robot should be able to stop following a line when it reaches a wall at the end of its path.

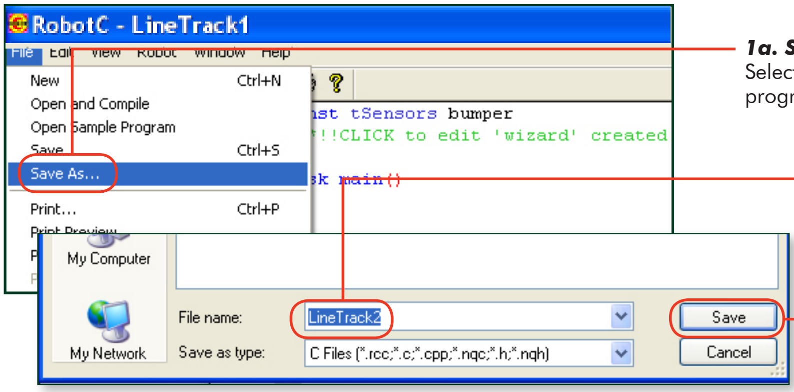


In principle, we should be able to do this pretty easily, all we need to do is change the "looping forever" part to "loop while the touch sensor is unpressed."

Line Tracking **Better** (cont.)

In this lesson, you will adapt your line tracking program to stop when a Touch Sensor is pressed, and then make it more robust by replacing risky nested loops with if-else statements.

1. Save your existing program from the previous lesson under a new name, "LineTrack2".



```
nst tSensors bumper
/!!CLICK to edit 'wizard' created
sk main()
```

File name: LineTrack2
Save as type: C Files (*.rcc;*.c;*.cpp;*.nqc;*.h;*.nqh)

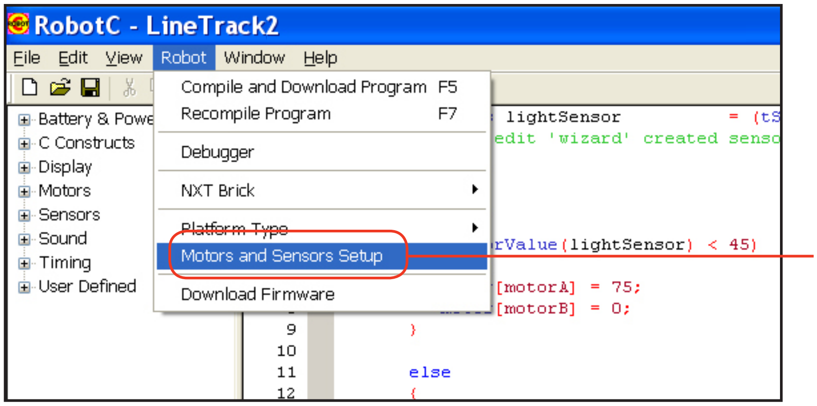
1a. Save program As...
Select File > Save As... to save your program under a new name.

1b. Name the program
Give this program the name "LineTrack2".

1c. Save the program
Press Save to save the program with the new name.

---

2. Open the Motors and Sensors Setup menu.

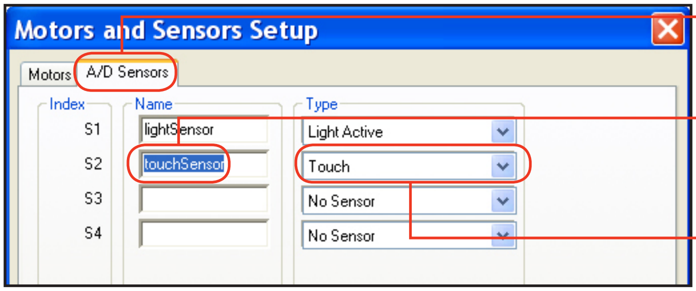


```
 lightSensor        = (tS
edit 'wizard' created senso
rValue(lightSensor) < 45)
[motorA] = 75;
[motorB] = 0;
}
else
{
```

2. Open "Motors and Sensors Setup"
Select Robot > Motors and Sensors Setup to open the Motors and Sensors Setup menu.

---

3. You will be adding a second sensor for this lesson. Configure a Touch Sensor called "touchSensor" on S2.



Index	Name	Type
S1	lightSensor	Light Active
S2	touchSensor	Touch
S3		No Sensor
S4		No Sensor

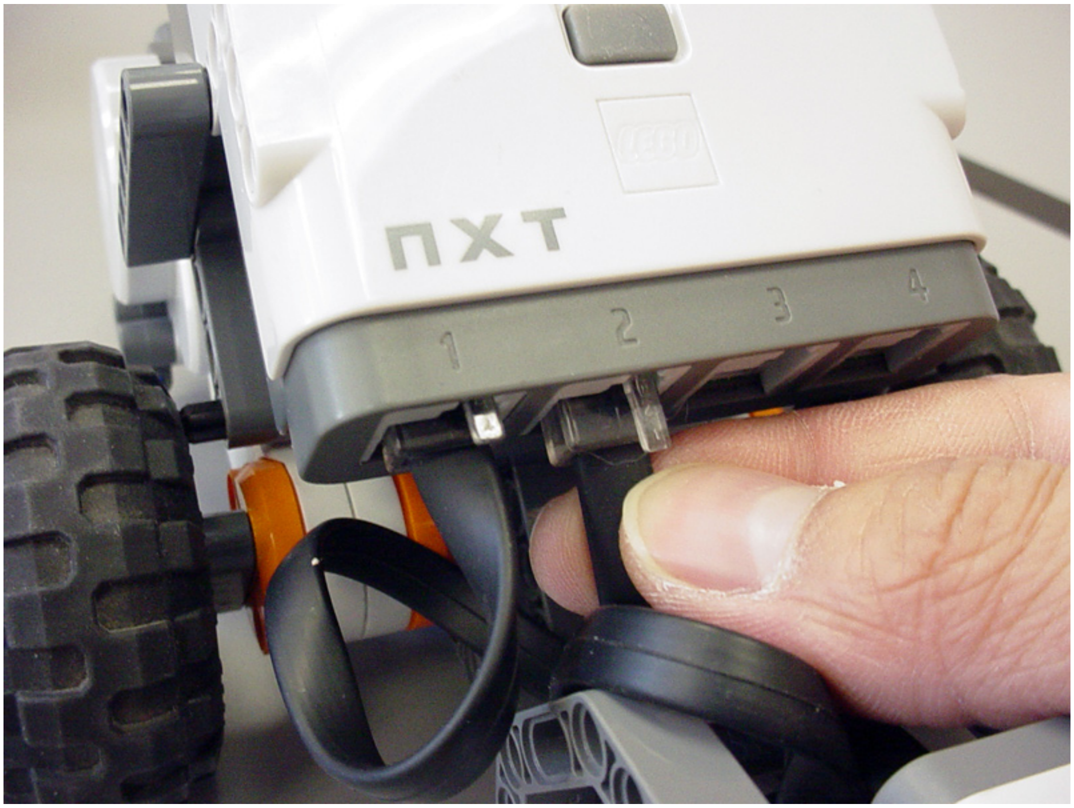
3a. Open A/D Sensors Tab
Click the A/D Sensors tab

3b. Name the sensor
Name the Touch Sensor on port S2 "touchSensor".

3c. Set Sensor Type
Identify the Sensor Type as a "Touch" sensor.

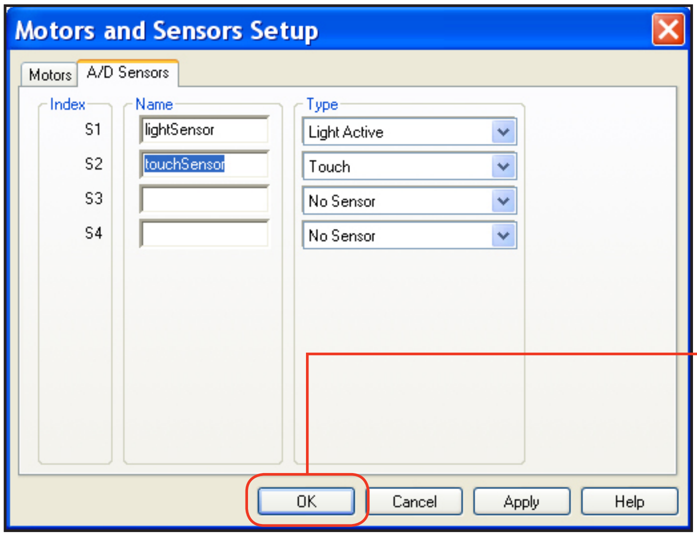
Line Tracking **Better** (cont.)

4. On your physical robot, plug the Touch Sensor into Port 2.



---

5. Press OK on the Motors and Sensors Setup menu.

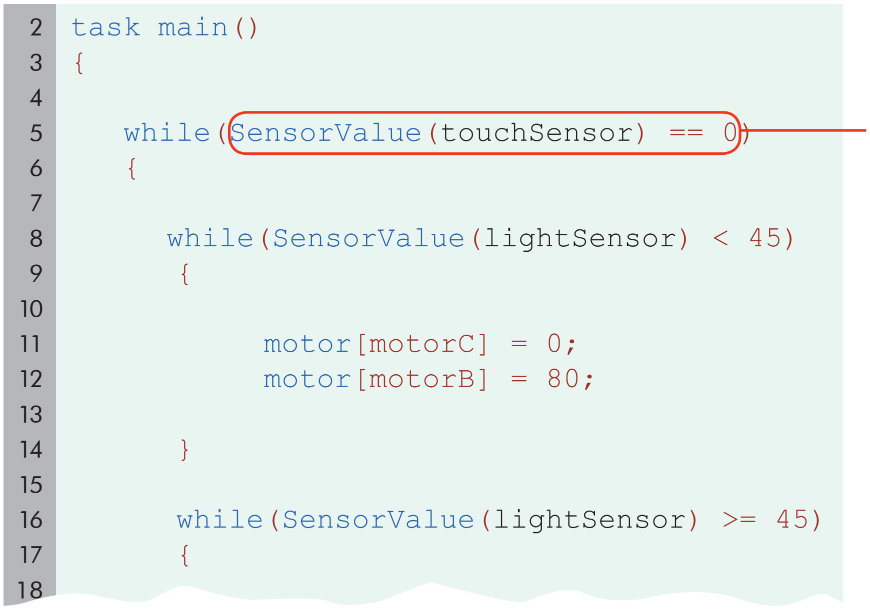


Index	Name	Type
S1	lightSensor	Light Active
S2	touchSensor	Touch
S3		No Sensor
S4		No Sensor

5. Press OK
Accept the changes to the sensor setup and close the window.

Line Tracking **Better** (cont.)

6. Replace the "forever" condition `1==1` with the condition "the touch sensor is unpressed", the same condition you used to "run until pressed" in the Wall Detection (Touch) lesson. This condition will be true when the `SensorValue` of `touchSensor` is equal to 0.



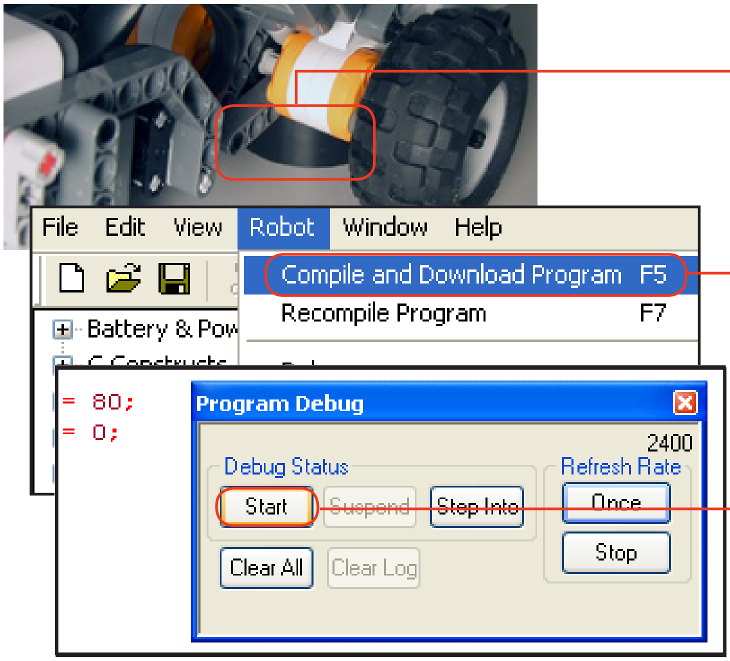
```
task main()
{

    while(SensorValue(touchSensor) == 0)
    {

        while(SensorValue(lightSensor) < 45)
        {

            motor[motorC] = 0;
            motor[motorB] = 80;

        }

         while(SensorValue(lightSensor) >= 45)
         {

```

6. Modify this code
Change the condition in parentheses to check whether the "touch sensor is unpressed" instead.

The condition will be true when the touch sensor's value is equal to 0.

---

7. Elevate ("block up") the robot so that you can test it without its wheels touching the ground. Note that the light sensor now hangs in the air. Download and run your program.



7a. Block up the robot
Place an object under the robot so that its wheels don't reach the table. The robot can now run without moving.

7b. Download the program
Click Robot > Compile and Download Program.

7c. Run the program
Click "Start" on the onscreen Program Debug window, or use the NXT's on-brick menus.

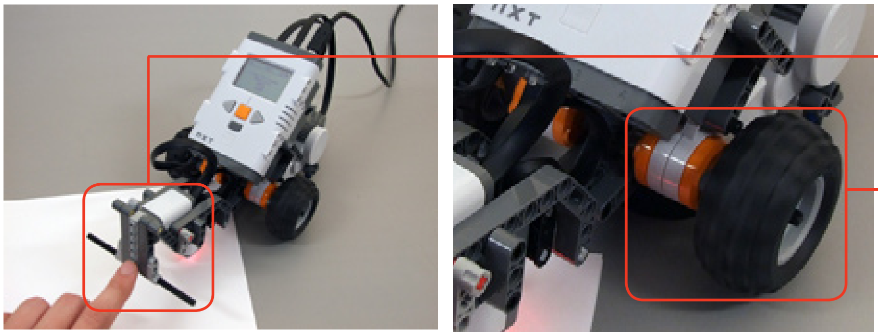
Line Tracking **Better** (cont.)

Checkpoint
Check that your Line Tracking behavior is correctly responding to light and dark by placing light- and dark-colored objects or paper under the light sensor.

Simulated dark line
Using a dark-colored object (or the naturally low value of the sensor when held in the air like this), confirm that the robot exhibits the correct motor behaviors when the sensor sees "dark".

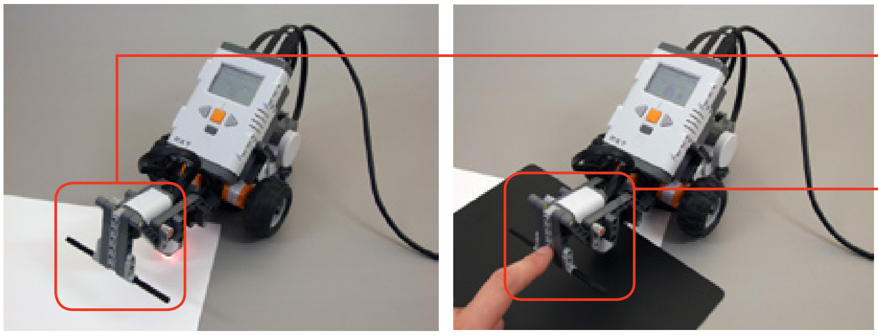
Simulated light surface
Place a sheet of white paper under the sensor to simulate the robot traveling off the line and onto the light table surface. Watch for the motors to change behaviors accordingly.

We modified the program so that the (condition) of the while() loop would only be true as long as the Touch Sensor was unpressed. When the sensor is pressed, the loop should end, and move on.



Touch the Sensor
Press in the bumper on the robot to trigger the Touch Sensor.

Observe motors
Do the motors stop like they should at the end of the program?



Light/Dark again
Release the Touch sensor, and see if the robot still responds to light and dark.

Light/Dark pressed
Hold down the Touch Sensor bumper, and try light/dark again. Does anything happen?

Line Tracking **Better** (cont.)

The robot responds strangely. When you pressed the touch sensor, it didn't respond. But when you held the touch sensor and waved the paper underneath it, the robot did stop. The touch sensor seems to be doing its job of stopping the loop… sometimes? Let's step through the code.

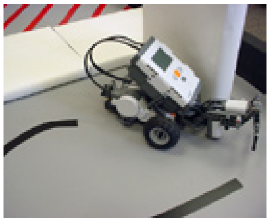
Key concept: While() loops do not continually monitor their (conditions). They only check when the program reaches the "while" line containing the condition.

```
task main()
{

   while(SensorValue(touchSensor) == 0)
   {

      while(SensorValue(lightSensor) < 45)
      {

         motor[motorC] = 0;
         motor[motorB] = 80;

      }

      while(SensorValue(lightSensor) >= 45)
      {
```

a. Touch Sensor check
The program checks the condition only at this point. It's true when we start, so the program goes "inside" the loop.

b. Inner loop
As long as the robot continues to see dark, it enters and remains in this loop.

What was the program was doing while the robot saw the dark object (or dark space below its sensor)? The program reached and went inside the while(dark) loop, (b) above, and remained inside as long as the Light Sensor continued seeing dark. Consider which lines check the Touch Sensor. While the program was inside the inner while() loop, was it ever able to reach those lines?



```
task main()
{

   while(SensorValue(touchSensor) == 0)
   {

      while(SensorValue(lightSensor) < 45)
      {

         motor[motorC] = 0;
         motor[motorB] = 80;

      }
```

Code must reach this point
The Touch Sensor is only checked when the program reaches this line.

Code is stuck here
Until the Light Sensor stops seeing dark, the program doesn't leave this loop.

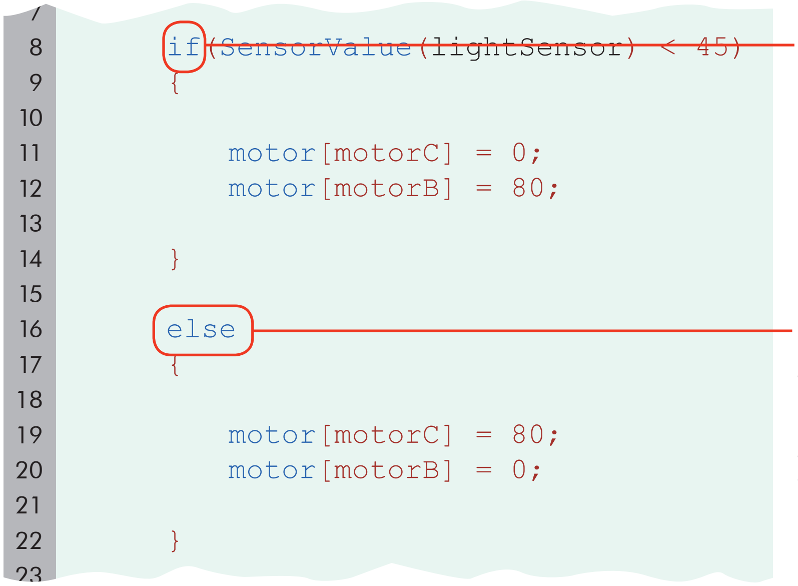
The current program contains flawed logic. Until the robot stops seeing dark, there's no way for the program to reach the line that checks the touch sensor! This "stuck in the inner loop" problem will always be a danger any time we place one loop inside another, a structure called a "nested loop". We were only able to get the robot to recognize touch by waving the light object in front of it to force it out of the while(dark) loop, and back around to check the Touch Sensor again.

Line Tracking **Better** (cont.)

The solution requires a little shift in thinking. The program as it is now involves running trough an "inner" while loop, where it has the potential to get stuck, oblivious to the outside world. We need to get rid of the nested loop. If, instead, we break down the robot's actions into a series of tiny, instantaneous decisions that will always pick the correct direction, we can avoid the need to go "inside" a loop that might not end in time. Enter the **`if-else`** statement.

---

7. Replace the inner `while()` loops with a simpler, lightweight decision-making structure called a conditional statement, or if-else statement.



```
        if(SensorValue(lightSensor) < 45)
        {

            motor[motorC] = 0;
            motor[motorB] = 80;

        }

        else
        {

            motor[motorC] = 80;
            motor[motorB] = 0;

        }
```

7a. Modify this code
Replace `while` with `if`.

If the light sensor value is less than 45, run the code between the curly braces, once only, then move on.

7b. Modify this code
Replace the `while()` line with the keyword `else`.

If the code in the `if` statement's brackets did not run, the code in the `else` statement's brackets will instead (once). This should only happen when the light sensor is seeing a value >= 45 (i.e .light).

In the same way that the `while` loop started with the word "while", the `if-else` starts with the word "if". It, like the while loop, is followed immediately by a condition in parentheses. In fact, it uses the same condition as the old program to check the light sensor. The difference is that the if-else statement will only run the commands in the brackets once, regardless of the light or touch sensor readings.

If the SensorValue of the lightSensor is less than the threshold, then the code directly after will execute, once. The else, followed by another set of curly braces, represents what the program should do if the condition is *not* true.

```
if(condition)
{
    true-commands;
}
else
{
    false-commands;
}
```

General form

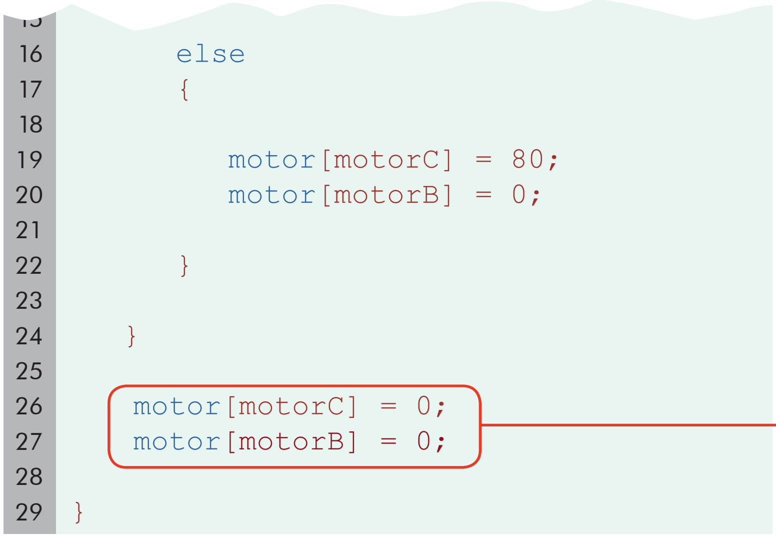
Conditional (if-else) loops always follow the pattern shown here.

If the (condition) is true, the true-commands will run.
If the (condition) is false, the false-commands will run instead.

Note, however, that whichever set of commands is chosen, they are only run *once*, and not looped!

Line Tracking **Better** (cont.)

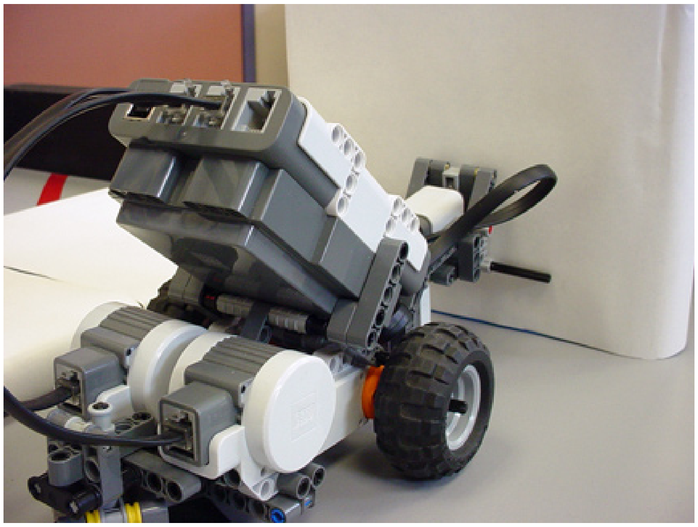
8. As a final touch, add a Stop motors behavior into the program, right before the final bracket. This ensures that you'll see an immediate reaction when the robot gets out of the loop.



```
      else
      {

         motor[motorC] = 80;
         motor[motorB] = 0;

      }

   }

   motor[motorC] = 0;
   motor[motorB] = 0;

}
```

8. Add this code
Stop both motors. Because these lines come outside the `while()` loop, they will run after the `while()` loop has completed.

End of Section
Save your program, download, and run.

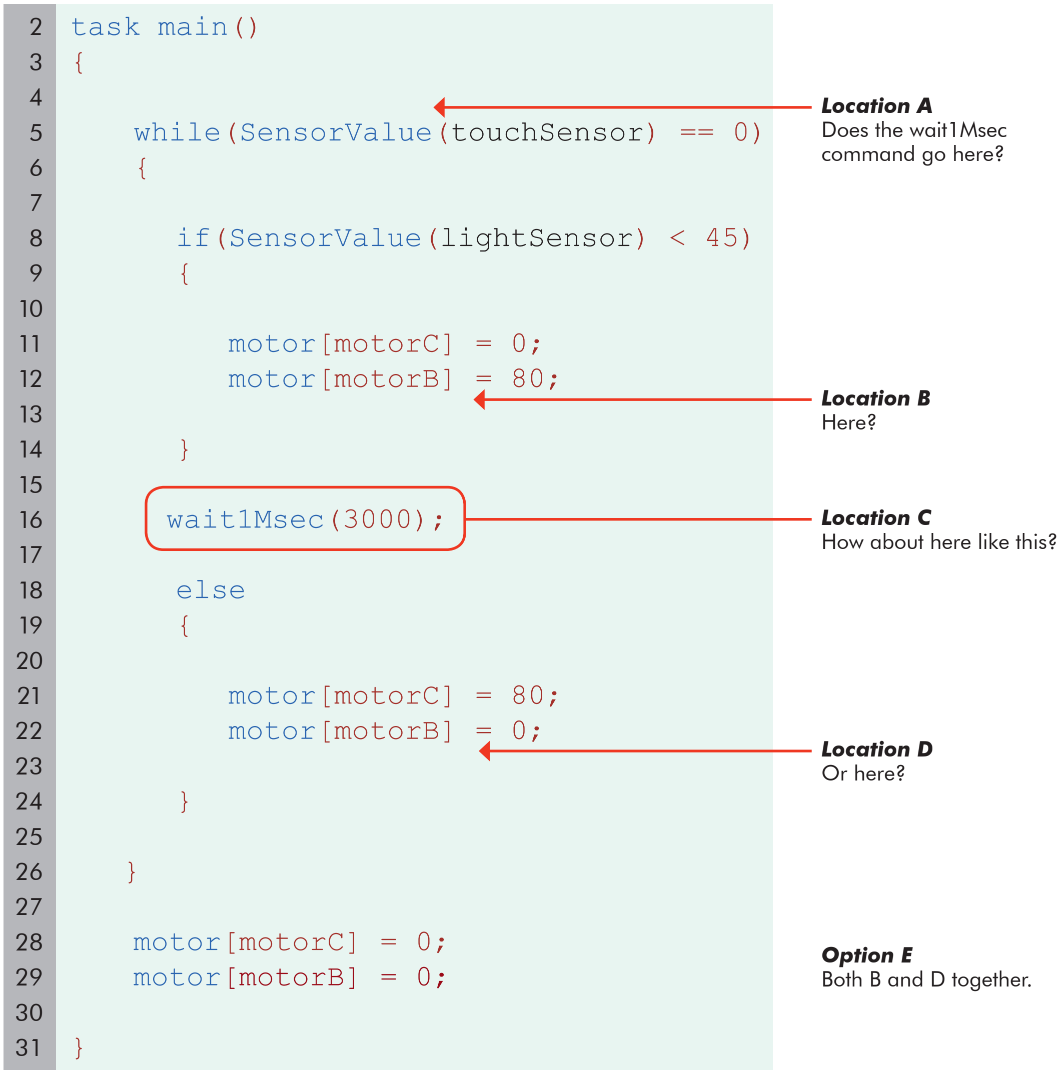


The robot no longer gets stuck in the "inner" while() loop, and successfully tracks the line until the touch sensor is triggered.

Line Tracking **Timer Lesson**

The behavior we programmed in the previous lesson is great for those situations where you want the robot to follow a line straight into a wall, and stop. However, let's see if there are any good ways to make the robot line track until something else happens.

To make the robot go straight for 3 seconds, we gave it motor commands, followed by a `wait1Msec(time)` command. How would this work with line tracking?



```
task main()
{

    while(SensorValue(touchSensor) == 0)
    {

      if(SensorValue(lightSensor) < 45)
      {

         motor[motorC] = 0;
         motor[motorB] = 80;

      }

      wait1Msec(3000);

      else
      {

         motor[motorC] = 80;
         motor[motorB] = 0;

      }

    }

    motor[motorC] = 0;
    motor[motorB] = 0;

}
```

Location A
Does the wait1Msec command go here?

Location B
Here?

Location C
How about here like this?

Location D
Or here?

Option E
Both B and D together.

Which one of the above locations is the right place to put the `wait1Msec` command?

The correct answer is: **none**. There is no right place to put a `wait1Msec` command to get the robot to line track for 3 seconds. Wait1Msec does not mean "continue the last behavior for this many milliseconds,"it means, "go to sleep for this many milliseconds."

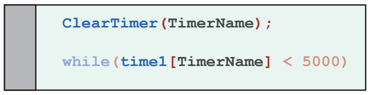
You've really told the robot to put its foot on the gas pedal, and go to sleep. That doesn't work when the robot needs to watch the road. Instead, we'll keep the robot awake and attentive, using a Timer (rather than just Time) to decide when to stop.

Line Tracking **Timer** (cont.)

Your robot is equipped with four Timers, T1 through T4, which you can think of as Time Sensors, or if you prefer, programmable stopwatches.

Using the Timers is pretty straightforward: you reset a timer with the **`ClearTimer()`** command, and it immediately starts counting time.

Then, when you want to find out how long it's been since then, you just use **`time1[`***`TimerName`***`]`**, and it will give you the value of the timer, in the same way that **`SensorValue(`***`SensorName`***`)`** gives you the value of a sensor.



```
ClearTimer(TimerName);

while(time1[TimerName] < 5000)
```

Timer Tips

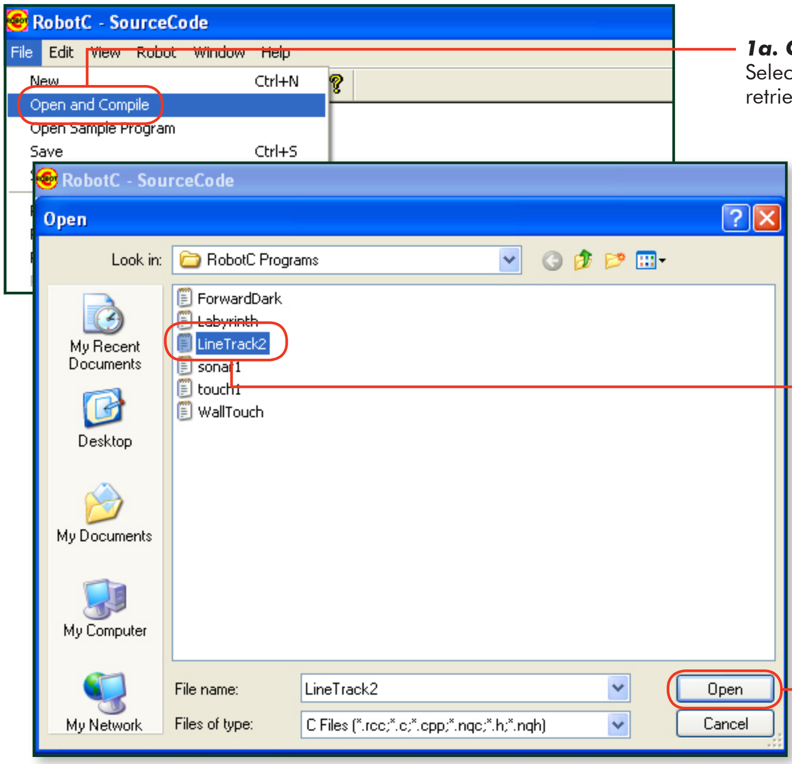
Timers should be reset when you are ready to start counting.

time1[*TimerName*] represents the timer value in milliseconds since the last reset. It is shown here being used to make a while loop run until 5 seconds have elapsed.

Line Tracking Timer (cont.)

In this lesson you will learn how to use Timers to make a line-tracking behavior run for a set amount of time.

1. Open the Touch Sensor Line Tracking program "LineTrack2".



File name: LineTrack2

Files of type: C Files (*.rcc;*.c;*.cpp;*.nqc;*.h;*.nqh)

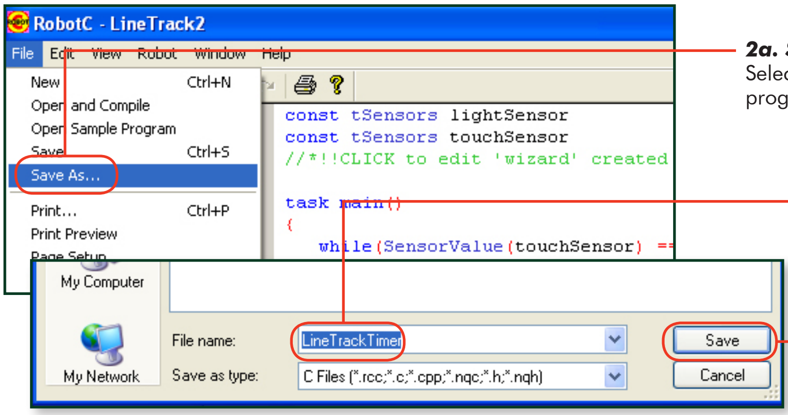
1a. Open Program
Select File > Open and Compile to retrieve your old program.

1b. Select the program
Select "LineTrack2".

1c. Open the program
Press Open to open the saved program.

---

2. Save this program under a new name, "LineTrackTimer". (Note the "r" at the end of "timer")



```
const tSensors lightSensor
const tSensors touchSensor
//*!!CLICK to edit 'wizard' created

task main()
{
   while(SensorValue(touchSensor) ==
```

File name: LineTrackTimer

Save as type: C Files (*.rcc;*.c;*.cpp;*.nqc;*.h;*.nqh)

2a. Save program As...
Select File > Save As... to save your program under a new name.

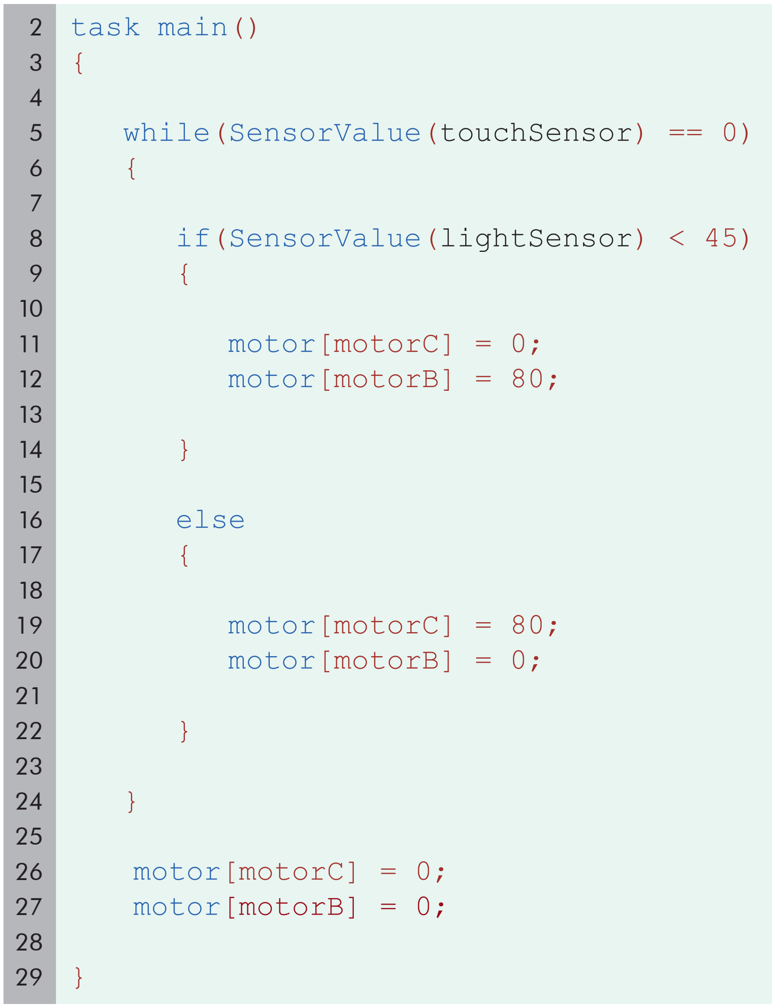
2b. Name the program
Give this program the name "LineTrackTimer".

2c. Save the program
Press Save to save the program with the new name.

Line Tracking **Timer** (cont.)

Checkpoint

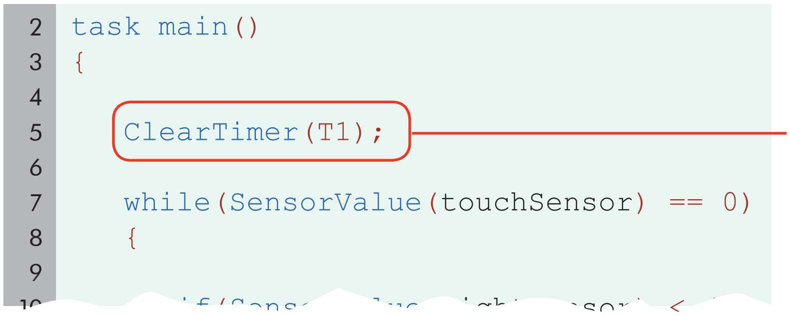
The program on your screen should again look like the one below.



```
task main()
{

   while(SensorValue(touchSensor) == 0)
   {

      if(SensorValue(lightSensor) < 45)
      {

         motor[motorC] = 0;
         motor[motorB] = 80;

      }

      else
      {

         motor[motorC] = 80;
         motor[motorB] = 0;

      }

   }

   motor[motorC] = 0;
   motor[motorB] = 0;

}
```

---

3. Before a timer can be used, it has to be cleared, otherwise it may have an unwanted time value still stored in it.



```
task main()
{

   ClearTimer(T1);

   while(SensorValue(touchSensor) == 0)
   {

```

3. Add this code

Reset the Timer T1 to 0 and start it counting just before the loop begins.

Line Tracking **Timer** (cont.)

4. Now, change the while loop's (condition) to check the timer instead of the touch sensor. The robot should line track while the timer T1 reads less than 3000 milliseconds.

```
task main()
{

   ClearTimer(T1);

   while(time1[T1] < 3000)
   {

      if(SensorValue(lightSensor) < 45)
      {

         motor[motorC] = 0;
         motor[motorB] = 80;

      }

      else
      {

```

4. Modify this line
Base the decision about whether to continue running, on how much time has passed since T1's last reset.

End of Section
Download and Run.

Line Tracking for Time(r)
The robot tracks the line for a set amount of time. But is time really what you want to measure?

ROBOTC gives you four different timers to work with: T1, T2, T3, and T4. They can be reset and run independently, in case you need to time more than one thing. You reset them the same way – **`ClearTimer(T2);`** – and you check them the same way – **`time1[T2]`**.

Still, there's the issue of timing itself. Motors, even good ones, aren't perfectly precise. By assuming that you're going a certain speed, and therefore will go a certain distance in a set amount of time, you are making a pretty bold assumption.

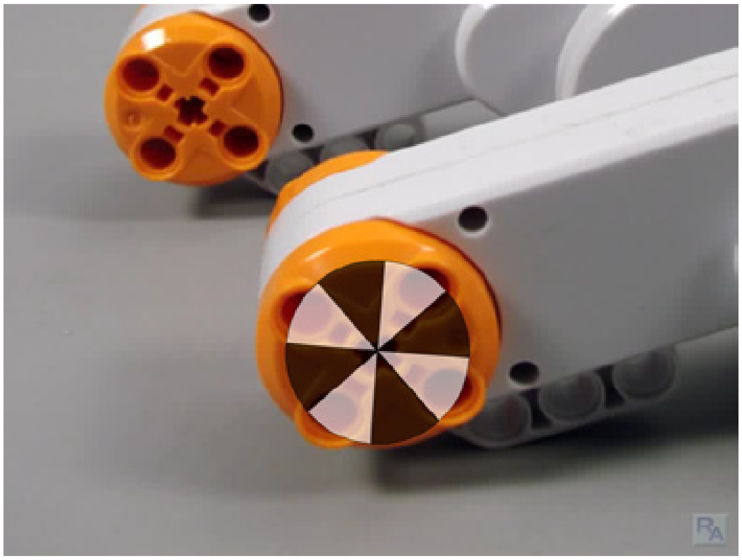
In the next part of this lesson, you'll find out how to track a line for a certain distance, instead of tracking for time and hoping that it equates to the correct distance.

Line Tracking Rotation

In this lesson we'll find out how to watch for distance, instead of watching for *time* and hoping that the robot moves the correct distance, like in our previous program.

NXT Motors
Rotation sensors are built into every NXT motor.

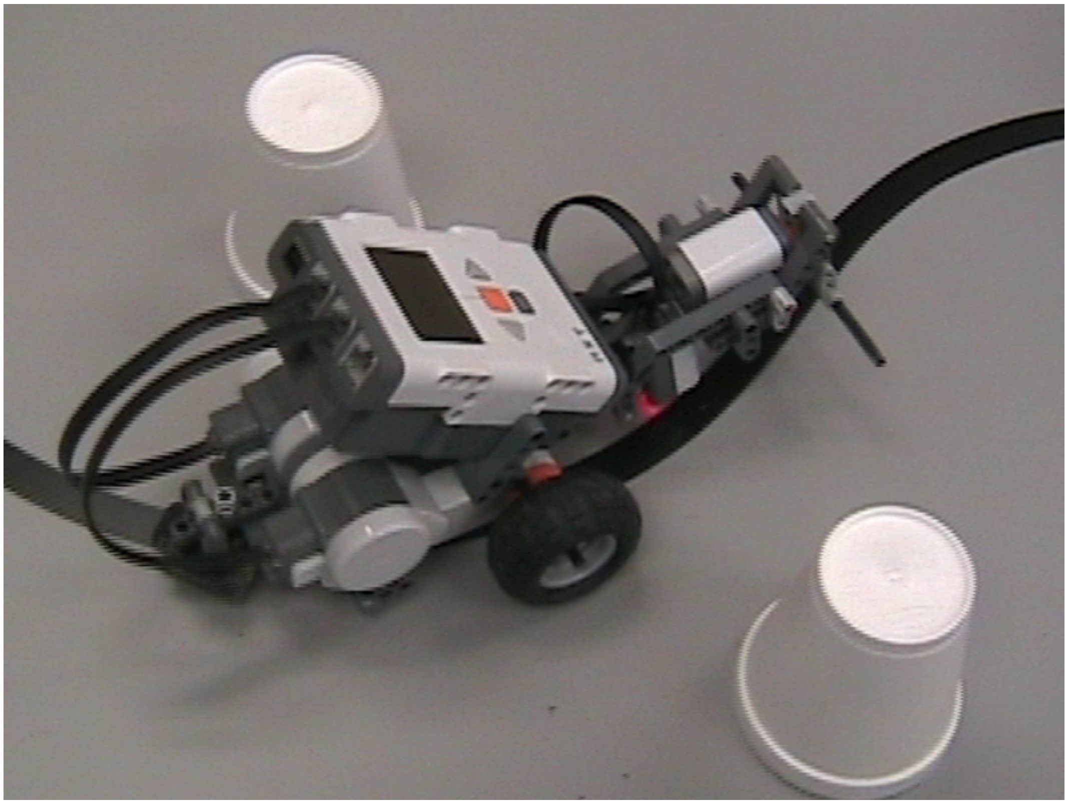
A rotation sensor is a patterned disc attached to the inside of the motor. By monitoring the orientation of the disc as it turns, the sensor can tell you how far the motor has turned, in degrees. Since the motor turns the axle, and the axle turns the wheel, the rotation sensor can tell you how much the wheel has turned. Knowing how far the wheel has turned can tell you how far the robot has traveled. Setting the robot to move until the rotation sensor count reaches a certain point allows you to accurately program the robot to travel a set distance.



Line Tracking **Rotation** (cont.)

Review
The last program we're going to visit in the Line Tracking lesson is perhaps the most useful form, but it's taken us awhile to get here. Progress in engineering and programming projects is often made in this "iterative" way, by making small, directed improvements that build upon one another. Let's quickly review what we have done in some of the previous lessons.

We started with figuring out that a **line tracking behavior** consists of bouncing back and forth between light and dark areas in an effort to follow the edge of a line.

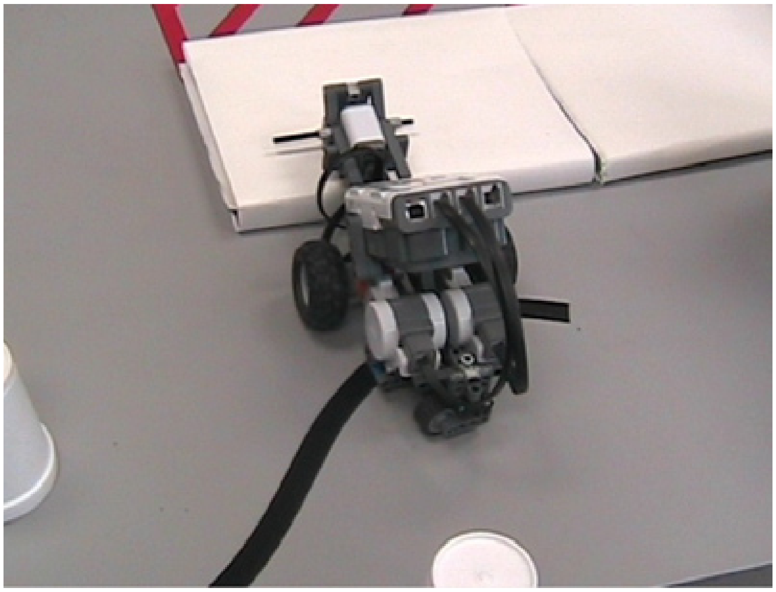


Line Tracking **Rotation** (cont.)

We then implemented a naive version of the line tracking behavior using **while() loops**, inside other while() loops.

```
task main()
{

   while(1 == 1)
   {

     while(SensorValue(lightSensor) < 45)
      {

           motor[motorC] = 0;
           motor[motorB] = 80;

      }

     while(SensorValue(lightSensor) >= 45)
      {

           motor[motorC] = 80;
           motor[motorB] = 0;

      }

   }

}
```

But, we found that the program could get stuck inside one of those inner loops, preventing it from checking the sensor that we wanted to use to stop the tracking.



Line Tracking **Rotation** (cont.)

We then implemented **if-else conditional statements**, which allow instantaneous sensor checking, and thus avoid the “nesting” of loops inside other loops, which had caused the program to get stuck.

```
      if(SensorValue(lightSensor) < 45)
      {

         motor[motorC] = 0;
         motor[motorB] = 80;

      }

      else
      {

         motor[motorC] = 80;
         motor[motorB] = 0;

      }

    }
```

Then, we upgraded from checking a Touch Sensor, to being able to use an independent **timer** to determine how long to run the line tracker.

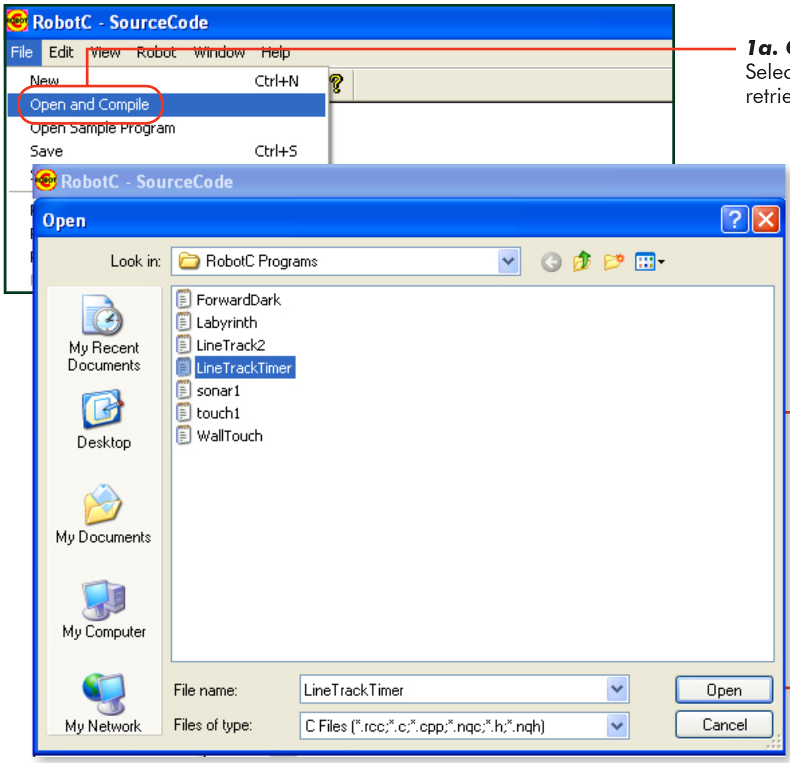
```
task main()
{

   ClearTimer(T1);

   while(time1[T1] < 3000
   {

      if(SensorValue(lightSensor) < 45)
      {

         motor[motorC] = 0;
         motor[motorB] = 80;

      }

      else
      {

```

Line Tracking **Rotation** (cont.)

Now, let’s improve upon the Timer-based behavior by using a sensor more fundamentally connected to the quantity we wish to measure: distance traveled, using the Rotation Sensor.

In this lesson you will learn how to use the Rotation Sensors built into every NXT motor to make a line tracking behavior run for a set distance.

1. Start by opening the Line Tracking Timer Program “LineTrackTimer”.



File name: LineTrackTimer

Files of type: C Files (*.rcc;*.c;*.cpp;*.nqc;*.h;*.nqh)

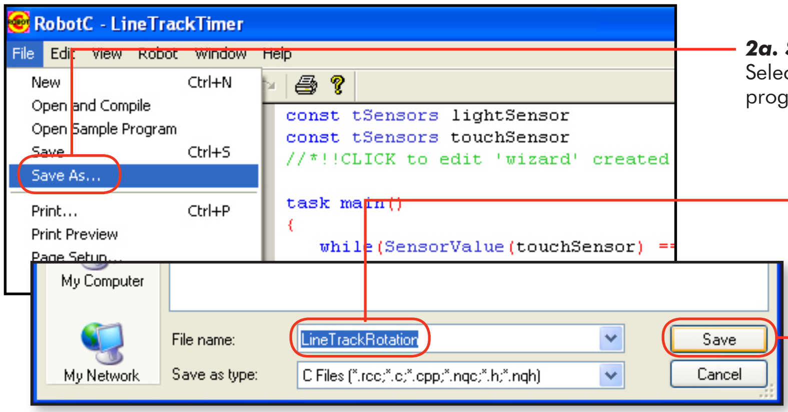
1a. Open Program
Select File > Open and Compile to retrieve your old program.

1b. Select the program
Select “LineTrackTimer”.

1c. Open the program
Press Open to open the saved program.

---

2. Save this program under a new name, “LineTrackRotation”.



```
const tSensors lightSensor
const tSensors touchSensor
//*!!CLICK to edit 'wizard' created

task main()
{
   while(SensorValue(touchSensor) ==
```

File name: LineTrackRotation

Save as type: C Files (*.rcc;*.c;*.cpp;*.nqc;*.h;*.nqh)

2a. Save program As...
Select File > Save As... to save your program under a new name.

2b. Name the program
Give this program the name “LineTrackRotation”.

2c. Save the program
Press Save to save the program with the new name.

Line Tracking Rotation (cont.)

Checkpoint

Your starting program for this lesson should look like the one below.

```
task main()
{

   ClearTimer(T1);

   while(time1[T1] < 3000)
   {

      if(SensorValue(lightSensor) < 45)
      {

         motor[motorC] = 0;
         motor[motorB] = 80;

      }

      else
      {

         motor[motorC] = 80;
         motor[motorB] = 0;

      }

   }

   motor[motorC] = 0;
   motor[motorB] = 0;

}
```

It's time to start changing the program to use the Rotation sensors. Rotation sensors have **no guaranteed starting position**, so, you must first reset the rotation sensor count. It will take the place of the equivalent reset code used for the Timer.

In the robotics world, the term **"encoder"** is often used to refer to any device that measures rotation of an axle or shaft, such as the one that spins in your motor. Consequently, the ROBOTC word that is used to access a Rotation Sensor value is **`nMotorEncoder[`*`MotorName`*`]`**.

Unlike the Timer, which has its own ClearTimer command, the rotation sensor (motor encoder) value must be manually set back to zero to reset it. The command to do so will look like this:

Example:

```
nMotorEncoder[motorC] = 0;
```

Line Tracking **Rotation** (cont.)

3. Start with the left wheel, attached to Motor C on your robot. Reset the rotation sensor on that motor to 0.

```
task main()
{

   nMotorEncoder[motorC] = 0;

   while(time1[T1] < 3000)
   {

      if(SensorValue(lightSensor) < 45)
      {

         motor[motorC] = 0;
         motor[motorB] = 80;

      }

      else
      {

         motor[motorC] = 80;
         motor[motorB] = 0;

      }

   }

   motor[motorC] = 0;
   motor[motorB] = 0;

}
```

3. Modify this code
Instead of resetting a Timer, reset the rotation sensor in MotorC to a value of 0. Replace `ClearTimer(T1);` with `nMotorEncoder[motorC]=0;`

Line Tracking **Rotation** (cont.)

4. Reset the other motor’s rotation sensor, `nMotorEncoder[motorB] = 0;`

```
task main()
{

    nMotorEncoder[motorC] = 0;
    nMotorEncoder[motorB] = 0;

    while(time1[T1] < 3000)
    {

        if(SensorValue(lightSensor) < 45)
        {

            motor[motorC] = 0;
            motor[motorB] = 80;

        }

        else
        {

            motor[motorC] = 80;
            motor[motorB] = 0;

        }

    }

    motor[motorC] = 0;
    motor[motorB] = 0;

}
```

4. Add this code
Reset the rotation sensor in MotorB to 0 as well.

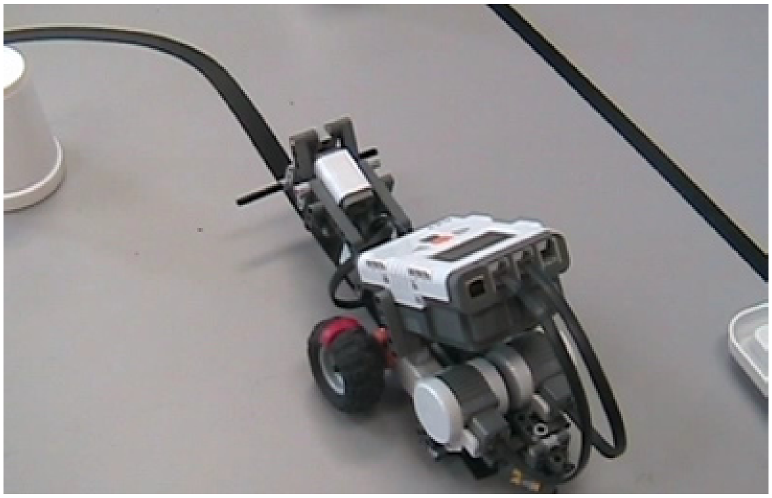
Line Tracking **Rotation** (cont.)

5. The NXT motor encoder measures in degrees, so it will count 360 for every full rotation the motor makes. Change the **`while()`** loop's condition to make this loop run while the **`nMotorEncoder`** value of motorC is less than 1800 degrees, five full rotations.

```
task main()
{

   nMotorEncoder[motorC] = 0;
   nMotorEncoder[motorB] = 0;

   while(nMotorEncoder[motorC] < 1800)
   {

      if(SensorValue(lightSensor) < 45)
      {

         motor[motorC] = 0;
         motor[motorB] = 80;

      }

      else
      {

```

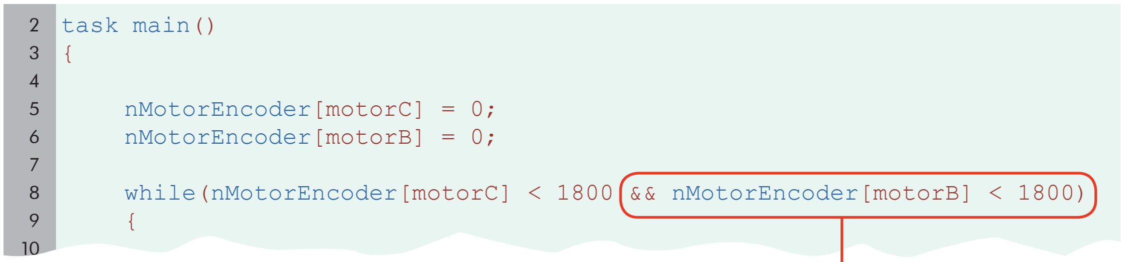
5. Modify this code
Set MotorC to run for five full rotations or 1800 degrees.

Checkpoint
Save, download and run your program. You may want to mark one of the wheels with a piece of tape so that you can count the rotations.



Line Tracking **Rotation** (cont.)

6. We only checked one wheel and not the other. Add a check for the other motor's encoder value to the condition. The {condition} will now be satisfied and loop as long as BOTH motors remain below the distance threshold of 1800 degrees.



```
task main()
{

    nMotorEncoder[motorC] = 0;
    nMotorEncoder[motorB] = 0;

    while(nMotorEncoder[motorC] < 1800 && nMotorEncoder[motorB] < 1800)
    {
```

6. Add this code
This change sets the condition to run while "the motor encoder on motorC reads less than 1800 degrees, AND the motor encoder for motorB also reads less than 1800 degrees.

End of Section
Download and run this program, and you will see that on curves going to the left, where the right motor caps out at 1800 first, this program will stop sooner than the one that just waited for the left motor (remember, the left motor is traveling less when making a left turn).

Take a step back, and look at what you have. Your robot is now able to perform a behavior using one sensor, while watching another sensor to know when to stop. Using the rotation sensor means that your robot can now travel for a set distance along the line, and be pretty sure of how far it's gone. These capabilities can be applied to more than just line tracking, however. You can now build any number of environmentally-aware decision-making behaviors, and run them until you have a good reason to stop. This pattern of while and conditional loops is one of the most frequently used setups in robot programming. Learn it well, and you will be well prepared for many roads ahead.