

Greetings Roboticists,

These days, robotics and intelligent systems are found everywhere— smart cars, smart houses, smart buildings, smart phones, healthcare technology, internet search engines, automated security systems, all phases of the shipping industry... intelligent systems are ubiquitous. Students, as future innovators, need to know to use them.

Robots elicit curiosity from people of all ages; there is something that fascinates people when they see a robot moving around making decisions on its own. This natural attraction can open up opportunities for inspiration and enlightenment in both conventional and unconventional ways. In fact, robotics may be the premier integrator in education today. When students study robotics, they learn about engineering, electronics, and programming. They gain equally valuable experience in managing projects, analyzing systems, accessing information, working in teams, and problem solving.

Carnegie Mellon and LEGO are working together to design research-based educational tools that promote mathematical and engineering competency, as well as technological and scientific literacy for all generations of students. The *Teaching ROBOTC for LEGO MINDSTORMS* training CD enables students to take their first step toward becoming competent programmers, engineers, and innovators.

In these lessons, students are given opportunities to design, build, program and troubleshoot tabletop robots. These projects require a diverse and well-rounded skillset, from measurement to analysis, calculation to communication, individual initiative to group collaboration. Engineering is a complex and multi-faceted discipline, one which reflects the challenges and demands that tomorrow will make of its citizens.

Today, we are finding that more high schools and colleges are using MINDSTORMS and other robots to introduce engineering competencies and control concepts. Programming is an elusive key skill that unlocks the potential of all these intelligent systems for students and educators. Teaching programming builds a foundation for the future. *Teaching ROBOTC for LEGO MINDSTORMS* is a tool that we hope will help you do that.

Best regards,

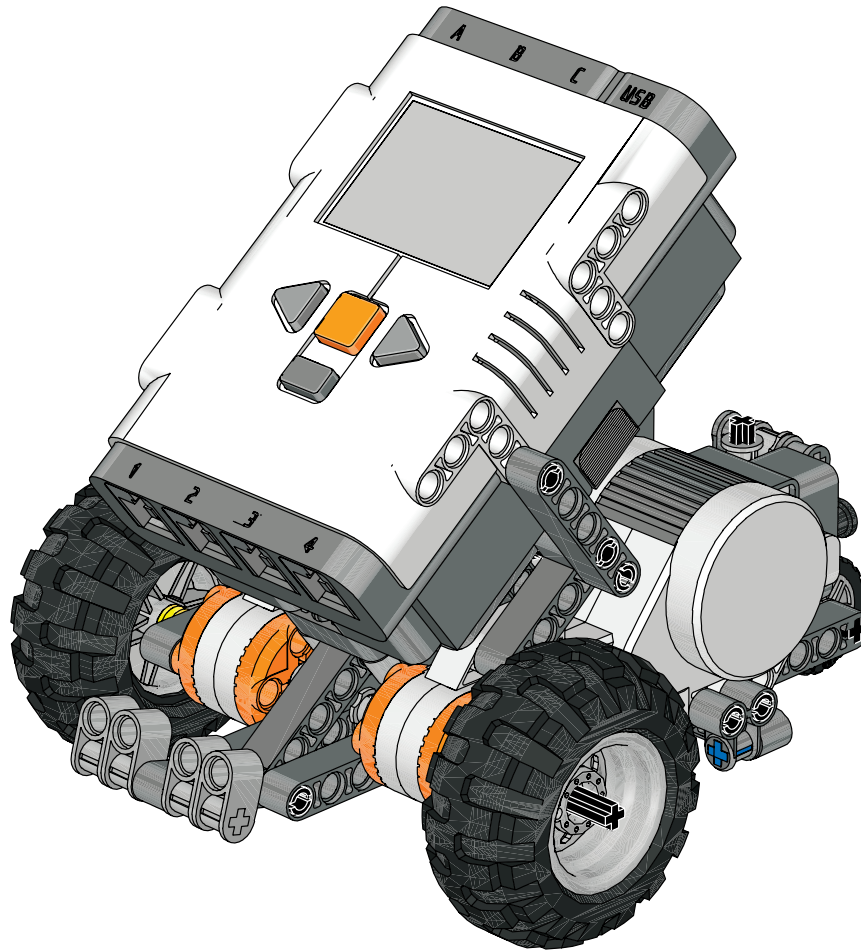


Robin Shoop,
Director of Educational Outreach
Carnegie Mellon Robotics Academy

Carnegie Mellon

Robot Educator Model

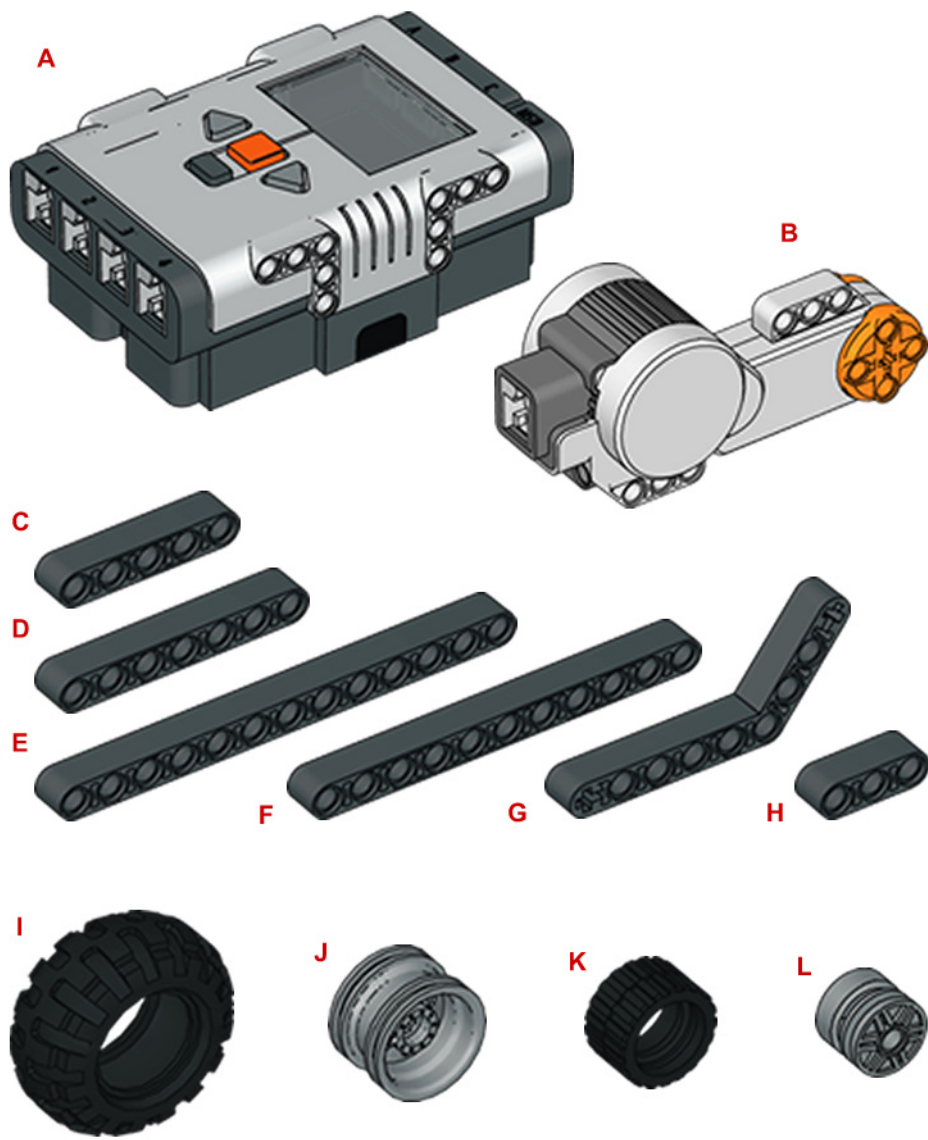
Building Instructions



Parts Page 1

- A** 1x NXT Brick
- B** 2x NXT Servo Motor
- C** 3x 5-Module Beam
- D** 2x 7-Module Beam
- E** 2x 13-Module Beam
- F** 1x 11-Module Beam
- G** 1x 4x6 Angular Beam
- H** 3x 3-Module Beam
- I** 2x 56x26 Tire
- J** 2x 30x20 Hub
- K** 1x 24x14 Tire
- L** 1x 18x14 Hub

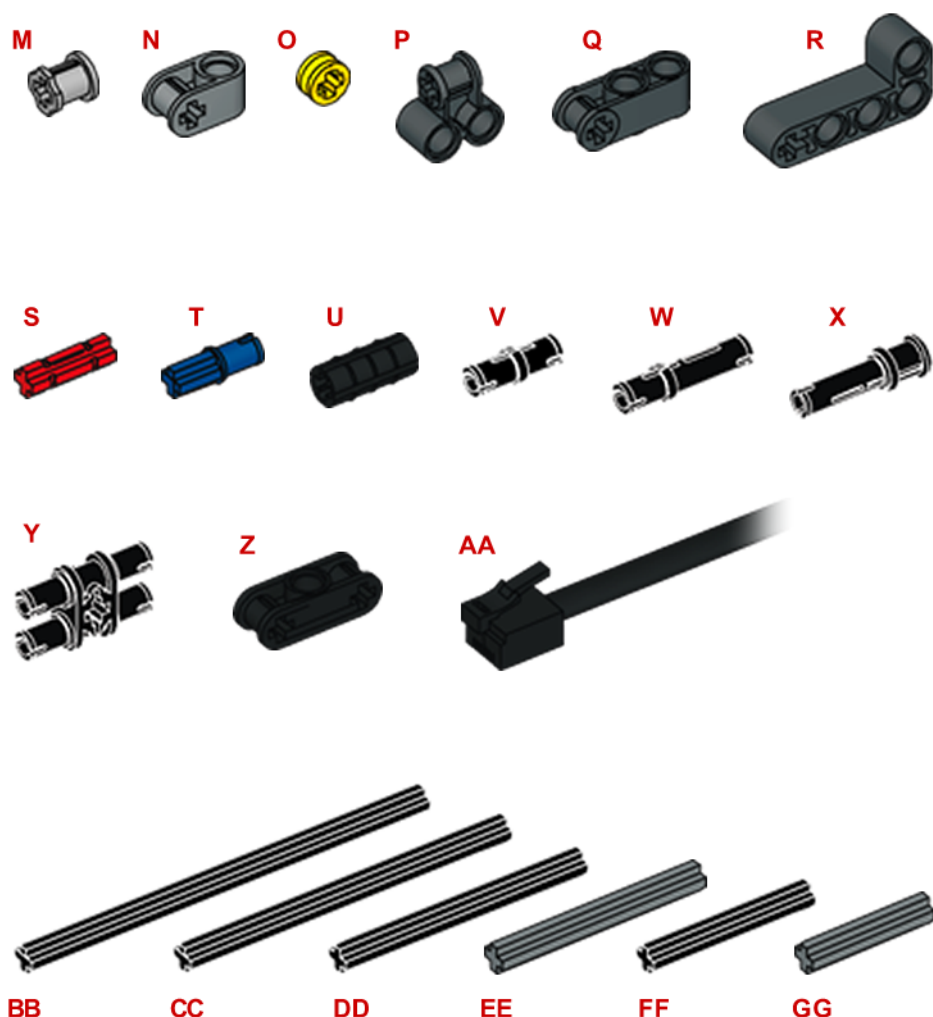
A = Part Number
1x = Amount Needed

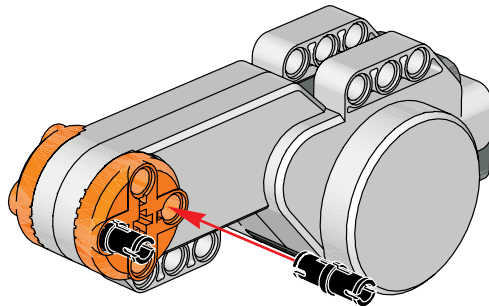
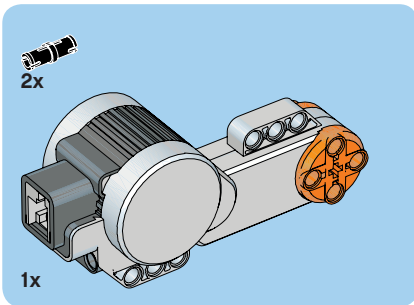
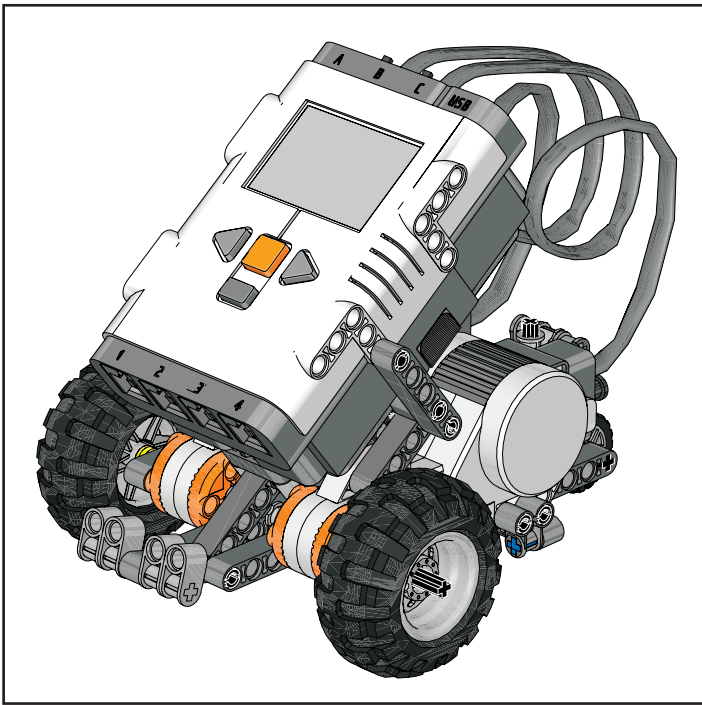


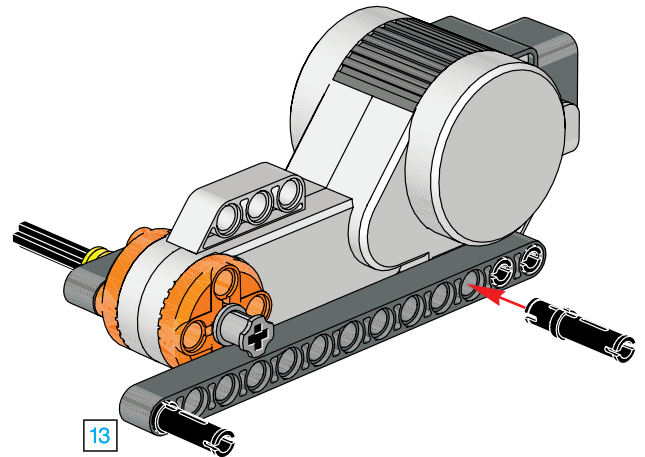
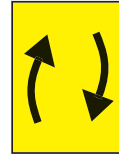
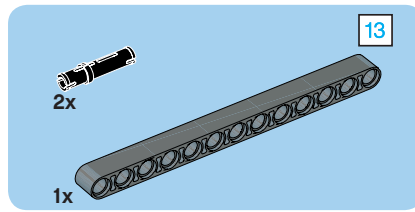
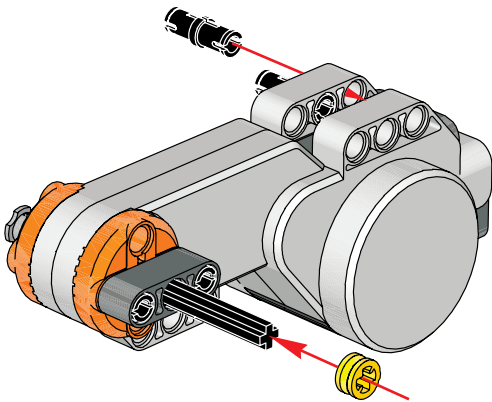
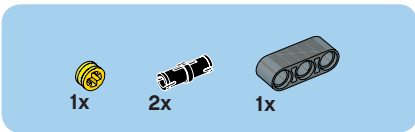
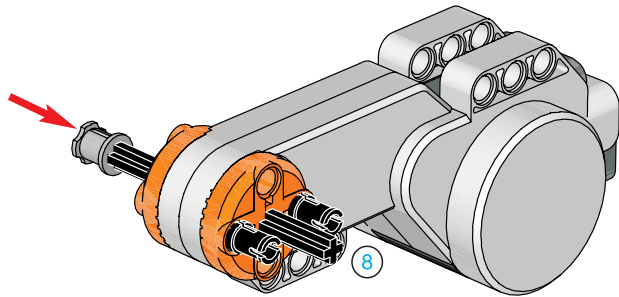
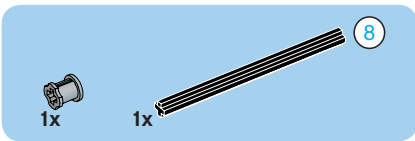
Parts Page 2

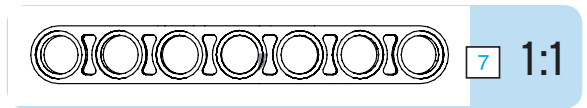
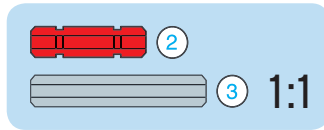
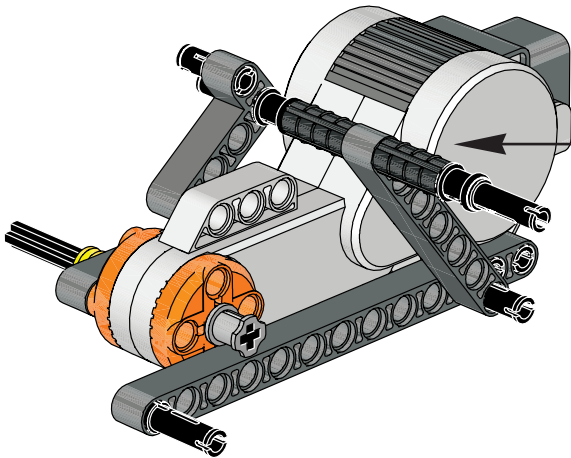
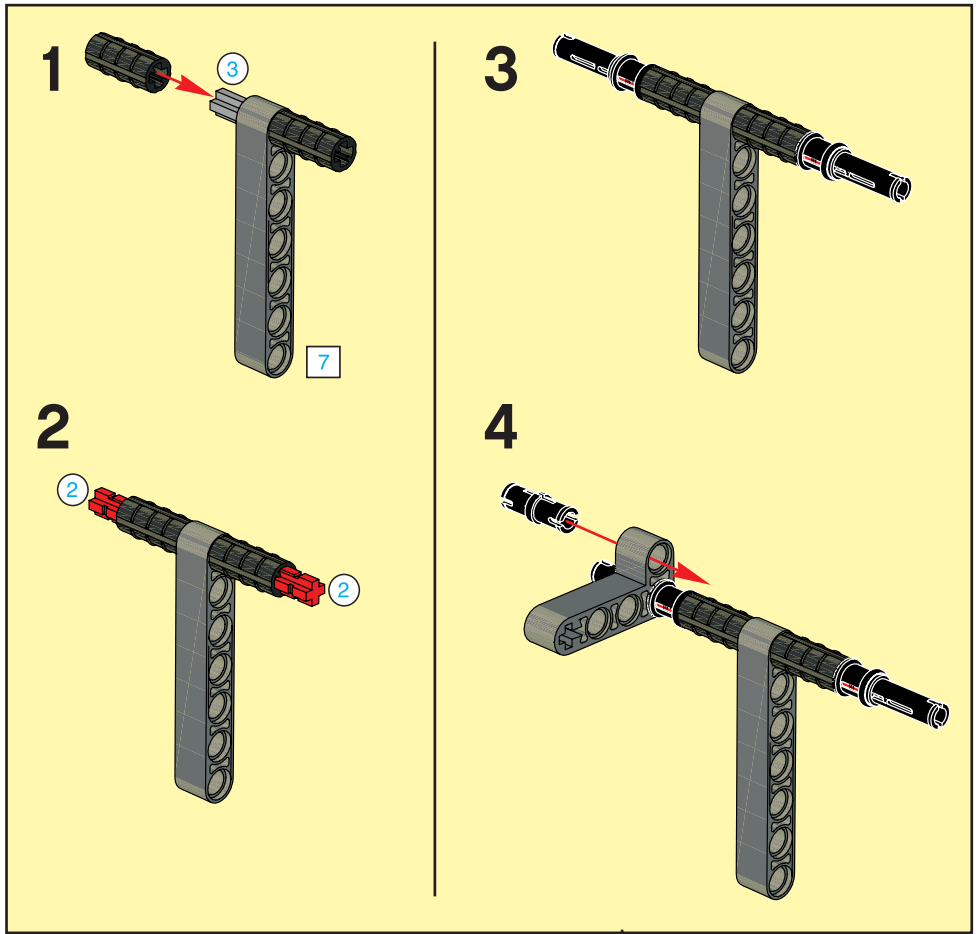
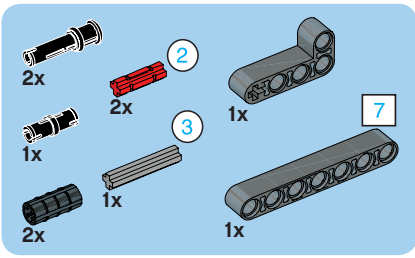
- M** 6x Bushing
- N** 8x 2-Module Cross Block
- O** 7x 1/2-Module Bushing
- P** 2x 2x1-Module Cross Block
- Q** 2x 3-Module Cross Block
- R** 4x 4x2 Angular Beam
- S** 2x 2-Module Axle
- T** 2x Connector Peg with Friction Axle
- U** 2x Axle Extender
- V** 18x Connector Peg with Friction
- W** 4x 3-Module Connector Peg with Friction
- X** 2x Connector Peg with Bushing
- Y** 2x 2-Module Double Connector Peg
- Z** 3x Double Cross Block
- AA** 2x Cable
- BB** 1x 10-Module Axle
- CC** 2x 8-Module Axle
- DD** 1x 6-Module Axle
- EE** 3x 5-Module Axle
- FF** 3x 4-Module Axle
- GG** 2x 3-Module Axle

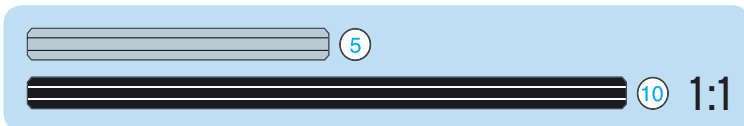
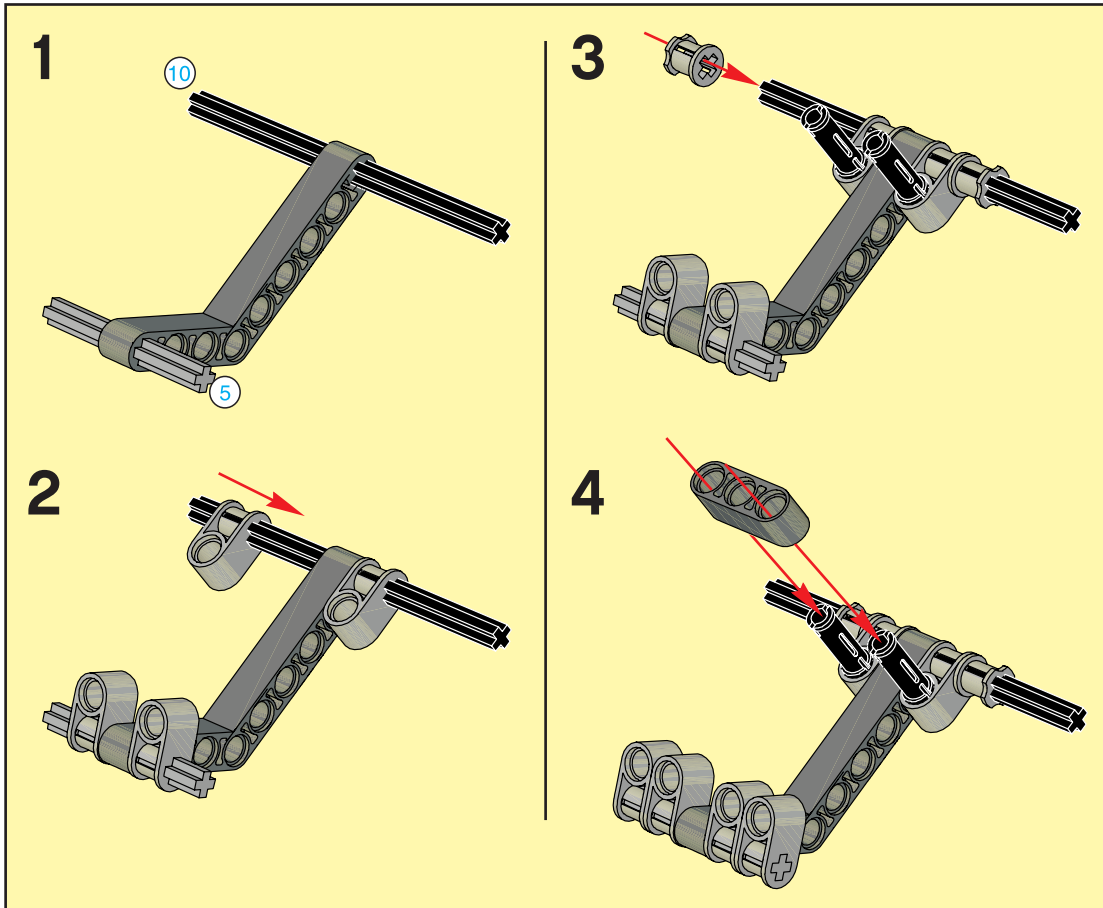
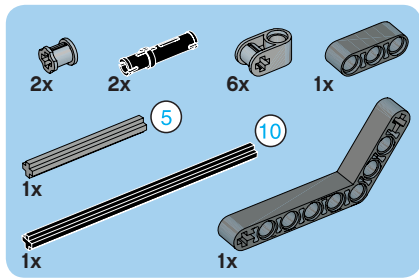
A = Part Number
1x = Amount Needed

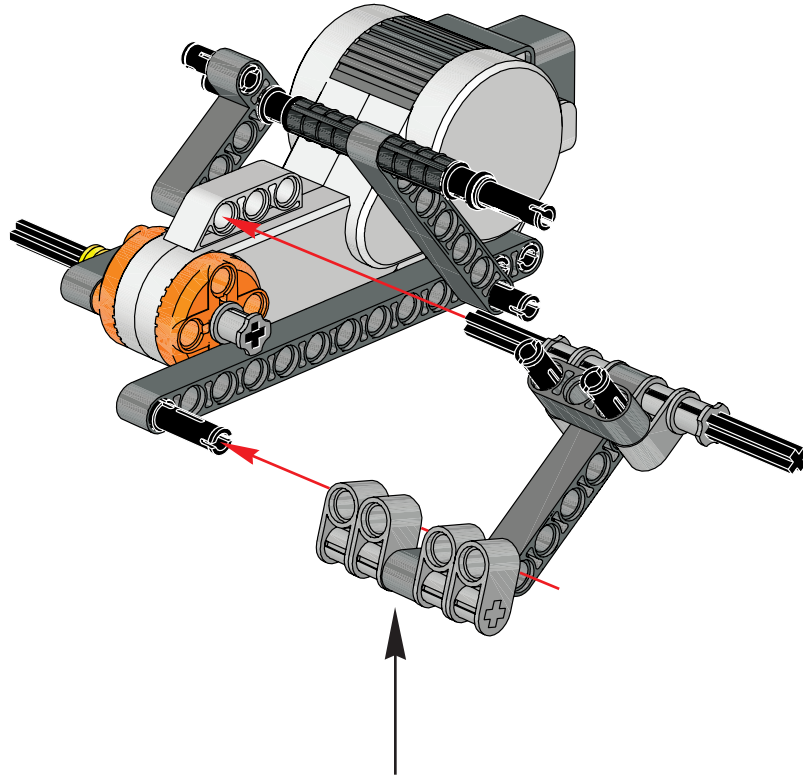


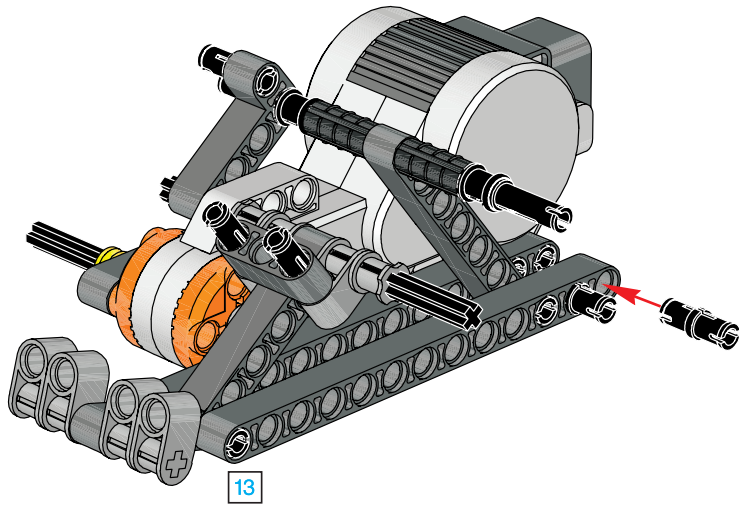
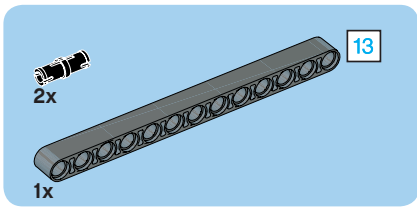






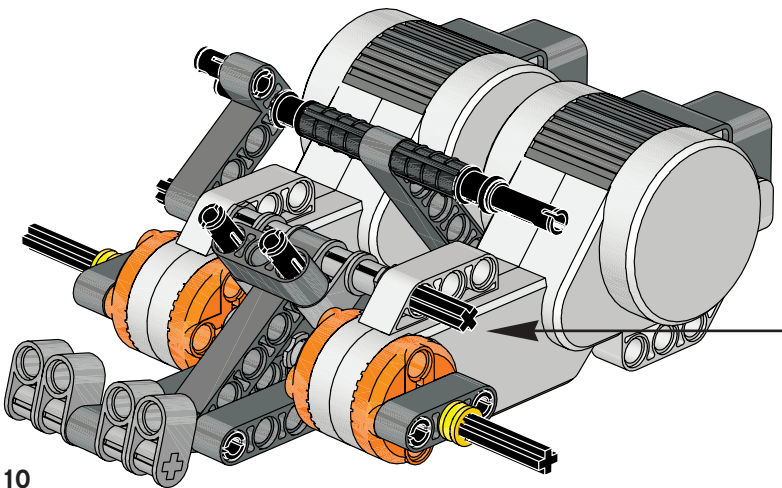
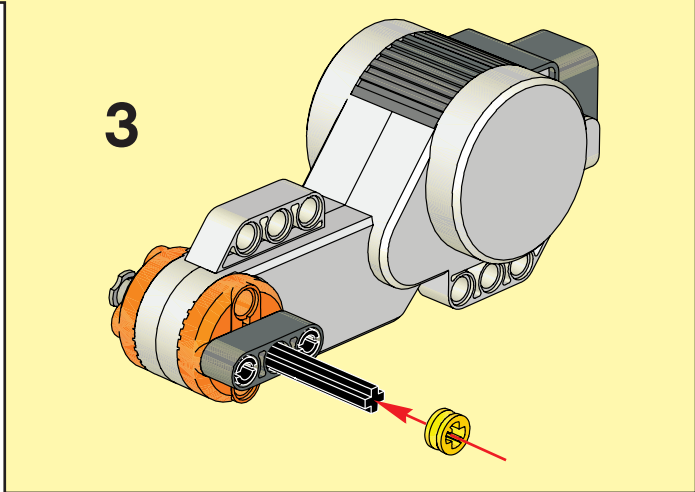
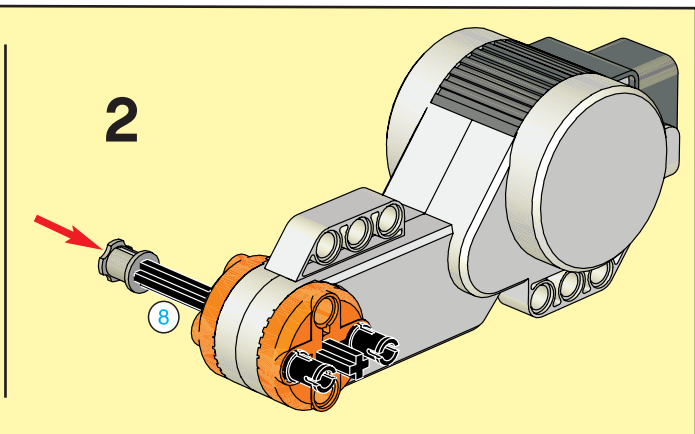
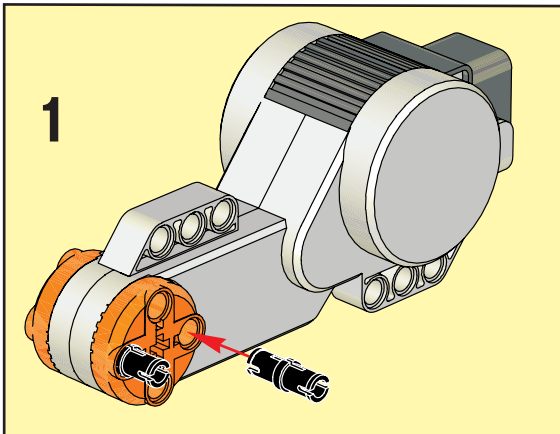
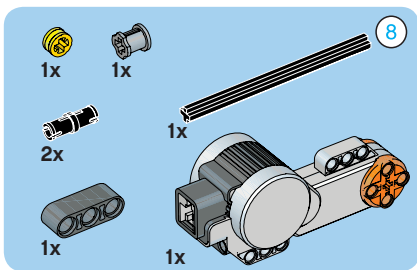


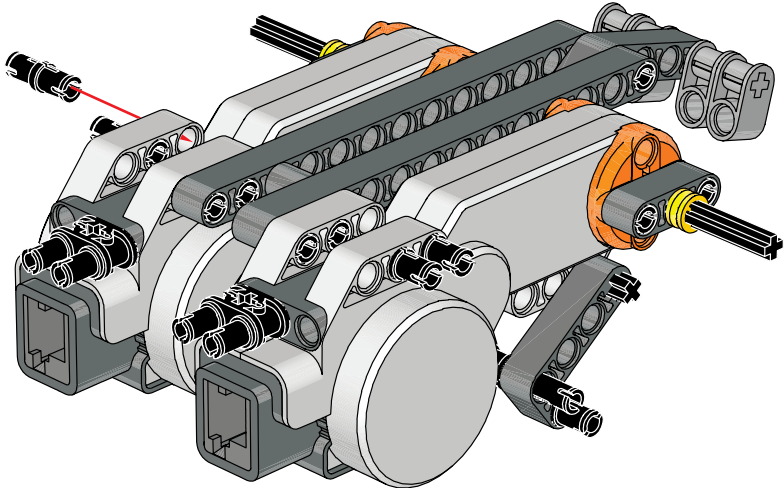
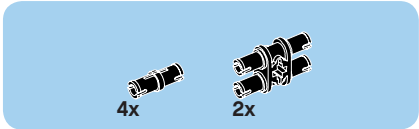
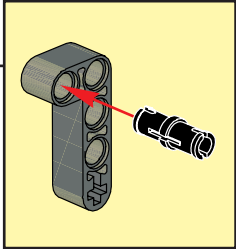
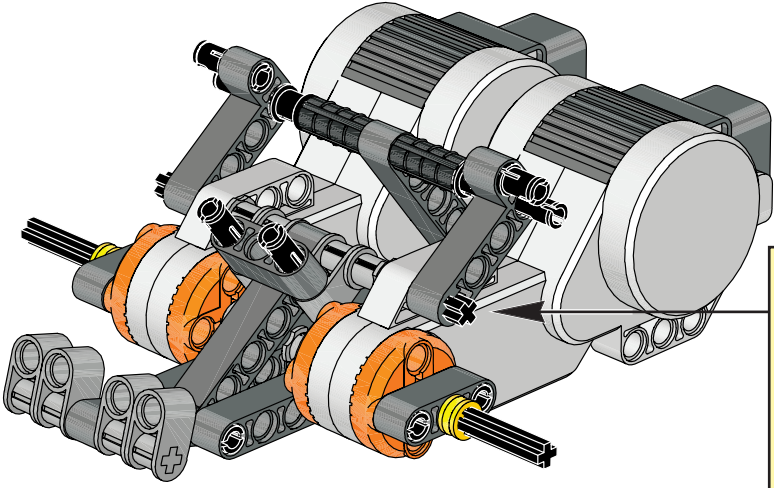
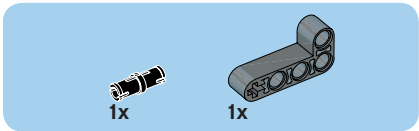


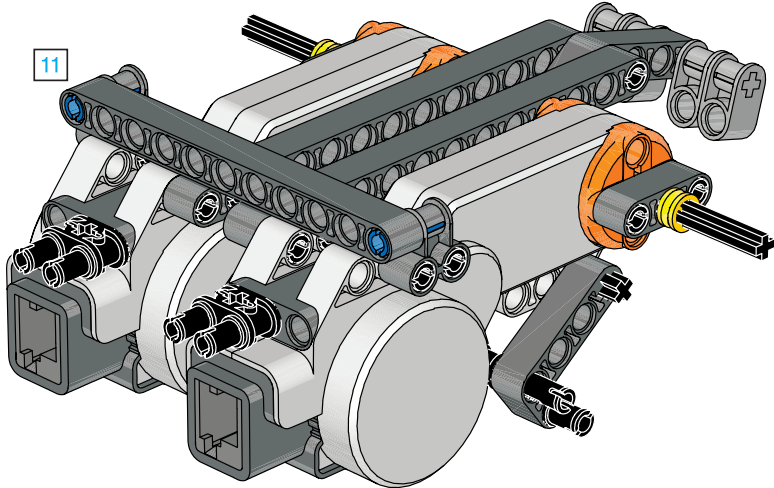
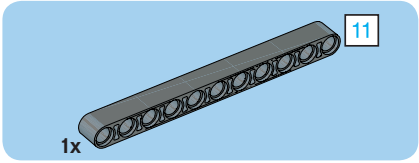
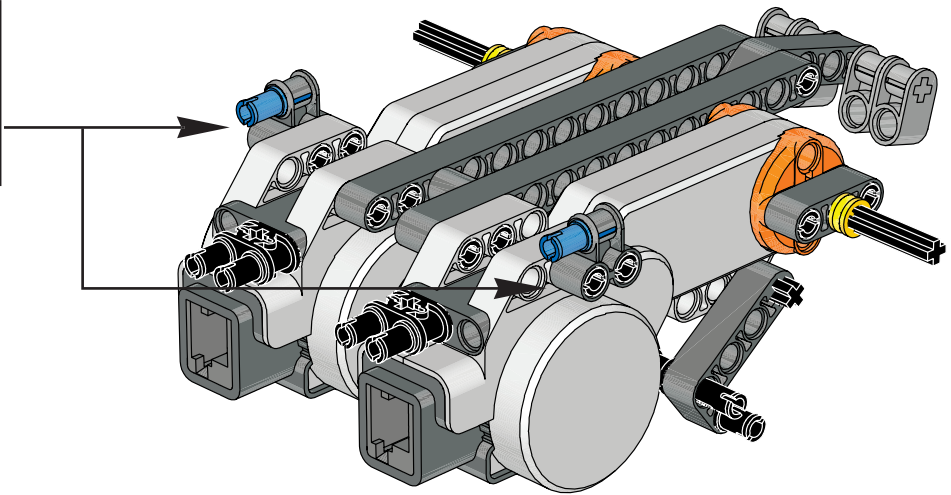
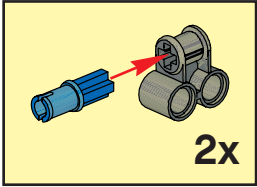
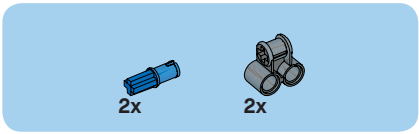


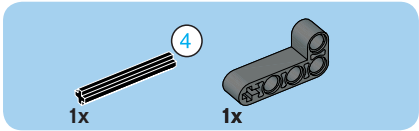
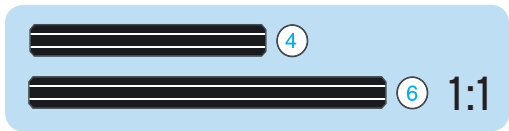
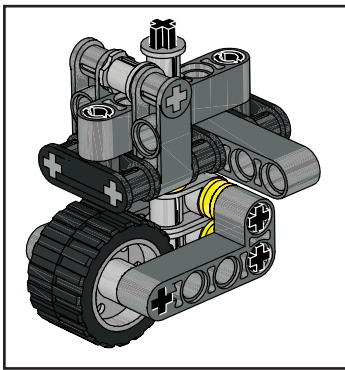
13

1:1

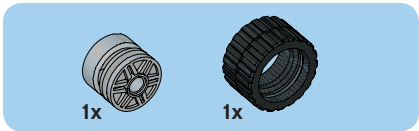
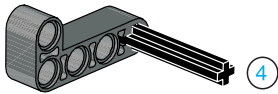




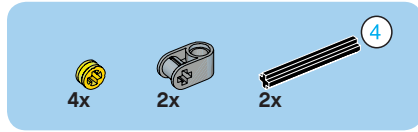
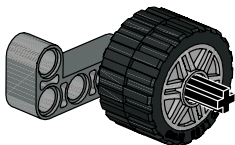




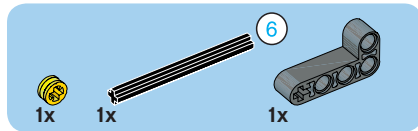
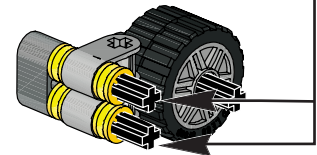
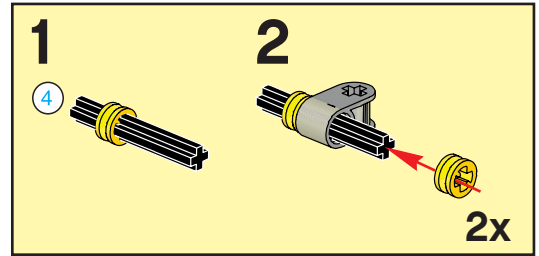
1



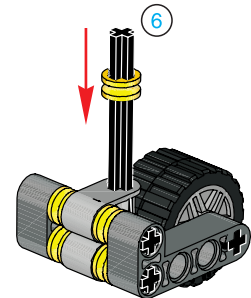
2

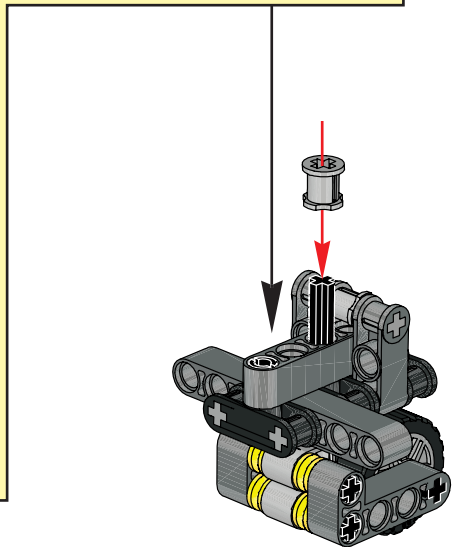
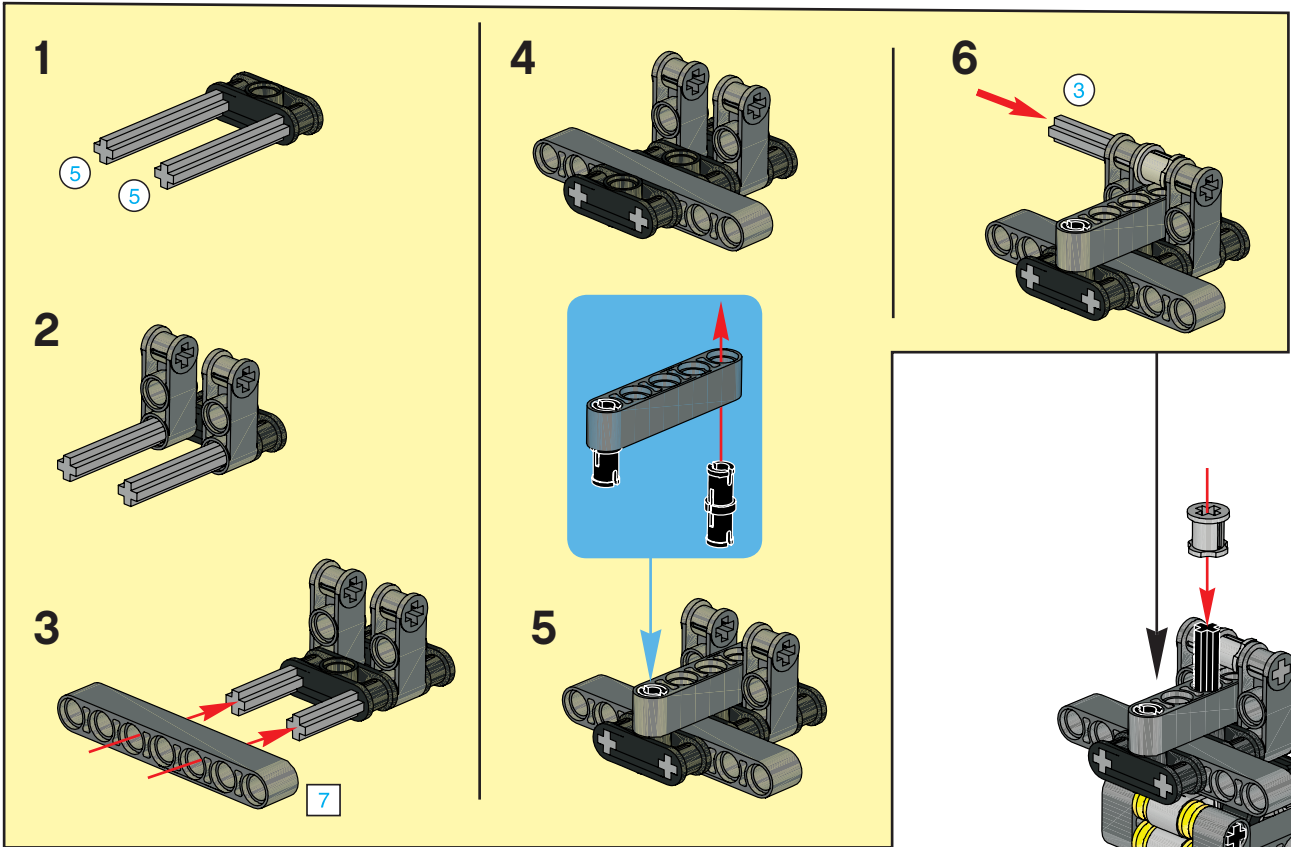
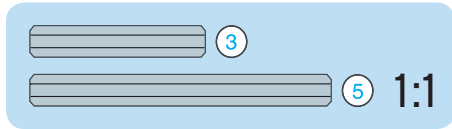
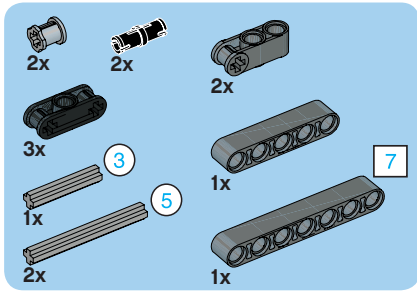


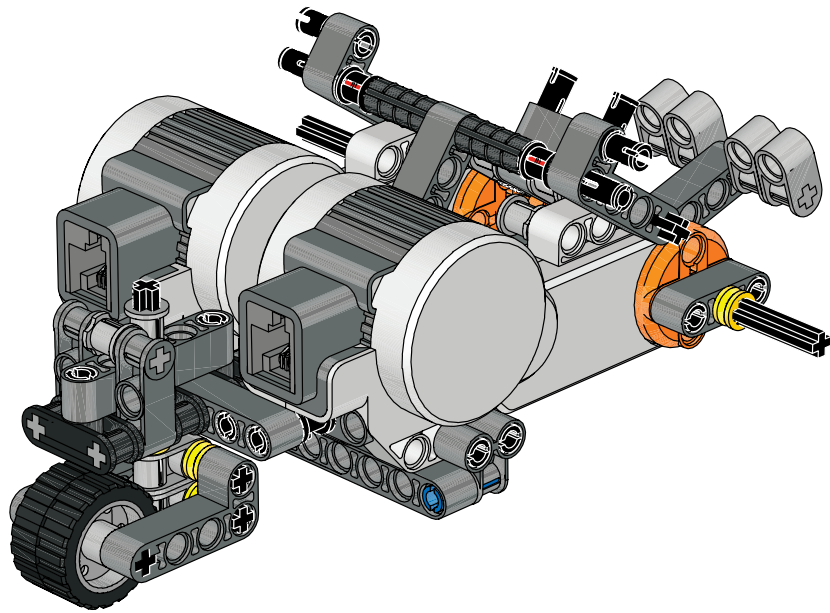
3

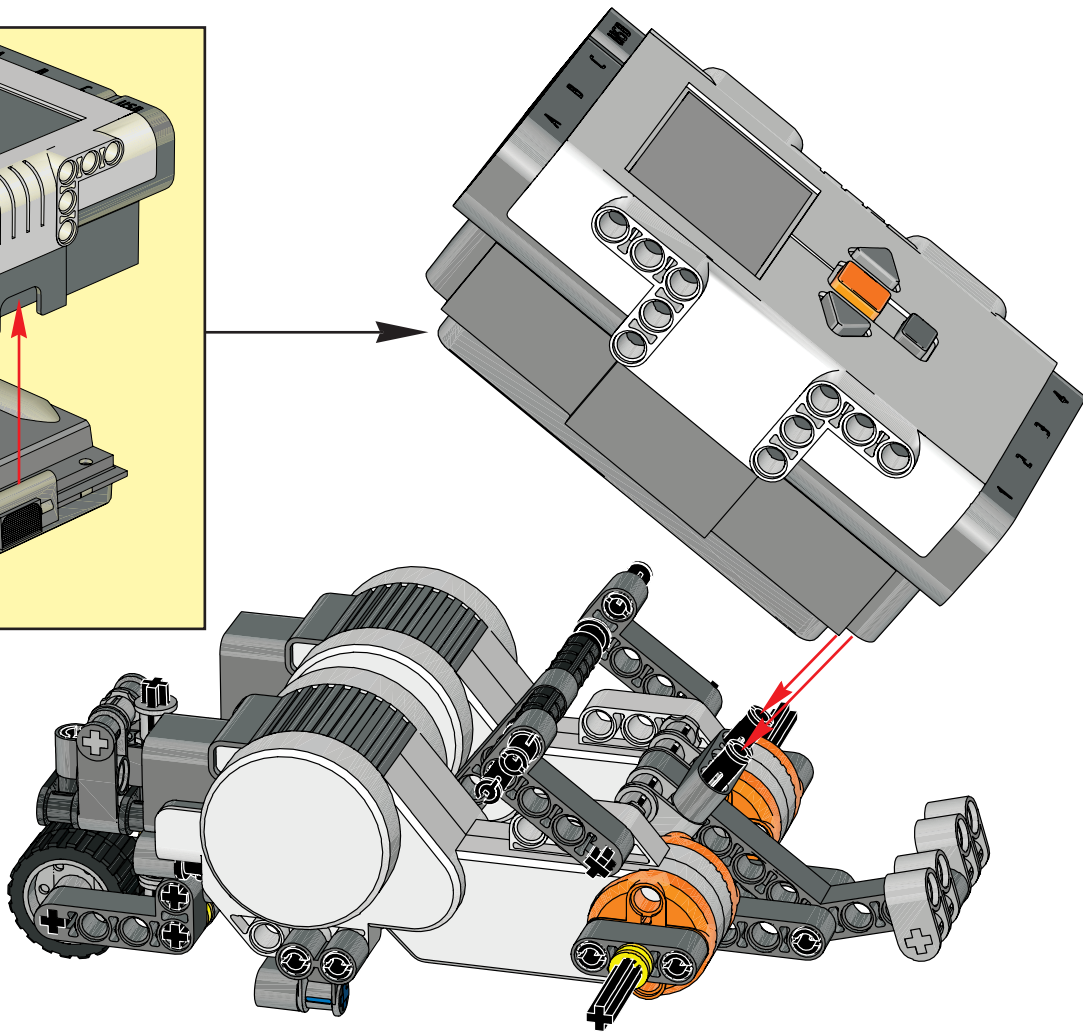
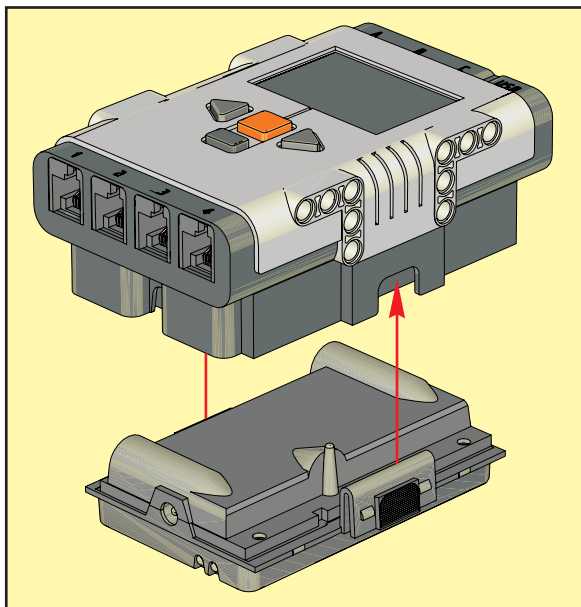
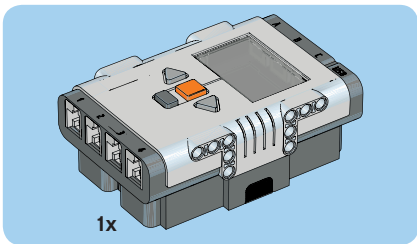


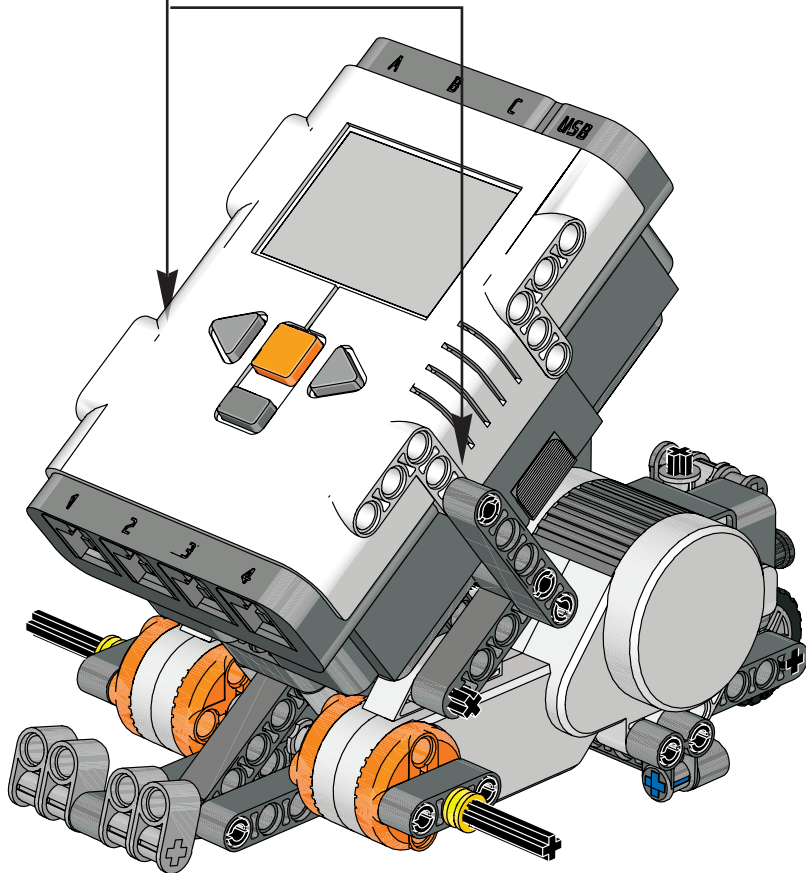
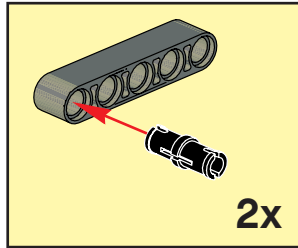
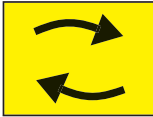
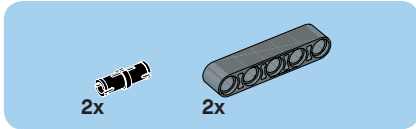
4

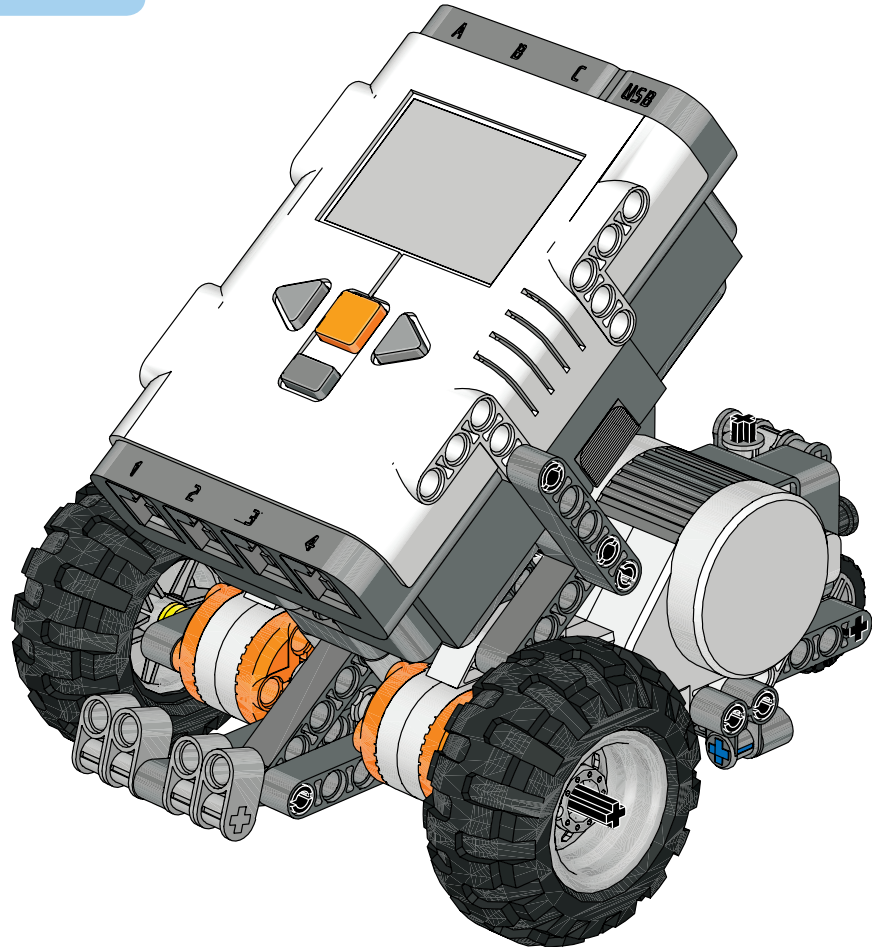


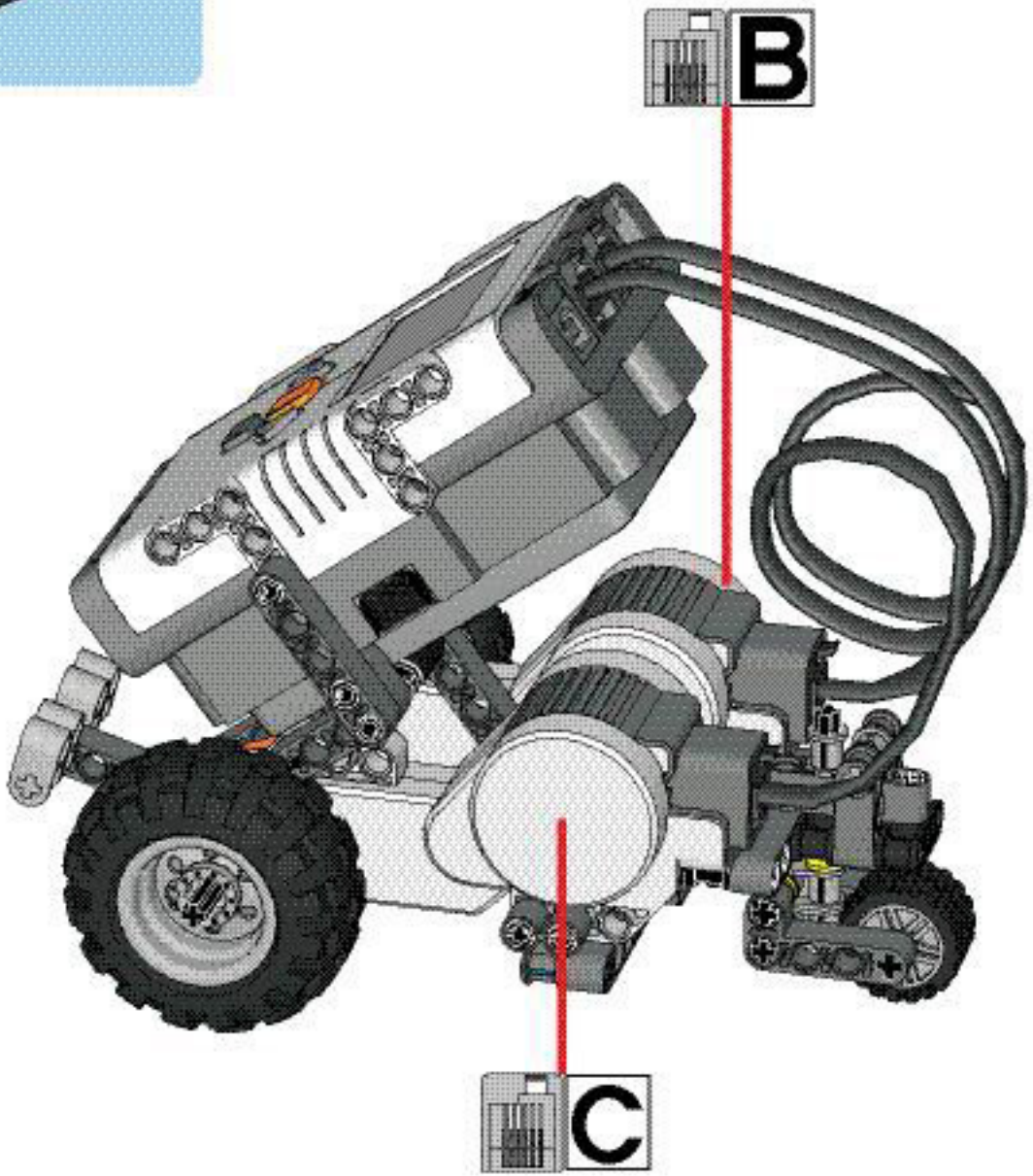




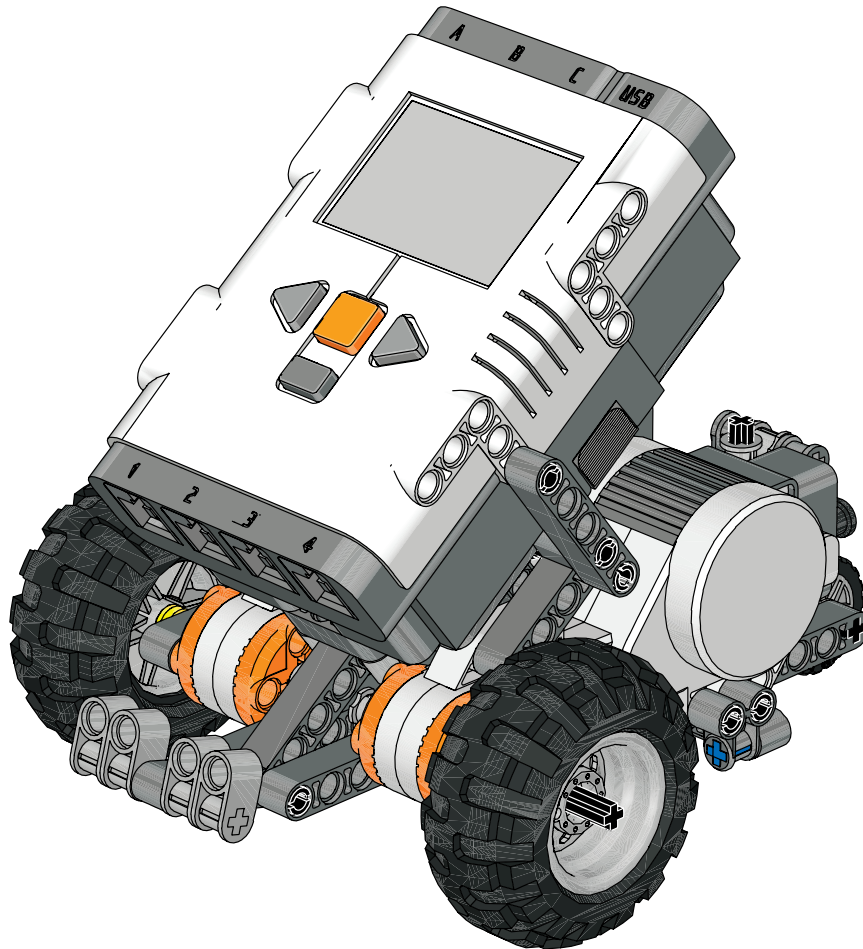








Your **Robot Educator Model** is now complete!



Setup

Firmware

You have installed ROBOTC and built the REM bot, but the robot is not yet able to understand ROBOTC programs. You must first download firmware onto your NXT. Firmware is the operating system for your robot. Once loaded on the brick, the firmware will allow the NXT to load and run ROBOTC programs.

You will need:

1. Your NXT
2. A computer with ROBOTC installed
3. A USB connector cable (A-B, included with 9797 base set)

1. Plug one end of the USB cable into your NXT, and the other into your computer.
If the robot is not on press the orange button on your NXT brick.



1a. Connect the USB cable

Plug one end of the USB cable into your robot, and the other into your computer to allow communication between them.



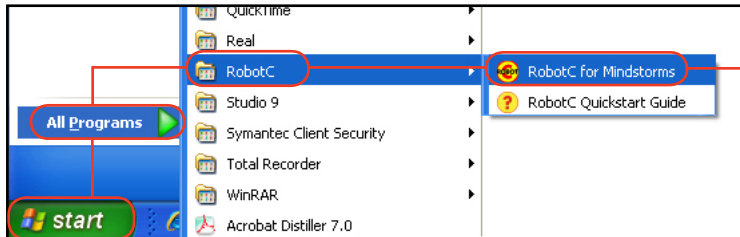
1b. Turn NXT on

Press the orange square on your NXT brick to turn your robot on if it is not already.

Setup

Firmware

- Open up the ROBOTC program. To start ROBOTC go to the Start Menu, Programs or All Programs, RobotC and finally choose "RobotC for Mindstorms".

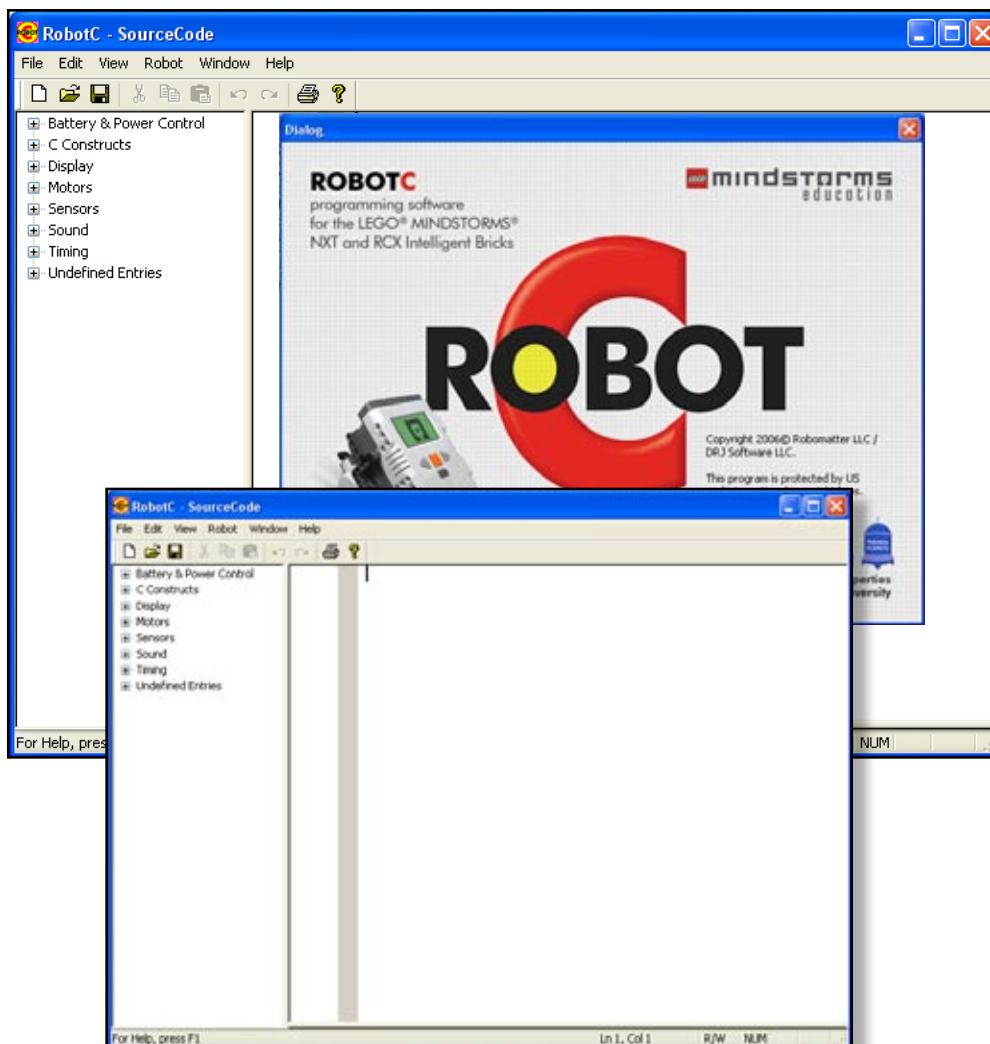


2. Open ROBOTC for Mindstorms

Select the Start Menu > Programs or All Programs > RobotC > RobotC for Mindstorms to open up the ROBOTC program.

Checkpoint

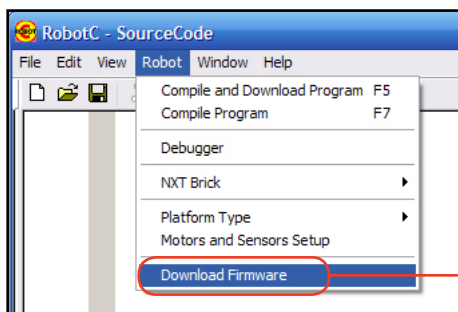
This what your screen should look like. The ROBOTC Dialog box will disappear after a few seconds. What is left is the main ROBOTC window.



Setup

Firmware

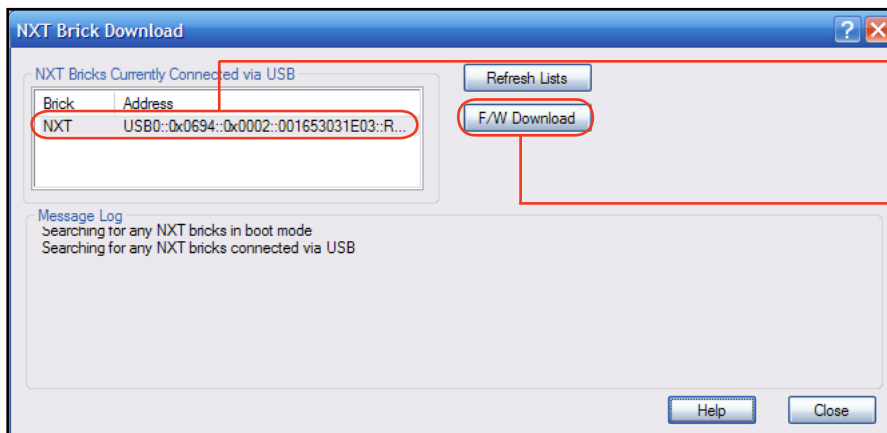
3. Go to the “Robot” menu, then select “Download Firmware”.



3. Download Firmware

Select Robot > Download Firmware to open up the NXT Brick Download menu.

4. The NXT Brick Download menu will appear. In the white box in the upper left, you will see your NXT’s name and device address. Make sure the line for your NXT is selected, then click on the “F/W Download” button.



4a. Select NXT

Select your NXT in the window. Normally, there will only be one listed.

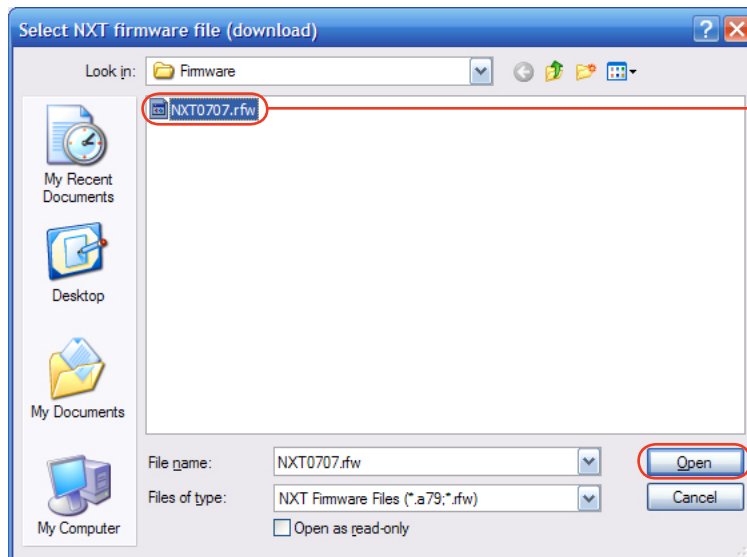
4b. Select F/W Download

Select “F/W Download” to download the firmware to your NXT brick.

Setup

Firmware

5. A list of available firmware files will appear. If there is only one firmware file listed, select it. If there is more than one, choose the firmware file (.rfl) with the highest number. Click "Open" to begin downloading the firmware.



5a. Select the (.rfl) file

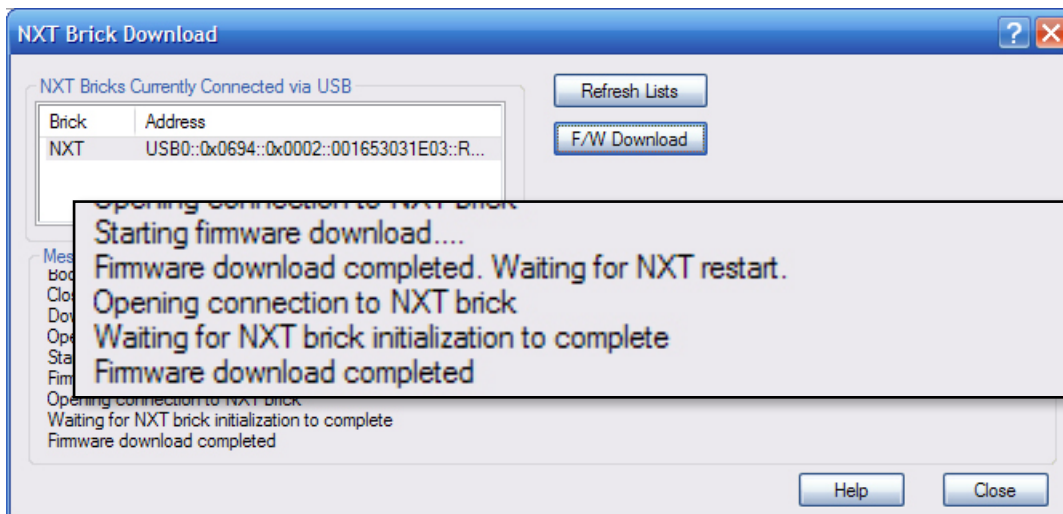
Select the firmware file to download to your robot. If more than one is shown, select the one with the highest number.

5b. Select Open

Once you have selected the file, click "Open" to begin downloading the firmware.

End of Section

The message log will show the progress of the firmware download. Your robot will appear to turn off while the firmware is being loaded. When the process is complete, you will see a line at the end of the log stating, "Firmware download completed".



Setup

Download Firmware Quiz

NAME _____ DATE _____

1. Mark each of the following statements as either 'T' for True or 'F' for False.

- _____ The firmware must be downloaded every time you wish to run a program on the NXT.
- _____ Without a firmware loaded, your robot cannot run any programs.
- _____ Once the ROBOTC firmware is loaded, you will be able to run both ROBOTC and normal NXT language programs.
- _____ All firmwares are identical, so as long as one is loaded, you can run any program.
- _____ Firmware and programs are the same thing.
- _____ You can download the firmware in ROBOTC by using the Robot menu command "Compile and Download".
- _____ Without a firmware loaded, your robot cannot run any programs.

Setup

Download Program

Your robot is ready to go! All that's left is for you to tell it what to do by sending it a program. A program is a set of commands that tell the robot what to do and how to react to its environment. Once written, a program must be transferred ("downloaded") to the robot before it can be run.

This is the program you will download onto the NXT.

```

1  task main()
2  {
3
4  motor[motorC] = 100;
5  wait1Msec(3000);
6
7  }

```

1. Normally, you would type this program directly into ROBOTC. For your convenience, however, there is an already-completed copy provided in the Sample Programs folder. Follow the steps below to open this program.

1a. Open Sample Program
Select File > Open Sample Program to find the saved program.

1b. Select Training Samples
Open the Training Samples folder to find the "MotorC Forward" saved program.

1c. Select the program
Select the "MotorC Forward" program from the Training Samples Folder.

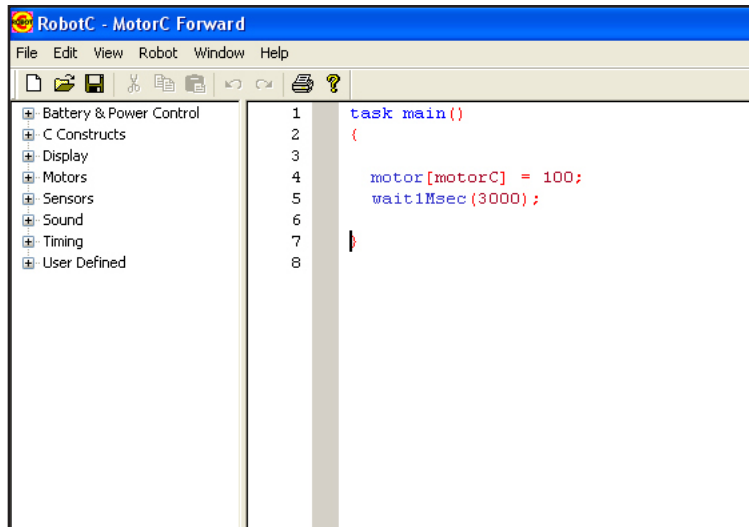
1d. Open the program
Press "Open" to open the saved MotorC Forward program.

Setup

Download Program

Checkpoint

The program should appear in the right-hand pane of the window.

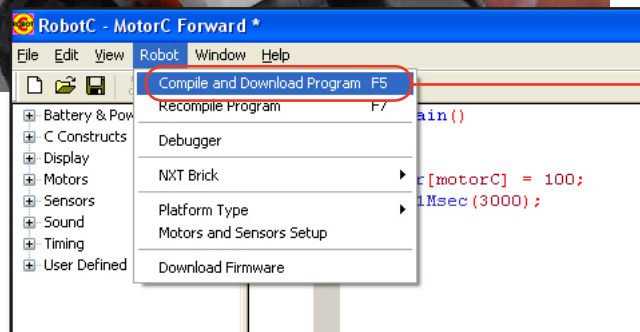


- Download the program to the robot by first turning it on, then using the "Compile and Download" command from the "Robot" menu.



2a. Turn NXT on

Press the orange square on your NXT brick if it is not already on.



2b. Compile and Download

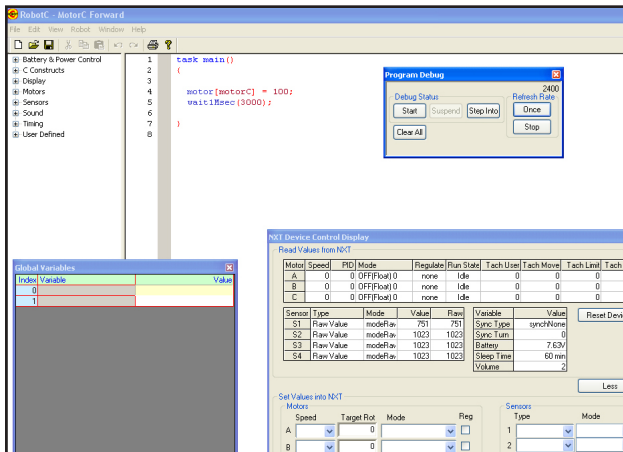
Select Robot > Compile and Download Program to download the MotorC Forward program.

Setup

Download Program

Checkpoint

Several new windows should appear. If you get an error, make sure that the robot is turned on and plugged in to the computer with the USB cable, then try again.

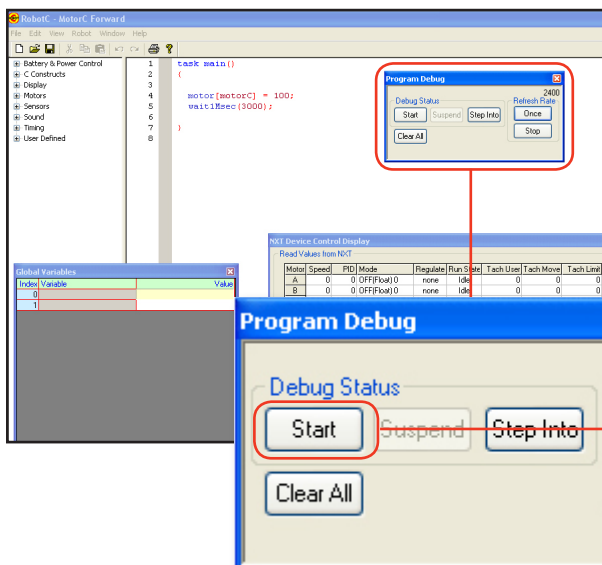


- Place the robot on an open area on the floor or table. In the Program Debug window, press the button labeled "Start". The ROBOTC debug windows appear when the download is complete.



3a. Place robot

Place the robot in an open area, on the floor or table, with the USB cord connected.



3b. Select "Start"

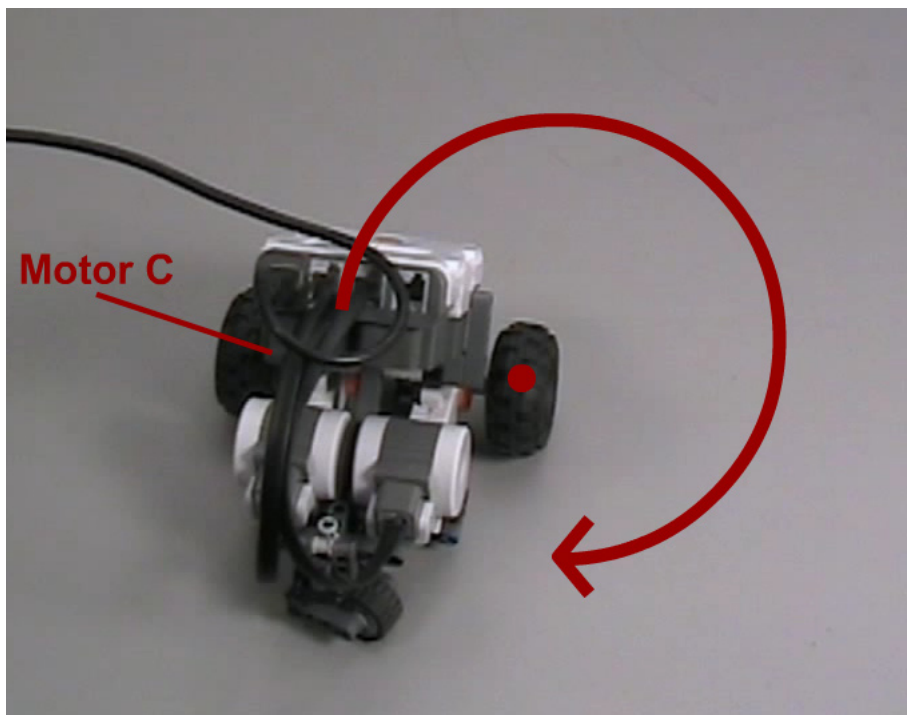
Select the "Start" button to run the MotorC Forward program.

Setup

Download Program

Checkpoint

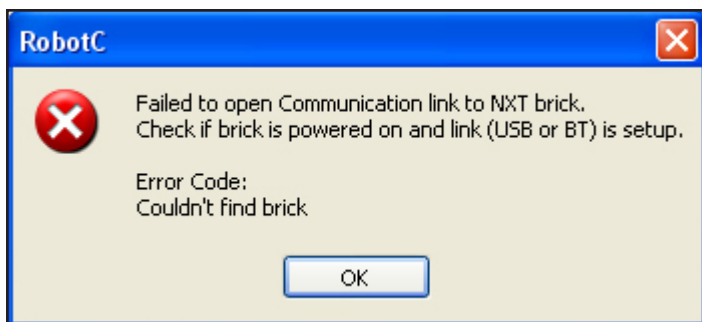
The program we just downloaded told the robot to run one of the motors for three seconds. This causes the robot to move in a circle or perform a pivot turn.



Setup

Download Program

If you get an error, make sure that the robot is turned on and plugged in to the computer with the USB cable, then try again.



Setup

Download Program

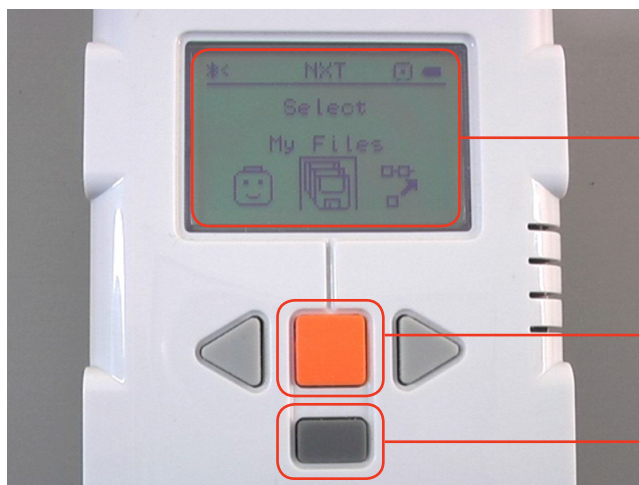
End of Section.

The program must be loaded onto the robot while it is plugged in to the PC, but it can run either attached, or unattached.

To run it unattached, first unplug the USB cable.



Make sure your NXT is on, and take a look at your robot's screen. You should be seeing the main menu, and "My Files" should be displayed. Press the orange button.



Go to the Main menu

Highlight "My Files".

Press the orange button

Press the orange button to go into the "My Files" menu.

Return to the Main menu

Pressing the dark gray button a few times will take you back to the Main menu.

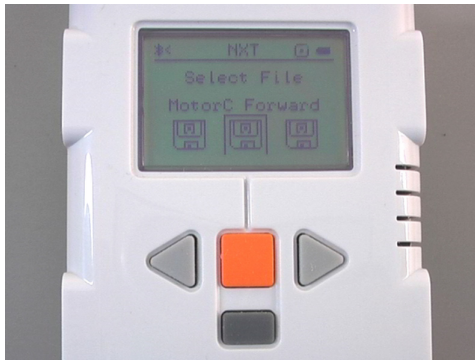
Setup

Download Program



Select "Software Files"

Press the orange button again to go into the "Software Files" menu.



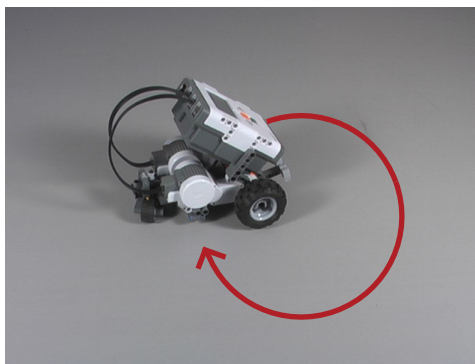
Select your program

Navigate to your program using the right and left arrow buttons. When you find the name of your program, press the orange button.



Run the program

Press the orange button one more time to run the program.



Observe the robot

The robot should now move in a circle.

Setup

Download Sample Quiz

NAME _____ DATE _____

1. Number the following steps in the order that you need to do them in order to successfully run a program. Put an 'X' next to any steps that are not a necessary part of the process.

- _____ Write or open an existing program file.
- _____ Press the dark grey button on the NXT.
- _____ Say clearly to the robot, "Run Program."
- _____ Check that the robot is plugged in and turned on.
- _____ Navigate to the Try Me menu using the NXT's LCD screen and buttons.
- _____ Press the Start button on the Program Debug window.
- _____ Open the Robot menu and select Compile and Download.

Fundamentals

Thinking About Programming **Programmer & Machine**

In this lesson, you will learn about the roles of the programmer and the robot, and how the two need to work together in order to accomplish their goals.

Robots are made to perform useful tasks. Each one is designed to solve a specific problem, in a specific way.



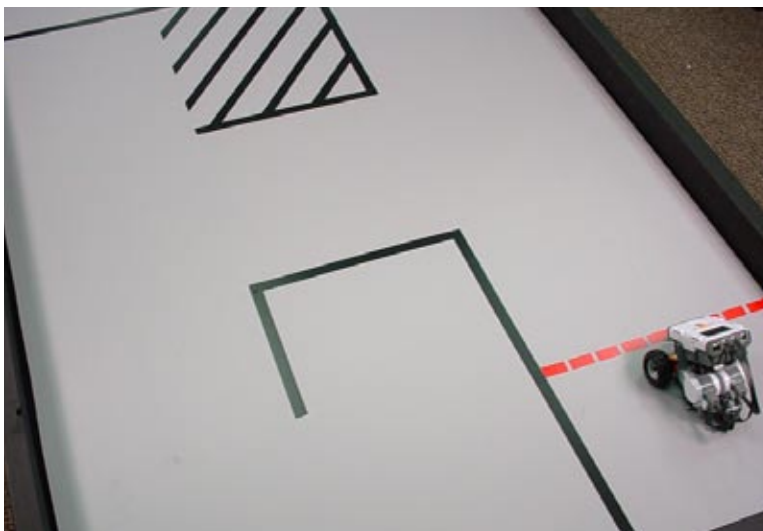
Robotic Tractor

Problem:

Drive safely through a field which may contain obstacles

Solution:

Move towards the destination, making small detours if any obstacles are detected



Labyrinth Robot

Problem:

Get through the maze

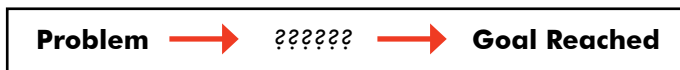
Solution:

Move along a predetermined path in timed segments

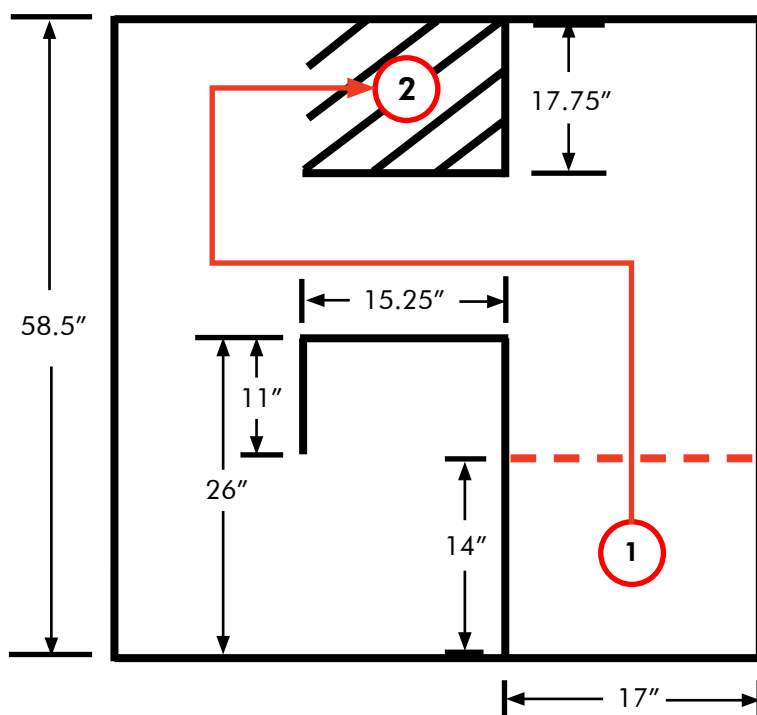
Fundamentals

Thinking about Programming **Programmer & Machine** (cont.)

Let's take a closer look at this last robot. How does it do that? How does it know to do that?



Creating a successful robot takes a team effort between humans and machines.

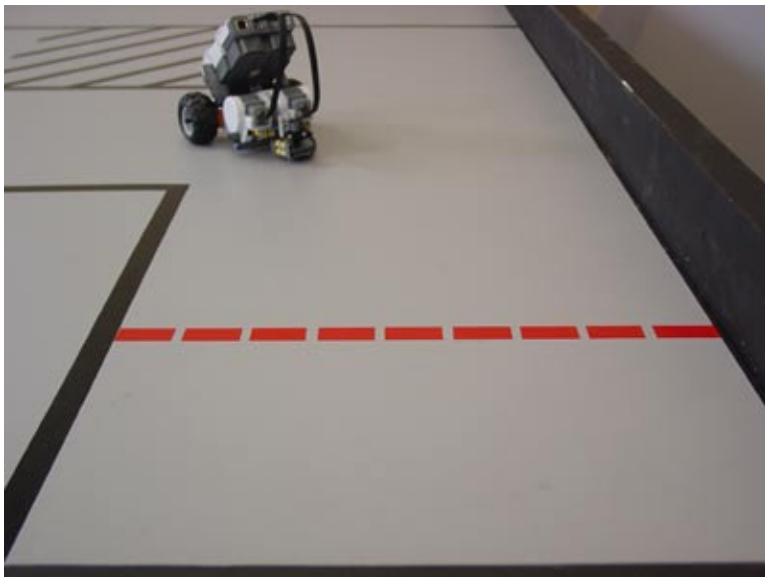


Role of the Programmer

The human is responsible for identifying the task, planning out a solution, and then explaining to the robot what it needs to do to reach the goal.

Fundamentals

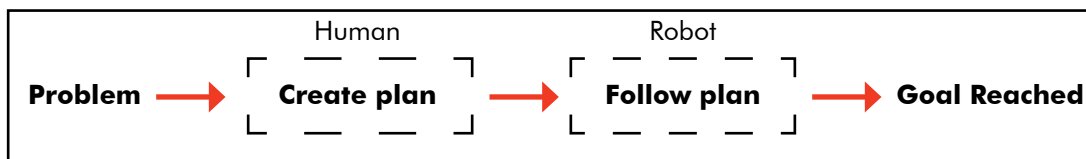
Thinking about Programming **Programmer & Machine** (cont.)



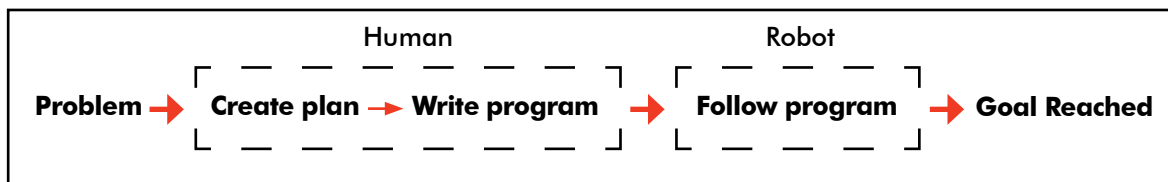
Role of the Robot

The machine is responsible for following the instructions it is given, and thereby carrying out the plan.

The human and the robot can accomplish the task together by dividing up the responsibilities. The human programmer must come up with the plan and communicate it to the robot, and the robot must follow the plan.



Because humans and machines don't normally speak the same language, a special language must be used to translate the necessary instructions from human to robot. There are many such languages, with ROBOTC being one of them. These human-to-robot languages are called "programming" languages, and instructions written in them are called "programs".



Fundamentals

Thinking about Programming **Programmer & Machine** (cont.)

End of Section

The human who writes the program is called the programmer. The programmer's job, therefore, is to identify the problem that the robot must solve, to create a plan to solve it, and to turn that plan into a program that the robot can understand. The robot will then run the program, and perform the task.

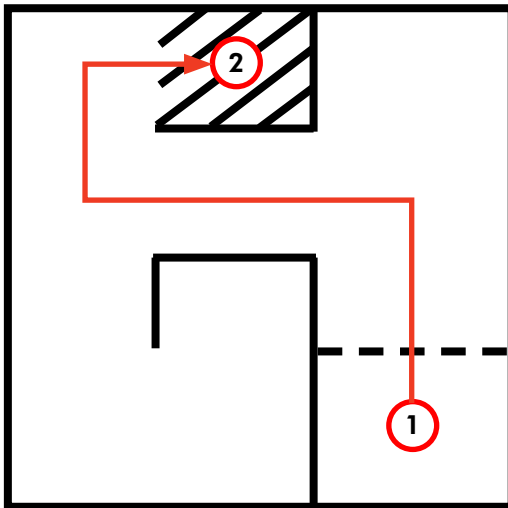
Finally, take note: the robot only follows the program, it does not think for itself. Just as it can be no *stronger* than it is built, the robot can be no *smarter* than the program that the human programmer gave it. You, as programmer, will be responsible for planning and describing to the robot exactly what it needs to do to accomplish its task.

Fundamentals

Thinking About Programming Planning & Behaviors

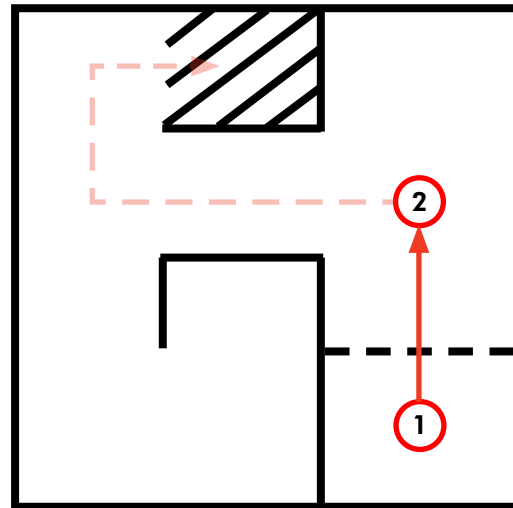
In this lesson, you will learn how thinking in terms of “behaviors” can help you to see the logic behind your robot’s actions, and break a big plan down into practical parts.

“Behaviors” are a very convenient way to talk about what the robot is doing, and what it must do. Moving forward, stopping, turning, looking for an obstacle... these are all behaviors.



Complex Behavior

Some behaviors are big, like “solve the maze.”



Basic or Simple Behavior

Some behaviors are small, like “go forward for 3 seconds.” Big behaviors are actually made up of these smaller ones.

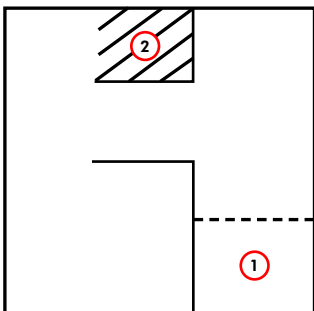
As you begin the task of programming, you should also begin thinking about the robot’s actions in terms of behaviors. Recall that as programmer, your primary responsibilities are:

- **First**, to formulate a plan for the robot to reach the goal,
- And **then**, to translate that plan into a program that the robot can follow.

The plan will simply be *the sequence of behaviors that the robot needs to follow*, and the program will just be those behaviors translated into the programming language.

Fundamentals

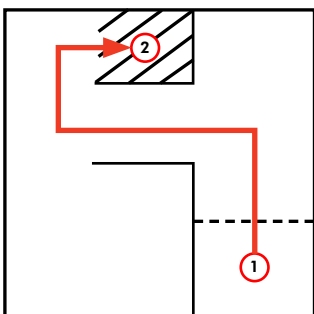
Thinking about Programming **Planning & Behaviors** (cont.)



1. Examine problem

To find a solution, start by examining the problem.

Here, the problem is to get from the starting point (1) to the goal (2).



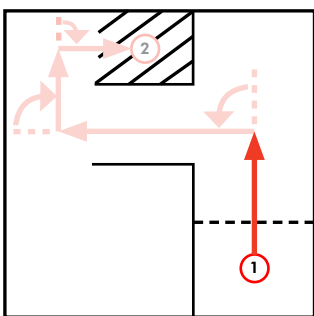
Follow the path to reach the goal

2. Broad solution

Try to see what the robot needs to do, at a high level, to accomplish the goal.

Having the robot follow the path shown on the left, for example, would solve the problem.

You've just identified the first behavior you need! Write it down.



Follow the path to reach the goal

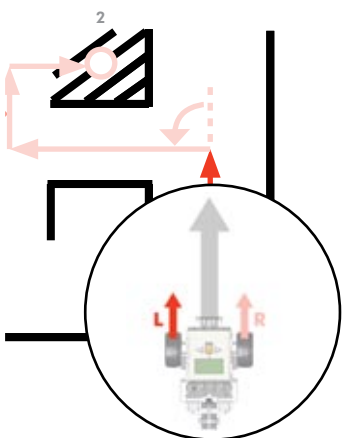
Go forward 3 seconds
Turn left 90°
Go forward 5 seconds
Turn right 90°
Go forward 2 seconds
Turn right 90°
Go forward 2 seconds

3. Break solution into smaller behaviors

Now, start trying to break that behavior down into smaller parts.

Following this path involves moving forward, then turning, then moving forward for a different distance, then turning the other way, and so on. Each of these smaller actions is also a behavior.

Write them down as well, taking care to keep them in the correct sequence.



Go forward for 3 seconds

Turn on left motor
Turn on right motor
Wait 3 seconds
Turn off left motor
Turn off right motor

4. Break into even smaller pieces

If you then break down these behaviors into even smaller pieces, you'll get smaller and smaller behaviors, with more and more detail. Keep track of them as you go.

Eventually, you'll reach commands that you can express directly in the programming language.

For example, ROBOTC has a command to turn on one motor. When you reach a behavior that says to turn on one motor, you can stop breaking it down, because it's now ready to translate.

Fundamentals

Thinking about Programming **Planning & Behaviors** (cont.)

Large behavior

Follow the path to reach the goal

Go forward 3 seconds
 Turn left 90°
 Go forward 5 seconds
 Turn right 90°
 Go forward 2 seconds
 Turn right 90°
 Go forward 2 seconds

Smaller behaviors

Go forward for 3 seconds

Turn on left motor
 Turn on right motor
 Wait 3 seconds
 Turn off left motor
 Turn off right motor

Turn left 90°

Reverse left motor
 Turn on right motor
 Wait 0.8 seconds
 Turn off left motor
 Turn off right motor

Go forward for 5 seconds

Turn on left motor
 Turn on right motor
 Wait 5 seconds

ROBOTC-ready behaviors

1. Turn on left motor
 2. Turn on right motor
 3. Wait 3 seconds
 4. Turn off left motor
 5. Turn off right motor
-
6. Reverse left motor
 7. Turn on right motor
 8. Wait 0.8 seconds
 9. Turn off left motor
 10. Turn off right motor
-
11. Turn on left motor
 12. Turn on right motor
 13. Wait 5 seconds
 - ...

Step by step

1. Start with a large-scale behavior that solves the problem.
2. Break it down into smaller pieces. Then break the smaller pieces down as well.
3. Repeat until you have behaviors that are small enough for ROBOTC to understand.

When all the pieces have reached a level of detail that ROBOTC can work with – like the ones in the **“ROBOTC-ready behaviors”** list above – take a look at the list you’ve made. These behaviors, in the order and way that you’ve specified them, represent the plan that the robot needs to follow in order to accomplish the goal.

Because the steps are still written in English, they should be relatively easy to understand for the human programmer.

As the programmer becomes more experienced, the organization of the behaviors in English will start to include important techniques from the programming language itself, like if-else statements and loops. This hybrid language, halfway between English and the programming language, is called **pseudocode**, and is an important tool in helping to keep larger programs understandable.

1. Turn on left motor
2. Turn on right motor
3. Wait 3 seconds
4. Turn off left motor
5. Turn off right motor

Simple pseudocode

Your list of behaviors to perform in a specific order are a simple form of pseudocode.

```

if (the light sensor sees light)
{
    turn on left motor
    hold right motor still
}
    
```

Later pseudocode

As your programming skills grow, your pseudocode will include more complex logic, but will still serve the same purpose: to help you find and express the necessary robot behaviors in simple English.

Fundamentals

Thinking about Programming **Planning & Behaviors** (cont.)

End of Section

By starting with a very large solution behavior, and breaking it down into smaller and smaller sub-behaviors, you have a logical way to figure out what the robot needs to do in order to accomplish its task.

By recording the behaviors in English, you have taken the first steps toward good pseudocoding practice, allowing you to easily review the behaviors and their organization as you prepare to translate them to program code.

The only step remaining is to translate your behaviors from English pseudocode to ROBOTC programming language.

Fundamentals

Thinking about Programming Quiz

NAME _____ DATE _____

1. What is the role of the programmer?

2. Break the complex behavior "get ready for school in the morning" into at least five smaller behaviors, and list them below.

Fundamentals

Programming in ROBOTC **ROBOTC Rules**

In this lesson, you will learn the basic rules for writing ROBOTC programs.

ROBOTC is a text-based programming language based on the standard C programming language.

Commands to the robot are written as text on the screen, processed by the ROBOTC compiler into a machine language file, and then loaded onto the robot, where they can be run. Text written as part of a program is called "code".

```

1 task main()
2 {
3
4     motor[motorC] = 100;
5     wait1Msec(3000);
6
7 }
```

Program Code

Text written as part of a program is called "code".

You type code just like normal text, but you must keep in mind that capitalization is important to the computer. Replacing a lowercase letter with a capital letter or a capital letter with lowercase, will cause the robot to become confused.

```

1 Task main()
2 {
3
4     motor[motorC] = 100;
5     wait1Msec(3000);
6
7 }
```

Capitalization

Capitalization (paying attention to UPPERCASE vs. lowercase) is important in ROBOTC.

Capitalizing the 'T' in task causes ROBOTC to no longer recognize this command.

As you type, ROBOTC will try to help you out by coloring the words it recognizes. If a word appears in a different color, it means ROBOTC knows it as an important word in the programming language.

```

1 task main()
2 {
3
4     motor[motorC] = 100;
5     wait1Msec(3000);
6
7 }
```

Code coloring

ROBOTC automatically colors key words that it recognizes.

Compare this correctly-capitalized "task" command with the incorrectly-capitalized version in the previous example. The correct one is recognized as a command and turns blue.

Fundamentals

Programming in ROBOTC **ROBOTC Rules** (cont.)

And now, we will look at some of the important parts of the program code itself.

The most basic kind of statement in ROBOTC simply gives a command to the robot. The `motor[motorC];` statement in the sample program you downloaded is a simple command. It instructs the motor plugged into the Motor C port to turn on at 100% power.

| | |
|--|--|
| <pre> 1 task main() 2 { 3 4 motor[motorC] = 0; 5 wait1Msec(3000); 6 7 }</pre> | <p>Simple statement A straightforward command to the robot. This statement tells the robot to turn on the motor attached to motor port C at 100% power.</p> <p>Simple statement (2) This is also a simple statement. It tells the robot to wait for 3000 milliseconds (3 seconds).</p> |
|--|--|

Statements are run in order, as quickly as the robot is able to reach them. Running this program on the robot turns the motor on, then waits for 3000 milliseconds (3 seconds) with the motor still running, and then ends.

| | |
|--|---|
| <pre> 1 task main() 2 { 3 4 1st motor[motorC] = 0; 5 2nd wait1Msec(3000); 6 7 } End</pre> | <p>Sequence Statements run in English reading order (left-to-right, top-to-bottom). As soon as one command is complete, the next runs. These two statements cause the motors to turn on (<i>1st command</i>), and then the robot immediately begins a three second wait (<i>2nd command</i>) while the motors remain on.</p> <p>End When the program runs out of statements and reaches the } symbol in task main, all motors stop, and the program ends.</p> |
|--|---|

Fundamentals

Programming in ROBOTC **ROBOTC Rules** (cont.)

How did ROBOTC know that these were two separate commands?

Was it because they appeared on two different lines?

No. Spaces and line breaks in ROBOTC are only used to separate words from each other in multi-word commands. Spaces, tabs, and lines don't affect the way a program is interpreted by the machine.

```

1 task main()
2 {
3     motor[motorC] = 0;
4     wait1Msec(3000);
5 }
6
7
```

Whitespace

Spaces, tabs, and line breaks are generally unimportant to ROBOTC and the robot.

They are sometimes needed to separate words in multi-word commands, but are otherwise ignored by the machine.

So why ARE they on separate lines? For the programmer. Programming languages are designed for humans and machines to communicate. Using spaces, tabs, and lines helps the human programmer to read the code more easily. Making good use of spacing in your program is a very good habit for your own sake.

```

1 task main() {motor[motorC
2 ]=0;wait1Msec(3000);}
```

No Whitespace

To ROBOTC, this program is the same as the last one. To the human programmer, however, this is close to gibberish.

Whitespace is used to help programs be readable to humans.

But what about ROBOTC? How DID it know where one statement ended and the other began? It knew because of the semicolon at the end of each line. Every statement ends with a semicolon. It's like the period at the end of a sentence.

```

1 task main()
2 {
3     motor[motorC] = 0;
4     wait1Msec(3000);
5 }
6
7
```

Semicolons

Like periods in an English sentence, semicolons mark the end of every ROBOTC statement.

Checkpoint

Statements are commands to the robot. Each statement ends in a semicolon so that ROBOTC can identify it, but each is also usually written on its own line to make it easier for humans to read. Statements are run in "reading" order, left to right, top to bottom, and each statement is run as soon as the previous one is complete. When there are no more statements, the program will end.

Fundamentals

Programming in ROBOTC **ROBOTC Rules** (cont.)

ROBOTC uses far more punctuation than English. Punctuation in programming languages is usually used to separate important areas of code from each other. Most ROBOTC punctuation comes in pairs.

Punctuation pairs, like the parentheses and square brackets in these two statements, are used to mark off special areas of code. Every punctuation pair consists of an **“opening”** punctuation mark and a **“closing”** punctuation mark. The punctuation pair designates the area **between them** as having special meaning to the command that they are part of.

```

1 task main()
2 {
3
4     motor[motorC] = 100;
5     wait1Msec(3000);
6
7 }
```

Punctuation pair: Square brackets []

The code written between the square brackets of the motor command indicate what motor the command should use.

```

1 task main()
2 {
3
4     motor[motorC] = 100;
5     wait1Msec(3000);
6
7 }
```

Punctuation pair: Parentheses ()

The code written between the parentheses of the wait1Msec command tell it how many milliseconds to wait.

Checkpoint

Paired punctuation marks are always used together, and surround specific important parts of a statement to set them apart.

Different commands make use of different punctuation. The motor command uses square brackets and the wait1Msec command uses parentheses. This is just the way the commands are set up, and you will have to remember to use the right punctuation with the right commands.

Fundamentals

Programming in ROBOTC **ROBOTC Rules** (cont.)

Simple statements do the work in ROBOTC, but Control Structures do the thinking.

These are pieces of code that control the flow of the program's commands, rather than issue direct orders to the robot.

Simple statements can only run one after another in order, but control statements allow the program to **choose the order that statements are run**. For instance, they may choose between two different groups of statements and only run one of them, or sometimes they might repeat a group of statements over and over.

One important structure is the **task main**. Every ROBOTC program includes a special section called task main. This control structure determines what code the robot will run as part of the main program.

```

1 task main()
2 {
3
4     motor[motorC] = 100;
5     wait1Msec(3000);
6
7 }
```

Control structure: task main

The control structure "task main" directs the program to the main body of the code. When you press "Start" or "Run" on the robot, the program immediately goes to task main and runs its code.

The left and right curly braces { } belong to the task main structure. They surround the commands which will be run in the program.

```

while (SensorValue(touchSensor) == 0)
{
    motor[motorC] = 100;
    motor[motorB] = 100;
}
```

Control structure preview

Another control structure, the while loop, repeats the code between its curly braces { } as long as certain conditions are met.

Normally, statements run only once, but with a while loop, they can be told to repeat over and over for as long as you want!

Checkpoint

Control structures like task main decide which lines of code are run, and when. They control the "flow" of your program, and are vital to your robot's ability to make decisions and respond intelligently to its environment.

Fundamentals

Programming in ROBOTC **ROBOTC Rules** (cont.)

Programming languages are meant to be readable by both humans and machines.

Sometimes, the programmer needs to leave a note for human readers to help understand what the code is doing. For this, ROBOTC allows “comments” to be made.

Comments are text that the computer will ignore. A comment can therefore contain notes, messages, and symbols that may help a human, but would be meaningless to the computer. ROBOTC will simply skip over them. Comments appear in green in ROBOTC.

```

1 // Move motor C forward with 100% power
2
3 task main()
4 {
5
6     /*
7      Motor C forward with 100% power
8      Do this for 3 seconds
9     */
10
11     motor[motorC] = 100;
12     wait1Msec(3000);
13
14 }
```

Comments: // Single line

Any section of text that follows a //double slash on a line, is considered a comment, although code to the left of the // is still considered normal.

Comments: /* Any length */

A comment can be created in ROBOTC using another type of paired punctuation, which starts with /* and ends with */

This type of comment can span multiple lines, so be sure to include both the opening and closing marks!

End of Section

What you have just seen are some of the primary features of the ROBOTC language. **Code** is entered as text, which builds **statements**. Statements are used to issue commands to the robots. **Control structures** decide which statements to run at what times. **Punctuation**, both single like **semicolons** and **paired like parentheses**, are used to set apart important parts of commands.

A number of features in ROBOTC code are designed to help the human, rather than the computer. **Comments** let programmers leave notes for themselves and others, and **whitespace** like tabs and spaces helps to keep your code organized and readable.

Fundamentals

ROBOTC Programming Quiz

NAME _____ DATE _____

1. What punctuation mark signals the end of a simple statement?

2. Give an example of paired punctuation.

1. Control structures such as task main or if-else:
- a. Issue direct commands to the robot's motors
 - b. Are only there for the human programmer's benefit, and are ignored by the robot
 - c. Control the "flow" of commands: they choose which commands to run and when
 - d. Are a form of paired punctuation

Movement

Labyrinth Challenge

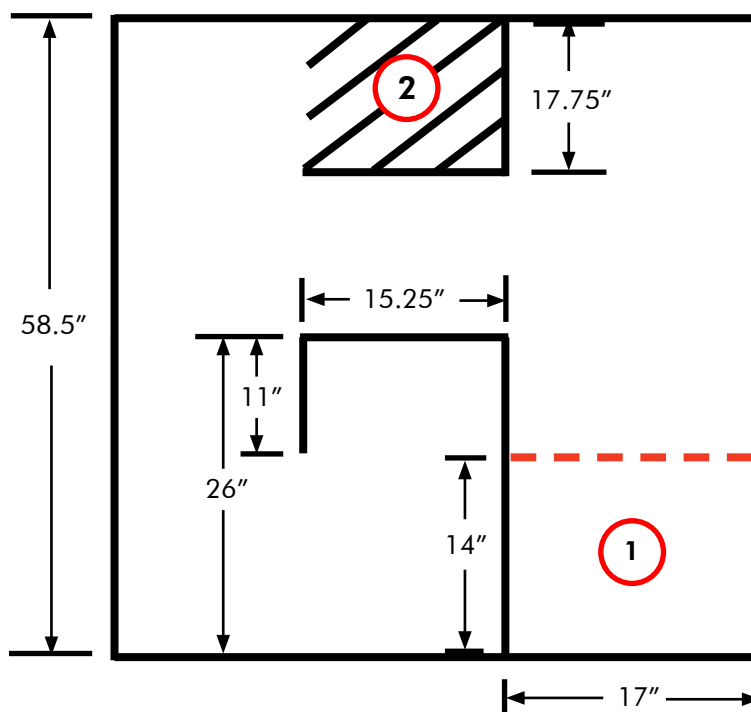
Challenge Description

This challenge features a sequence of turns that the robot must perform in order to get to the "end" of the Labyrinth. The robot must first begin at the starting point, and get to the goal area by completing turning and forward movement behaviors. The robot must NOT cross any lines.

Materials Needed

- Black electrical tape
- Red electrical tape
- Scissors (or cutting tool)
- Ruler (or straight edge)

Board Specifications



- 1 Robot must begin here, and then maneuver the robot to get to the goal area.
- 2 Robot must reach this goal area without crossing any black lines (Goal area lines).

Movement

Moving Forward Code Dissection

Now that you understand the steps needed to download a program, which you learned through downloading a sample program, let's take a step back and figure out what all of this "code" is really doing.

Here on the main screen we have several lines of code. Let's walk through what each of these commands "does".

```

1 task main()
2 {
3
4     motor[motorC] = 100;
5     wait1Msec(3000);
6
7 }

```

task main()
This line creates a task called main, which contains the behaviors that we want the robot to do.

task main() marks the beginning of a structure.

{ body }
{ and } "braces" surround the body of the structure. The lines between the braces tell the program what commands to follow as part of the main task.

As you know, the code currently tells the robot to move in a circle. More literally, it tells the robot to move "Motor C" forward for 3 seconds. Moving only one motor, or wheel, will make your robot go in circles. The details of each command are as follows:

motor[] command

The motor[] command tells the robot to set a motor to run at a given power level. The example below (taken from the program you ran) sets motor C to run at 100% power forward. Note that every command in ROBOTC must end with a semicolon, just as every English statement must end with a period.

Example:

```
motor[motorC] = 100;
```

wait1Msec() command

The command "wait1Msec" tells the robot to wait, for the given time in milliseconds. The number within the parenthesis is the number of milliseconds that you want the robot to wait. 3000 milliseconds is equal to 3 seconds, so the robot moves for 3 seconds.

Example:

```
wait1Msec(3000);
```

Movement

Moving Forward Code Dissection (cont.)

In this lesson, you will modify the existing program code to create a Moving Forward behavior with the robot.

- Before making any changes, save the program with a new name. Go to "File", "Save As", and rename this program to "Labyrinth".

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

1b. Browse to an appropriate folder
Browse to or create a folder (on your desktop, in your documents folder, etc.) that you will use to store your programs.

1c. Rename program
Give this program the new name "Labyrinth".

1d. Save
Click Save.

- Add a new line after the first motor command.

```

1 task main()
2 {
3
4     motor[motorC] = 100;
5      
6     wait1Msec(3000);
7
8 }

```

- 2. Add this space**
This is where we will add the second motor command in the next step.

Movement

Moving Forward **Code Dissection** (cont.)

3. In order to make the robot go forward, you'll want both motor C *and* motor B to run forward. The command `motor[motorC]=100;` made Motor C move at 100% power. Add a command that is exactly the same, but addresses Motor B instead.

```

1  task main()
2  {
3
4      motor[motorC] = 100;
5      motor[motorB] = 100;
6      wait1Msec(3000);
7
8  }
```

3. Add this code

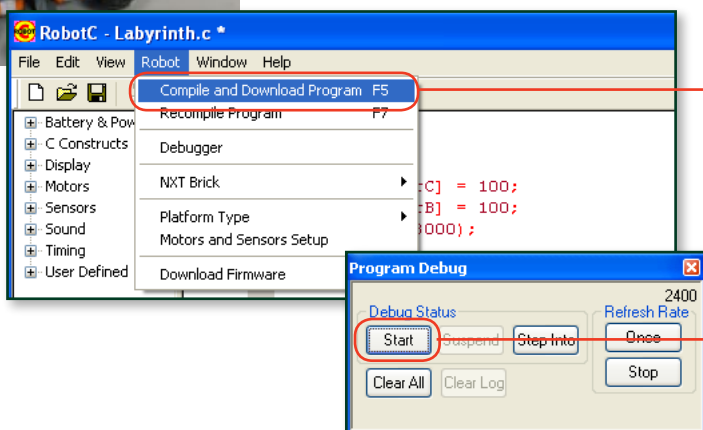
This code is exactly the same as the line above it, except that it is directed at Motor B (right wheel) instead of Motor C (left wheel).

4. Make sure your robot is on and that the robot is plugged in with the USB cable, then go to the menu "Robot" > "Compile and Download".



4a. Check connection

Ensure that your robot is turned on and plugged in to the computer through the USB cable



4b. Compile and Download

Select Robot > Compile and Download Program to send your program to the robot.

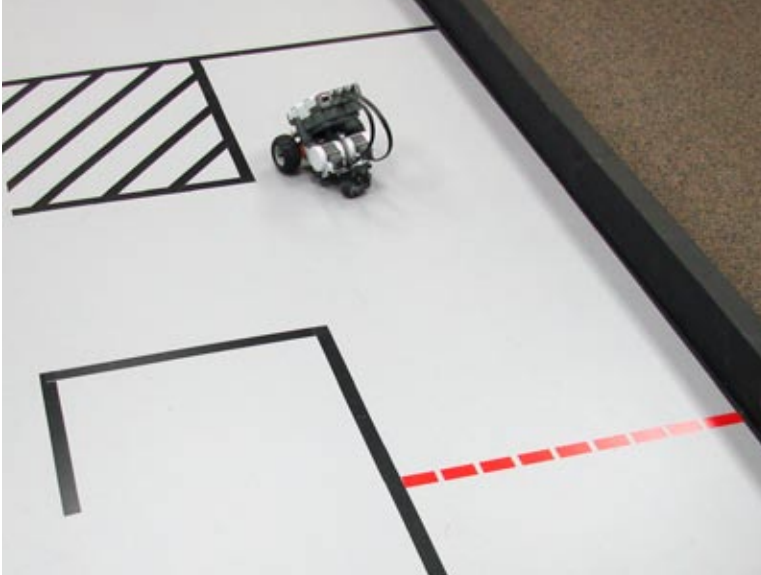
4c. Press Start

Press the Start button on the Program Debug menu that appears, to run the program.

Movement

Moving Forward **Code Dissection** (cont.)

5. Once the program is downloaded, you can either unplug the bot and navigate to your program to run it, or you can keep it connected to the computer and click on the "Start" button.



End of Section

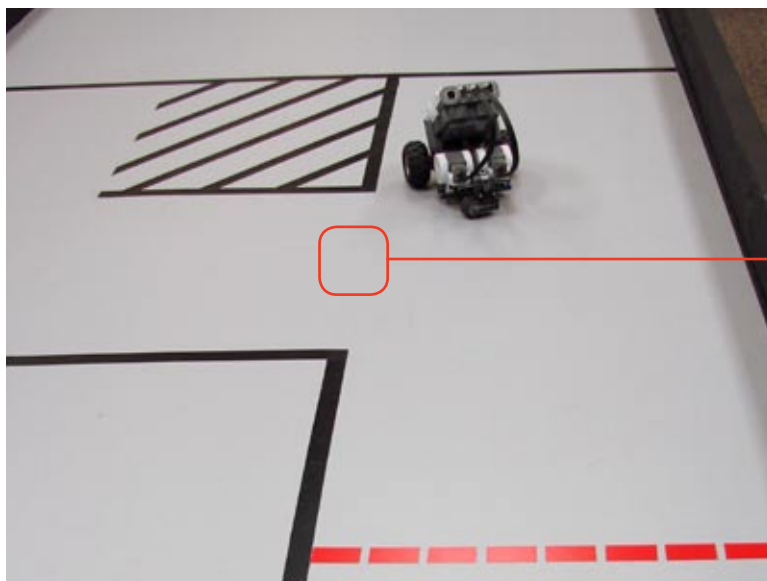
By examining what each line of code in the Sample Program did, we were able to figure out a way to turn on the other motor on the robot as well. Both motors running together created a forward movement. Proceed to the next section to begin experimenting with the other parts of the program.

Movement

Moving Forward Timing Lesson

In this lesson, you will learn how to adjust the time (and consequently, the distance) the robot travels in the Moving Forward behavior.

The robot moves forward for 3 seconds. This is a great start, but the end needs work.



Missed turn

The robot has traveled too far and cannot make the first turn in the maze.

1. Adjust the amount of time the robot lets its motors run, by changing the number value inside the wait1Msec command.

```

1 task main()
2 {
3
4     motor[motorC] = 100;
5     motor[motorB] = 100;
6     wait1Msec(2000);
7
8 }
```

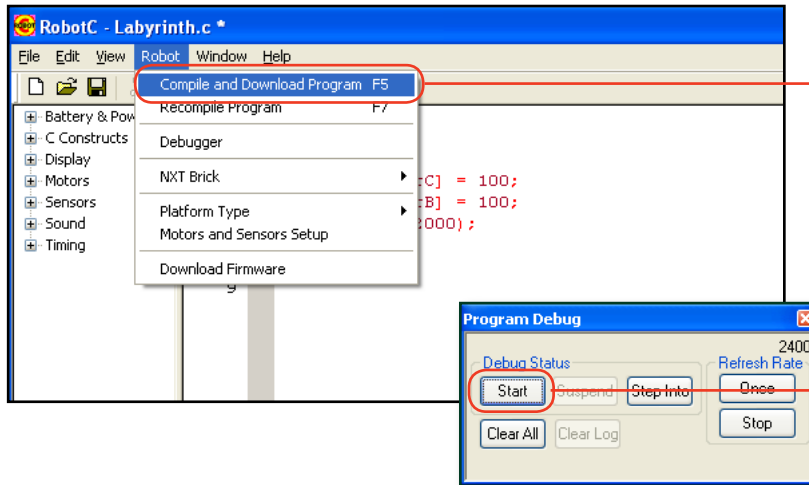
1. Modify this code

Change the 3000 milliseconds in the wait1Msec command to 2000 milliseconds.

Movement

Moving Forward **Timing** (cont.)

2. Compile and Download the program by going to "Robot" > "Compile and Download".



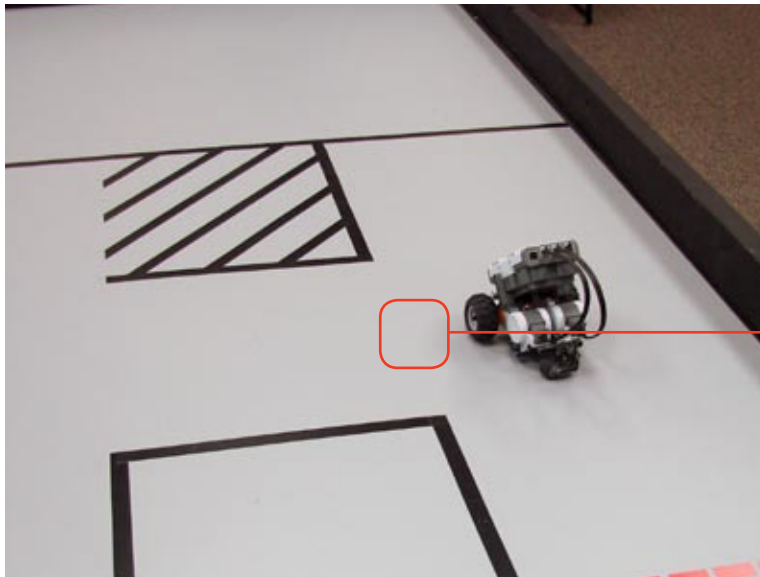
2a. Compile and Download

Select Robot > Compile and Download Program to send your program to the robot.

2b. Press Start

Press the Start button on the Program Debug menu that appears, to run the program.

End of Section. The wait1Msec command controlled how long the robot let its motors run. By shortening the duration from 3000ms to 2000ms, we adjusted the total distance traveled as well.



Ready to turn

The robot stops in a good position to begin its next maneuver, a left turn toward the next part of the path.

Movement

Moving Forward Quiz

NAME _____ DATE _____

1. In the program below, which line or lines control how long the robot will move?

```
1 task main()  
2 {  
3     motor[motorC] = 100;  
4     motor[motorB] = 100;  
5     wait1Msec(2000);  
6 }
```

- a. Line 1
- b. Lines 4 & 5
- c. Line 6
- d. This robot moves forever

2. Look at the code below. Write a second block of code that would cause the robot to move at half the speed, but still move approximately the same distance.

```
1 task main()  
2 {  
3     motor[motorC] = 100;  
4     motor[motorB] = 100;  
5     wait1Msec(2000);  
6  
7  
8  
9  
10  
11  
12  
13 }
```

Movement

Speed and Direction **Motor Power**

In this lesson, you will modify the existing program to make your robot move at a slower speed. This should result in more consistent movement.

Moving at slower speeds can help your robot to be more consistent. All you need to do is alter the motor commands to turn the motors on with a power level lower than 100%.

1. Change the power levels in your motor[] commands to move at half speed.

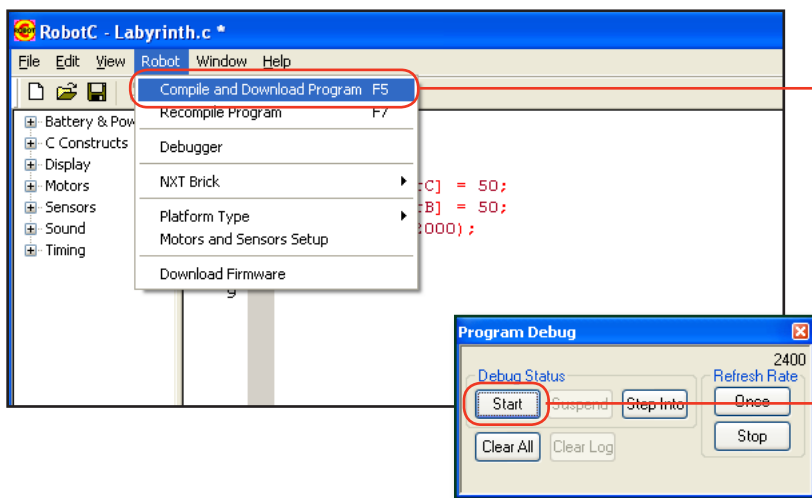
```

1  task main()
2  {
3
4      motor[motorC] = 50;
5      motor[motorB] = 50;
6      wait1Msec(2000);
7
8  }
```

1. Modify this code

Change the old 100 (100% power) to 50 (50% power) to make the robot move at half power. Do this for both motor commands.

2. Download and run your program. Note that downloading automatically saves your program.



2a. Compile and Download

Select Robot > Compile and Download Program to send your program to the robot.

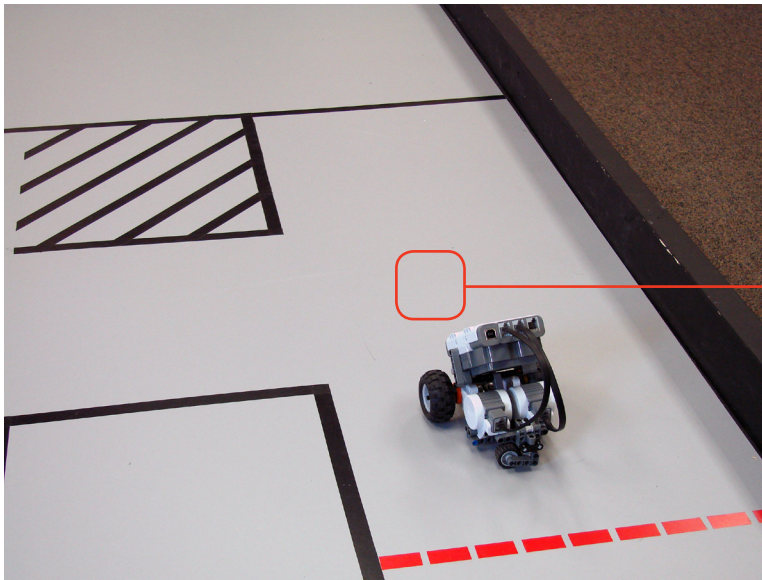
2b. Press Start

Press the Start button on the Program Debug menu that appears, to run the program.

Movement

Speed and Direction **Motor Power** (cont.)

Checkpoint. The numeric value assigned to each motor in the motor[] commands represents the % of power that the motors will run with. So far, we've changed them from full power to half. Since your robot is traveling slower, it will now need to travel longer to go the same distance.



Distance changed

Traveling for the same amount of time, but at a slower pace, causes the robot to stop short of its destination.

3. Since the power has been halved, try doubling the time.

```

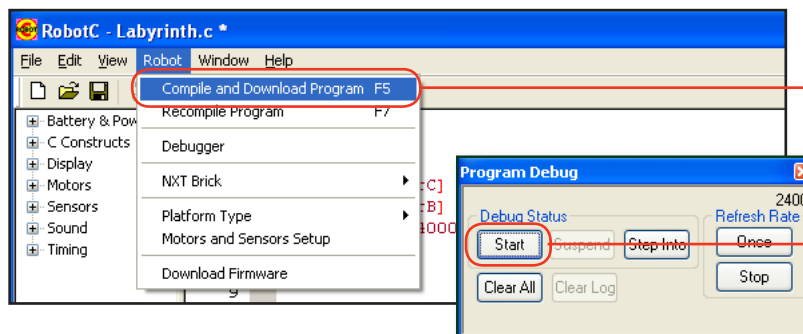
1  task main()
2  {
3
4      motor[motorC] = 50;
5      motor[motorB] = 50;
6      wait1Msec(4000);
7
8  }

```

3. Modify this code

Since the motors are traveling at half power, double the 2000ms duration to 4000ms.

4. Download and run again.



4a. Compile and Download

Select Robot > Compile and Download Program to send your program to the robot.

4b. Press Start

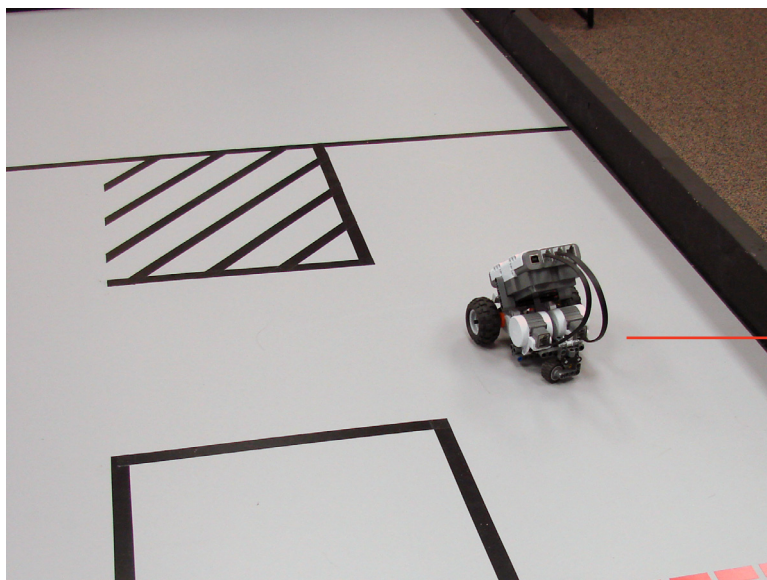
Press the Start button on the Program Debug menu that appears, to run the program.

Movement

Speed and Direction **Motor Power** (cont.)

End of Section

Your robot is traveling approximately the same distance, but at a slower speed than before. Traveling at this speed, the robot is able to maneuver more consistently, and its behaviors are easier to see and identify.



Back again

The robot now travels the correct distance again, but at a slower speed than before.

Movement

Speed and Direction **Turn and Reverse**

In this lesson, you will learn how to make the robot turn and back up using different combinations of motor powers, and how to perform multiple actions in a sequence.

Setting both motors to half power makes the robot go slower. What do other combinations of motor powers do?

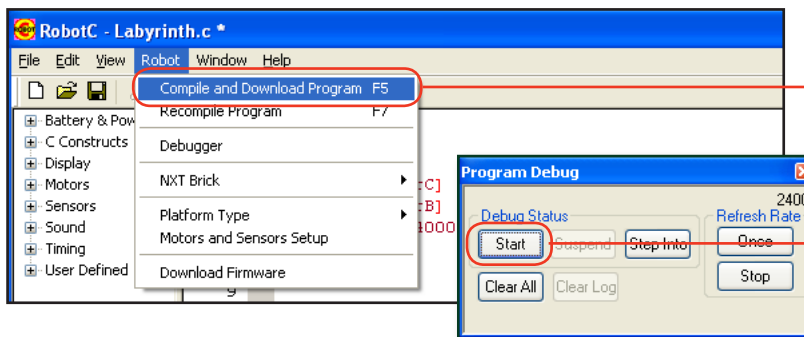
1. Negative numbers make the motor spin in reverse, up to -100% power.

```

1  task main()
2  {
3
4      motor[motorC] = -100;
5      motor[motorB] = -100;
6      wait1Msec(4000);
7
8  }
```

1a. Modify this code

Change both motors to run at -100% power.

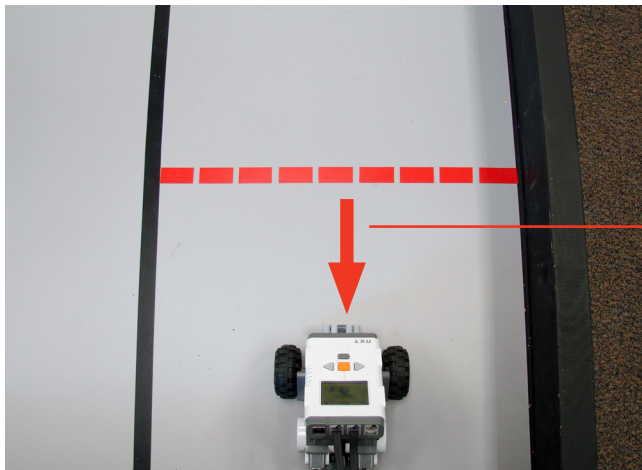


1b. Compile and Download

Select Robot > Compile and Download Program to send your program to the robot.

1c. Press Start

Press the Start button on the Program Debug menu that appears, to run the program.



1d. Move Backward

The robot runs in reverse with both motors set to -100% power.

Movement

Speed and Direction **Turn and Reverse** (cont.)

2. A motor power of 0 makes the robot stop.

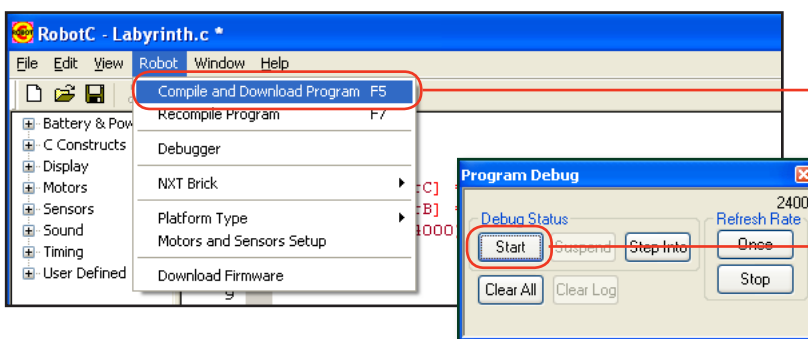
```

1  task main()
2  {
3
4      motor[motorC] = 0;
5      motor[motorB] = 0;
6      wait1Msec(4000);
7
8  }

```

2a. Modify this code

Change both motors to run at 0% power.

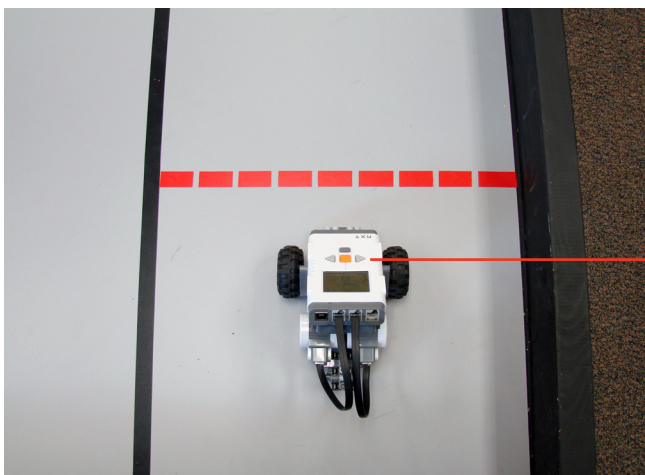


2b. Compile and Download

Select Robot > Compile and Download Program to send your program to the robot.

2c. Press Start

Press the Start button on the Program Debug menu that appears, to run the program.



2d. Braking

The robot holds its position and applies braking with both motors set to 0% power. Try pushing the robot while the program is running.

Movement

Speed and Direction **Turn and Reverse** (cont.)

3. Giving different powers to the two motors causes the robot to turn in various ways. Giving them opposite powers causes the robot to turn in place.

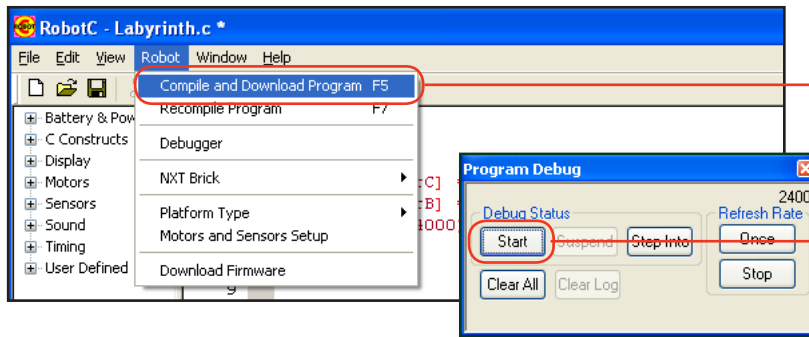
```

1  task main()
2  {
3
4      motor[motorC] = 100;
5      motor[motorB] = -100;
6      wait1Msec(4000);
7
8  }

```

3a. Modify this code

Change the motors to run at 100% power in opposite directions.

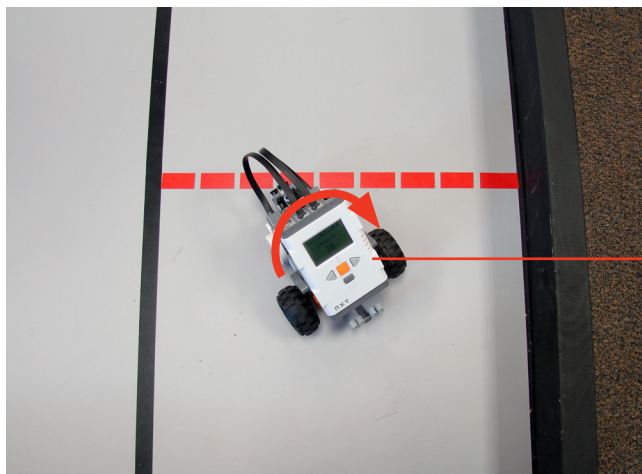


3b. Compile and Download

Select Robot > Compile and Download Program to send your program to the robot.

3c. Press Start

Press the Start button on the Program Debug menu that appears, to run the program.



3d. Point Turn Right

Making the left wheel go forward while the right wheel goes backward causes a "point turn" in place to the right.

Movement

Speed and Direction **Turn and Reverse** (cont.)

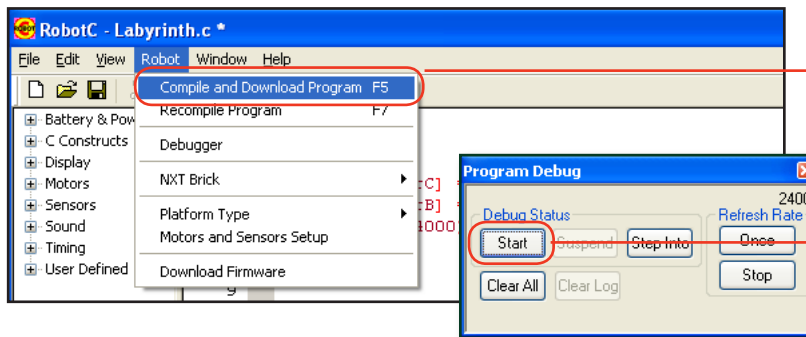
4. Making one wheel move while the other remains stationary causes the robot to “swing turn” with the stationary wheel acting as a pivot.

```

1  task main()
2  {
3
4      motor[motorC] = 100;
5      motor[motorB] = 0;
6      wait1Msec(4000);
7
8  }

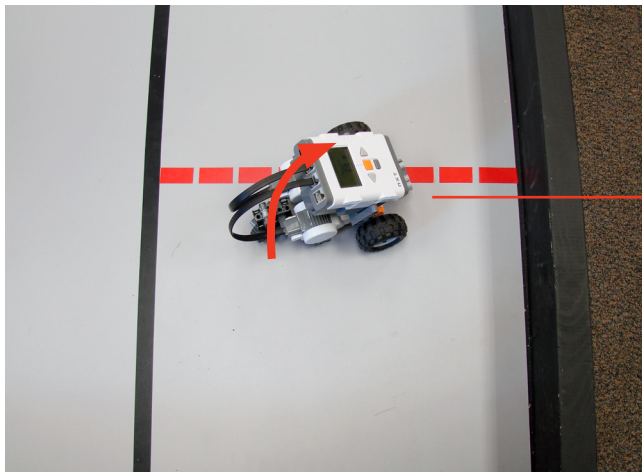
```

4a. Modify this code
Make one wheel move while the other holds its position.



4b. Compile and Download
Select Robot > Compile and Download Program to send your program to the robot.

4c. Press Start
Press the Start button on the Program Debug menu that appears, to run the program.



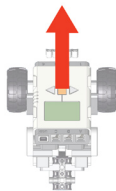
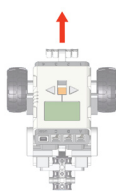
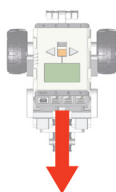
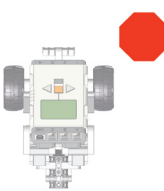
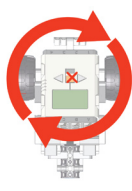
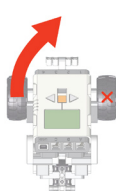
4d. Swing Turn Right
Making the left wheel go forward while holding the right wheel stationary causes a “swing turn” around the stationary wheel.

Movement

Speed and Direction **Turn and Reverse** (cont.)

Checkpoint

The following table shows the different types of movement that result from various combinations of motor powers. Remember, these commands only set the motor powers. A wait1Msec command is still needed to tell the robot how long to let them run.

| Motor commands | Resulting movement |
|--|---|
| <pre>motor[motorC]=100; motor[motorB]=100;</pre> |  |
| <pre>motor[motorC]=50; motor[motorB]=50;</pre> |  |
| <pre>motor[motorC]=-100; motor[motorB]=-100;</pre> |  |
| <pre>motor[motorC]=0; motor[motorB]=0;</pre> |  |
| <pre>motor[motorC]=100; motor[motorB]=-100;</pre> |  |
| <pre>motor[motorC]=100; motor[motorB]=0;</pre> |  |

Movement

Speed and Direction **Turn and Reverse** (cont.)

6. Finally, the robot will need to be able to perform multiple actions in a sequence. Commands in ROBOTC are run in order from top to bottom, so to have the robot perform one behavior after another, simply add the second one below the first in the code.

```

1  task main()
2  {
3
4      motor[motorC] = 50;
5      motor[motorB] = 50;
6      wait1Msec(4000);
7
8      motor[motorC] = -50;
9      motor[motorB] = 50;
10     wait1Msec(800);
11
12 }

```

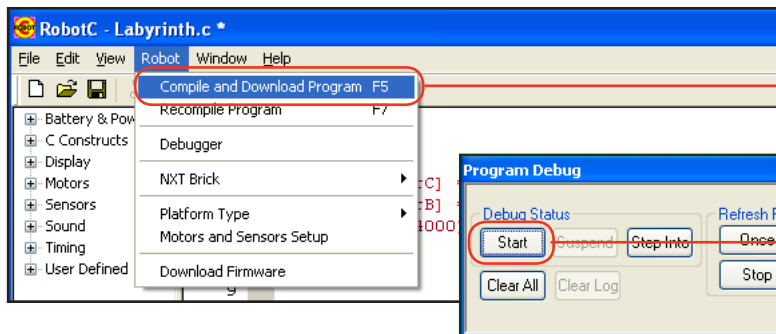
6a. Modify this code

Restore the first behavior to a half-power forward movement.

6b. Add this code

Adding a left-point-turn behavior after the moving-forward behavior will make the robot move then turn.

The turn needs only about 0.8 seconds (800ms) to complete.

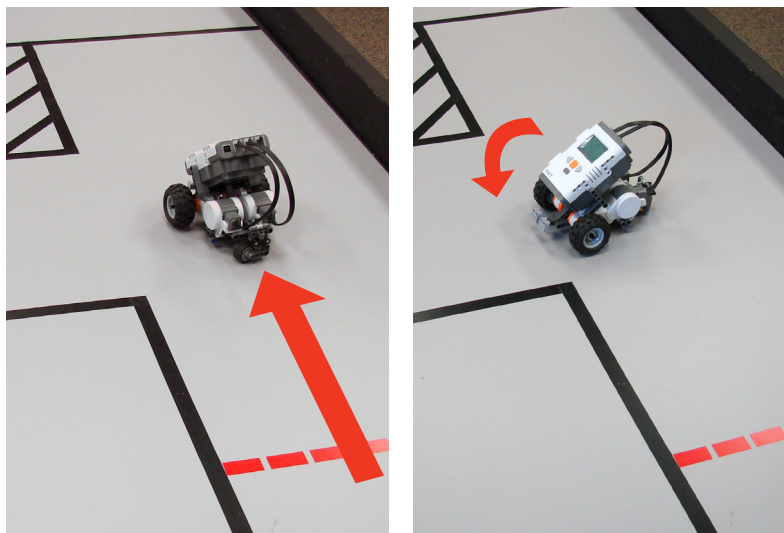


6c. Compile and Download

Select Robot > Compile and Download Program to send your program to the robot.

6d. Press Start

Press the Start button on the Program Debug menu that appears, to run the program.



6e. Behavior Sequences

Placing behaviors one after another in the code tells your robot to perform them in sequence.

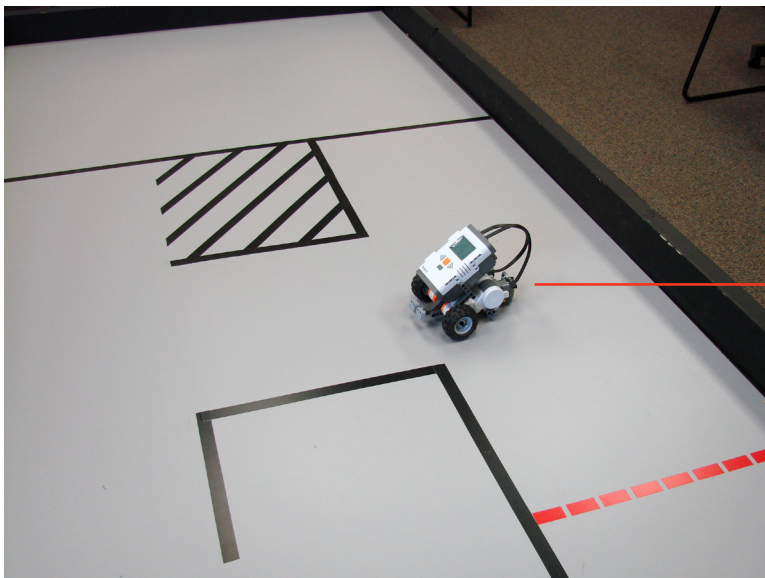
The moving-forward behavior in lines 4-6 of the program is done first (at left). The turning behavior in lines 8-10 follows immediately afterward.

Movement

Speed and Direction **Turn and Reverse** (cont.)

End of Section

You now know how to program all the necessary behaviors to navigate the Labyrinth. However, even at lowered speeds, the robot's movements are not as precise as we might like. Continue on to the Improved Movement section to learn how to clean up the robot's motion.



One down...

The robot has completed the first leg of its journey, and is ready for the next!

Movement

Speed and Direction Quiz

NAME _____ DATE _____

1. In the code example below, the speed of the robot could be changed by manipulating the:

```
1 task main()
2 {
3     motor[motorC] = 100;
4     motor[motorB] = 100;
5     wait1Msec(4000);
6 }
```

- a. motor brackets.
- b. motor time.
- c. motor power.
- d. motor sensor.

2. In the section below, write code that makes the robot perform the following tasks in order:

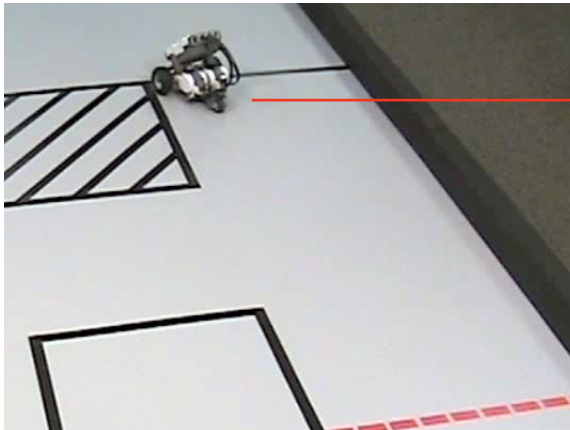
- a. Move forward at half speed for 3 seconds, then
- b. Turn in place to the right for half a second (at any speed), then
- c. Move reverse at full speed for 1 second

```
1 task main()
2 {
3
4
5
6
7
8
9
10
11
12
13
14 }
```

Movement

Improved Movement **Manual Straightening**

You know how to make the robot move, and you've made improvements to its performance by having it brake and maneuver at a slower speed. Even so, you have probably noticed by now that the robot's idea of "straight" ... isn't.



Off course

This robot has drifted noticeably to the left while running.

Even when you set the motors to go the same speed, the robot turns a little. Recall that a turn results from two motors moving at different speeds.

```

1  task main()
2  {
3
4      motor[motorC] = 50;
5      motor[motorB] = 50;
6      wait1Msec(4000);
7
8      motor[motorC] = -50;
9      motor[motorB] = 50;
10     wait1Msec(800);
11
12 }

```

Same speed?

If both motors are set the same, shouldn't they go the same speed and therefore move straight?

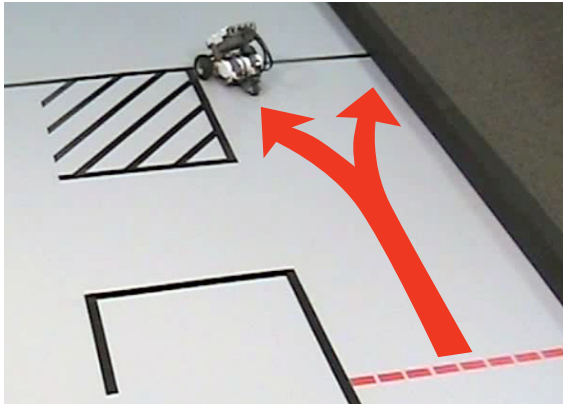
Actually, SPEEDS aren't set with the motor[] commands. Motor POWER is. However, not all motors are created equal. Various factors in the robot's construction, and the manufacturing process for the motors themselves cause different amounts of energy to be lost to friction in each motor.

This means that even though both motors start with the same power at the plug, the amount of power that reaches the wheel to move the robot can vary quite a bit. Even with the same POWER being applied, SPEEDS may differ. And as you know, wheels moving at different speeds make the robot turn, even if just a little bit. So to fix this situation, let's do the logical thing, we'll change the power so the motors end up going the same speed.

Movement

Improved Movement **Manual Straightening** (cont.)

In this lesson, you will manually adjust your motor command powers to make your robot go straight, watching for patterns to make the process smoother in the future.



Lesson Note

The example robot used in this lesson drifts slightly to the left. If your robot drifts in the other direction, simply apply the following steps to the other motor.

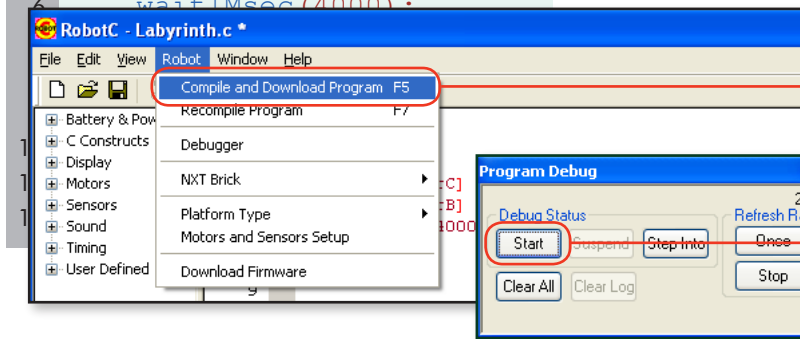
1. We can't speed up the slower motor, because it's already going full power. So instead, we'll have to slow down the faster one. The robot shown in this example has veered left, indicating that the right motor is going faster than the left.

```

1 task main()
2 {
3
4     motor[motorC] = 50;
5     motor[motorB] = 45;
6     wait1Msec(4000);

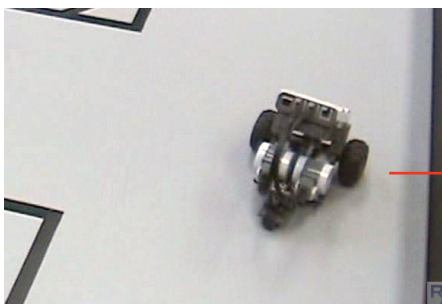
```

1a. Modify this code
Reduce the faster motor's power by 5% in the moving-forward behavior.



1b. Compile and Download
Select Robot > Compile and Download Program.

1c. Press Start
Press the Start button on the Program Debug menu.



1d. Observe behavior
Did the robot go straight?
This one curves to the right now.

Movement

Improved Movement **Manual Straightening** (cont.)

2. We seem to have overcorrected, and our robot now curves in the opposite direction. So we'll adjust our guess, and go with something in between the original and our last guess.

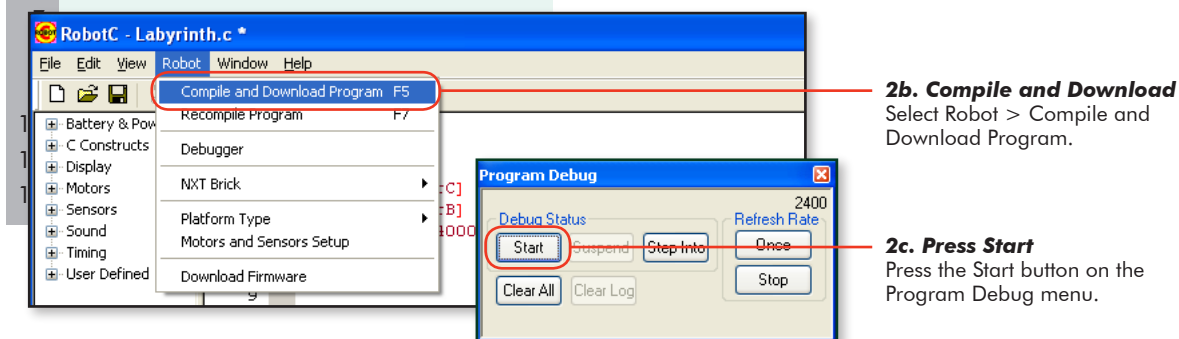
```

1  task main()
2  {
3
4      motor[motorC] = 50;
5      motor[motorB] = 48;
6      wait1Msec(4000);

```

2a. Modify this code

50 was too high, and 45 too low. Choose a value in between, like 48.

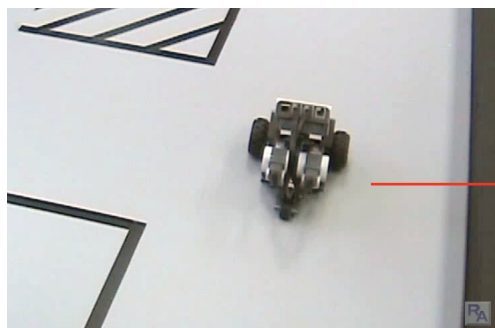


2b. Compile and Download

Select Robot > Compile and Download Program.

2c. Press Start

Press the Start button on the Program Debug menu.



2d. Observe behavior

Did the robot go straight?
It looks a lot better now.

End of Section

This method of manual straightening works, but it's unwieldy. One big problem is that it requires reprogramming any time something changes. Running on a different table surface, negotiating a slope, running after the batteries have run down, and even tuning up the robot will all force you to re-adjust these values.

Worse still, the program values don't work on every robot. In the example, we had to change our motor to 48%, but you probably had to do something quite different with yours. Worse yet, there are obstacles out there that can't be accounted for by programming your robot hours or weeks in advance. Manual adjustment to robot power levels can work, but there must be a better way...

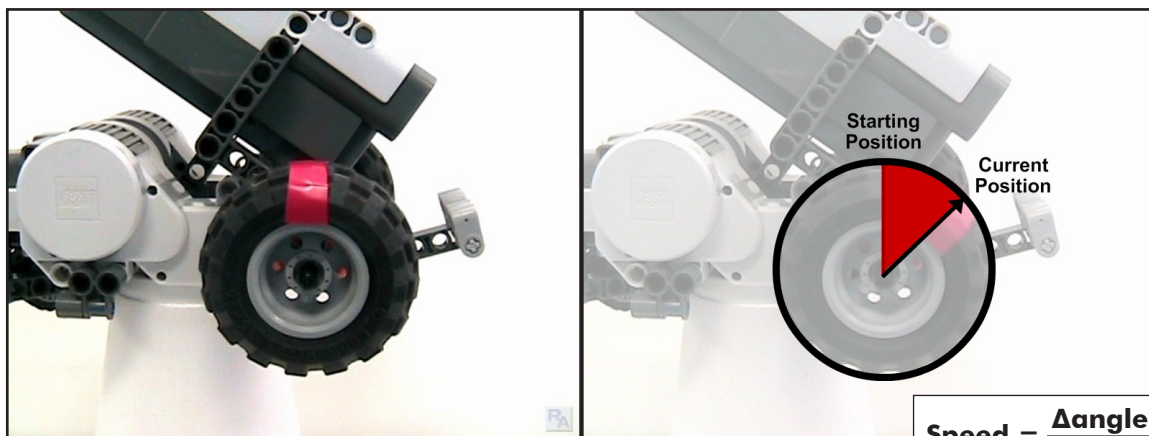
Movement

Improved Movement Principles of PID

We found that we could make a robot move straighter by adjusting power levels so that its wheels move at the same SPEED rather than just being driven with the same power. However, manual adjustment has severe limitations. What if we could find a way to make those adjustments automatically?

In this lesson, you will learn how the PID speed control algorithm works.

Using the rotation sensors built into the NXT motors, the robot can be aware of how far each wheel has moved. By comparing the motor's current position to its position a split second ago, the robot can calculate how fast the wheel is moving.



Starting position ($t=0$)

The initial position of the wheel as it starts turning.

A short time later... ($t=0.1s$)

1/10th of a second later, the wheel has turned slightly. Since both the change in position and the change in time are known, the robot can calculate the rate of turn.

$$\text{Speed} = \frac{\Delta \text{angle}}{\Delta \text{time}}$$

Suppose the wheel turned 30 degrees in the 0.1 seconds shown above. The robot would automatically calculate the speed as:

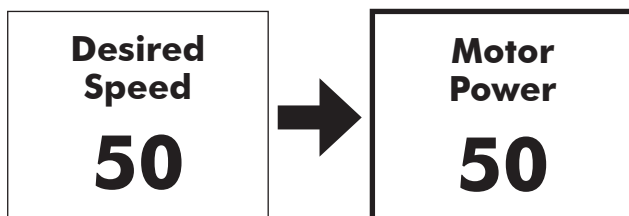
$$\text{Speed} = \frac{\Delta \text{angle}}{\Delta \text{time}} \rightarrow \text{Speed} = \frac{30^\circ}{0.1 \text{sec}} \rightarrow \text{Speed} = 300^\circ/\text{sec}$$

This speed is translated into a "speed rating" in the NXT firmware so that a speed rating of 100 would correspond to an "ideal motor" running at 100% power.

Since the robot can now tell how fast the wheel is actually turning, it can use PID to tune the motor power levels to make sure it is running at the correct speed. If the motor's actual speed is lower than it should be, the PID algorithm will increase its power level. If the motor is ahead, PID will slow it down. On the following page, we'll find out how it works.

Movement

Improved Movement Principles of PID (cont.)

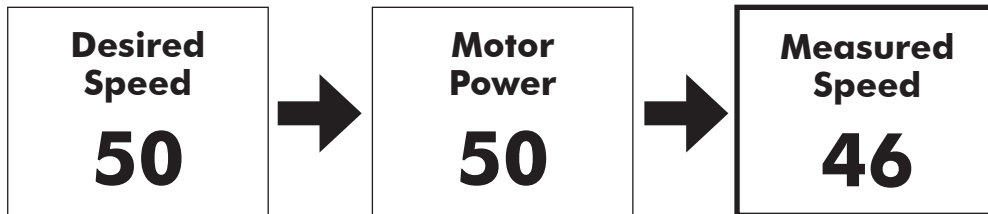


1. Motor Power

The motor is told to run at a power level that will *theoretically* produce the correct speed.

Without PID control, this is the only step used.

Without PID engaged, motor control is an “open loop” process. Motor power is set, but no mechanism is in place to see whether the desired speed is actually being achieved, and no corrections can be made.



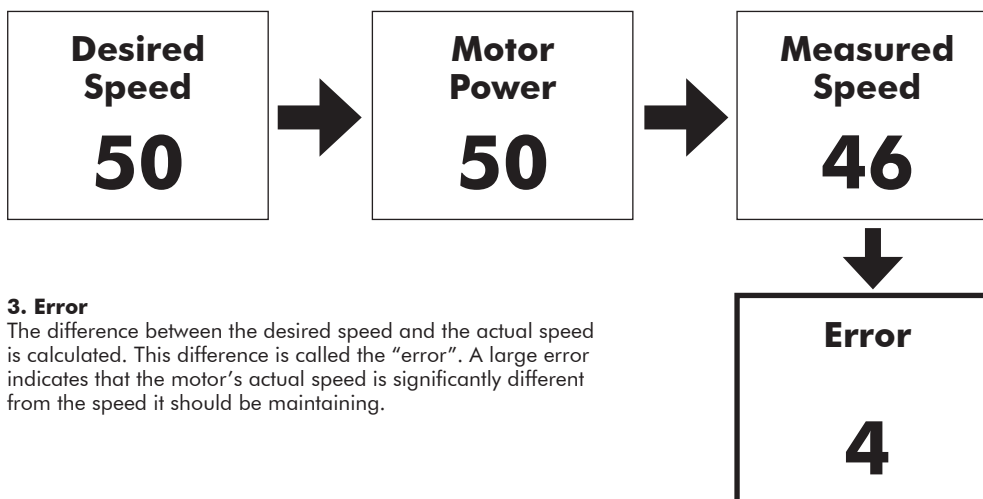
2. Measured Speed

With PID, the robot will also measure the actual speed of the motor, by measuring the position of the wheel over time (as shown on the previous page).

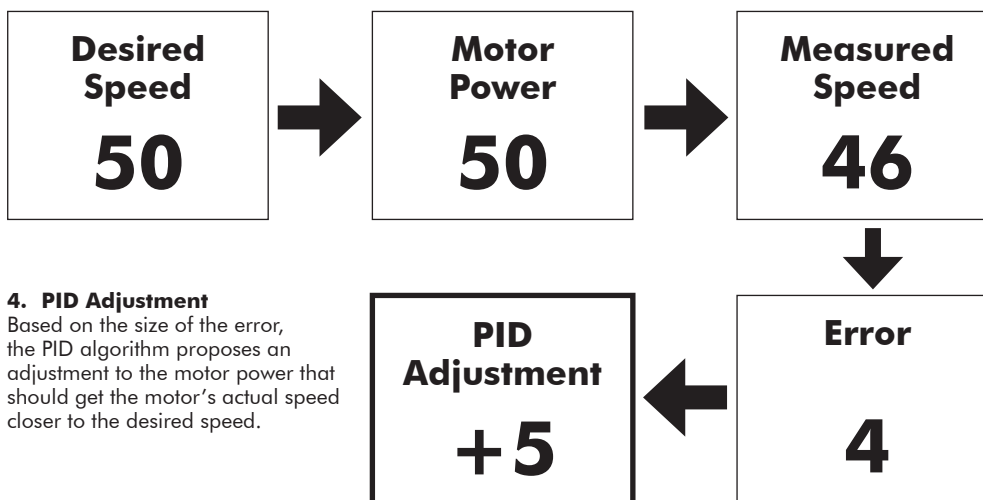
Real motors very rarely match up perfectly with “ideal” values, therefore the actual speed is different when given the “theoretical” power.

Movement

Improved Movement Principles of PID (cont.)



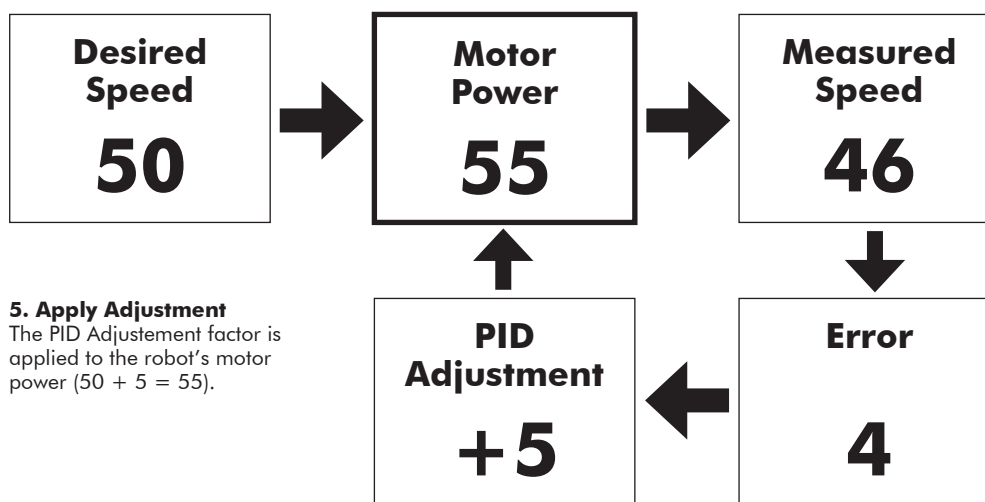
How far off is the speed? The "error" term is simply the difference between the measured speed and the desired speed.



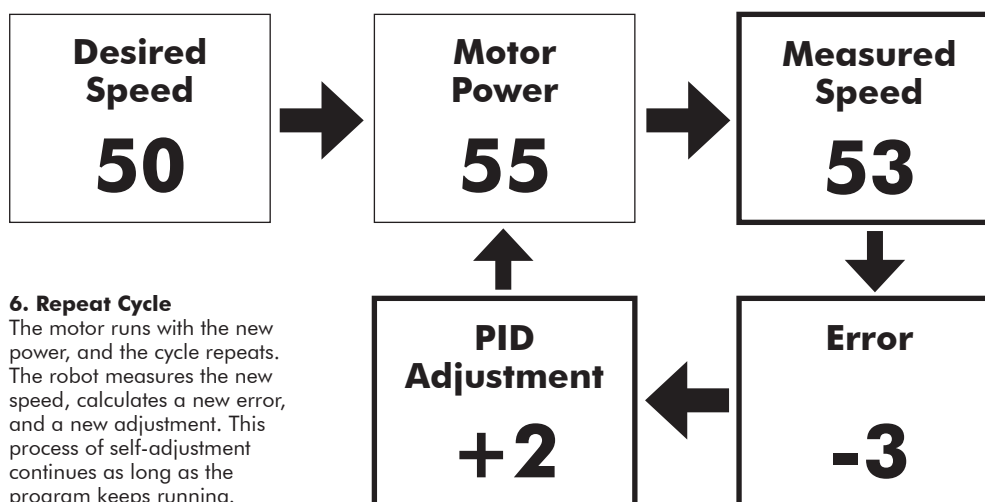
Based on the size of the error term, and how the error has been changing over time (has it been getting bigger or smaller?), the PID algorithm calculates an adjustment to the motor power that should help the motor's actual speed to get closer to the desired speed.

Movement

Improved Movement Principles of PID (cont.)



The new motor power is calculated by adding the PID adjustment factor to the original power.



The adjustment is applied to the motor power. The speed is measured again. The error is recalculated (hopefully it is now smaller!). A new adjustment factor is determined. The cycle continues forever, always ready to catch and compensate for any factor that may make the robot go at the wrong speed.

Movement

Improved Movement Principles of PID (cont.)

End of Section

This setup, where the robot monitors and adjusts its speed based on measurements it takes itself, is called “closed loop” control. The term refers to the “loop” relationship formed by output (motor power) and feedback (speed measurement, error, and PID adjustment factor).

PID gives your robot the ability to intelligently self-adjust its motor power levels to the correct values to maintain a desired speed. The closed-loop system monitors the “error” difference between how fast the robot is going and how fast it should be, and makes adjustments to the motor’s power level accordingly.

Movement

Improved Movement PID Programming

ROBOTC includes a PID algorithm already built into the firmware. In order to take advantage of PID speed control, you must first enable it in your program.

In this lesson, you will learn how to enable PID speed control for your robot's motors, using ROBOTC's built-in motor control features.

1. Start with your moving-and-turning Labyrinth program. Save your program with a new name: "LabyrinthPID".

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

1b. Browse to an appropriate folder
Browse to or create an appropriately named folder within your program folder to save your program.

1c. Rename program
Give this program the new name "LabyrinthPID".

1d. Save
Click Save.

Movement

Improved Movement PID Programming (cont.)

2. PID control must be enabled for each motor on the robot.

```

1  task main()
2  {
3
4      nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
5      nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;
6
7      motor[motorC] = 50;
8      motor[motorB] = 50;
9      wait1Msec(30000);
10
11     motor[motorC] = -50;
12     motor[motorB] = 50;
13     wait1Msec(800);
14
15 }

```

2a. Add this code

Enable PID control on both motors by setting their nMotorPIDSpeedControl modes to mtrSpeedReg.

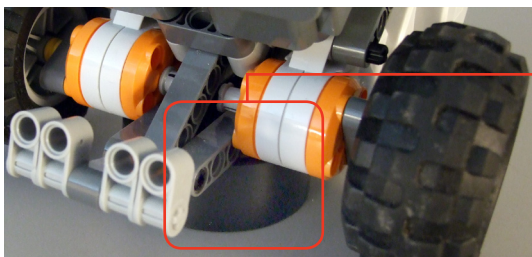
2b. Modify this code

Restore the motor command settings to 50%.

2c. Modify this code

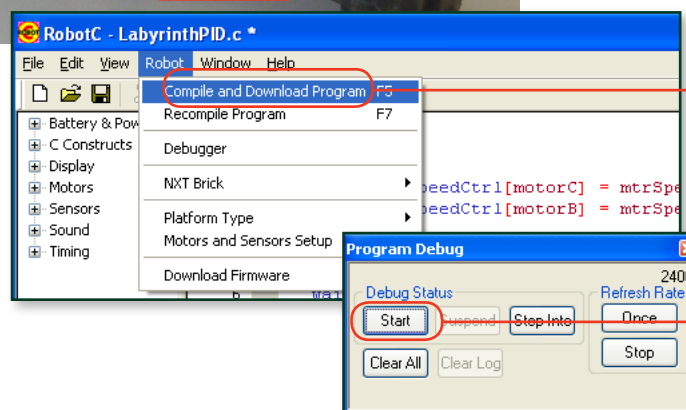
We want enough time to see and test the effects of PID control. Change this value to 30 seconds (30000 ms).

3. Download and run. Keep your robot plugged in.



3a. Block up the robot

Place an object under the robot so that its wheels can't reach the table. This lets you run the robot without having to chase it around.



3b. Download and Compile

Click Robot > Download Program.

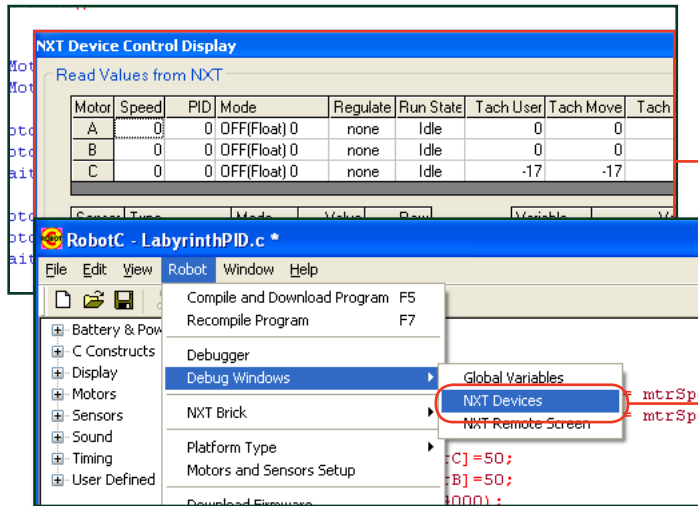
3c. Run the program

Click "Start" on the onscreen Program Debug window.

Movement

Improved Movement PID Programming (cont.)

4. A window should appear called the “NXT Device Control Display”. If it doesn’t appear...

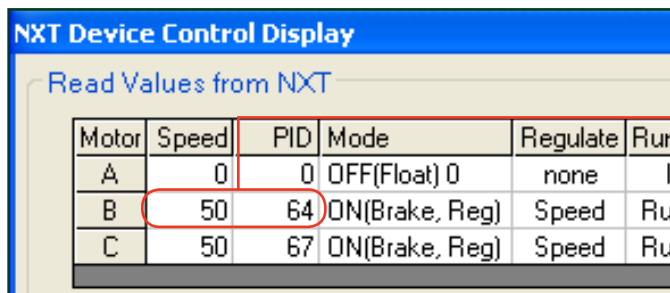


4. NXT Device Control Display
Make sure this window is showing. If not, open it through Robot > Debug Windows > NXT Devices.

Checkpoint

This debugger window is a troubleshooting tool that can help you see what your robot is doing, and what it thinks it’s doing. The lines we’re interested in are highlighted above: “Speed” and “PID” for Motors C and B.

The Speed column shows the desired speed for the motor, which we set to be 50%. The PID column shows the actual amount of power that the robot is giving the motor to make it move at that speed.

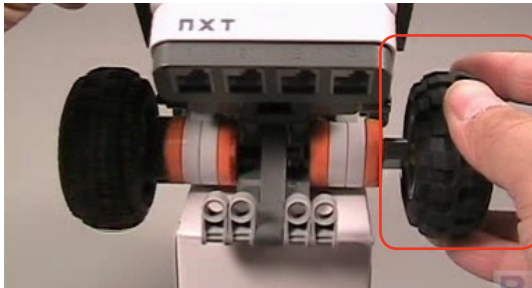


Adjusted motor power
The PID algorithm is having to give this motor 64% power to achieve 50% speed. This is typical, because the motor needs additional power to overcome friction.

Movement

Improved Movement PID Programming (cont.)

5. Hold one wheel in place and watch the power values on its corresponding motor.



5a. Hold wheel

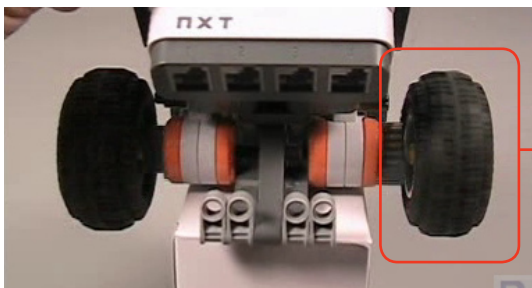
Grab one of the wheels on the robot and hold it so it stops. In the picture, motor C's wheel is being held.

| NXT Device Control Display | | | | | | |
|----------------------------|-------|------|----------------|----------|-----|--|
| Read Values from NXT | | | | | | |
| Motor | Speed | PID | Mode | Regulate | Run | |
| A | 0 | 0 | OFF(Float) 0 | none | | |
| B | 50 | 68 | ON(Brake, Reg) | Speed | Run | |
| C | 50 | 100 | ON(Brake, Reg) | Speed | Run | |
| Sensor | Type | Mode | Value | Raw | | |

5b. Observe motor power

The PID algorithm will notice that the motor's measured speed is falling behind where it should be, and will increase the motor's power level to try to bring the speed up.

6. Release the wheel and observe its reaction.



6a. Release the wheel

Let go of the wheel so it can turn freely again.

| NXT Device Control Display | | | | | | |
|----------------------------|-------|------|----------------|----------|-----|--|
| Read Values from NXT | | | | | | |
| Motor | Speed | PID | Mode | Regulate | Run | |
| A | 0 | 0 | OFF(Float) 0 | none | | |
| B | 50 | 68 | ON(Brake, Reg) | Speed | Run | |
| C | 50 | 35 | ON(Brake, Reg) | Speed | Run | |
| Sensor | Type | Mode | Value | Raw | | |

6b. Observe motor power

Now that the wheel is going too fast, the motor will decrease its power until it reaches the correct speed.

Movement

Improved Movement PID Programming (cont.)

7. End the program and return the timing to what it was before.

```
1 task main()
2 {
3
4     nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
5     nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;
6
7     motor[motorC] = 50;
8     motor[motorB] = 50;
9     wait1Msec(4000);
10
11    motor[motorC] = -50;
12    motor[motorB] = 50;
13    wait1Msec(800);
14
15 }
```

7. Modify this code
Change the timing back to 4000ms (still at 50% speed).

End of Section

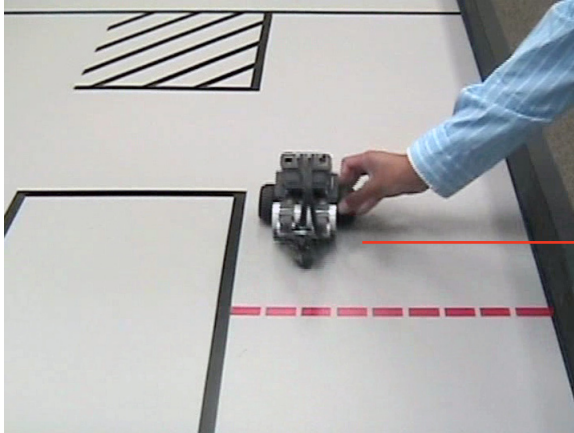
PID control is a great way to make your robot's movement more consistent. The algorithm monitors how fast the motors are turning versus how far they should be, and adjusts the motors' power levels to keep them on track. This allows the robot to automatically adjust for minor variations both in the environment and in the motors themselves.

Movement

Improved Movement **Synchronized Motors**

When we started, we said that we wanted the robot to go straight. Its motors should move at the same speed. PID control gave us that in a roundabout way: by asking both motors to maintain a target speed, and giving them both the same target, they moved the same speed. Sort of.

If we run into a tough spot like this, how should the robot react?



Stuck

The wheel is being held firmly in place... what should the other wheel do?

Using PID, the other motor will keep running at the speed it was set to, and the robot will begin to spin in a circle as if ordered to turn.

However, if going straight is the priority, then we need to change our perspective slightly. We'll need to enforce identical speeds on the two motors as our first priority, not just tell both motors to seek the same target independently. **The sameness of the values is more important than the exact speed.**

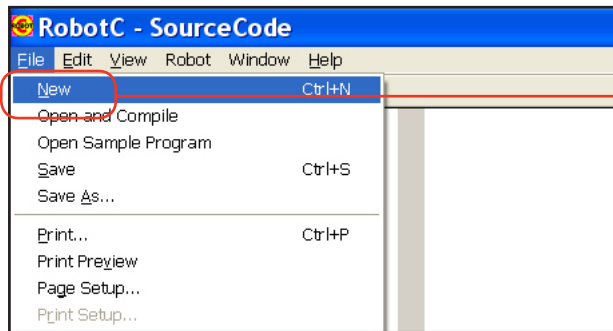
ROBOTC includes a feature called **Motor Synchronization**, which allows you to pair two motors together, and define their speeds relative to each other. If you tell them that their goal is to stay exactly together with one another as they move, then they will, even if it means the faster one has to stop and wait. The goal of keeping both motors together takes precedence over reaching the "ideal" speed.

Movement

Improved Movement **Synchronized Motors** (cont.)

In this lesson, you will learn how to use Motor Synchronization to ensure that both motors run at the same speed, even if something unexpected happens to one of them.

1. Open ROBOTC and start a new program.



1. Create new program

Select File > New to create a blank new program.

2. Add the basic framework for a program.

```

1  task main()
2  {
3
4
5  }
```

2. Add this code

Add a task main() {}.

3. Engage Motor Synchronization on the robot, with the sync mode set to "synchBC".
The special term **synchBC** defines B and C as the motors to be synchronized.

```

1  task main()
2  {
3
4      nSynchedMotors = synchBC;
5
6  }
```

3. Add this code

Engage Motor Synchronization for Motors B and C, with B set as the master.

Movement

Improved Movement **Synchronized Motors** (cont.)

Checkpoint

The program will now operate motors B and C in Synchronized mode. The order of the letters BC in “synchBC” does matter, because the two motors in a synchronized setup are not completely equal. Of the pair, one of the two motors will take the lead, and the other will play a more reactive role.

The motor B (the first letter in “synchBC”) is called the Master motor, and C (the second one) is called the Slave motor. **All commands to the motor pair, such as speed or braking commands, are issued through the Master motor.**

The Slave motor, C in this case, doesn’t receive a speed command. Instead, we give it a **ratio command**. This ratio is defined as a percentage of the first motor’s position. For moving forward, you always want the two motors to be at the same position, so we’ll set the Slave motor ratio to be 100% of the Master motor’s.

4. Set the slave motor to run at 100% of the master motor’s speed.

```

1  task main()
2  {
3
4      nSyncedMotors = synchBC;
5      nSyncedTurnRatio = 100;
6
7  }
```

4. Add this code

Set the turn ratio for the slave motor (C) to be 100%. Slave motor C will now attempt to maintain exactly 100% of the master motor B’s speed.

Note that the master motor’s speed has not been set yet, so the slave motor B will initially be running at 100% of 0 (i.e. stopped).

5. Set the master motor to a desired speed of 50, and let the robot run for 4 seconds.

```

1  task main()
2  {
3
4      nSyncedMotors = synchBC;
5      nSyncedTurnRatio = 100;
6
7      motor[motorB] = 50;
8      wait1Msec(4000);
9
10 }
```

5. Add this code

Set a desired speed of 50 for the master motor. Master motors are automatically PID speed regulated.

Movement

Improved Movement **Synchronized Motors** (cont.)

6. Save your program as "LabyrinthSynch".

6a. Save program As...
Select File > Save As... to save your program under a new name.

6b. Browse to an appropriate folder
Browse to or create an appropriately named folder within your program folder to save your program.

6c. Rename program
Give this program the new name "LabyrinthSynch".

6d. Save
Click Save.

5. Download and Run.

7a. Compile and Download
Click Robot > Compile and Download Program.

7b. Run the program
Click "Start" on the onscreen Program Debug window.

Movement

Improved Movement **Synchronized Motors** (cont.)

Checkpoint

The motors are now constantly updating themselves to maintain identical positions as they move. If one motor happens to stop, the other motor will adjust, and maintain 100% of the new position!

Finally, motor synchronization is useful for far more than just going straight. Cleaning up turning is also quite easy. As you saw when you first encountered turns, all you need to do is set the motors to move at different speeds. To turn in place, the motors should go different speeds. For a point turn, they should be completely opposite. The Slave motor should go -100% of the Master motor's speed.

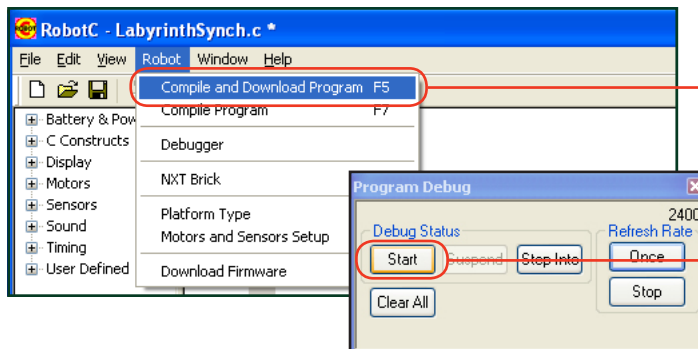
8. Change the sync ratio to -100% to make the robot turn instead of moving straight.

```

1  task main()
2  {
3
4      nSyncedMotors = synchBC;
5      nSyncedTurnRatio = -100;
6
7      motor[motorB] = 50;
8      wait1Msec(4000);
9
10 }
```

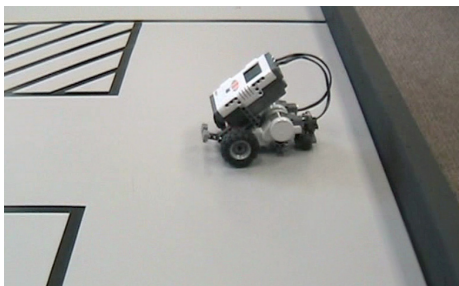
8. Modify this code
Change the sync ratio 100% to -100% to make the motors turn in exactly opposite directions.

9. Download and Run.



9a. Compile and Download
Click Robot > Compile and Download Program.

9b. Run the program
Click "Start" on the onscreen Program Debug window.



Movement

Improved Movement **Synchronized Motors** (cont.)

End of Section

Motor synchronization allows you to control your robot in a way that prioritizes motor alignment over motor speed. This is a trade-off, but one that may be favorable when the most important thing is getting your robot to go straight.

Movement

Improved Movement Quiz

NAME _____ DATE _____

1. What factor or factors affect the robot's ability to move in a straight line?

- a. Motor manufacturing tolerances
 - b. Robot weight distribution
 - c. Frictional forces in the robot's drive train
 - d. All the above
-

2. "Closed-loop" control describes a system:

- a. that monitors its own performance and adjusts its output to achieve a desired outcome.
 - b. whose specifications are kept secret.
 - c. in which a Loop control structure with matching opening and closing punctuation is used.
 - d. which is ring-shaped.
-

3. The command `nSyncedTurnRatio=100`; would tell the slave motor to turn:

- a. at the same rate and in the same direction as the master.
 - b. at the same rate and in the opposite direction of the master.
 - c. at 100 degrees per second, in the same direction as the master.
 - d. at full power forward.
-

4. The PID algorithm adjusts:

- a. the power level of an individual motor to achieve a target speed.
 - b. two motors' powers to keep them together at all times.
 - c. a motor's gear ratio to achieve a target power.
 - d. the amount of friction in a motor to make it run more smoothly.
-

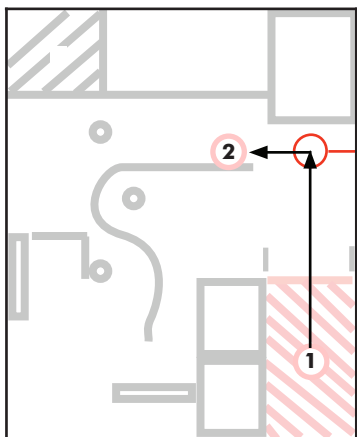
5. Write the piece of code that would establish a Synchronized relationship between motors B and C, with C as the master and B as the slave in the space below.

1
2
3
4

Sensing

Wall Detection **Touch vs. Timing**

We've learned a lot about how to make the robot move, including how to make it go forward and backward for specific lengths of time, how to adjust its speed, and how to make it go as straight as possible. But motor control alone won't be enough to let the robot to stay on the obstacle course below, because we don't know exactly where the robot will start.



Turn left

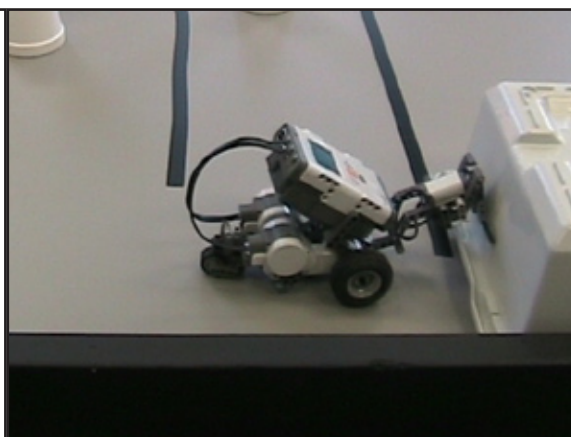
To get from position 1 to position 2, the robot has to turn left just in front of the wall, at the red circle. In this challenge, we don't know exactly where the robot will start in the red hatched area. It is therefore impossible to make the robot turn in the correct place using motor control only.

We know we want the robot to make a left turn just in front of the obstacle course wall. What we need is a way for the robot to find out where that wall is, and adjust its course accordingly. In this lesson, we'll attach a Touch Sensor to the robot and use it to detect the wall. By using feedback from the sensor, we can make the robot turn in the correct place no matter how far away from the wall it started.



Touch Sensor

The Touch Sensor, above, can enable the robot to detect physical contact with objects like walls.



Touch Sensor detecting a wall

A robot uses sensors to gather information from the environment and uses the information to plan movement.

Sensing

Wall Detection **Touch vs. Timing** (cont.)

In this lesson, you will learn to use feedback from a Touch Sensor to let the robot detect a solid object and adjust its course accordingly.

1. Add the Touch Sensor attachment to the robot (if it doesn't have one already). Connect the sensor to Port 1 on the NXT brick. The bumper assembly helps the sensor to detect collisions that are not centered directly on the sensor's orange contact surface.



1. **Build the Touch Sensor attachment**
Building instructions are available through the main lesson menu. Connect the Touch Sensor to port 1.

2. Load the program "nxt_wait_for_push.c" on the NXT.

2a. Open sample program
Click File > Open Sample Program.

2b. Open Touch folder
Double-click the "Touch" folder to open it.

2c. Open *nxt_wait_for_push*
Double-click "*nxt_wait_for_push.c*" to open the program.

Sensing

Wall Detection **Touch vs. Timing** (cont.)

Checkpoint

The program should look like the one below.

```

c
p/p
const tSensors touchSensor = (tSensors)
//*!!CLICK to edit 'wizard' created sensor & mot
//*****
//                               Wait for Push
//                               RobotC on NXT
//

```

3. Note that the program has 3 major parts. (Lines 1-35 have been omitted, since they contain only comments that do not affect how the program works.)

Auto `const tSensors touchSensor = (tSensors) S1;`

Touch Sensor setup

At the top of the program is a special line that tells ROBOTC to look for a Touch Sensor on Port 1, and to call it "touchSensor".

```

36 task main()
37 {
38     while(SensorValue(touchSensor) == 0)
39     {
40         motor[motorA] = 100;
41         motor[motorB] = 100;
42     }
43
44     motor[motorA] = -75;
45     motor[motorB] = -75;
46
47     wait1Msec(1000);
48 }
49 }

```

While() loop

Next, we have the while() loop. It's called a "while" loop because it will do something *while* certain conditions continue.

Movement commands

Finally, we have two sets of movement commands: one *inside* the while() loop, and one *right after* the while() loop. The positioning of these commands inside and outside of the loop is important, but otherwise, these are the same commands you have already used to move the robot in previous programs.

Checkpoint

You will learn more about the Sensor Setup and while() loop parts of the program later in this lesson. For now, however, **look carefully at the motor commands**. Which motor ports do they address? What ports are your motors plugged into? Do they match?

The sample program assumes your motors would be on ports A and B, but your robot's design has them on C and B! The program will not work without modifications. Software (programs) and hardware (like the physical robot) are dependent on each other to produce correct behaviors.

Sensing

Wall Detection **Touch vs. Timing** (cont.)

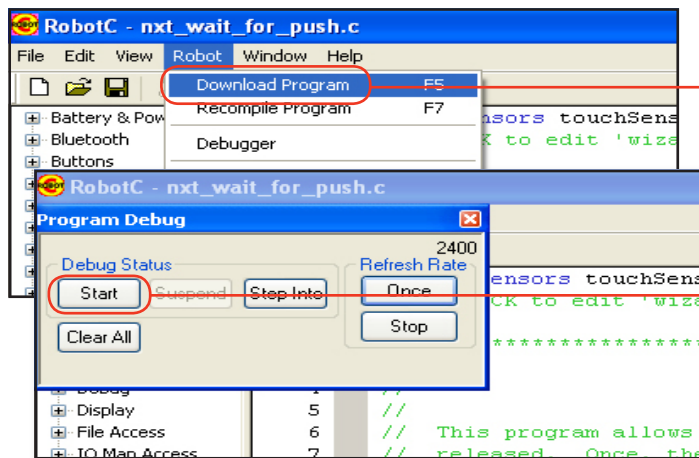
4. Modify the motor[] commands to send power to the correct motors by changing all the motorA references to motorC (motorB is the same in both).

```
Auto const tSensors touchSensor = (tSensors) S1;

36 task main()
37 {
38     while(SensorValue(touchSensor) == 0)
39     {
40         motor[motorC] = 100;
41         motor[motorB] = 100;
42     }
43
44     motor[motorC] = -75;
45     motor[motorB] = -75;
46
47     wait1Msec(1000);
48 }
49 }
```

4. Modify this code
Change the motorA references to instead use motorC, where your left motor is actually attached.

5. Download and run the program.



5a. Download the program
Click Robot > Download Program.

5b. Run the program
Click "Start" on the onscreen Program Debug window.

Sensing

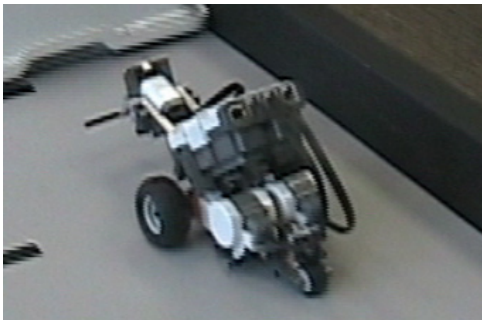
Wall Detection **Touch vs. Timing** (cont.)

6. Run the program on the Obstacle Course board. Observe the sample program's behaviors.



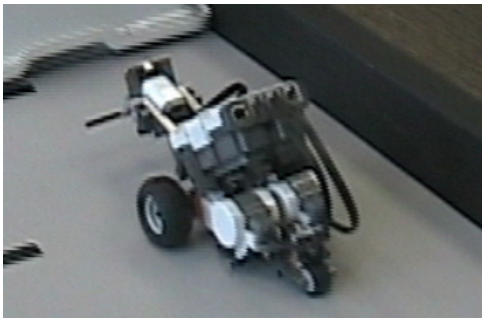
6a. Forward until touch

The robot runs forward as long as the touch sensor is not pressed in.



6b. React to touch

When the touch sensor is pressed, the robot will back up for one second, then stop.



6c. End

The program ends after one touch-and-reverse cycle.

End of Section

We've taken a crucial step forward in solving the problem of getting the robot to adjust its course when it touches a wall by adding a Touch Sensor attachment, downloading a program, and demonstrating that the robot will reverse its direction when it reaches a solid object. The next step is to understand the program, so that you can write one like it yourself.

Sensing

Wall Detection Configuring Sensors

Now that we've seen the wall detection program work, we're going to take it apart piece by piece to understand how it works. In this lesson we'll examine the first section of the program, where we set up the sensors. This configuration process tells the robot which sensors are present, and which ports they're connected to.

In ROBOTC, sensor configuration is done through the Motors and Sensors Setup dialog, which we'll go through in this lesson. You don't have to, and shouldn't, type any code inside the sensor configuration section at all, unless you're an experienced programmer.

```
Auto const tSensors touchSensor = (tSensors) S1;
```

```
36 task main()
37 {
38     while(SensorValue(touchSensor) == 0)
39     {
40         motor[motorC] = 100;
41         motor[motorB] = 100;
42     }
43
44     motor[motorC] = -75;
45     motor[motorB] = -75;
46
47     wait1Msec(1000);
48 }
49 }
```

Touch Sensor set up

At the top of the program is a special line that tells ROBOTC to look for a Touch Sensor on Port 1, and to call it "touchSensor". Don't type in this area unless you know what you're doing!

Sensing

Wall Detection **Configuring Sensors** (cont.)

In this lesson, you will learn how to use the Motors and Sensors Setup dialog to configure the Touch Sensor.

1. Begin by saving the program under a new name. You can't save changes directly to the sample programs, and you want to have a copy of the program for yourself anyway.

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

1b. Browse to an appropriate folder
Browse to or create an appropriately named folder within your program folder to save your program.

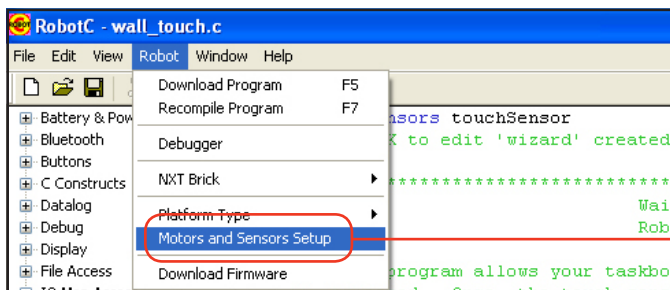
1c. Rename program
Give this program the new name "wall_touch".

1d. Save
Click Save.

Sensing

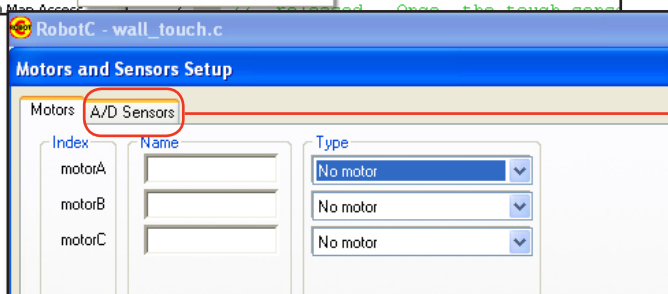
Wall Detection **Configuring Sensors** (cont.)

2. Open the Motors and Sensors Setup menu, and select the A/D Sensors tab.



2a. Open "Motors and Sensors Setup"

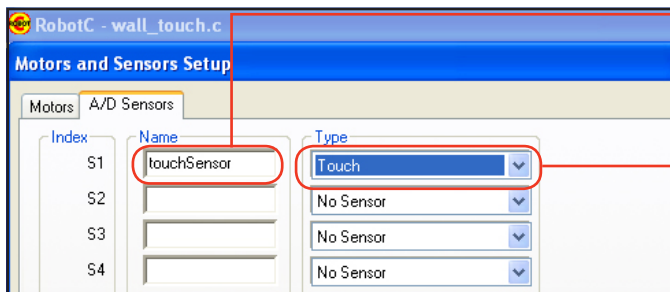
Select Robot > Motors and Sensors Setup to open the Motors and Sensors Setup menu.



2b. Select the A/D Sensors tab

Click the "A/D Sensors" tab" on the Motors and Sensors Setup menu.

3. Note the Motors and Sensors Setup menu configuration. To the right of S1 are boxes indicating the name and type of sensor attached to sensor port 1.



3a. Sensor "Name"

Assigns the name "touchSensor" to the sensor on port 1. "touchSensor" is a name chosen for convenience, following certain rules (see below).

3b. Sensor "Type"

Identifies the sensor attached to sensor port 1 as a Touch Sensor.

Naming Things in ROBOTC

Here are some basic rules for giving names to things (such as Sensors) in ROBOTC:

- Words that are already part of the ROBOTC language (like "while" or "motor") cannot be used as names
- Names may not contain spaces
- Names may not contain punctuation
- Names may not START with a number, but may contain them anywhere else
- CaPiTaLiZaTiOn maTTeRs

Sensing

Wall Detection **Configuring Sensors** (cont.)

4. Let's try changing some settings to see what happens. Move all the Touch Sensor entries in the menu from S1 to S2.

4a. Delete "touchSensor" from S1
Delete the name "touchSensor" from the S1 Name box. The Type box for S1 will change to read No Sensor after your cursor leaves the Name area.

4b. Enter "touchSensor" in S2
Type the name "touchSensor" in the S2 Name box.

4c. Change S2 Type to Touch
Select Touch from the S2 Type dropdown menu.

4d. Click OK
Click OK to save your sensor configuration changes.

Checkpoint

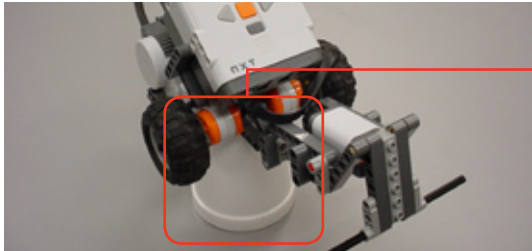
The first line of the program should now look like this. Make sure that the first line of the program contains "S2" and not "S1". The sensor on S2 is named touchSensor and set to work as a Touch Sensor.

```
Auto const tSensors touchSensor = (tSensors) S2;
```

Sensing

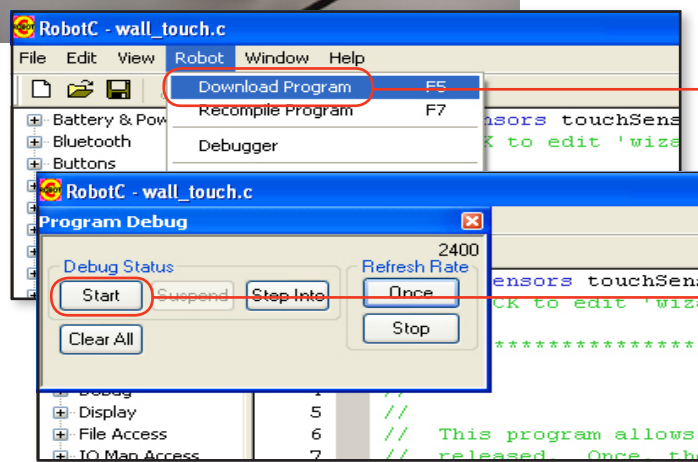
Wall Detection **Configuring Sensors** (cont.)

5. Download and run the program, but before you run it, pick up or block up the robot so it doesn't run into anything.



5a. Block up the robot

Place an object under the robot so that its wheels can't reach the table. This lets you run the robot without having to chase it around.



5b. Download the program

Click Robot > Download Program.

5c. Run the program

Click "Start" on the onscreen Program Debug window.

6. While the program is running, press the Touch Sensor. The robot continues to move forward, rather than reversing direction.



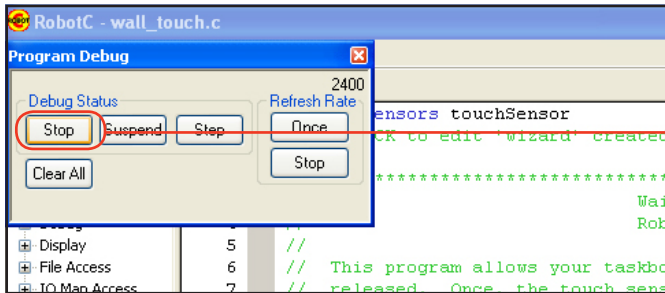
6. Press the touch sensor

Press the orange button on the Touch Sensor and observe the robot's reaction (or lack thereof).

Sensing

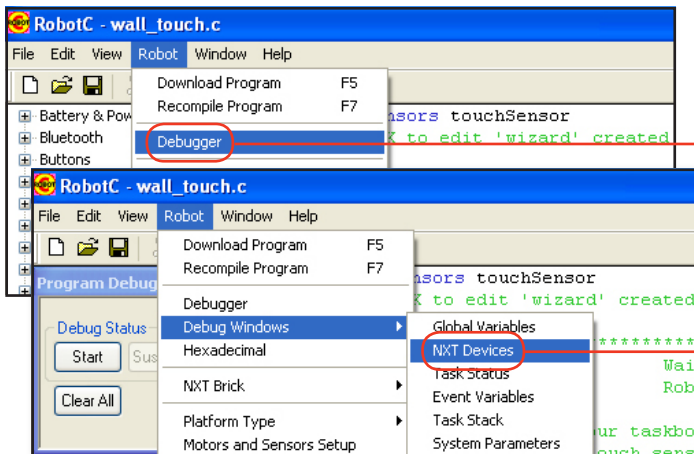
Wall Detection **Configuring Sensors** (cont.)

7. Stop the program to conserve battery power.



7. Stop the program
Click "Stop" on the Program Debug window.

8. Bring up the NXT Device Control Display window to find out why the Touch Sensor no longer makes the robot reverse direction. If the NXT Device Control Display window is already visible, skip this step.



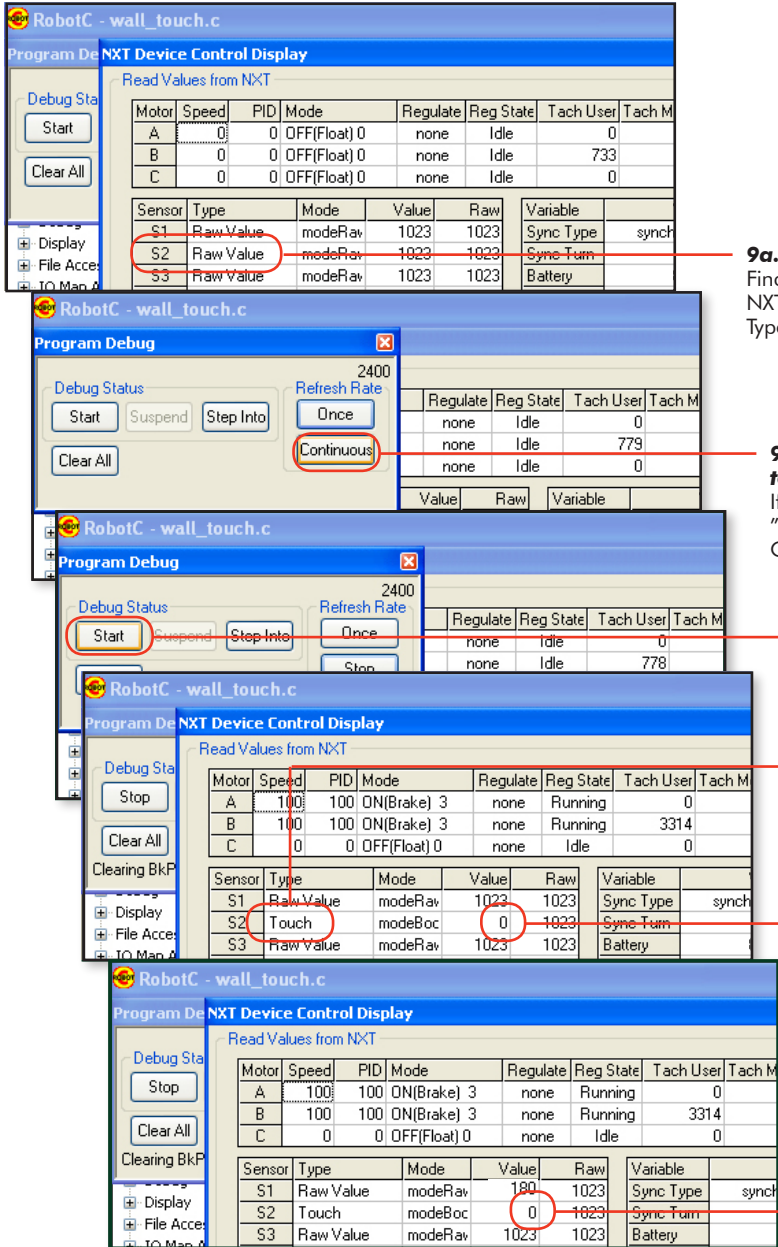
8a. Bring up the Debugger
Select Robot > Debugger.

8b. Bring up the Device Window
Select Robot > Debug Window > NXT Devices.

Sensing

Wall Detection **Configuring Sensors** (cont.)

9. Observe the changes in the NXT Device Control Display window when you run the program, and when you touch the Touch Sensor.



9a. Observe S2
Find the S2 box under Sensor in the NXT Device Control Display. Under Type, it should say "Raw Value".

9b. Change Refresh Rate to Continuous
If you see a button labeled "Continuous", press it. Otherwise, skip this step.

9c. Start the program
Click Start in the Program Debug window.

9d. Observe "Type" of S2
The "Type" of sensor on S2 is now a Touch Sensor, just as we set it to be.

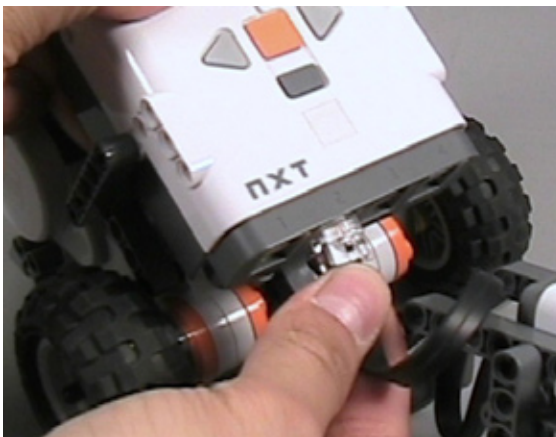
9e. Observe "Value" of S2
A Touch Sensor will show a "Value" of 1 if the sensor is pressed, and a value of 0 otherwise. What does this value indicate?

9f. Press the Touch Sensor and watch the S2 value
Press the Touch Sensor, and watch for a change (or lack of change) in the "Value" box in S2. Should it change?

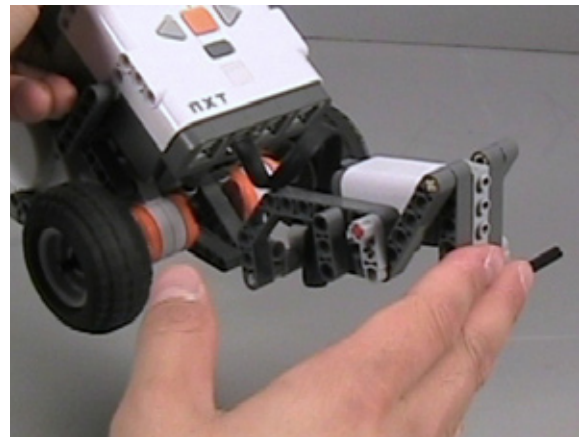
Sensing

Wall Detection **Configuring Sensors** (cont.)

10. Even when you press the Touch Sensor, the S2 value remains 0. This makes sense, because the Touch Sensor is attached to Port 1, not Port 2. Try connecting the Touch Sensor to port 2 and see what happens.



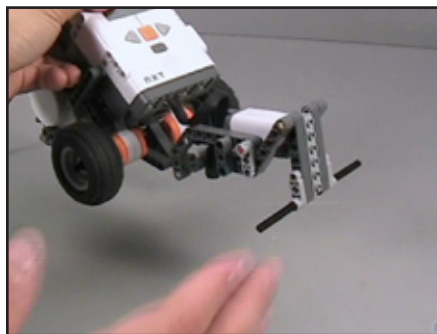
10a. Switch sensor ports
Disconnect the Touch Sensor from port 1 and reconnect it to port 2 on the NXT.



10b. Press the Touch Sensor and watch the S2 value
Press the Touch Sensor, and watch for a change (or lack of change) in the S2 Value in the NXT Device Control Display.

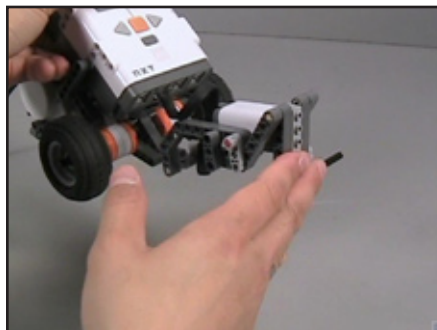
Checkpoint

Now when you press the Touch Sensor, the S2 value turns to 1. A value of 1 indicates “pressed” on a Touch Sensor. Also, the program now works as it did before. When you press the Touch Sensor, the motor now reverses for one second and stops.



| Mode | Value | Raw | Var |
|---------|-------|------|-----|
| modeRav | 1023 | 1023 | Syr |
| modeBoc | 0 | 1023 | Syr |
| modeRav | 1023 | 1023 | Bal |
| modeRav | 1023 | 1023 | Sle |

Not Pressed
The value for the sensor S2 is 0 while the Touch Sensor remains unpressed



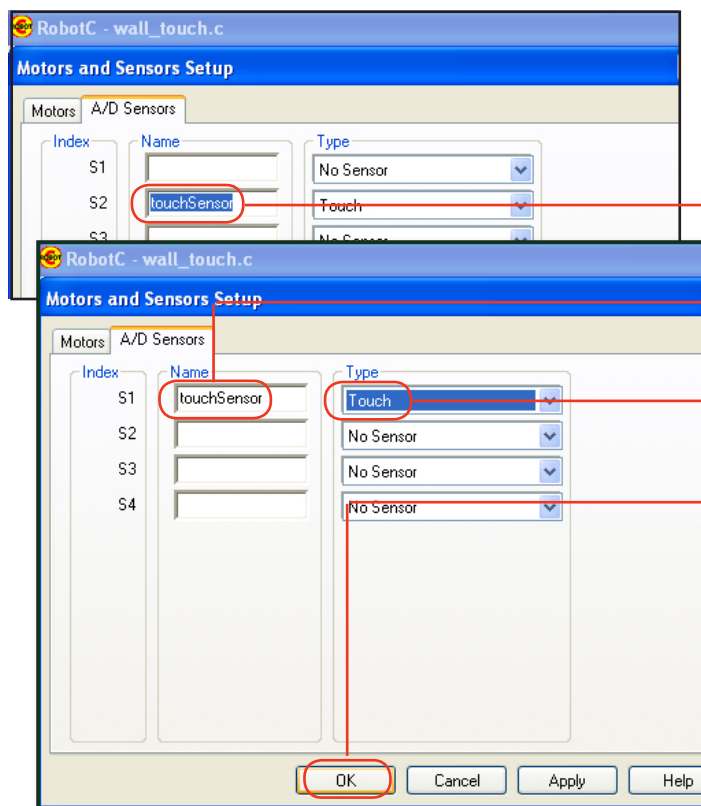
| Mode | Value | Raw | Var |
|---------|-------|------|-----|
| modeRav | 1023 | 1023 | Syr |
| modeBoc | 1 | 102 | Syr |
| modeRav | 1023 | 1023 | Bal |
| modeRav | 1023 | 1023 | Sle |

Pressed
The value for the sensor S2 is 1 when the Touch Sensor is pressed.

Sensing

Wall Detection **Configuring Sensors** (cont.)

- 11.** Use the Motors and Sensors Setup menu to change the sensor settings back to the way they were so we can move on with the program.



11a. Delete "touchSensor" from S2
Delete "touchSensor" from the S2 Name box.

11b. Enter "touchSensor" in S1
Type "touchSensor" in the S1 Name box.

11c. Change S1 Type to "Touch".
Select "Touch" from the S1 "Type" dropdown menu.

11d. Click OK
Click OK to confirm the change.



- 11e. Switch sensor ports**
Disconnect the Touch Sensor from port 2 and reconnect it to port 1

End of Section

You have successfully used the Motors and Sensors Setup menu to configure the Touch Sensor to work on port 2, and now changed it back to port 1. This is the universal process for configuring sensors in ROBOTC. You also learned to use the NXT Device Control Window to view sensor values. Finally, you also saw the two values the Touch Sensor can provide: 0 (unpressed) and 1 (pressed). It's time to move on to the next lesson, where you will examine the part of the program called the while loop.

Sensing

Wall Detection The while() Loop

Your robot's ability to sense and respond to touch revolves around a structure in the program called a while() loop. The while() loop in this program uses the Touch Sensor feedback to decide whether the robot should continue on its current course, or back up and turn.

In this lesson, you will learn what a while() loop is and how it works.

Below is the code for the sample program's while() loop. Reading this statement out loud tells you pretty much exactly what it does:

*"While the sensor value of the **Touch Sensor** is equal to zero, run motors C and B at 100% power."*

```

38     while (SensorValue(touchSensor) == 0)
39     {
40         motor[motorC] = 100;
41         motor[motorB] = 100;
42     }

```

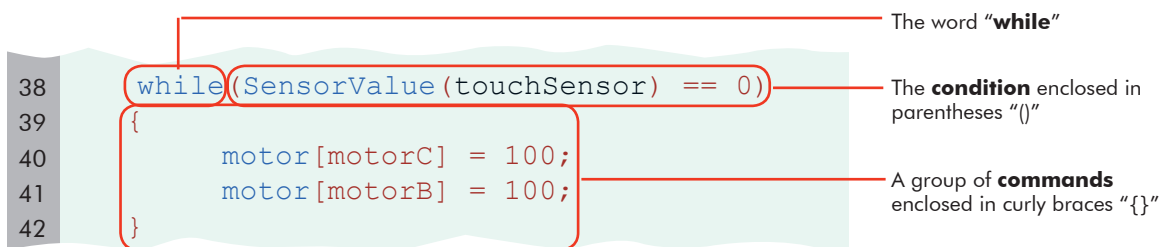
The decision-making nature of the while() loop may not be apparent at first, but making decisions that control the flow of the program is actually the while() loop's main purpose. The while() loop above instructs the program to use the Touch Sensor's status to **decide** how long to keep the motors running.

When the program reaches most commands, it runs them, and then moves on. When the program reaches the while() loop, however, it steps "inside" the loop, and stays there as long as the while() loop decides that it should. The loop also specifies a set of commands that the robot will repeat over and over as long as the program remains inside the loop.

The programmer specifies in advance *under what **conditions** the program should remain in the loop*, and *what **commands** the robot should repeat while inside the loop*.

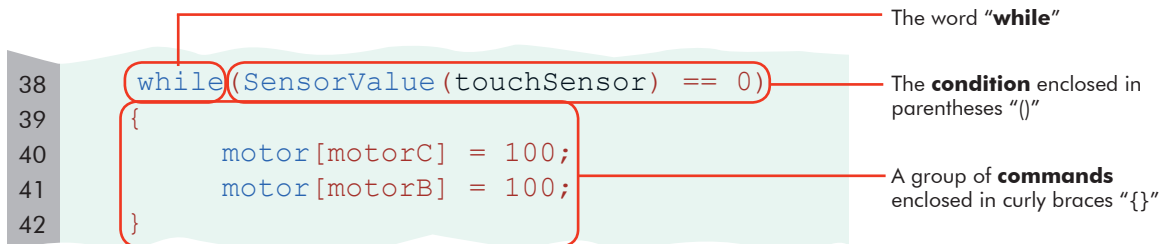
The while() loop therefore has three parts, in order:

- The word "**while**"
- The **condition** enclosed in parentheses "()"
- A group of **commands** enclosed in curly braces "{}"



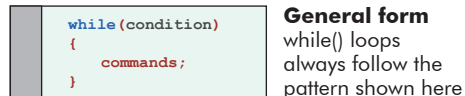
Sensing

Wall Detection **The while() Loop** (cont.)



while

A while() loop always starts with the word “while”.



General form

while() loops always follow the pattern shown here

The (condition)

The statement in parentheses specify the *condition(s)* under which the loop should continue looping. These conditions are specified in the form of a true-or-false statement, like the one in the example above, “The sensor value of the Touch Sensor is equal to zero”. The statement is either *true* (the value IS zero) or it is *false* (the value IS NOT zero).

The *true* (or *false*) value of the statement determines whether the loop will continue or end. As long as the condition is true, the while loop will continue to run. If the condition becomes false, the loop will end and the program will move on to the commands that come after it.

Example

In the code above, the condition is “The sensor value of the Touch Sensor is equal to zero.” This (condition) statement is true as long as the Touch Sensor reads zero. Recall from the previous lesson that the Touch Sensor reads 0 whenever its button is not pressed in, and it reads 1 when the button is pressed in.

So, as long as the Touch Sensor button is NOT pressed, the sensor value will be zero, and the condition will be true. As long as the condition remains true, the commands inside the curly braces will run. If the Touch Sensor is ever pressed, its value will become 1, not 0, and the condition will become false. The loop would then end.

The {commands}, sometimes called the “body”

These are the commands that are run while the condition is true. The commands inside the braces are run in order. When they have all been run, the program goes back to check the condition again. If the (condition) is still true, the loop continues and the {commands} are run again. In the code shown above, the {commands} are to run both of the robot’s motors at full power forward, and the program will do that as long as the touch sensor remains unpressed.

End of Section

The while() loop allows the program to make a decision about program flow, based on a true-or-false statement. It works by checking to see if a (condition) is true, then, if it is true, running a group of {commands}, and looping back to recheck the (condition). If the (condition) ever stops being true, the while() loop skips over the {commands}, and moves on to the next section of the program.

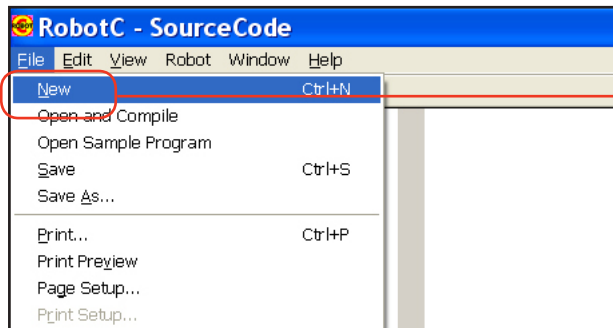
In the wall_touch program, the robot will move forward at full power while the Touch Sensor remains unpressed, then exit the loop and move on to the rest of the program. A well-planned choice of commands to follow the loop tell the robot to back away from the obstacle afterwards.

Sensing

Wall Detection **Putting it Together**

Now that you have examined and worked with the pre-written "Wait for Touch" program, it's time to write one on your own.

1. Start with a new program.



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

2. Remember the program has to do three things:

- Configure the sensor port to recognize a Touch Sensor on Port 1
- Create a while() loop that runs forward while the touch sensor is unpressed
- Back away from the obstacle afterwards

Sensing

Wall Detection **Putting it Together** (cont.)

3. So let's do them in order, starting with the first: configure the sensor port.

3a. Open "Motors and Sensors Setup"
Select Robot > Motors and Sensors Setup to open the Motors and Sensors Setup menu.

3b. Select the A/D Sensors tab
Click the "A/D Sensors" tab on the Motors and Sensors Setup menu.

3c. Give S1 the Name "bumper"
In the "Name" box next to S1, type "bumper".

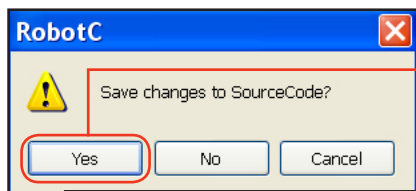
3d. Designate S1 as a Touch Sensor
Select "Touch" from the dropdown box in the "Type" area.

3e. Click OK
Click the "OK" button to save your changes.

Sensing

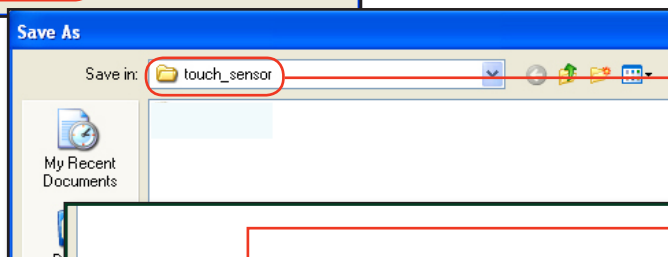
Wall Detection **Putting it Together** (cont.)

4. ROBOTC will want you to save your program at this point. Save your program with your other programs as "touch1".



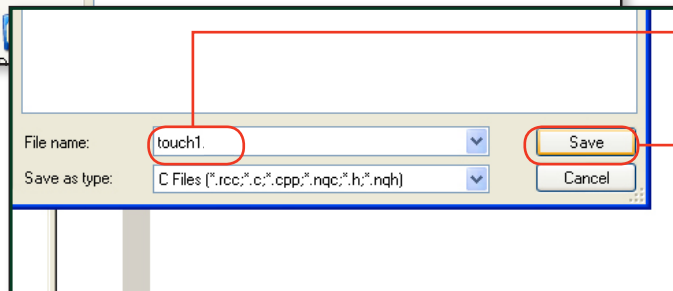
1a. Select "Yes"

Save your program when prompted.



1b. Browse to an appropriate folder

Browse to or create an appropriately named folder within your program folder to save your program.



1c. Name program

Give this program the name "touch1".

1d. Save

Click Save.

Sensing

Wall Detection **Putting it Together** (cont.)

5. Create task main().

```
Auto const tSensors bumper = (tSensors) S1;
Auto /*!!CLICK to edit 'wizard' created sensor
1
1 task main()
1 {
1
1
1
1 }
```

5. Add this code

Create the basic task main() {}.

6. Create the while() loop.

```
Auto const tSensors bumper = (tSensors) S1;
Auto /*!!CLICK to edit 'wizard' created sensor
1
1 task main()
1 {
1
1 while()
1 {
1
1
1
1
1 }
1 }
```

6. Add this code

Add the while() loop: the word "while", the parentheses to hold the condition, and the curly braces to hold the commands.

Checkpoint

Your program should now look like this, with a while() loop within task main(). Line numbers will not update until you compile, so the line of 1s is normal.

```
Auto const tSensors bumper = (tSensors) S1;
Auto /*!!CLICK to edit 'wizard' created sensor
1
1 task main()
1 {
1
1 while()
1 {
1
1
1
1
1 }
1 }
```

Sensing

Wall Detection **Putting it Together** (cont.)

7. Write the condition.

```
Auto const tSensors bumper = (tSensors) S1;
Auto /*!!CLICK to edit 'wizard' created sensor
1
1 task main()
1 {
1
1     while (SensorValue(bumper) == 0)
1     {
1
1
1     }
1
1 }
```

6. Add this code

The condition should test whether the Touch Sensor is unpressed. Thus, the condition is that the Touch Sensor value is equal to zero. Recall that == means "is equal to".

7. Tell the robot what to do while the Touch Sensor is unpressed: go forward at a prudent 50% power (since are expecting to run into an object at some point).

```
Auto const tSensors bumper = (tSensors) S1;
Auto /*!!CLICK to edit 'wizard' created sensor
1
1 task main()
1 {
1
1     while (SensorValue(bumper) == 0)
1     {
1
1         motor[motorC] = 50;
1         motor[motorB] = 50;
1
1     }
1
1 }
```

6. Add this code

Turn Motors A and B on forward at full speed. Because this code is inside the while loop's {} braces, they will be run repeatedly as long as the condition remains true.

Sensing

Wall Detection **Putting it Together** (cont.)

9. Tell the robot what to do after the Touch Sensor is pressed and the while loop ends.

```
Auto const tSensors bumper = (tSensors) S1;
Auto /*!!CLICK to edit 'wizard' created sensor
1
1 task main()
1 {
1
1 while (SensorValue (bumper) == 0)
1 {
1
1 motor[motorC] = 50;
1 motor[motorB] = 50;
1
1 }
1
1 motor[motorC] = -50;
1 motor[motorB] = -50;
1 wait1Msec (1000);
1
1 }
```

9a. Add this code

Run motors A and B backward at 50% power. Because this code comes after the } of the while loop, it will be run only after the loop is done.

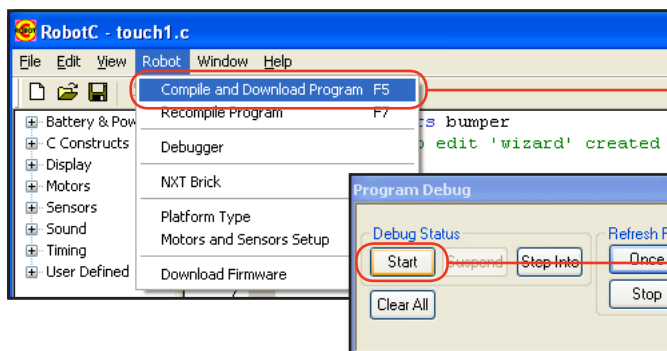
Thus, the robot will back up AFTER the loop ends.

9b. Add this code

Leave the motors running for 1 second.

End of Section

Download and run your program. Congratulations, you have now programmed your robot to use a sensor to detect and respond to its environment! In fact, you've just created your first true robot. The ability to use sensor feedback to govern its own behavior is what sets a robot apart from other machines.



Download the program

Click Robot > Download Program.

Run the program

Click "Start" on the onscreen Program Debug window.



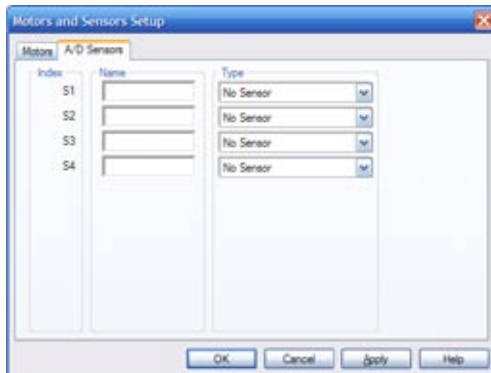
Forward until touch

The robot runs forward as dictated by the while() loop, then, when the touch sensor is pressed and the loop ends, the program continues on to the backing-up commands.

Sensing

Wall Detection (Touch) Quiz

NAME _____ DATE _____



1. The recommended method of configuring sensors in ROBOTC is to use the Motors and Sensors Setup menu shown above.
- True
 - False

2. What form does feedback from the Touch Sensor take?
- A number between 0 and 255, indicating how hard the button is being pressed.
 - A number, either 0 or 1, indicating Pressed or Not Pressed.
 - Pounds per square inch of pressure.
 - A or B, depending on how the user configures the sensor.

3. In plain English (or pseudocode), describe what the following code does.

```

1 while (SensorValue(touchSensor) == 1)
2 {
3     motor[motorC] = 100;
4     motor[motorB] = 0;
5 }

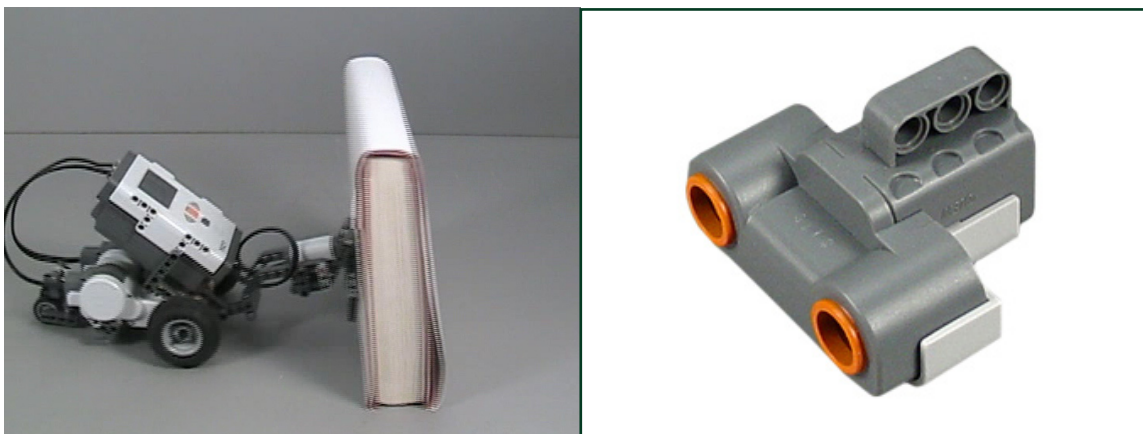
```

4. In general, when does a while loop run the body of its code?
- When its condition is true.
 - When its condition is false.
 - When the sensor gives feedback above the value of threshold.
 - If the motor has been given a power level and assigned a wait state in milliseconds.

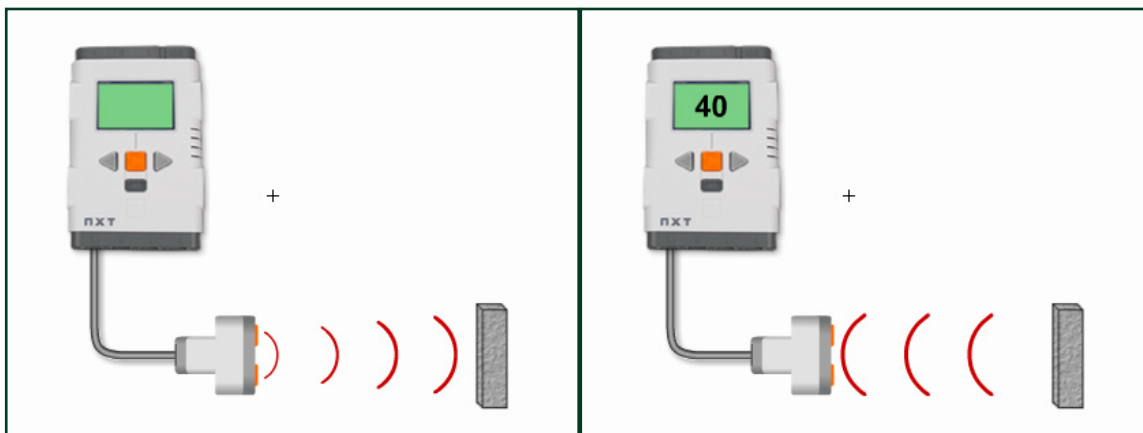
Sensing

Wall Detection **A Sonic Sojourn**

Robots are precise, reliable, intelligent machines, but only when they are programmed to both sense and respond appropriately. Using a sensor which can only detect an obstacle by contact has drawbacks. You would rather not have to bump into something to know it's there, and neither would your robot.



Above right is an Ultrasonic Sensor. Using the same physical principle that a bat or a submarine uses to find its way around, the Ultrasonic Sensor measures distances using sound. It then tells the robot how far away the nearest object in front of it is.



Ultrasonic Sensor detecting a wall (1)

The Ultrasonic Sensor sends out ultrasonic sound waves.

Ultrasonic Sensor detecting a wall (2)

The sound waves hit an obstacle and deflect back. The Ultrasonic Sensor receives the deflected sound waves, then calculates the difference between the time it sent the sound waves and the time it received them. Since the waves travel at a known speed (the speed of sound), the Ultrasonic Sensor can then calculate the distance to the obstacle (in this case, 40 centimeters).

The program you'll write in this lesson will work in a very similar way to the Touch Sensor program you wrote in the previous unit, but instead of using a Touch Sensor to detect obstacles by contact, it will use an Ultrasonic Sensor to detect them at a distance.

Sensing

Wall Detection **A Sonic Sojourn** (cont.)

In this lesson, you will learn to use feedback from an Ultrasonic Sensor to make the robot detect a solid object and stop when it's 25 cm away.

1. Build the Ultrasonic Sensor attachment, and connect it to your robot.



1. Build the Ultrasonic Sensor attachment

Building instructions are available through the main lesson menu. Connect the Ultrasonic Sensor to port 1.

2. Open the "wall_touch" program you wrote for the previous section.

2a. Open Program
Select File > Open and Compile to retrieve your old program.

2b. Select the program
Select "wall_touch".

2c. Open the program
Press Open to open the saved program.

Sensing

Wall Detection **A Sonic Sojourn** (cont.)

Checkpoint

The program should look like the one below.

```

RobotC - wall_touch.c
File Edit View Robot Window Help
Battery & Power Con Auto const tSensors bumper
Bluetooth Auto /*!!CLICK to edit 'wizard' created
Buttons Auto
C Constructs Auto
Datalog 37
Debug 37
Display 38
File Access 39
IO Map Access 39
task main()
{
while(SensorValue(bumper) == 0)
{

```

3. Save this program under a new name, "sonar1".

3a. Save program as...
Select File > Save As... to save your program under a new name.

3b. Browse
Browse to and/or create an appropriate folder.

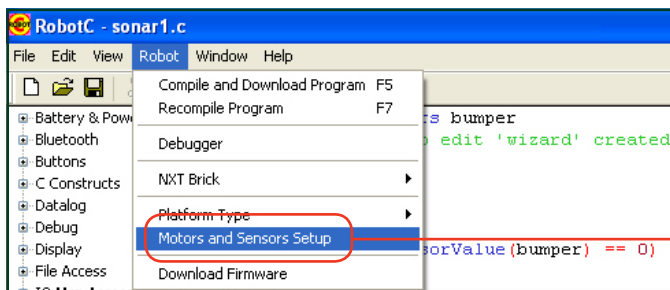
3c. Name the program
Give this program the name "sonar1".

3d. Save the program
Press Save to save the program with the new name.

Sensing

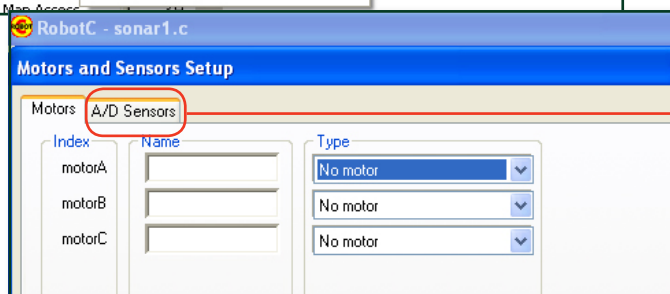
Wall Detection **A Sonic Sojourn** (cont.)

4. Open the Motors and Sensors Setup menu, and go to the Sensors tab.



4a. Open "Motors and Sensors Setup"

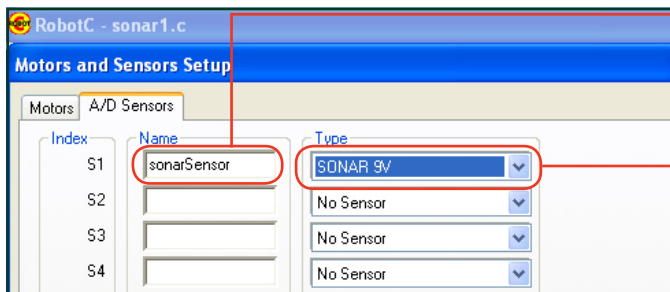
Select Robot > Motors and Sensors Setup to open the Motors and Sensors Setup menu.



4b. Select the A/D Sensors tab

Click the "A/D Sensors" tab" on the Motors and Sensors Setup menu.

5. Use the Motors and Sensors Setup interface to name the S1 sensor "sonarSensor", then select "SONAR 9V" as its type.



5a. Name sensor "sonarSensor"

Enter the name "sonarSensor" in the S1 name box.

5b. Make type "SONAR 9V"

Use the dropdown box to make "SONAR 9V" the sensor type.

Sensing

Wall Detection **A Sonic Sojourn** (cont.)

Checkpoint

Your program should look like this. The while() loop is the focal point of its structure.

```

Auto const tSensors sonarSensor = (tSensors) S1;
Auto /**!!CLICK to edit 'wizard' created sensor
1
2 task main()
3 {
4
5   while (SensorValue (bumper) == 0)
6   {
7
8     motor[motorC] = 50;
9     motor[motorB] = 50;
10
11  }
12
13  motor[motorC] = -50;
14  motor[motorB] = -50;
15  wait1Msec (2000);
16
17 }

```

while

The keyword while signals the beginning of the while loop.

The (condition)

As long as the (condition) is satisfied, the loop will continue repeating.

The {commands}

These commands are repeated over and over while the (condition) remains true.

The program uses the while() loop to check a certain (condition) to see whether it should keep looping or not. The (condition) right now is satisfied as long as the bumper is 0, or unpressed. The robot keeps running as long as this is true.

But now we're using the Ultrasonic Sensor. Having the (condition) look for a sensor value of 0 no longer makes sense, because the Ultrasonic Sensor can report a large range of values, not just one or zero. Remember, the Ultrasonic Sensor measures distance. It gives you a number that indicates the number of centimeters to the nearest detectable object in front of the sensor. It could be 1, 250, or anything in between.

The while() loop, however, doesn't want 250 different values, it just wants to make one decision: continue looping or go on to the next section of the program. The task is to get the robot to stop around 25 cm away from the obstacle. Ask yourself when the robot needs to run, and when it needs to stop. "The robot should run while..."

We'd like the robot to move forward while it is more than 25 cm away from the box, that is, while the distance to the box is greater than 25 (centimeters). Once the robot gets closer than 25cm, it should stop and move on to the next part of the program.

So, let's try that.

Sensing

Wall Detection **A Sonic Sojourn** (cont.)

5. Change the loop's condition to make it run while the Ultrasonic Sensor's value is greater than 25cm.

```

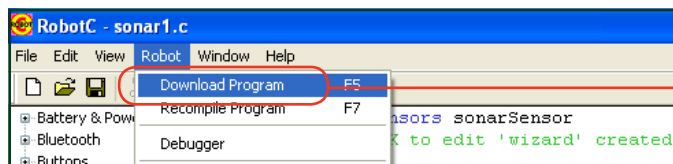
Auto const tSensors sonarSensor = (tSensors) S1;
Auto /*!!CLICK to edit 'wizard' created sensor
1
2 task main()
3 {
4
5     while (SensorValue(sonarSensor) > 25)
6     {
7
8         motor[motorC] = 50;
9         motor[motorB] = 50;
10
11     }
12
13     motor[motorC] = -50;
14     motor[motorB] = -50;
15     wait1Msec(2000);
16
17 }

```

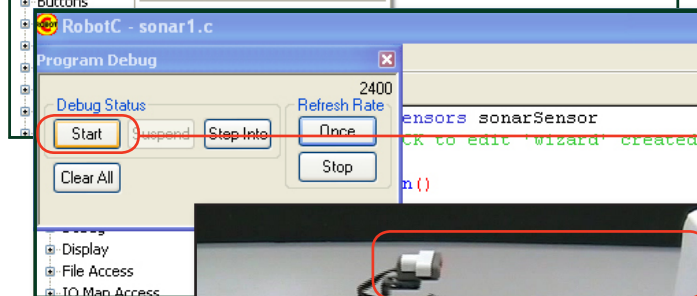
5b. Change sensor name
Change the sensor name in the while () loop condition to "sonarSensor".

5b. Modify this code
Change the while() loop condition's value so that it will check whether the sonarSensor's value is greater than 25 cm.

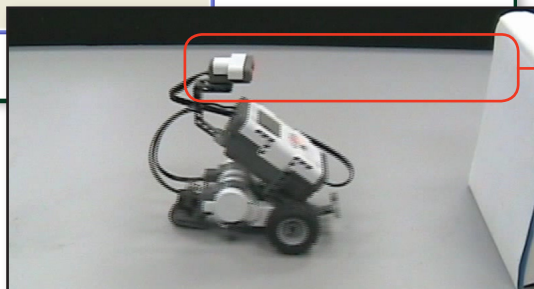
6. Download and run the program. Disconnect the robot and move it onto the course if needed.



6a. Download the program
Click Robot > Download Program.



6b. Run the program
Click "Start" on the onscreen Program Debug window, or use the NXT's on-brick menus.



6c. 25cm Stop
The robot runs forward until the Ultrasonic Sensor detects an object < 25 cm away.

Sensing

Wall Detection **A Sonic Sojourn** (cont.)

7. So we've succeeded in making the robot stop when it's 25 centimeters from an obstacle. Now let's try making the robot stop at some other distance from an obstacle.

```

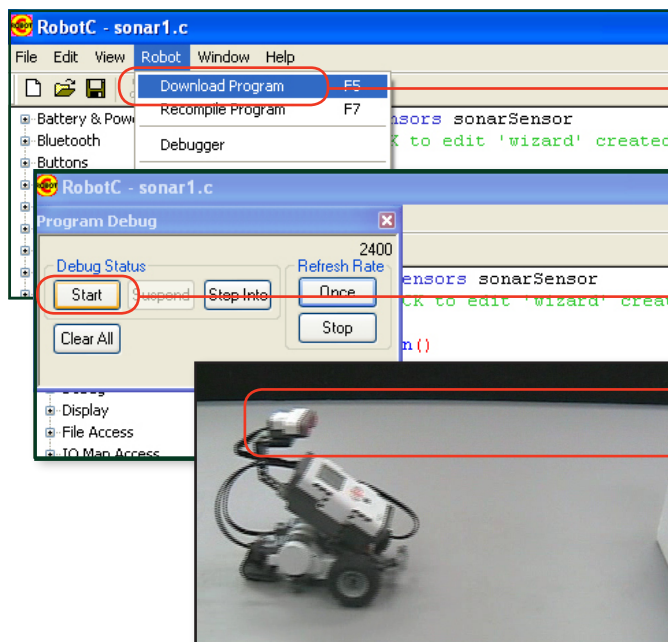
Auto const tSensors sonarSensor = (tSensors) S1;
Auto /*!!CLICK to edit 'wizard' created sensor
1
2 task main()
3 {
4
5     while (SensorValue (sonarSensor) > 40)
6     {
7
8         motor[motorC] = 50;
9         motor[motorB] = 50;
10
11    }
12
13    motor[motorC] = -50;
14    motor[motorB] = -50;
15    wait1Msec (2000);
16
17 }

```

7. Modify this code

Change the while() loop condition's value so that it will check whether the sonarSensor's value is greater than 40 cm.

8. Download and run the program. Disconnect the robot and move it onto the course if needed.



8a. Download the program

Click Robot > Download Program.

8b. Run the program

Click "Start" on the onscreen Program Debug window.

8c. 40cm Stop

The robot runs forward until the Ultrasonic Sensor detects an object < 40 cm away.

Sensing

Wall Detection **A Sonic Sojourn** (cont.)

End of Section

You have modified your program to stop when the robot detects an object closer than a specified distance.

The number that you use to determine how far the robot stops is called a threshold. Thresholds are values that set a cutoff in a range of values, so that even though there are many possible values, every one of them will fall either above the threshold or below it.

In the case of the Ultrasonic Sensor, we set the threshold to 25 in our initial program, and made the distinction that values “greater than 25” will let the loop continue running, while values less than or equal to 25 will make the loop stop.

Then we changed the threshold to a different distance value, and saw how it affected the robot’s behavior. By using thresholds, we can make use of the range of values an Ultrasonic Sensor provides to make a robot stop at whatever distance from an obstacle we want.

Sensing

Wall Detection (Ultrasonic) Quiz

NAME _____ DATE _____

1. The Ultrasonic sensor uses sound to determine:

- a. Direction
 - b. Contact
 - c. Temperature
 - d. Distance
-

2. In ROBOTC, the _____ command is used to find the value of a sensor.

- a. `SensorDistance(sensor_name)`
 - b. `SensorValue(sensor_name)`
 - c. `Sensor(sensor_name)`
 - d. `SensorMeasurement(sensor_name)`
-

3. In the ROBOTC program, it is not necessary to specify the units returned by the ultrasonic sensor because the NXT Ultrasonic Sensor always measures in:

- a. inches.
 - b. centimeters.
 - c. fractional units.
 - d. decimal units.
-

4. The ultrasonic sensor sends and then receives the deflected sound waves and uses the difference between the time sent and time received to calculate the distance from an object.

- a. True
 - b. False
-

5. A robot at the museum is programmed to ask visitors to please step back if they come within six feet of a very fragile glass sculpture.

How might a threshold be used to implement this behavior?

Sensing

Forward Until Dark **Light Sensor**

In this lesson, you will learn how the Light Sensor works, and how its feedback compares to the Touch and Ultrasonic Sonar sensors.



Detects: Physical contact

Feedback: 0 if unpressed, 1 if pressed

Typical use: Bumper

Sample code: `while (SensorValue(touchSensor) == 0)`
will run the while() loop as long as the touch sensor is not pressed.

Touch Sensor

The Touch Sensor detects physical contact with the orange trigger, and returns a SensorValue of 1 if it is pressed in, or 0 if it is not.



Detects: Distance to object

Feedback: Range to object in centimeters (1-250)

Typical use: Obstacle detection and avoidance

Sample code: `while (SensorValue(sonarSensor) > 25)`
will run the while() loop as long as there is no object detected within 25 cm.

Ultrasonic Sensor

The Ultrasonic (sometimes called Sonar) Sensor sends out pulses of sound and measures the time it takes for the sound waves to bounce off an object and return. Since the speed of sound is known, the sensor calculates the distance based on the time, and reports the distance in centimeters as its SensorValue

Sensing

Forward until Dark **Light Sensor** (cont.)

And now, let's look at a new sensor.



Detects: Reflected + Ambient light

Feedback: Brightness (0-100)

Typical use: Line detection

Sample code: `while (SensorValue(LightSensor) > 40)`
will run the `while()` loop as long as the light sensor value remains brighter than 40.

Light Sensor

The Light Sensor (in the normal Active mode) shines a light out in a cone in front of it, and measures how much light comes back to it, from either reflection or ambient sources. See additional explanation below.

This is the Light Sensor. When turned on, it shines a cone of red light out from the red LED, and measures how much of it comes back into the light detector through the clear lens.



Red lens

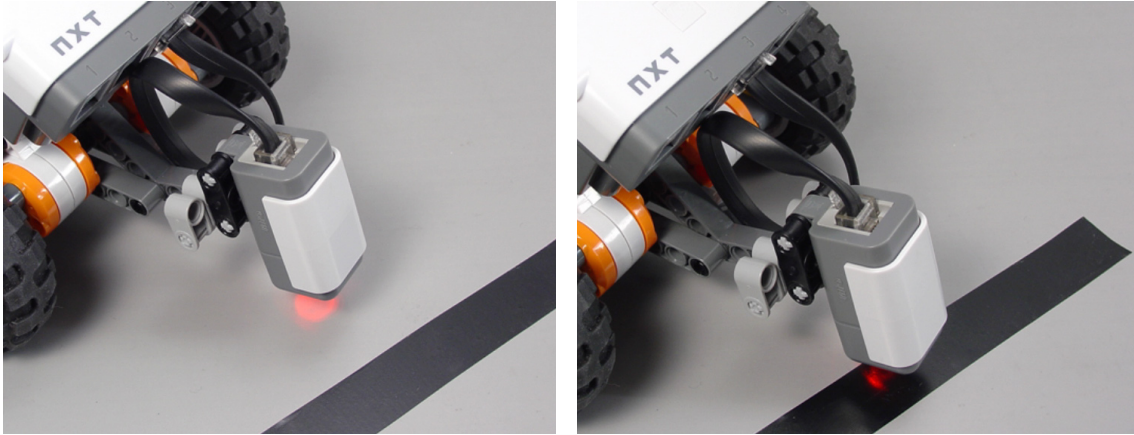
A cone of red light shines out from the red LED.

Clear lens

A light detector measures how much light comes back.

Sensing

Forward until Dark **Light Sensor** (cont.)



The light sensor can detect the basic colors of objects and surfaces by aiming directly at them at close range. Light-colored surfaces, like this bright grey table, reflect a large amount of the light, and produce a high sensor reading. Dark-colored surfaces, like this strip of black electrical tape, reflect very little light, and produce a low sensor reading.

High readings vs. low readings can therefore be used to distinguish light surfaces from dark ones. To make this work for the while() loop, we'll need to use the same technique we used with the Ultrasonic Sensor: set a threshold value to create a "cutoff" point between light and dark.

The sensor gives a light intensity reading of 0-100. But unlike the Ultrasonic Sensor, where the number was in centimeters, the Light Sensor's values are relative only, and do not correspond to any set system of units. In fact, any light source – lamps, sunlight, shadows – and even the height of the light sensor off the table can affect how much light the Light Sensor sees for the same surface. So how can you set a fair cutoff (threshold) between light and dark?

In the next section, you will use the NXT's View Mode to see for yourself what sorts of numbers you get for different surfaces. You will use these real-world readings as reference values for light and for dark. Your readings will give you measured "anchors," that take into account the colors of surfaces, and lighting conditions, and will allow you to make a proper choice of threshold.

Sensing

Forward until Dark **Thresholds 201**

Reminder! Light sensor readings and other numbers used in this printed guide may not be right for your environment. Your room's lighting and the position of the sun and shadows will cause light sensor readings to vary. Measure often!

So higher is brighter, and lower is darker, but if you remember from the last time we worked with a large range of values, we set a threshold to separate the two values we care about. Before we can set a threshold for the Light Sensor, we need to know what values mean "Light" and what values mean "Dark." Let's take some readings to find out.

In this lesson, you will learn how the Light Sensor works, and how its feedback compares to the Touch and Ultrasonic Sonar sensors.

1. View the Reflected Light values in View Mode.



1a. Turn on NXT

Turn on your NXT if it is not already on.



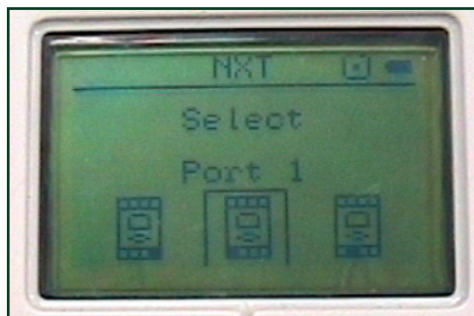
1b. Navigate to View Mode

Use the left and right arrow buttons to find the View option, and press the Orange button to select it.



1c. Select Reflected Light

Use the left and right arrow buttons to find the Reflected Light option, and press the Orange button to select it.



1d. Select Port 1

Make sure your Light Sensor is plugged into Port 1 on the NXT. Select Port 1 on screen.

Caution! Do not choose "Light Sensor*!"

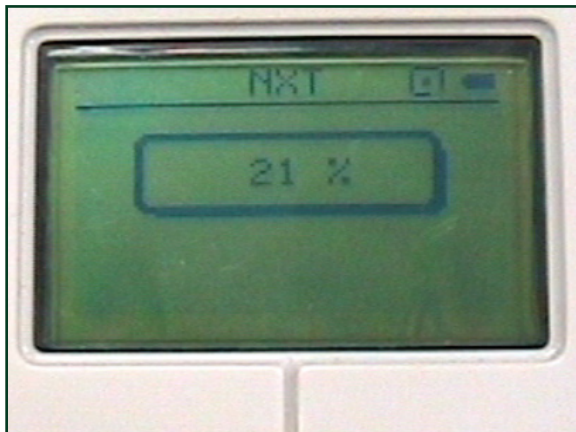
Light Sensor* (and all sensors with a * at the end of their names) refers to the old RCX-generation Light Sensor, and will not provide the correct readings for the NXT Light Sensor.

Sensing

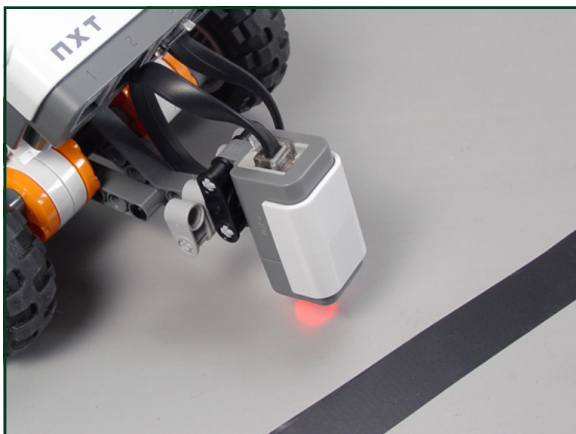
Forward until Dark **Thresholds 201** (cont.)

Checkpoint

You are now seeing the sensor's value live, in real time.

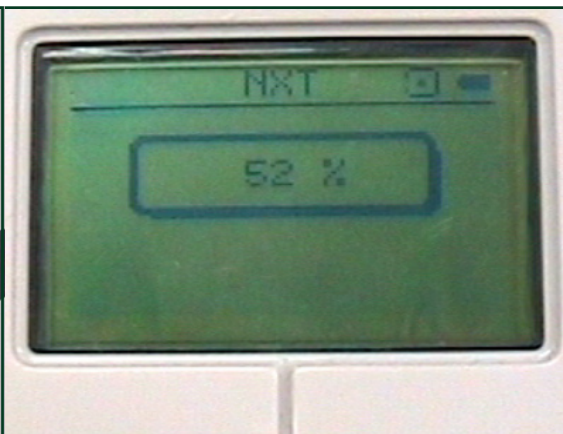


- Place the robot so the light sensor is over the light surface, move your hand away (it can cast a shadow and mess up your readings), and record the reading on the screen.



2a. Place robot over light surface

Position the robot so that the light sensor shines on a light-colored surface.



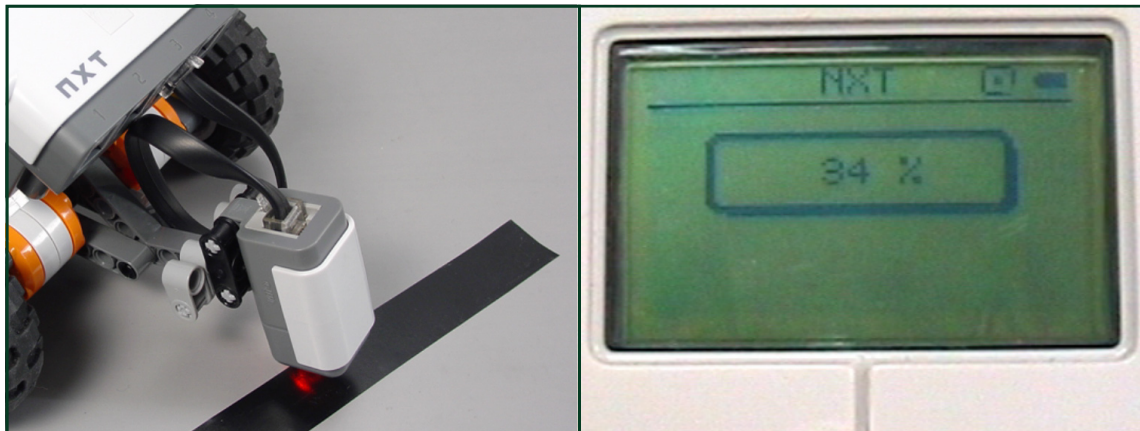
2b. Record "light" sensor value

On a separate sheet of paper, write down the Light Sensor value for a "light" surface.

Sensing

Forward until Dark **Thresholds 201** (cont.)

3. Now, place the light sensor over a part of the dark line, and record that reading.



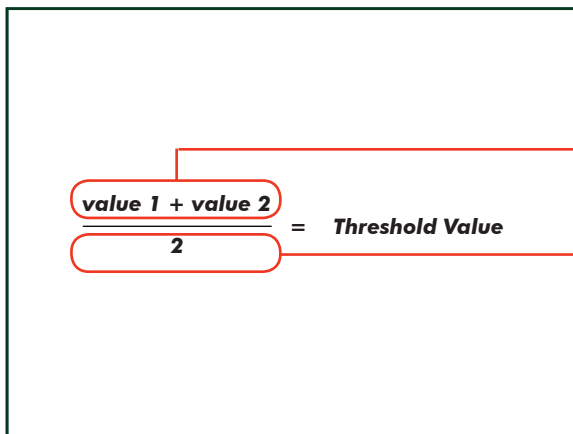
2a. Place robot over dark surface

Position the robot so that the light sensor shines on a dark-colored line.

2b. Record "dark" sensor value

On a separate sheet of paper, write down the Light Sensor value for a "dark" surface.

4. A fair place to set the cutoff line is right in the middle between these two values. That would be the average of these two values.



4a. Add "light" and "dark" values

The first step in finding an average is to find the sum of the two values.

4b. Divide sum by 2

Since there were two values (light and dark), divide the sum by 2 to find the average.

4c. Write down Threshold value

This average is fairly situated, exactly between the other two values. Record this

End of Section

With the threshold set at the point indicated by the red line, the world of light sensor readings can now be divided into two categories: "light" values above the threshold, and "dark" values below the threshold. This distinction will allow your robot to find the line, by looking for the "dark" surface on the ground.

The threshold you have calculated marks the cutoff line for your lighting conditions. Any sensor values above the threshold value will now be considered light, and any below it will be considered dark.

Sensing

Forward until Dark **Wait for Dark**

Reminder! Light sensor readings and other numbers used in this printed guide may not be right for your environment. Your room's lighting and the position of the sun and shadows will cause light sensor readings to vary. Measure often!

So higher is brighter, and lower is darker, but if you remember from the last time we worked with a large range of values, we set a threshold to separate the two values we care about. Before we can set a threshold for the Light Sensor, we need to know what values mean "Light" and what values mean "Dark." Let's take some readings to find out.

In this lesson, you will use the Light Sensor and the Threshold you calculated in the previous section to adapt your Ultrasonic Wall Detector program to detect a dark line instead.

1. Open "sonar1", the Ultrasonic Sensor program from the Wall Detection (Ultrasonic) lesson.

The image shows two overlapping windows from the RobotC software. The top window is titled "RobotC - SourceCode" and has a menu bar with "File", "Edit", "View", "Robot", "Window", and "Help". The "File" menu is open, and "Open and Compile" is highlighted with a red circle. The bottom window is titled "RobotC - sonar1.c" and is an "Open" dialog box. The "Look in:" field shows "ultrasonic". The file list contains "sonar1.c", which is selected with a red circle. The "File name:" field contains "sonar1.c" and the "Files of type:" field is set to "C Files (*.rc;*.c;*.cpp;*.nqc;*.h;*.nh)". The "Open" button is also circled in red.

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

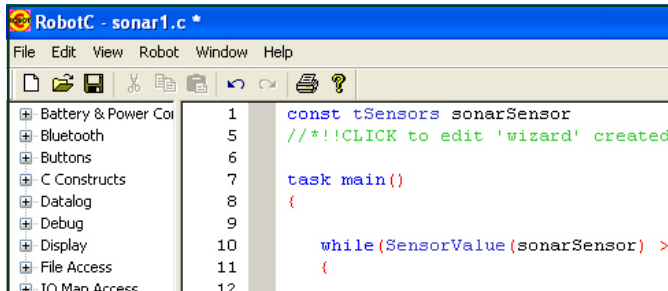
1b. Select the program
Select "sonar1".

2c. Open the program
Press Open to open the saved program.

Sensing

Forward until Dark **Wait for Dark** (cont.)

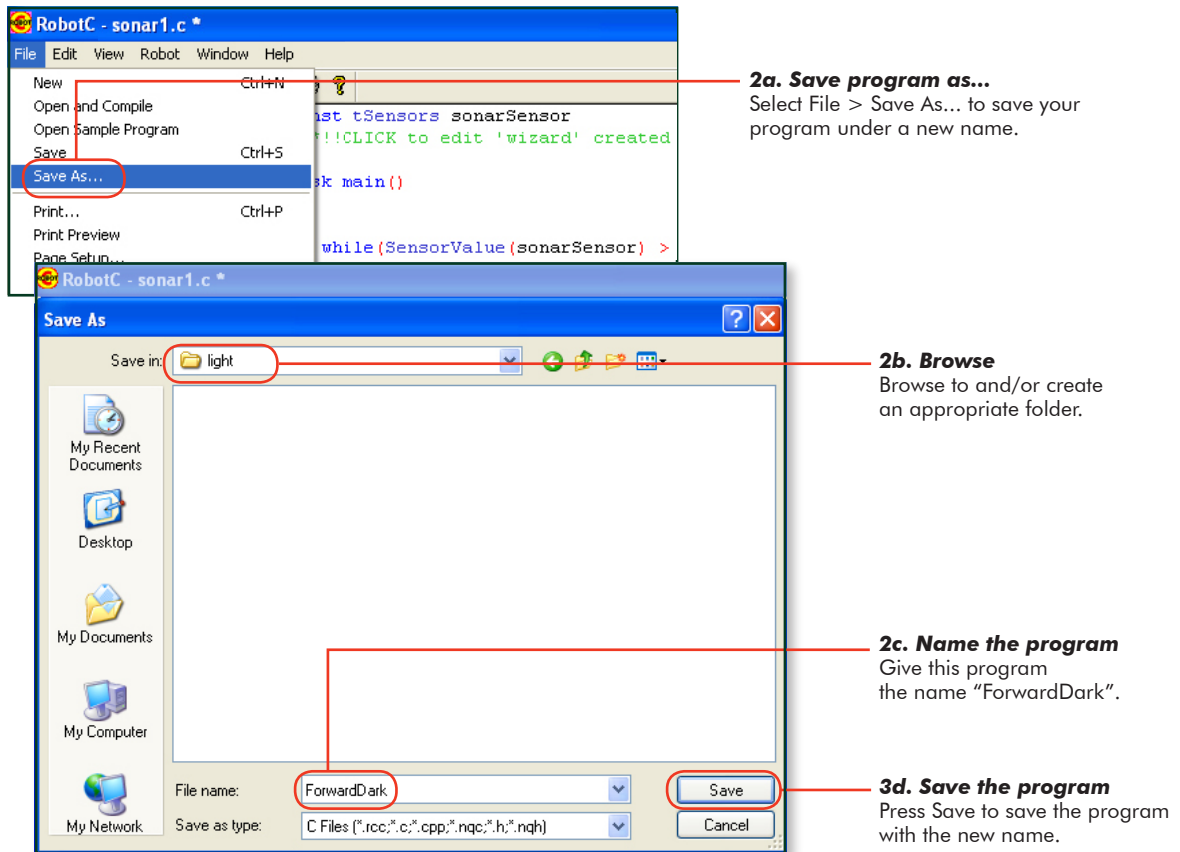
Checkpoint. The program should look like the one below.



```

RobotC - sonar1.c *
File Edit View Robot Window Help
[Icons]
Battery & Power Coi 1 const tSensors sonarSensor
Bluetooth 5 /**!!CLICK to edit 'wizard' created
Buttons 6
C Constructs 7 task main()
Datalog 8 {
Debug 9
Display 10 while (SensorValue (sonarSensor) >
File Access 11 {
IO Man Access 12
  
```

2. Because we're going to be changing the program, save it under the new name "ForwardDark".



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

2b. Browse
Browse to and/or create an appropriate folder.

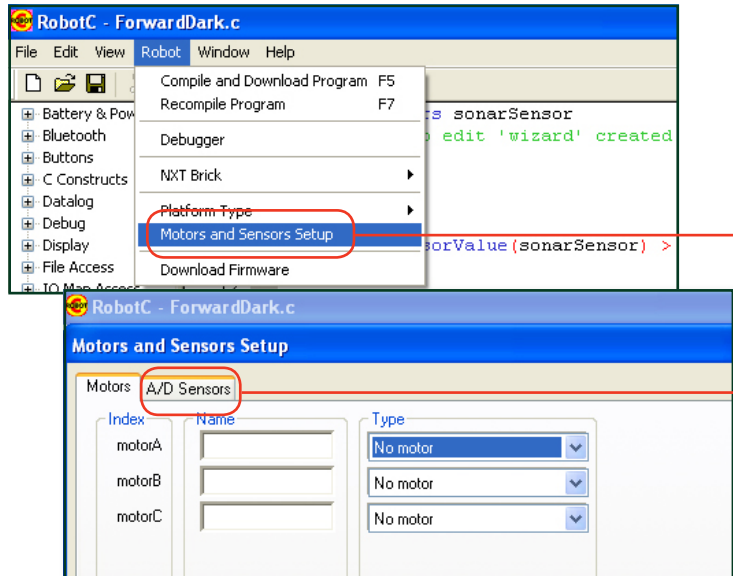
2c. Name the program
Give this program the name "ForwardDark".

3d. Save the program
Press Save to save the program with the new name.

Sensing

Forward until Dark **Wait for Dark** (cont.)

3. Open the Motors and Sensors Setup menu, and go to the Sensors tab.

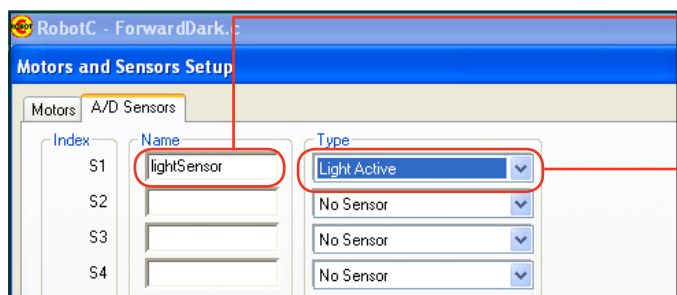


The screenshot shows the RobotC software interface. In the first part, the 'Robot' menu is open, and 'Motors and Sensors Setup' is highlighted. In the second part, the 'Motors and Sensors Setup' window is open, and the 'A/D Sensors' tab is selected.

3a. Open "Motors and Sensors Setup"
Select Robot > Motors and Sensors Setup to open the Motors and Sensors Setup menu.

3b. Select the A/D Sensors tab
Click the "A/D Sensors tab" on the Motors and Sensors Setup menu.

4. Use the Motors and Sensors Setup interface to name the S1 sensor "sonarSensor", then select "SONAR 9V" as its type.



The screenshot shows the 'Motors and Sensors Setup' window with the 'A/D Sensors' tab selected. The S1 sensor is named 'lightSensor' and its type is set to 'Light Active'.

4a. Name sensor "lightSensor"
Enter the name "lightSensor" in the S1 name box.

4b. Make type "Light Active"
Use the dropdown box to make "Light Active" the sensor type.

Sensing

Forward until Dark **Wait for Dark** (cont.)

5. Modify the (condition) in the while() loop to watch for the lightSensor value to be greater than (brighter than) the threshold.

```

Auto const tSensors lightSensor = (tSensors) S1;
Auto /*!!CLICK to edit 'wizard' created sensor
1
2 task main()
3 {
4
5     while (SensorValue(lightSensor) > 40)
6     {
7
8         motor[motorC] = 50;
9         motor[motorB] = 50;
10
11     }
12
13     motor[motorC] = 0;
14     motor[motorB] = 0;
15     wait1Msec(2000);
16
17 }

```

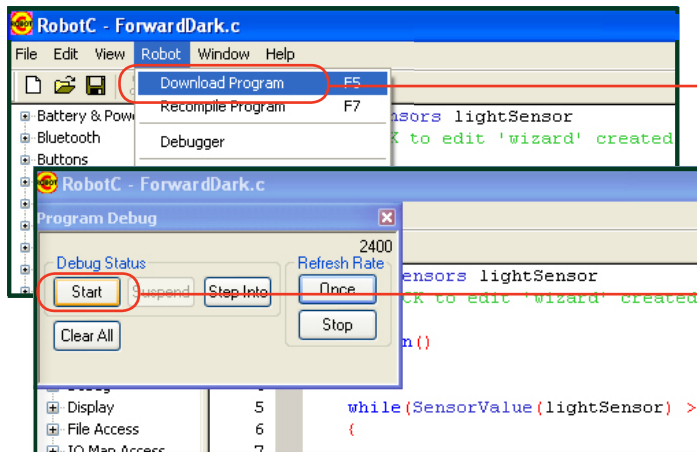
5a. Modify this code

Change the while() loop condition's value so that it will check whether the **Light Sensor's** value is greater than the **threshold value** you calculated in the last lesson.

5b. Modify this code

Change the speed of Motors C and B to 0 so that the robot stops when it reaches a black line, rather than reversing at 50% power.

6. Download and Run the program.



6b. Download the program

Click Robot > Download Program.

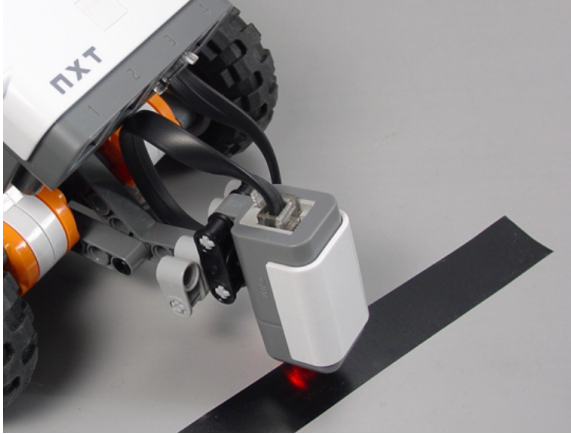
6c. Run the program

Click "Start" on the onscreen Program Debug window.

Tip: If your robot stops immediately or runs past the line without stopping, check your light sensor values using the View mode. Lighting conditions (position of the sun, room lighting) may have changed, and your threshold may need to be adjusted.

Sensing

Forward until Dark **Wait for Dark** (cont.)



End of Section

When the robot sees “dark” (a value below the threshold), the (condition) is no longer satisfied, and the program moves on to the stop commands, causing the robot to stop at the dark line.

As a final exercise, consider what would happen if you were to turn the lights off (or on) in the room where the robot is running. Make your prediction, and test it!

Sensing

Forward Until Dark Quiz

NAME _____ DATE _____

1. One reasonable way of finding a threshold for a light sensor would be to:
- use the output value of the LED.
 - sum up the high and low readings and then divide that by two.
 - use the high reading and subtract the distance traveled.
 - calculate the average of the ambient light in the room.
-

2. What type of light does the NXT light sensor use?
- Reflected halogen light
 - LED
 - Neon light
 - Fluorescent light
-

3. A high number reading from the light sensor could mean:
- the light sensor is seeing a dark surface which reflects a small amount of light.
 - the light sensor is seeing a dark surface which reflects no light.
 - the light sensor is seeing a light surface which reflects a large amount of light.
 - the light sensor was unable to detect either a light or dark surface and cannot make a consistent final reading.
-

4. A standard behavior to move until the robot sees a dark line on a light surface looks like the following code. Writing directly on the code, change the program above to look for a white line on a dark surface instead (assume the threshold value stays the same).

```
1 while(SensorValue(lightSensor) > 45)
2 {
3     motor[motorC] = 75;
4     motor[motorB] = 75;
5 }
```

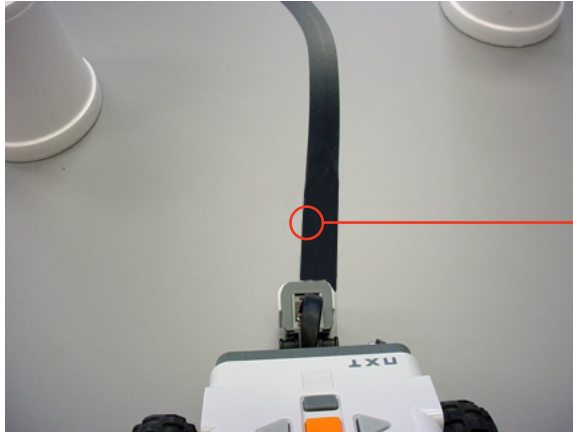
5. What does it mean when the Light Sensor is in "Active Mode"?
- It is actively generating its own light using the built-in emitter.
 - The light sensor is actively controlling the motors.
 - The light source is turned off, and the sensor is actively searching for outside light.
 - The light sensor is broken, and you need to actively find a replacement.

Sensing

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.

Sensing

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

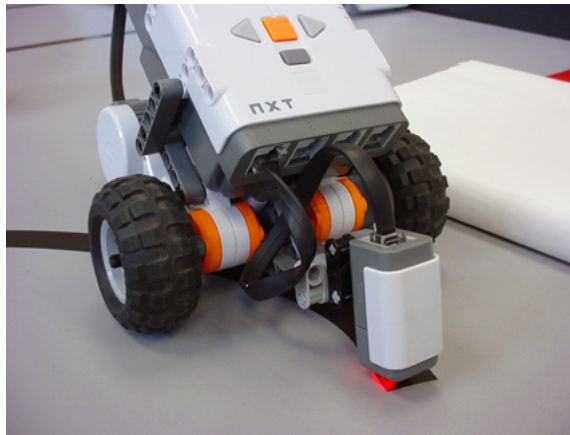
The robot is now over the light surface. The left edge of the line must be to the robot's right.



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:

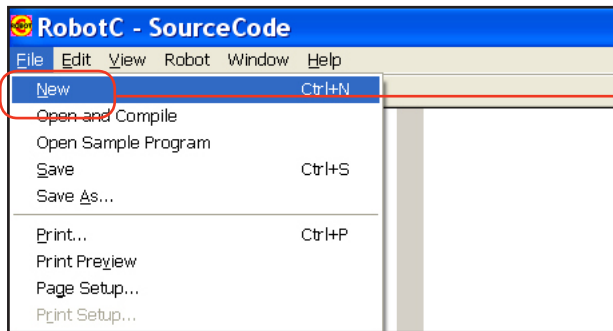
The robot will perform the line track behavior to the end of the line

Sensing

Line Tracking **Basic** (cont.)

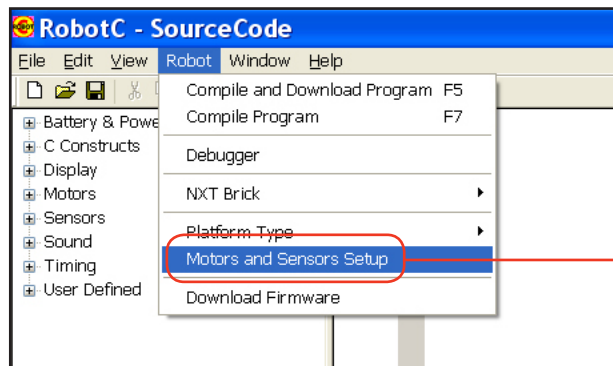
In this lesson you will learn how to use the light sensor to follow a line, using behaviors similar to the Wait for Dark (and Wait for Light) behaviors you have already worked with.

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".

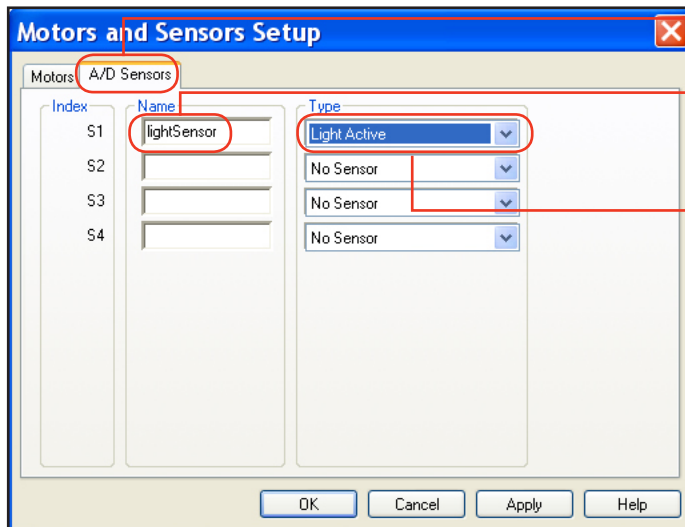


- 2. Open "Motors and Sensors Setup"**
Select Robot > Motors and Sensors Setup to open the Motors and Sensors Setup menu.

Sensing

Line Tracking **Basic** (cont.)

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

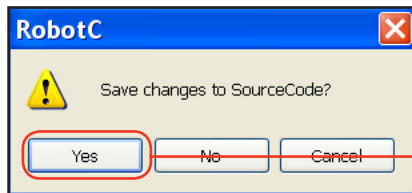


3a. Open A/D Sensors Tab
Click the A/D Sensors tab

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

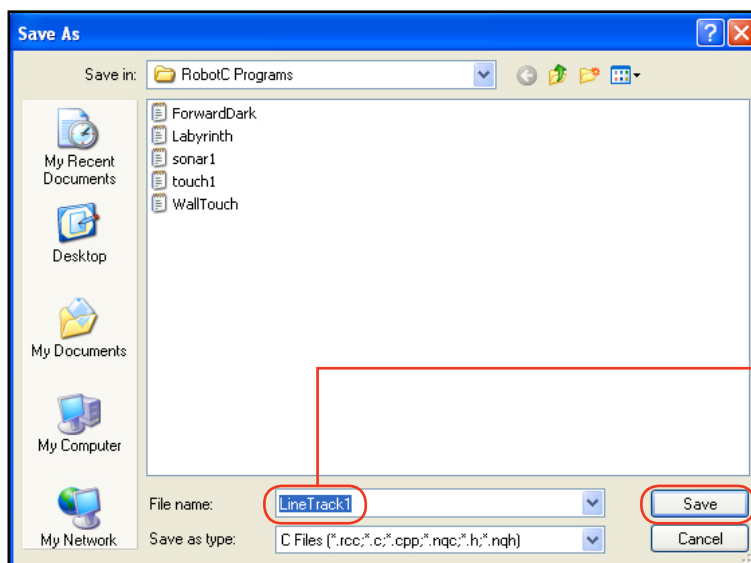
3c. Set Sensor Type
Identify the Sensor Type as a "Light Active" sensor.

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



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

5. Save this program as "LineTrack1".



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

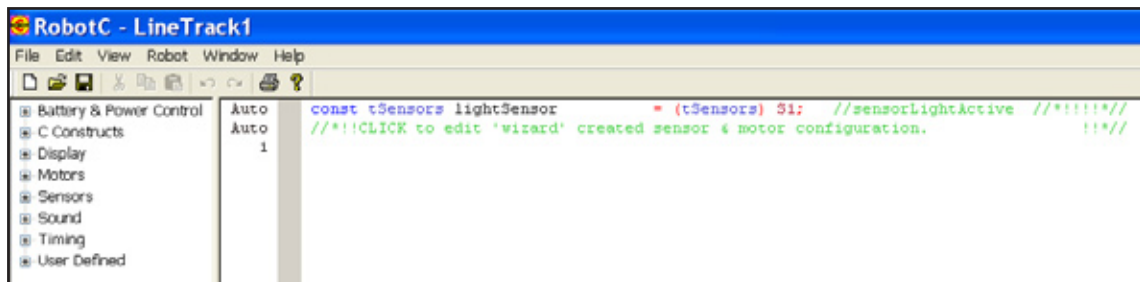
5b. Save the program
Press Save to save the program with the new name.

Sensing

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.



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

```
2 task main()  
3 {  
4  
5  
6 }
```

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).

```
2 task main()  
3 {  
4  
5 while (SensorValue(lightSensor) < 45)  
6 {  
7  
8 motor[motorC] = 0;  
9 motor[motorB] = 80;  
10  
11 }  
12  
13 }
```

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.

Sensing

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 as it continues to see the light table surface (i.e. until it sees the dark line again).

```
2 task main()
3 {
4
5     while (SensorValue(lightSensor) < 45)
6     {
7
8         motor[motorC] = 0;
9         motor[motorB] = 80;
10
11     }
12
13     while (SensorValue(lightSensor) >= 45)
14     {
15
16         motor[motorC] = 80;
17         motor[motorB] = 0;
18
19     }
20
21 }
```

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...)

Sensing

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".

```
2 task main()
3 {
4
5     while(1==1)
6     {
7
8         while(SensorValue(lightSensor) < 45)
9         {
10
11             motor[motorC] = 0;
12             motor[motorB] = 80;
13
14         }
15
16         while(SensorValue(lightSensor) >= 45)
17         {
18
19             motor[motorC] = 80;
20             motor[motorB] = 0;
21
22         }
23
24     }
25
26 }
```

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;
    }
}
```

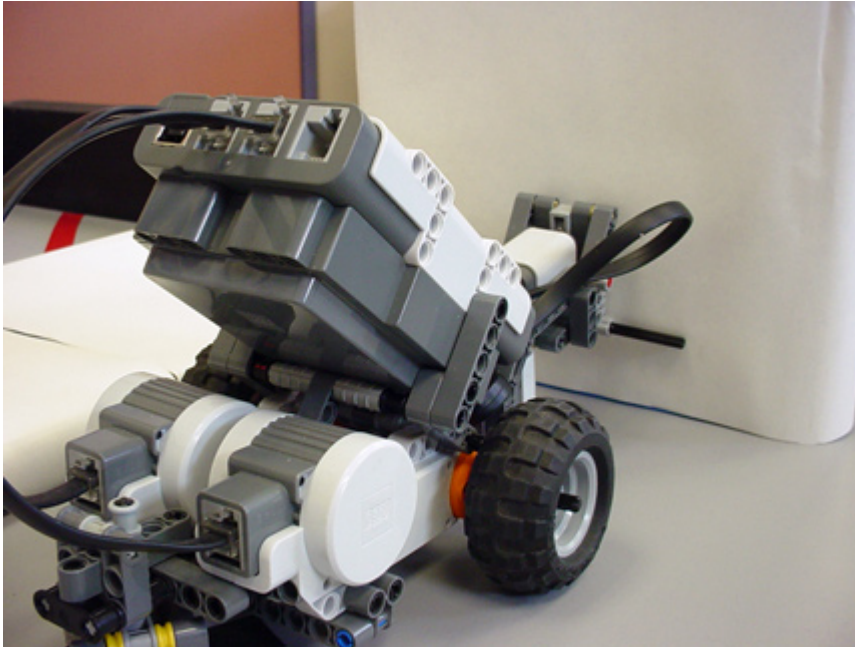
Line Tracking **Basic** (cont.)

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.



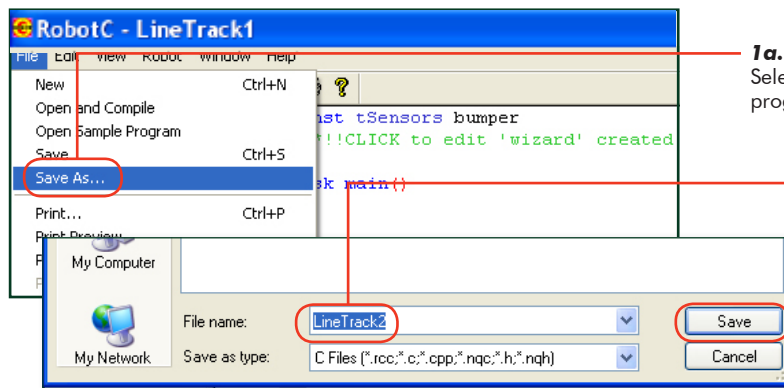
In principle, we should be able to do this pretty easily, all we need to do is change the “looping forever” part to “loop while the touch sensor is unpressed.”

Sensing

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".

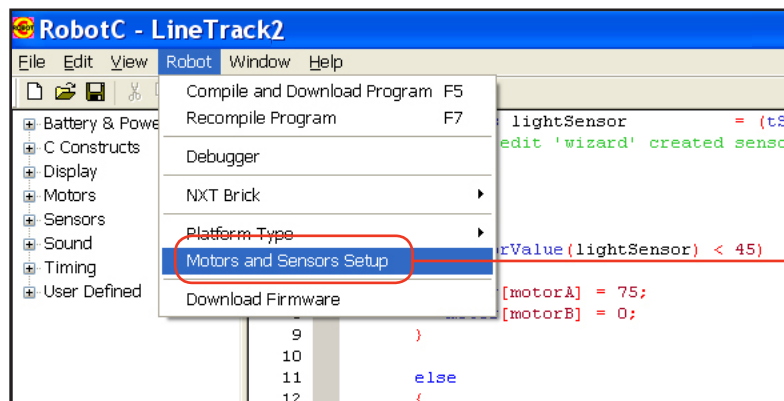


1a. Save program As...
Select File > Save As... to save your program under a new name.

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

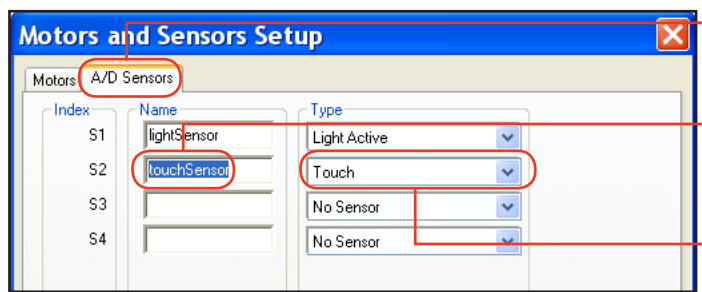
1c. Save the program
Press Save to save the program with the new name.

2. Open the Motors and Sensors Setup menu.



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.



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

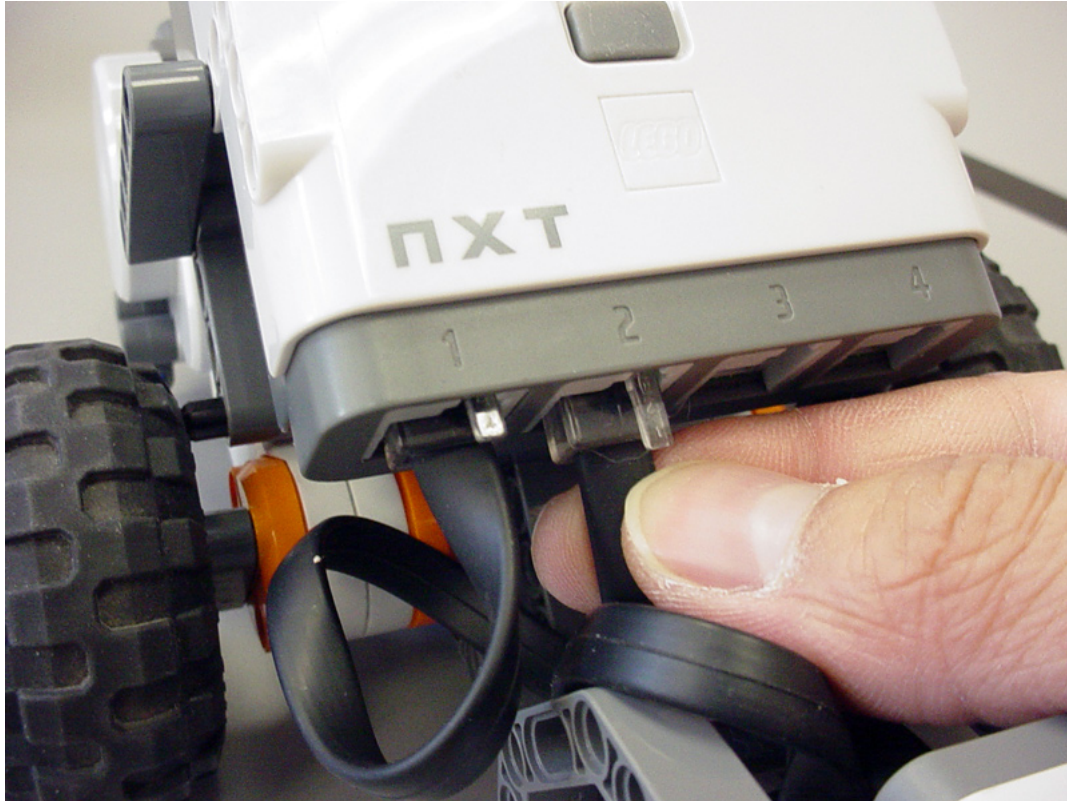
3b. Name the sensor
Name the Touch Sensor on port S2 "touchSensor".

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

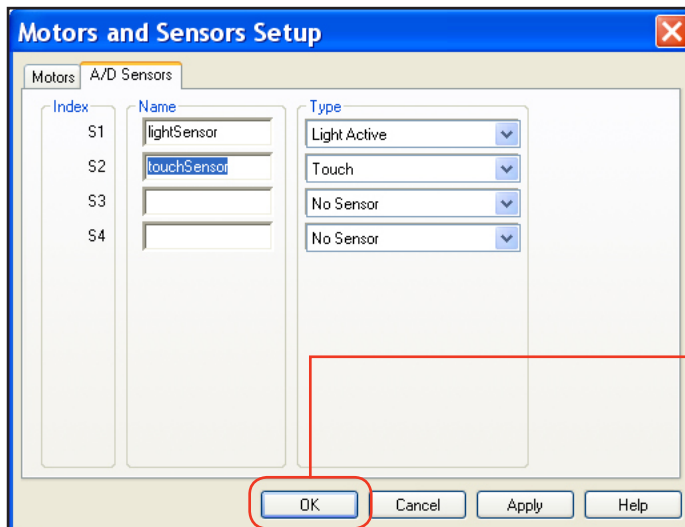
Sensing

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.



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

Sensing

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.

```
2 task main()  
3 {  
4  
5     while (SensorValue(touchSensor) == 0)  
6     {  
7  
8         while (SensorValue(lightSensor) < 45)  
9         {  
10  
11             motor[motorC] = 0;  
12             motor[motorB] = 80;  
13  
14         }  
15  
16         while (SensorValue(lightSensor) >= 45)  
17         {  
18
```

6. Modify this code

Change the condition in parentheses to check whether the “touch sensor is unpressed” instead.

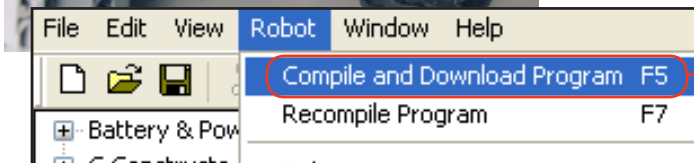
The condition will be true when the touch sensor’s value is equal to 0.

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.



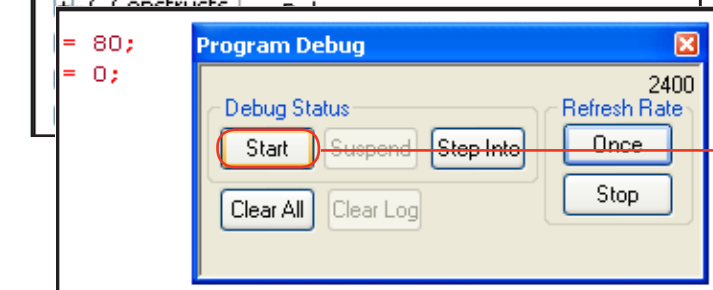
7a. Block up the robot

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.

Sensing

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

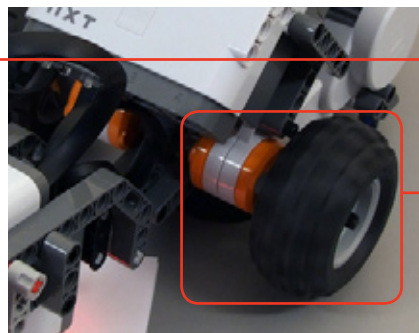
Place a sheet of white paper under the sensor to simulate the robot traveling off the line and onto the light table surface. Watch for the motors to change behaviors accordingly.

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.



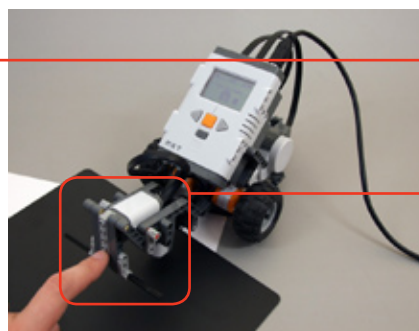
Observe motors

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?

Sensing

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.

```
2 task main()
3 {
4
5   while (SensorValue(touchSensor) == 0)
6   {
7
8     while (SensorValue(lightSensor) < 45)
9     {
10
11       motor[motorC] = 0;
12       motor[motorB] = 80;
13
14     }
15
16   while (SensorValue(lightSensor) >= 45)
17   {
18
```

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?



```
2 task main()
3 {
4
5   while (SensorValue(touchSensor) == 0)
6   {
7
8     while (SensorValue(lightSensor) < 45)
9     {
10
11       motor[motorC] = 0;
12       motor[motorB] = 80;
13
14     }
15
```

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.

Sensing

Line Tracking Better (cont.)

The solution requires a little shift in thinking. The program as it is now involves running through 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.

```
8      if (SensorValue(lightSensor) < 45)
9
10
11         motor[motorC] = 0;
12         motor[motorB] = 80;
13
14     }
15
16     else
17     {
18
19         motor[motorC] = 80;
20         motor[motorB] = 0;
21
22     }
23
```

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!

Sensing

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.

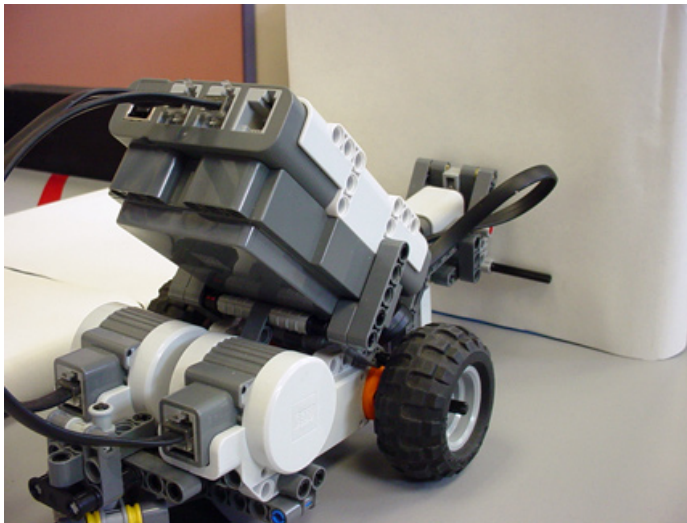
```
15  
16     else  
17     {  
18  
19         motor[motorC] = 80;  
20         motor[motorB] = 0;  
21  
22     }  
23  
24 }  
25  
26     motor[motorC] = 0;  
27     motor[motorB] = 0;  
28  
29 }
```

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?

```

2  task main()
3  {
4      while(SensorValue(touchSensor) == 0)
5      {
6          if(SensorValue(lightSensor) < 45)
7          {
8              motor[motorC] = 0;
9              motor[motorB] = 80;
10             }
11             }
12             }
13             }
14             }
15             }
16             }
17             }
18             }
19             }
20             }
21             }
22             }
23             }
24             }
25             }
26             }
27             }
28             }
29             }
30             }
31             }

```

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.

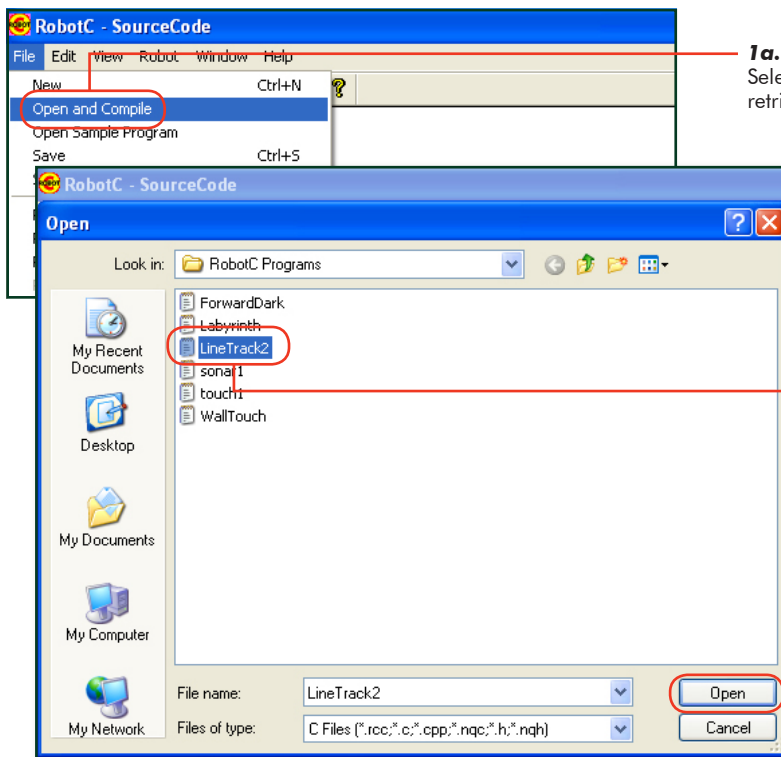
`time1[TimerName]` represents the timer value in milliseconds since the last reset. It is shown here being used to make a while loop run until 5 seconds have elapsed.

Sensing

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".

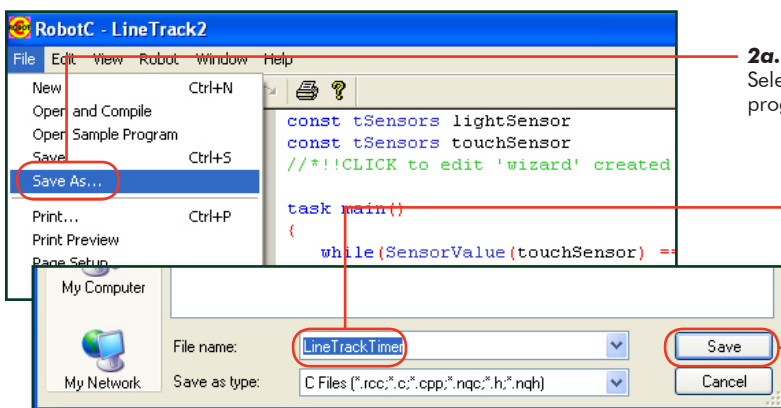


1a. Open Program
Select File > Open and Compile to retrieve your old program.

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")



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.

```
2 task main()
3 {
4
5     while (SensorValue(touchSensor) == 0)
6     {
7
8         if (SensorValue(lightSensor) < 45)
9         {
10
11             motor[motorC] = 0;
12             motor[motorB] = 80;
13
14         }
15
16         else
17         {
18
19             motor[motorC] = 80;
20             motor[motorB] = 0;
21
22         }
23
24     }
25
26     motor[motorC] = 0;
27     motor[motorB] = 0;
28
29 }
```

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

```
2 task main()
3 {
4
5     ClearTimer(T1);
6
7     while (SensorValue(touchSensor) == 0)
8     {
9
10         if (SensorValue(lightSensor) < 45)
```

3. Add this code

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

Sensing

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.

```
2 task main()
3 {
4
5     ClearTimer(T1);
6
7     while(time1[T1] < 3000)
8     {
9
10        if(SensorValue(lightSensor) < 45)
11        {
12
13            motor[motorC] = 0;
14            motor[motorB] = 80;
15
16        }
17
18        else
19        {
20
21            motor[motorC] = 80;
22            motor[motorB] = 0;
```

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**

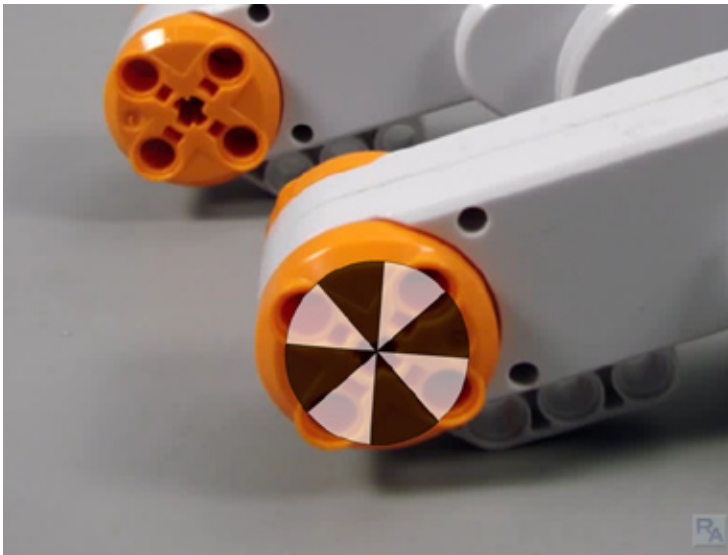
In this lesson we'll find out how to watch for distance, instead of watching for *time* and hoping that the robot moves the correct distance, like in our previous program.



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.

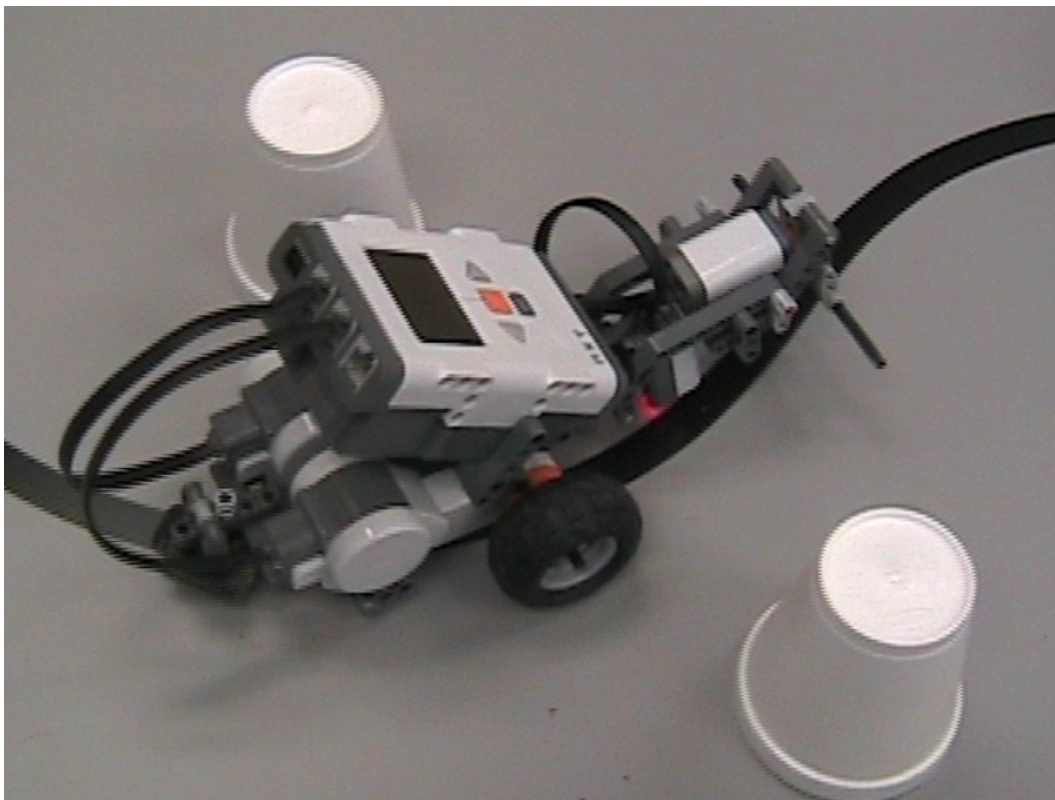


Line Tracking **Rotation** (cont.)

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.



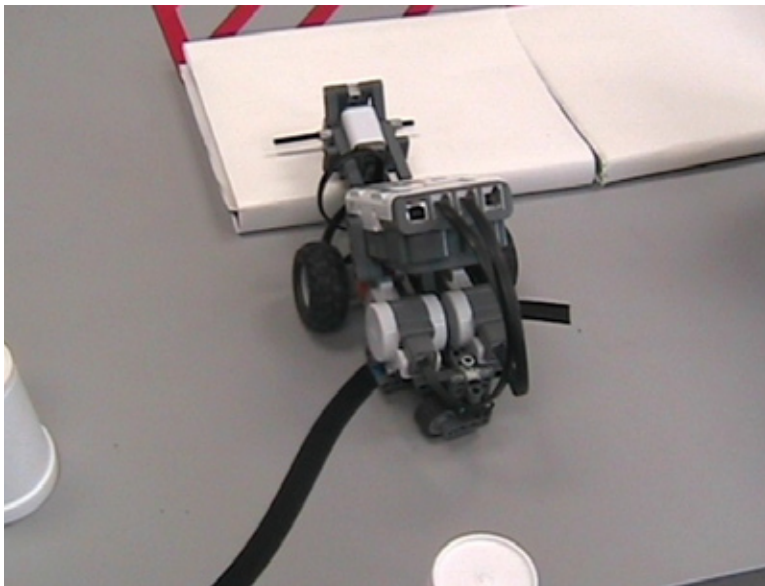
Sensing

Line Tracking **Rotation** (cont.)

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

```
2  task main()
3  {
4
5      while(1 == 1)
6      {
7
8          while(SensorValue(lightSensor) < 45)
9          {
10
11              motor[motorC] = 0;
12              motor[motorB] = 80;
13
14          }
15
16          while(SensorValue(lightSensor) >= 45)
17          {
18
19              motor[motorC] = 80;
20              motor[motorB] = 0;
21
22          }
23
24      }
25
26 }
```

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.

```
7
8     if(SensorValue(lightSensor) < 45)
9     {
10
11         motor[motorC] = 0;
12         motor[motorB] = 80;
13
14     }
15
16     else
17     {
18
19         motor[motorC] = 80;
20         motor[motorB] = 0;
21
22     }
23
24 }
```

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.

```
2 task main()
3 {
4
5     ClearTimer(T1);
6
7     while(time1[T1] < 3000
8     {
9
10        if(SensorValue(lightSensor) < 45)
11        {
12
13            motor[motorC] = 0;
14            motor[motorB] = 80;
15
16        }
17
18        else
19        {
20
```

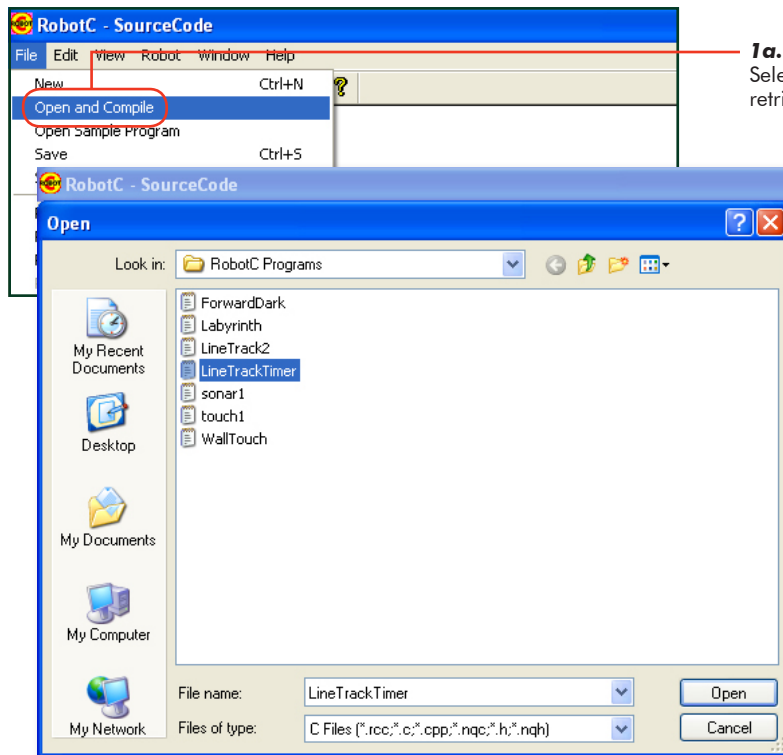
Sensing

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".

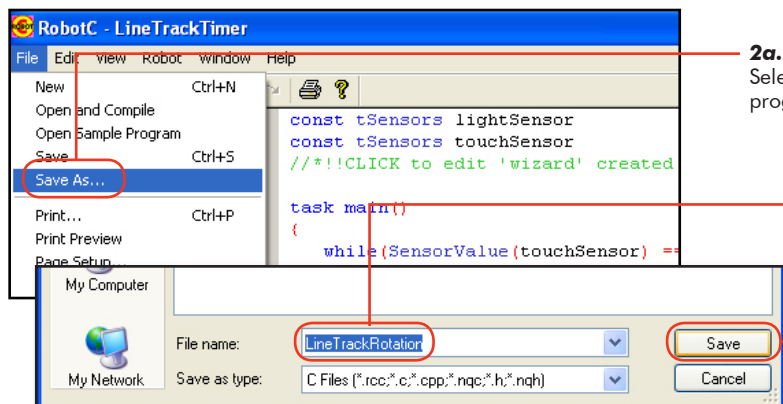


1a. Open Program
Select File > Open and Compile to retrieve your old program.

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".



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

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

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

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.

```
2 task main()
3 {
4
5     ClearTimer(T1);
6
7     while(time1[T1] < 3000)
8     {
9
10        if(SensorValue(lightSensor) < 45)
11        {
12
13            motor[motorC] = 0;
14            motor[motorB] = 80;
15
16        }
17
18        else
19        {
20
21            motor[motorC] = 80;
22            motor[motorB] = 0;
23
24        }
25
26    }
27
28    motor[motorC] = 0;
29    motor[motorB] = 0;
30
31 }
```

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;
```

Sensing

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.

```
2 task main()
3 {
4
5     nMotorEncoder[motorC] = 0;
6
7     while(time1[T1] < 3000)
8     {
9
10        if(SensorValue(lightSensor) < 45)
11        {
12
13            motor[motorC] = 0;
14            motor[motorB] = 80;
15
16        }
17
18        else
19        {
20
21            motor[motorC] = 80;
22            motor[motorB] = 0;
23
24        }
25
26    }
27
28    motor[motorC] = 0;
29    motor[motorB] = 0;
30
31 }
```

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;`

```
2 task main()
3 {
4
5     nMotorEncoder[motorC] = 0;
6     nMotorEncoder[motorB] = 0;
7
8     while(time1[T1] < 3000)
9     {
10
11         if(SensorValue(lightSensor) < 45)
12         {
13
14             motor[motorC] = 0;
15             motor[motorB] = 80;
16
17         }
18
19         else
20         {
21
22             motor[motorC] = 80;
23             motor[motorB] = 0;
24
25         }
26
27     }
28
29     motor[motorC] = 0;
30     motor[motorB] = 0;
31
32 }
```

4. Add this code

Reset the rotation sensor in MotorB to 0 as well.

Line Tracking **Rotation** (cont.)

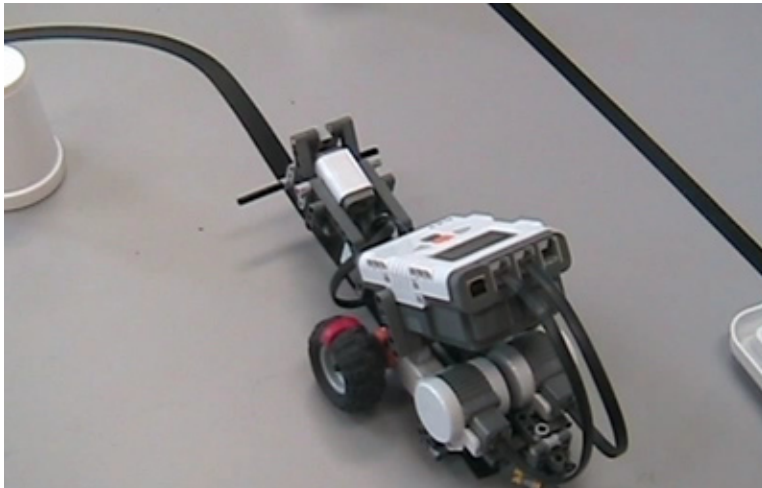
5. The NXT motor encoder measures in degrees, so it will count 360 for every full rotation the motor makes. Change the `while ()` loop's condition to make this loop run while the `nMotorEncoder` value of motorC is less than 1800 degrees, five full rotations.

```
2 task main()
3 {
4
5     nMotorEncoder[motorC] = 0;
6     nMotorEncoder[motorB] = 0;
7
8     while(nMotorEncoder[motorC] < 1800)
9     {
10
11         if(SensorValue(lightSensor) < 45)
12         {
13
14             motor[motorC] = 0;
15             motor[motorB] = 80;
16
17         }
18
19         else
20         {
21
```

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.



Sensing

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.

```
2 task main()
3 {
4
5     nMotorEncoder[motorC] = 0;
6     nMotorEncoder[motorB] = 0;
7
8     while(nMotorEncoder[motorC] < 1800 && nMotorEncoder[motorB] < 1800)
9     {
10
```

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.

Sensing

Line Tracking Quiz

NAME _____ DATE _____

1. Why can nested loops cause a problem in a program?

2. List two ways in which the "Line Tracking (Rotation)" program improves upon the "Line Tracking (Basic)" program. Explain why they are actually improvements.

3. Answer the questions about the following segment of code:

```
1  if(SensorValue(lightSensor) > 45)
2  {
3      motor[motorC] = 75;
4  }
5  else
6  {
7      motor[motorB] = -75;
8  }
```

a. What will the robot do if the light sensor reads a value of 64?

a. What if it reads a value of 45?

Sensing

Speed Based on Volume Values & Assignment (Part 1)

The Sound Sensor is the last of the standard NXT sensors. In essence it's a kind of microphone which senses amplitude (how loud or soft a sound is), but not anything else about it. The Sound Sensor, like the Light Sensor, reports values from 0-100 which do not correspond to any specific standard scale.



Sound Sensor

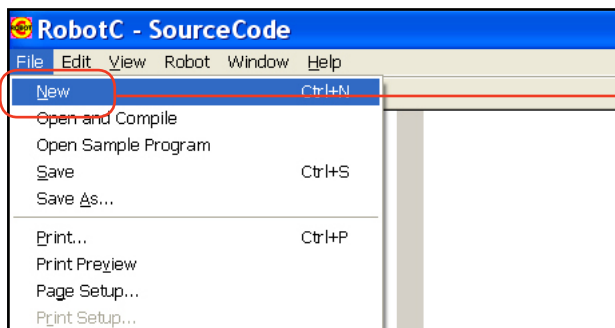
The Sound Sensor has an orange foam pad which resembles a microphone

Sensing

Speed Based on Volume Values and Assignment (Part 1) (cont.)

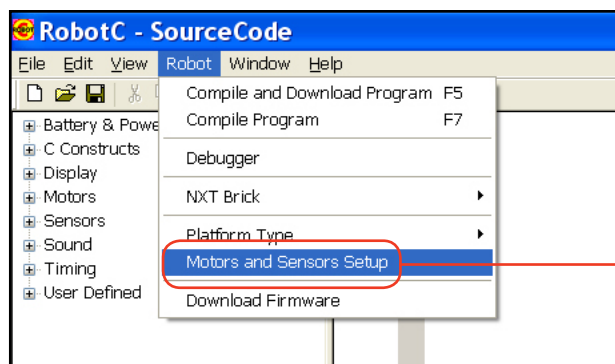
In this lesson you will learn how to use the Sound Sensor to manipulate your robot's motors

1. Start by opening a new program.



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

2. Open the Motors and Sensors Setup menu to configure the Sound Sensor.

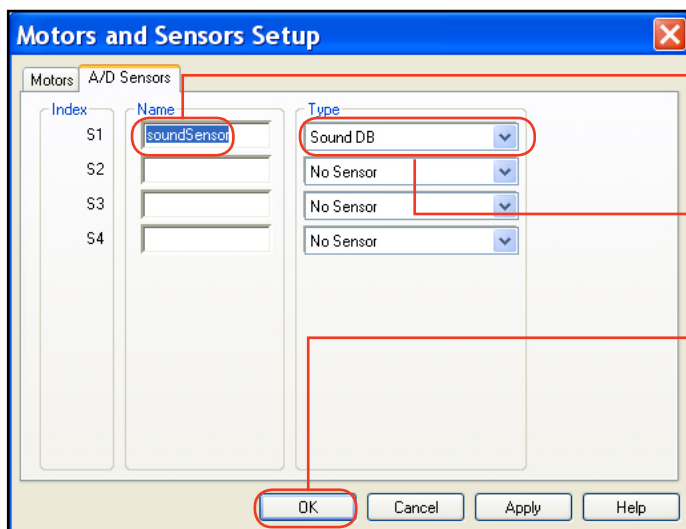


2. Open "Motors and Sensors Setup"
Select Robot > Motors and Sensors Setup to open the Motors and Sensors Setup menu.

Sensing

Speed Based on Volume Values and Assignment (Part 1) (cont.)

3. Configure the sensor on port 1 to be a "SoundDB" sensor named "soundSensor".

**3a. Name the sensor**

Name the Sound Sensor on port S1 "soundSensor".

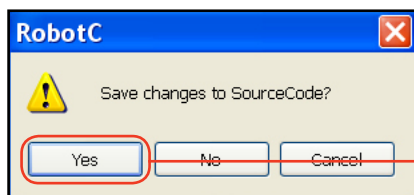
3b. Set Sensor Type

Identify the Sensor Type as a "Sound DB" sensor.

3c. Click OK

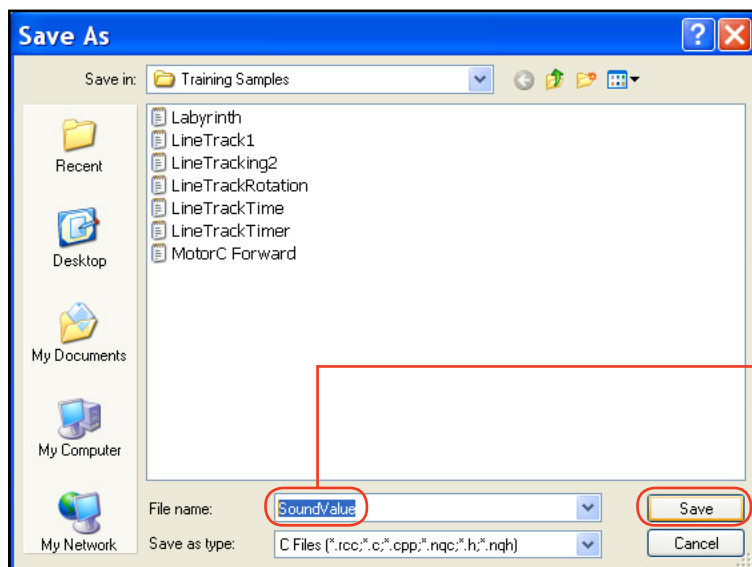
Click the "OK" button to save your changes.

4. You will be prompted to save the changes you have just made. Press Yes to save.

**4. Select "Yes"**

Save your program when prompted.

5. Save this program as "SoundValue".

**5a. Name the program**

Give this program the name "SoundValue".

5b. Save the program

Press Save to save the program with the new name.

Sensing

Speed Based on Volume Values and Assignment (Part 1) (cont.)

The Sound Sensor is now configured. Now, start the program by creating a task main() structure, then add a forward movement command for 10 seconds with both motors at 50% power.

```

1  task main()
2  {
3      motor[motorC] = 50;
4      motor[motorB] = 50;
5      wait1Msec(10000);
6  }
```

Checkpoint

Let's analyze what we're telling the robot to do. The basic motor command sets a given motor's power level. In this case, you're setting Motor C and B's power level to 50. 50 is just a number. If you wanted to set the power to 25, you would put 25 here. 100 works too. Really, any number value will do....

The Sound Sensor reading is also a number value. If the Sound Sensor is reading a sound level of 40, `SensorValue(soundSensor)` is the number value 40! We could simply put `SensorValue(soundSensor)` in place of the number we've been using, and the motor power would be set to the Sound Sensor's value! Let's try it.

```

const tSensors soundsensor
/**!!CLICK to edit 'wizard' created

task main()
{
    100      SensorValue(soundSensor)
           ↓
    motor[motorC] = 50;
    motor[motorB] = 50;
    wait1Msec(10000);
}
           ↓
           25
```

Sensing

Speed Based on Volume Values and Assignment (Part 1) (cont.)

6. Motor powers are number values. You can replace any number value with another, like changing a 50 to 75 or 100. `SensorValue(soundSensor)` is also a number. Replace 50 with the sensor value.

```
1  task main()  
2  {  
3      motor[motorC] = SensorValue(soundSensor);  
4      motor[motorB] = SensorValue(soundSensor);  
5      wait1Msec(10000);  
6  }
```

6. Modify the code

Replace the number values of 50, to the value of the Sound Sensor, S1.

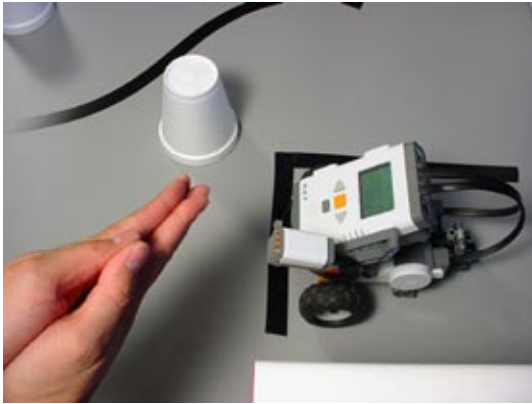
Checkpoint. In theory, our program should now work like this:

- The Sound Sensor reads the amount of sound in the environment
- The Sound Sensor sets the motor power to be equal to the sensor's numeric value
- The robot should run at a speed determined by the Sound Sensor reading – fast for loud, and slow for quiet

Sensing

Speed Based on Volume Values and Assignment (Part 1) (cont.)

7. Save, download, and run your program. Clap your hands to change the sound sensor value.



7a. Make some noise!

Run the program then clap your hands to change the sound sensor value.



7b. Observe the (lack of) reaction

The robot doesn't seem to do anything different...

End of Section

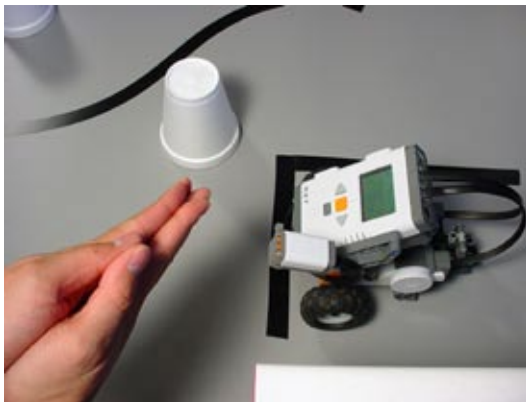
The robot's reaction to the level of sound in the environment was pretty disappointing – nothing happened. In the next section, we'll take a look at what's going on, where our understanding went wrong, and how the problem can be fixed.

Sensing

Speed Based on Volume Values & Assignment (Part 2)

In this lesson you will make the robot's motors use the Sound Sensor's values in real-time.

Try running the robot again, but make the sound **just as** you press the Start button.



Clap and Run

This time, clap (or talk into the Sound Sensor) just as you press the Start button.



Observe the behavior

The robot moves much faster.

The robot is clearly responding to sound levels, but not at the right time. Remember the line tracking behavior? The wait1Msec command tells the robot to go to sleep for a period of time. Going to sleep means the robot isn't watching the sound sensor or updating motor values! If we want to keep the motor's power level up to date with the sensor, we will need to make sure that the power level command gets run over and over. We'll need to use a while loop and a Timer.

Sensing

Speed Based on Volume Values and Assignment (Part 2) (cont.)

1. Delete the wait statement, and add a while() loop around the motor behaviors.

```

1  task main()
2  {
3      while()
4      {
5          motor[motorC] = SensorValue(soundSensor);
6          motor[motorB] = SensorValue(soundSensor);
7          wait1Msec(10000);
8      }
9  }

```

1a. Add this code
Place the while loop so that the motor commands go inside its curly braces.

The (condition) is not yet specified.

1b. Delete this line

We don't want the robot "sleeping" when it needs to update motor powers.

2. Timers must first be initialized, so add a ClearTimer(T1) just before the loop. Check the timer in our while loop condition, we use timer1[T1] less than 10,000 milliseconds, or 10 seconds.

```

1  task main()
2  {
3      ClearTimer(T1);
4      while(timer1[T1] < 10000)
5      {
6          motor[motorC] = SensorValue(soundSensor);
7          motor[motorB] = SensorValue(soundSensor);
8      }
9  }

```

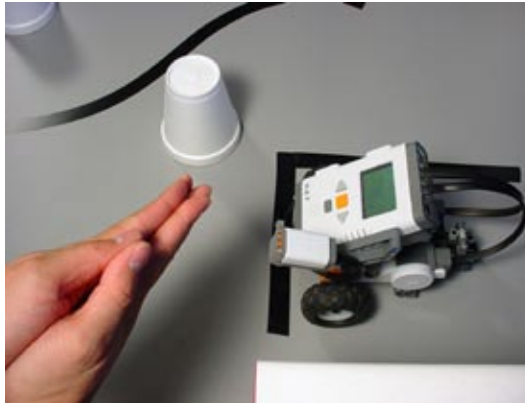
2a. Add this code
Timers must be reset before use.

2b. Add this code
The (condition) will now check whether the timer, T1, is less than 10000ms (10 seconds). The loop's {body} will run while this is true, i.e. less than 10 seconds have passed since the reset.

Sensing

Speed Based on Volume Values and Assignment (Part 2) (cont.)

3. Save, download, and run your program.



Run the program

Run the program and clap your hands repeatedly.



Observe the behavior

The robot moves depending on how much noise it detects!

End of Section

The robot is now checking the sensor repeatedly, and updating the motor power with the new sensor values as quickly as it can, over and over again. As a result, the robot is now responsive to new sound levels in the environment. Rather than just on or off, loud or soft, we've programmed the robot to change the motor power level in direct proportion to the sound level. This is a powerful way to use sensor values. It takes advantage of their numeric nature to link a sensor value with another numeric value, motor power output.

In the next Unit's challenges, you'll have additional opportunities to look even more deeply into the nature of numbers and other data types in ROBOTC. For the immediate future, we think you'll find this Volume Based on Speed behavior helpful on the Obstacle Course. See you on the field!

Sensing

Volume & Speed Quiz

NAME _____ DATE _____

1. The program below makes the robot:

```
1  const tSensors soundSensor          = (tSensors) S1;
2
3  task main()
4  {
5      motor[motorC] = SensorValue(soundSensor);
6      motor[motorB] = SensorValue(soundSensor);
7      wait1Msec(10000);
8  }
```

- travel at a speed that varies continually based on the value of the sound sensor, for 1 second.
- travel at a set speed based of the initial value of the sound sensor, for ten seconds.
- travel at a speed that varies continually based on the value of the sound sensor, for 10 seconds.
- travel at a set speed based of the initial value of the sound sensor, for one second.

2. Explain, in terms of “values”, why the amount of sound you made affected how quickly the robot moved in the Speed Based on Volume program.

Variables and Functions

Warehouse Challenge

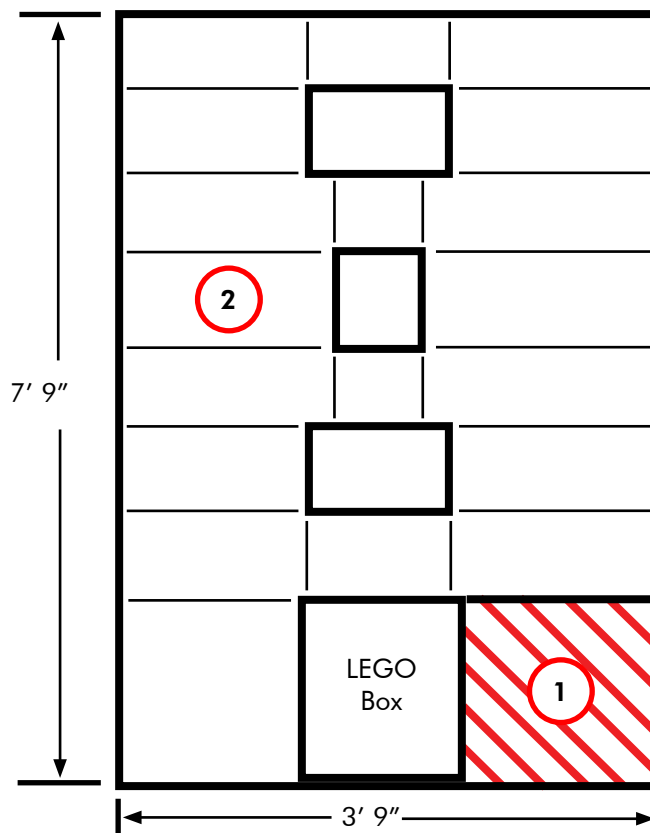
Challenge Description

This challenge provides the lines needed in order to investigate line counting, as well as many other behaviors. Books and the LEGO Box are used as obstacles, and lines are use for "markers".

Materials Needed

- Black electrical tape
- Red electrical tape
- Scissors (or cutting tool)
- Ruler (or straight edge)
- 3 Books
- 1 LEGO Box container

Board Specifications



Note: Diagrams are not drawn to scale

1 Starting area.

2 Goal area.

Variables and Functions

Automatic Threshold Values and Variables

In this lesson, we're going to look a little deeper into the world of "values," and pay special attention to the programming structures that are used to represent and store values, which are called "variables."

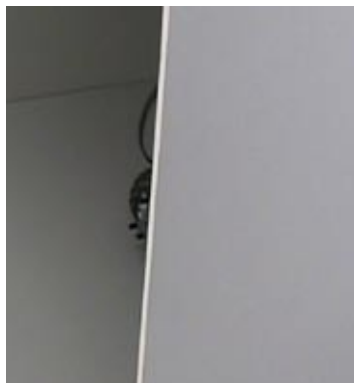
In the previous lesson, "Speed Based on Volume", the robot set its motor power levels based on sound sensor readings. To the robot, this was no different than setting the power level to 25, 50, or 100. These numbers – 25, 50, 100, Sound Sensor readings – are all interchangeable **values** that could be used to set the motor power levels.

There are some situations where values need to be stored for later use. For example, a robot sent into a cave to gather Light Sensor values needs to both **record those values** inside the cave and **be able to recall them** afterwards.



Robot enters the cave

The robot enters the cave (dark area on the right) to gather data.



Robot takes sensor readings

The robot must take and **store** sensor readings inside.



Robot returns

The robot backs out of the cave and displays the values from inside.

Without some way to store these values, they will be lost by the time the robot leaves the cave.

Variables are the robot's way of storing values for later use. They function as containers or storage for values. Values such as the cave robot's sensor reading can be placed in a variable when calculated (inside the cave), and retrieved at a later time (outside the cave) for convenient use. A variable is simply a place to store a value.

There are, however, different types of values. For instance, there are different types of numbers (integers versus decimals, to name just two), and there are values that aren't even numbers, like words. Since there are different types of values, there are **different types of variables** to hold them. In order to create (or "declare") a variable, the programmer must identify two key pieces of information: the **type** of value it will hold, and a **name** for the variable.

Variables and Functions

Automatic Threshold Values and Variables (cont.)

The **names** of variables can include anything that follows the general ROBOTC naming rules (see the “Wall Detection (Touch)” lesson for a list of rules). For **types**, ROBOTC breaks values down into a few simple categories.

Number values in ROBOTC are broken down into two different kinds of numbers:

Integer, or “int” values are numbers with no fractional or decimal component.

Integers

10, 0, -10

Non-Integers

~~10.5~~ ~~10.0~~

Floating point (“float”) numbers are so called because the decimal point “floats” around in the value, allowing decimal places to be used. Floating point numbers can be positive, negative, or zero, but they may also represent decimals. Floating point numbers take up more memory on the robot, and are slower to calculate with, so integer values are preferred when decimals aren’t necessary.

Floating Point Numbers

3.1456, 31.456, 0.0, -314.56

Other kinds of values also exist, including text like “Hello”, and logical values like True.

Strings (“string”): Text in ROBOTC is always a “string”. In ROBOTC, the word “Hello” is really a collection of letters – ‘H’, ‘e’, ‘l’, ‘l’, ‘o’ – “strung” together to form a single value. In fact, while all words are strings in ROBOTC, all strings are not words, and do not even have to be collections of letters. A string may be a series of numbers, or a series of mixed numbers and letters.

Strings

“Hello”, “my name is”, “a16Z”

Boolean (“bool”) values represent “truth” or “logic” values, in the form of “true” or “false”.

Boolean Values

true, false

Variables and Functions

Automatic Threshold Values and Variables (cont.)

To declare a variable, simply call out its type, then its name, then end with a semicolon.

`int lightValue;` will create a new integer-type variable named `lightValue`.

`bool isAwake;` will create a new true-or-false (Boolean) variable named `isAwake`.

Optionally, you can also assign a value to the variable at this point, but it is not necessary.

`int lightValue = 0;` will create a new integer-type variable named `lightValue`, with a starting value of 0.

`bool isAwake = true;` will create a new true-or-false (Boolean) variable named `isAwake`, with a starting value of true.

| Data Type | Description | Example | Code |
|------------------------|---|---|---------------------|
| Integer | Positive and negative whole numbers, as well as zero. | -35, -1, 0, 33, 100, 345 | <code>int</code> |
| Floating Point Decimal | Numeric values with decimal points. | -.123, 0.56, 3.0, 1000.07 | <code>float</code> |
| String | A string of characters that can include numbers, letters, or typed symbols. | "Counter reached 4", "STOP", "time to eat!" | <code>string</code> |
| Boolean | True or False. Useful for expressing the outcomes of comparisons. | true, false | <code>bool</code> |

End of Section

Things like motor powers and sensor readings are **values**. Values can be of different types, like integers or strings. When you need to store them, you can use a **variable** of the appropriate type to hold the value for later use. Variables must be declared by assigning them a suitable **type** and a **name**. Names must follow the usual ROBOTC naming rules, and should be chosen so that you will be able to remember what each variable is supposed to be doing when you read or troubleshoot your code later.

Variables and Functions

Automatic Threshold **Variables and Threshold**

Having to reprogram the robot every time the lighting conditions change is not efficient.

In this lesson, we will give the robot the ability to configure itself at the beginning of every run, with only a little human assistance.

When the program begins, the user will be prompted to “scan” a light surface with the Light Sensor, and then “scan” a dark surface. The robot will then calculate its own Light Sensor threshold, wait a few seconds, and proceed as normal.

We’ll begin by going through the threshold calculation process manually, and taking note of the important values that the robot will have to keep track of. Every time a number or value has to be remembered, make a note.



Scanning light

The robot's light sensor is first positioned over a **light** surface and told to read and store its value



Scanning dark

Then, the robot's light sensor is positioned over a **dark** surface and told to read and store its value

Variables and Functions

Automatic Threshold **Variables and Threshold** (cont.)

1. Turn on your NXT and navigate to the "View" mode using the gray arrows.



1a. Push the orange button

Turn on the robot by pushing the orange button. The screen should display "My Files" when it is on.



1b. Go to the "View" menu

Navigate to the "View" menu using the arrow buttons. Press the orange button to go into it.



1c. Select "Reflected Light"

Select "Reflected Light", not "Ambient Light". You will get different values otherwise.



1d. Select your port number

Select the port number that your Light sensor is plugged into.

2. Record your Light and Dark readings. Record these values.



2a. Record the light value

Place the robot on the light surface, and record the value that the Light sensor is reading.

Variables and Functions

Automatic Threshold Variables and Threshold (cont.)



2b. Record the dark value
Place the robot on the dark surface, and record the value that the Light sensor is reading.

5. Find the average of the light and dark readings by adding them together and dividing by two. This thresholdValue will be used for future comparison.

$$\text{light value} + \text{dark value} = \text{sum}$$

5a. $66 + 33 = 99$

$$\text{sum} / 2 = \text{average}$$

5b. $99 / 2 = 49.5$

Note: Get rid of the decimal number

5c. $49.5 = 49$

Get rid of the decimal
ROBOTC will get rid of the decimal automatically when using integers.

$$\text{average} = \text{threshold}$$

5d. $49 = \text{thresholdValue}$

Variables and Functions

Automatic Threshold **Variables and Threshold** (cont.)

Checkpoint

Four values were either recorded or calculated: light value, dark value, sum, and threshold.

$$\begin{array}{r} \text{light value} + \text{dark value} = \text{sum} \\ 66 + 33 = 99 \end{array}$$

Calculate "sum" value

The sum value is found by adding the light value and dark value.

$$\begin{array}{r} \text{sum} / 2 = \text{threshold} \\ 99 / 2 \approx 49 \end{array}$$

Calculate average/"threshold" value

The average is found by dividing the sum value by 2. The resulting average is the threshold value.

In order to write a program that will auto-calculate the value of threshold, we will need to create four variables to store the four values that the calculation needs. To declare each variable, a **name** and **type** must be specified. The name should help you to remember what the variable does. For this lesson these values will be named:

- lightValue
- darkValue
- sumValue
- thresholdValue

In addition to a name, the **type** of value (integer, floating point decimal, string, boolean value) that each variable will hold needs to be determined.

Light Sensors yield values that are whole numbers. So lightValue and darkValue will be "declared" as **integers**. Since the sum of two integers is also an integer, sumValue will be declared as an integer as well. Dividing by two might result in a decimal, but since the threshold is an estimate to begin with, rounding won't hurt it, and so thresholdValue will also be declared as an integer.

Declaring Variables

To create a variable, you must "declare" it with two pieces of information:

datatype then **name**;

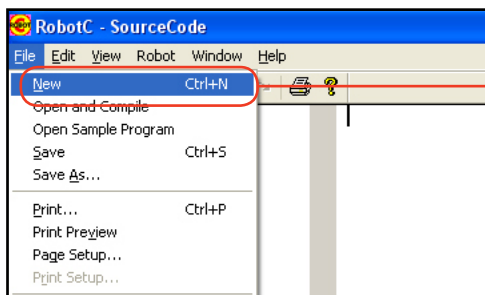
Example:

`int lightValue;` will create a new integer-type variable named lightValue.

Variables and Functions

Automatic Threshold **Variables and Threshold** (cont.)

7. Place the four variables declared as integers in a new program.



7a. Create new program
Select File > New to create a blank new program.

```

2  task main()
3  {
4
5
6
7
8
9
10 }
```

 A screenshot of the RobotC SourceCode editor showing the beginning of a program. The code is:


```

2  task main()
3  {
4
5
6
7
8
9
10 }
```

 A red box highlights the curly brackets and the space between lines 4 and 6. A red arrow points from the text '7b. Add this code' to this box.

7b. Add this code
These lines form the main body of the program, as they do in every ROBOTC program. Leave four lines between curly brackets for the variables.

```

2  task main()
3  {
4
5  int lightValue;
6  int darkValue;
7  int sumValue;
8  int thresholdValue;
9
10 }
```

 A screenshot of the RobotC SourceCode editor showing the variable declarations. The code is:


```

2  task main()
3  {
4
5  int lightValue;
6  int darkValue;
7  int sumValue;
8  int thresholdValue;
9
10 }
```

 A red box highlights the four lines of variable declarations (lines 5-8). A red arrow points from the text '7c. Add these lines' to this box.

7c. Add these lines
Declare the four variables, lightValue, darkValue, sumValue and thresholdValue as integers. Remember that typographic errors can keep the program from functioning!

End of Section

Four variables have been created to store the four values needed to calculate a Light Sensor threshold. In the next lesson we will write the remainder of the program.

Variables and Functions

Automatic Threshold Programming with Variables

In this lesson, you will learn how to store Light Sensor values in the variables you created, and how to use a Touch Sensor as a user interface button.

The robot will take the first Light Sensor reading over a light surface when the Touch Sensor is pressed, then take a second reading over a dark surface when the Touch Sensor is pressed a second time.

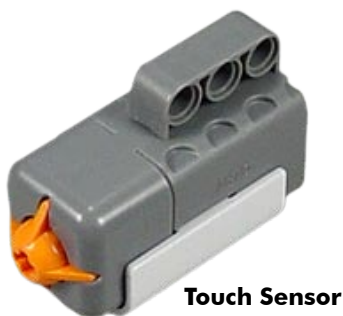
```

2  task main()
3  {
4
5     int lightValue;
6     int darkValue;
7     int sumValue;
8     int thresholdValue;
9
10 }
```

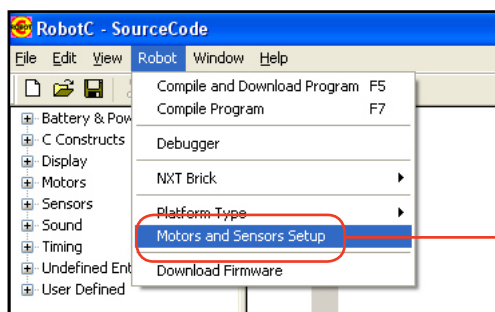
Existing program

Your program should currently look like this.

First, we'll configure the Light and Touch Sensors.



1. Open the Motors and Sensors Setup menu.



1. Open "Motors and Sensors Setup"

Select Robot > Motors and Sensors Setup to open the Motors and Sensors Setup menu and configure the sensors.

Variables and Functions

Automatic Threshold Programming with Variables (cont.)

2. ROBOTC will ask if you want to save your program. Click Yes, then save the program as "Autothreshold".

2a. Select "Yes"
Save your program when prompted.

2b. Name the program
Name the program "Autothreshold".

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

3. Select the A/D Sensors tab, and make Port 1 the Touch Sensor, named touchSensor, and Port 2 the Light Sensor, named lightSensor.

3a. Select "A/D Sensors" tab
Selecting this tab allows you view your sensors set up menu.

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

3c. Name the sensor
Name the Touch Sensor on port S1 "touchSensor".

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

3e. Name the sensor
Name the Light Sensor on port S2 "lightSensor".

3f. Click OK

Variables and Functions

Automatic Threshold Programming with Variables (cont.)

The next step is for the robot to take the first Light Sensor reading over a "light" surface when the Touch Sensor is pressed. Then, take the dark reading on the next Touch Sensor press.



Variables and Functions

Automatic Threshold Programming with Variables (cont.)

4. The robot should wait for the Touch Sensor to be pressed. A while() loop is used to check the touchSensor value to watch for a press. As long as the Touch Sensor isn't pressed, (SensorValue(touchSensor)==0) remains true, and the robot does nothing.

```

2  task main()
3  {
4
5    int lightValue;
6    int darkValue;
7    int sumValue;
8    int thresholdValue;
9
10 while (SensorValue(touchSensor)==0)
11 {
12 }
13
14 }
```

4. Add this code

This while() loop **idles** (i.e. runs an empty {} code block) while the Touch Sensor is not pressed.

5. After the Touch Sensor is pressed, record the Light Sensor's value to the variable lightValue. Assign the value of the sensor to the variable. LightSensor = SensorValue(lightSensor) Note: A single equals sign means, "set to the value of".

```

2  task main()
3  {
4
5    int lightValue;
6    int darkValue;
7    int sumValue;
8    int thresholdValue;
9
10 while (SensorValue(touchSensor)==0)
11 {
12 }
13
14 lightValue=SensorValue(lightSensor);
15
16 }
```

5. Add this code

This line puts the Light Sensor's value into the variable lightValue.

Variables and Functions

Automatic Threshold Programming with Variables (cont.)

6. Next, the robot records the dark value. Either retype the wait-for-press loop, and the storing of the value manually, or just highlight and copy the code you just wrote, (starting with “while” and ending with the semicolon) and paste another copy of it below. In this second recording, of course, you want to record the value to the dark Value.

```

2  task main()
3  {
4
5  int lightValue;
6  int darkValue;
7  int sumValue;
8  int thresholdValue;
9
10 while (SensorValue(touchSensor)==0)
11 {
12 }
13
14 lightValue=SensorValue(lightSensor);
15
16 while (SensorValue(touchSensor)==0)
17 {
18 }
19
20 darkValue=SensorValue(lightSensor);
21
22 }

```

6a. Add this code

This while () loop idles while the Touch Sensor is not pressed, just like the previous one.

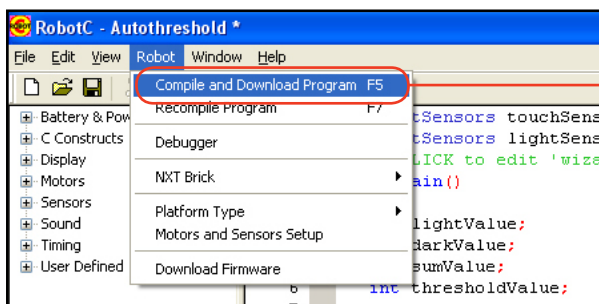
6b. Add this code

This line puts the Light Sensor's value into the variable “darkValue”.

Checkpoint

Check to see if the program is working. It is almost always better to write code in small bits and test often, rather than waiting to test a long section of code in which many mistakes could be hiding.

7. Compile, Download and run your program.



7. Compile and Download

Robot > Compile and Download Program

Variables and Functions

Automatic Threshold Programming with Variables (cont.)

8. Run the program. Put the Light Sensor over a light surface. Press the Touch Sensor.
Keep an eye the robot... it may not do what you expect!



9. The program seems to end immediately when the Touch Sensor is pressed.
That's not what we wanted!

End of Section

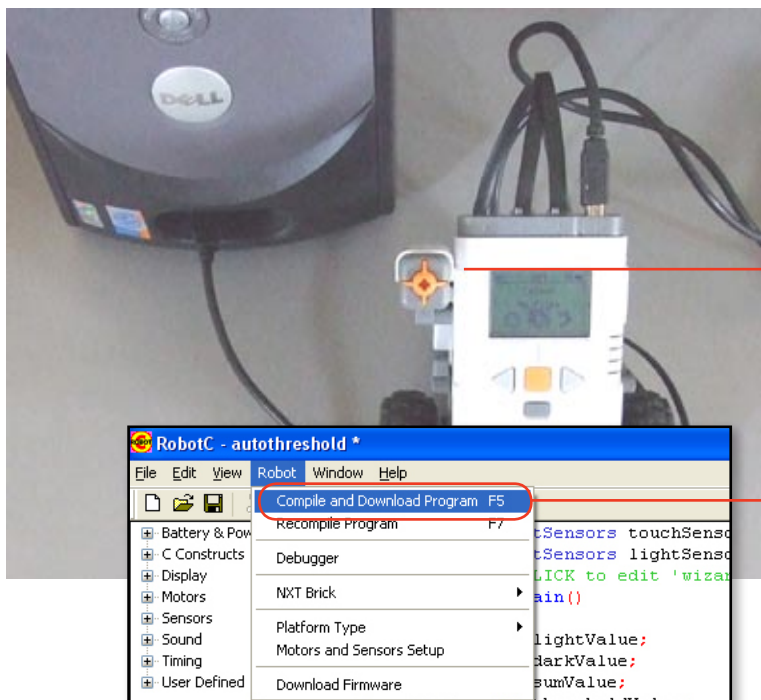
Something is wrong with the program. In the next lesson, the debugger will be used to fix the problem.

Variables and Functions

Automatic Threshold **Variables and the Debugger**

In this lesson, the Debugger windows will be used to determine why the program is not running properly. The debugger can be used to “freeze time” for the robot and allows you to step through the program at whatever speed you want.

1. Something is obviously wrong with the program. Download the program again, but this time, make sure the robot stays plugged into the computer, and watch the code window.



1a. Plug the robot back in

Robot has to be plugged into the computer, via USB, to be able to view the code window.

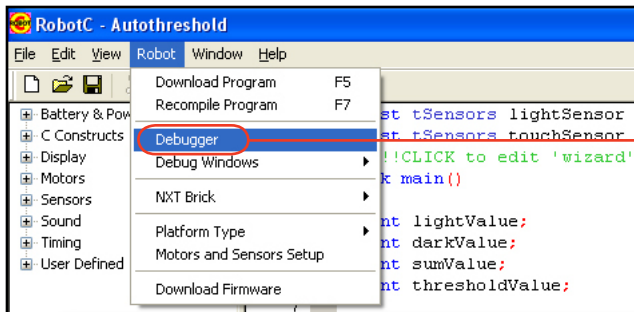
1b. Compile and download

Select Robot > Compile and Download Program. The option may just read “Download Program”, which is fine also.

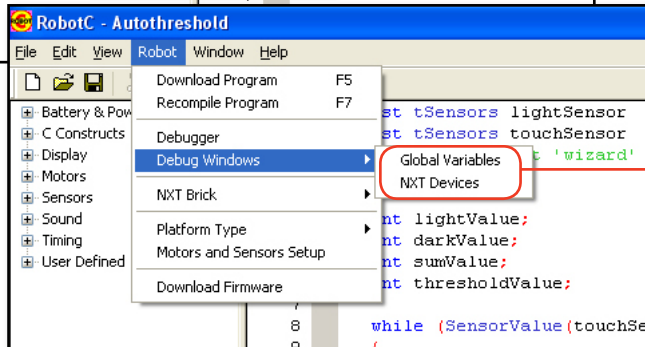
Variables and Functions

Automatic Threshold Variables and the Debugger (cont.)

- After you have downloaded the program to your robot, fix the problem by open up the Debugger, then select both the Global Variables and the NXT Devices options so both these windows are visible.



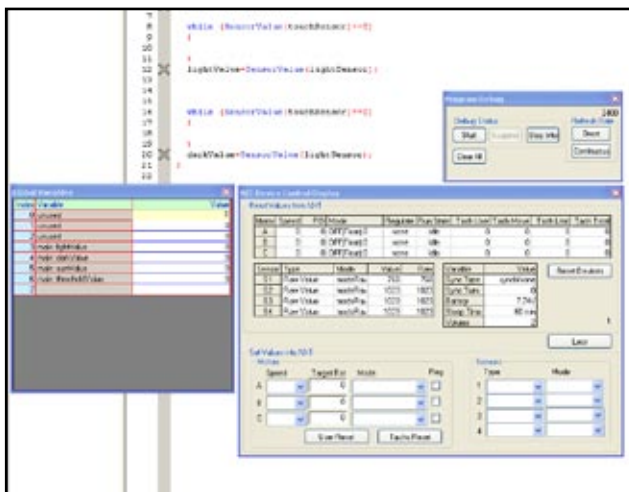
2a. View Debugger
Select Robot > Debugger to open up the Program Debug window.



2b. View Debugger Windows
Select Robot > Debug Windows and select both Global Variables and NXT devices if they are not already checked.

Checkpoint

The screen should look like the sample below with three windows visible: Program Debug, Global Variable and NXT Device Control Display.



Variables and Functions

Automatic Threshold **Variables and the Debugger** (cont.)

3. Run the program. Observe what happens when you push the Touch Sensor.

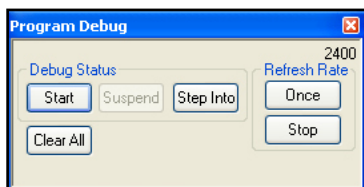


3. Push Touch Sensor

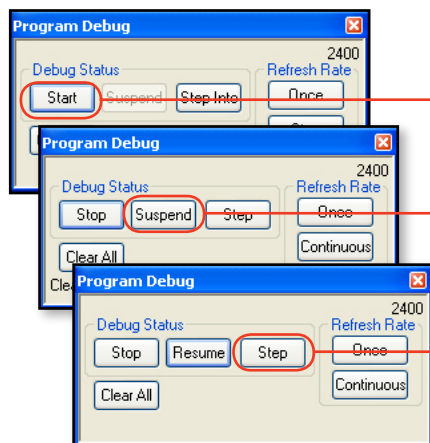
Pushing the Touch Sensor allows the program to move forward out of the while () loop.

Checkpoint

The button was pressed once, and the program shot straight to the end. You can tell the program is finished because the Start button on the Program Debug window is highlighted. (If the program was still running, the Suspend button would be highlighted.)



4. Run the program again, but this time use the Program Debug window to “freeze” time and step through the program while suspended. To do so, press the Suspend button, then the Step button.



4a. Press Start button

Press the Start button to get the program started.

4b. Press Suspend button

Press the Suspend button on the Program Debug window to “stop” time and leave the program right where it is.

4c. Press Step button

Press the Step button to go to the next line of code.

Variables and Functions

Automatic Threshold **Variables and the Debugger** (cont.)

5. Press the Touch Sensor and observe in the NXT Device Control Display that it is pushed and working properly.



5a. Push Touch Sensor

Pushing the Touch Sensor allows the program to move forward out of the while () loop.

| NXT Device Control Display | | | | | | | | | |
|----------------------------|-------|-----|--------------|----------|-----------|-----------|-----------|------------|------------|
| Read Values from NXT | | | | | | | | | |
| Motor | Speed | PID | Mode | Regulate | Run State | Tach User | Tach Move | Tach Limit | Tach Total |
| A | 0 | 0 | OFF(Float) 0 | none | Idle | 0 | 0 | 0 | 0 |
| B | 0 | 0 | OFF(Float) 0 | none | Idle | 0 | 0 | 0 | 0 |
| C | 0 | 0 | OFF(Float) 0 | none | Idle | 0 | 0 | 0 | 0 |

| Sensor | Type | Mode | Value | Raw | Variable | Value |
|--------|--------------|---------|-------|------|------------|----------|
| S1 | Touch | modeBoc | 1 | 180 | Sync Type | syncNone |
| S2 | Light Active | modePen | 67 | 426 | Sync Turn | 0 |
| S3 | Raw Value | modeRaw | 1023 | 1023 | Battery | 7.39V |
| S4 | Raw Value | modeRaw | 1023 | 1023 | Sleep Time | 60 min |
| | | | | | Volume | 2 |

5b. Observe the Touch Sensor

The value of the Touch Sensor, 1, means that it is pressed.

Since you have suspended the program, the robot's program remains "frozen" at the first while() loop (where the yellow line appears in the code). The NXT Device Control window on your PC screen, however, remains operational, and will continue to report the value of the sensors.

```

2  task main()
3  {
4
5    int lightValue;
6    int darkValue;
7    int sumValue;
8    int thresholdValue;
9
10   while (SensorValue(touchSensor) == 0)
11   {
12   }
13
14   lightValue = SensorValue(lightSensor);
15
16   while (SensorValue(touchSensor) == 0)
17   {
18   }
19
20   darkValue = SensorValue(lightSensor);
21
22 }
```

Line about to run

The program will run this step when the Step button is pressed again.

Because the line is a while loop, it will evaluate the (condition) and decide whether to loop, or move on.

Variables and Functions

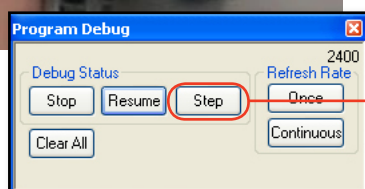
Automatic Threshold **Variables and the Debugger** (cont.)

6. While continuing to hold the Touch Sensor in the pushed position, click the Step button on the Program Debug control panel to allow the program to move past the while() loop.



6a. Push the Touch Sensor

Hold the Touch Sensor in the pushed position while pressing the Step button.



6b. Press Step button

Press the Step button while pushing the Touch Sensor to allow you to go to the next step of the code.

Since the Touch Sensor value is not 0 at the time the while loop checks, the program moves past the loop to the next step. The next line turns yellow now to indicate that this command is *about* to be executed.

```

2  task main()
3  {
4
5  int lightValue;
6  int darkValue;
7  int sumValue;
8  int thresholdValue;
9
10 while (SensorValue(touchSensor) == 0)
11 {
12 }
13
14 lightValue = SensorValue(lightSensor);
15
16 while (SensorValue(touchSensor) == 0)
17 {
18 }
19
20 darkValue = SensorValue(lightSensor);
21
22 }

```

Line that was run

When you pressed Step, this line was run. The (condition) was False because the touchSensor value was 1 (and not 0), so the program exited the loop and moved on.

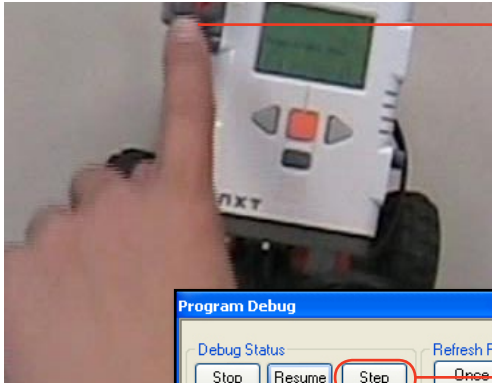
Line about to run

The program will run this line when the Step button is pressed again.

Variables and Functions

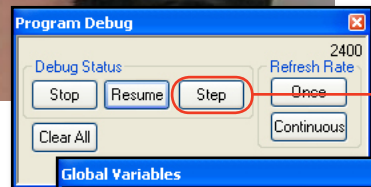
Automatic Threshold Variables and the Debugger (cont.)

7. Find the variable `lightValue` in the Global Variables window. Push the Touch Sensor. Keep it pushed in while pressing the Step button. The Light Sensor's value when the Step button was first pressed is now stored in the variable `lightValue`.



7a. Push the Touch Sensor

Hold the Touch Sensor in the pushed position while pressing the Step button.



7b. Press Step button

Press the Step button while pushing the Touch Sensor to enable the program to move to the next line of code.

| Index | Variable | Value |
|-------|----------------------|-------|
| 0 | unused | 0 |
| 1 | unused | 0 |
| 2 | unused | 0 |
| 3 | main::lightValue | 66 |
| 4 | main::darkValue | 0 |
| 5 | main::sumValue | 0 |
| 6 | main::thresholdValue | 0 |
| 7 | | |

7c. Stored Variable

The `lightValue` variable now equals the value of the Light Sensor when the Touch Sensor was first pushed, as shown in the Global Variables window.

```

13
14 lightValue=SensorValue(lightSensor);
15
16 while (SensorValue(touchSensor)==0)
17 {
18 }
19
20 darkValue=SensorValue(lightSensor);
21
22 }

```

Line that was run

When you pressed Step, this line was run, and stored the value of the Light Sensor in the variable.

Line about to run

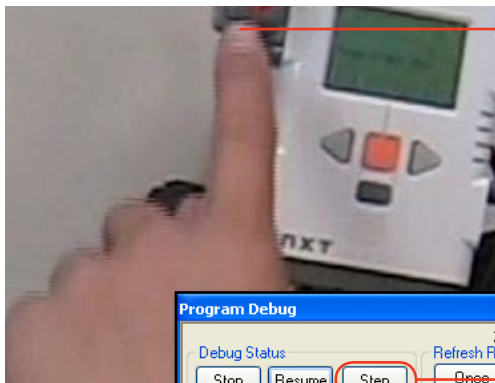
The program is now ready to run this next step when Step is pressed again.

Because the line is a while loop, it will evaluate the (condition) and decide whether to loop, or move on.

Variables and Functions

Automatic Threshold **Variables and the Debugger** (cont.)

8. While continuing to hold the Touch Sensor in, press the Step button several times to step through the rest of the program.



8a. Keep the Touch Sensor pressed

Hold the Touch Sensor in the pushed position while pressing the Step button.

8b. Press Step button

Press the Step button several times while pushing the Touch Sensor to step through to the end of the program.

```

13
14 lightValue=SensorValue(lightSensor);
15
16 while (SensorValue(touchSensor)==0)
17 {
18 }
19
20 darkValue=SensorValue(lightSensor);
21
22 }

```

Line that was run

When you pressed Step, this line was run. The (condition) was False because the touchSensor value was 1 (and not 0), so the program exited the loop and moved on.

8c. Press Step button again

The program moves to the next line of code, making the variable darkValue equal to the Light Sensor value the moment the Touch sensor was pressed.

```

13
14 lightValue=SensorValue(lightSensor);
15
16 while (SensorValue(touchSensor)==0)
17 {
18 }
19
20 darkValue=SensorValue(lightSensor);
21
22 }

```

8d. Press Step button again

The program moves to the next line of code, the last curly bracket, and the program ends.

Variables and Functions

Automatic Threshold **Variables and the Debugger** (cont.)

Checkpoint

Do you see what the problem is? When the Touch Sensor is **held down**, the program shoots straight through to the end of the program without stopping.

Why does it do this? Because we told it to. When the Touch Sensor was pressed, it took the program out of the first while loop. This was what we intended. But then, it quickly set the lightSensor variable, and then waited for the button to be pressed... which it still was, from the first press! The program immediately jumped past the second while loop. This is what we said, though certainly not what we wanted!

With the Step function, you could see this happening one step at a time. At normal speed, all this happens before you can take your finger off the button from the first press!

- Place a command between the while() loops telling the robot to wait for 1 second before looking for the Touch Sensor value again. This allows the human operator enough time to push, **and release**, the Touch Sensor.

```

2  task main()
3  {
4
5  int lightValue;
6  int darkValue;
7  int sumValue;
8  int thresholdValue;
9
10 while (SensorValue(touchSensor)==0)
11 {
12 }
13
14 lightValue=SensorValue(lightSensor);
15
16 wait1Msec(1000);
17
18 while (SensorValue(touchSensor)==0)
19 {
20 }
21
22 darkValue=SensorValue(lightSensor);
23
24 }
```

9. Add this code

Tells the robot to wait for 1 second before it starts looking for the Touch Sensor again.

End of Section

In this lesson, the debugger was used as a tool to diagnose why a program was not working properly. Stepping through the commands in a program one at a time allows you to slow down the program so the problem can be found.

Variables and Functions

Automatic Threshold **Threshold Calculations**

About half of the autothreshold calculator program is complete. In the previous lessons the Light and Dark values were recorded and stored in variables. In this lesson, you will use them to calculate the threshold value for the robot's environment.



Checkpoint

This is what the current program should look like.

```

2  task main()
3  {
4
5  int lightValue;
6  int darkValue;
7  int sumValue;
8  int thresholdValue;
9
10 while (SensorValue(touchSensor)==0)
11 {
12 }
13
14 lightValue=SensorValue(lightSensor);
15
16 wait1Msec(1000);
17
18 while (SensorValue(touchSensor)==0)
19 {
20 }
21
22 darkValue=SensorValue(lightSensor);
23
24 }
```


Variables and Functions

Automatic Threshold **Threshold Calculations** (cont.)

1. Starting at the end of the program, just before the closing brace of the task main pair, set the sumValue equal to the sum of lightValue and darkValue. The variable sumValue is now being used to store the result of lightValue plus darkValue.

```
2  task main()
3  {
4
5  int lightValue;
6  int darkValue;
7  int sumValue;
8  int thresholdValue;
9
10 while (SensorValue(touchSensor)==0)
11 {
12 }
13
14 lightValue=SensorValue(lightSensor);
15
16 wait1Msec(1000);
17
18 while (SensorValue(touchSensor)==0)
19 {
20 }
21
22 darkValue=SensorValue(lightSensor);
23
24 sumValue = lightValue + darkValue;
25
26 }
```

1. **Add this code**
Add lightValue and darkValue together, and store the result in the variable sumValue.

Variables and Functions

Automatic Threshold **Threshold Calculations** (cont.)

2. Set thresholdValue equal to sumValue divided by two. The variable thresholdValue now stores the threshold value calculated from the readings of light and dark surfaces.

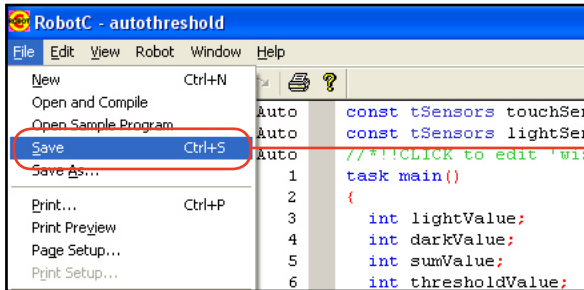
```
2  task main()
3  {
4
5    int lightValue;
6    int darkValue;
7    int sumValue;
8    int thresholdValue;
9
10   while (SensorValue(touchSensor)==0)
11   {
12   }
13
14   lightValue=SensorValue(lightSensor);
15
16   wait1Msec(1000);
17
18   while (SensorValue(touchSensor)==0)
19   {
20   }
21
22   darkValue=SensorValue(lightSensor);
23
24   sumValue = lightValue + darkValue;
25   thresholdValue = sumValue/2;
26
27 }
```

2. Add this line of code
Divide sumValue by 2, and store the result in the variable thresholdValue.

Variables and Functions

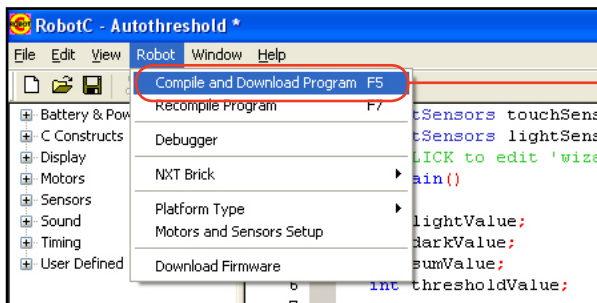
Automatic Threshold Threshold Calculations (cont.)

3. Save the Autothreshold program.



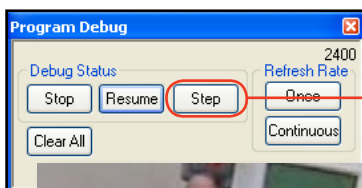
5. Save program
File > Save, to save your current autothreshold program.

4. Compile and download your program.



4. Compile and download
Robot > Compile and Download Program

5. Step through the program using the debugger, pushing the Touch Sensor at the appropriate times. Observe the variables window as sumValue stores the sum of lightValue and darkValue; and thresholdValue stores sumValue divided by two.



5a. Press Step button
Press the Step button in the Program Debug window to step through the program.



5b. Push the Touch Sensor
Push the Touch Sensor over light and dark surfaces at the appropriate times when you step through the program.

The screenshot shows the 'Global Variables' window. It contains a table with the following data:

| Index | Variable | Value |
|-------|----------------|-------|
| 3 | lightValue | 57 |
| 4 | darkValue | 24 |
| 5 | sumValue | 81 |
| 6 | thresholdValue | 0 |

5c. Observe variables
Observe the variables window as lightValue, darkValue, sumValue and thresholdValue are calculated.

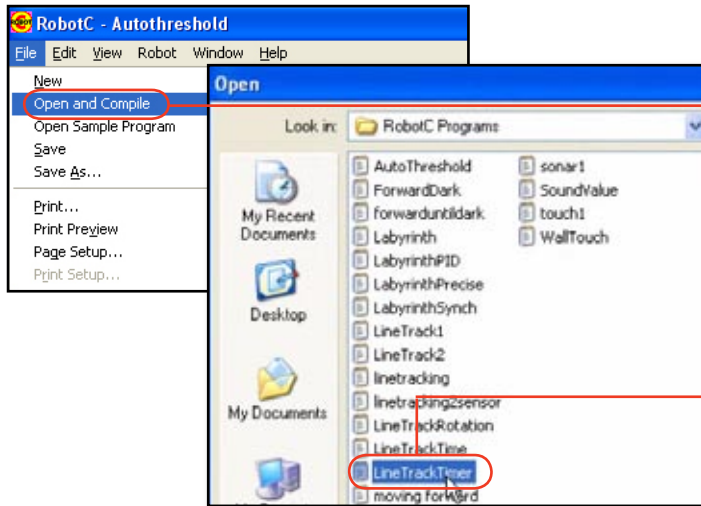
Checkpoint

The threshold is now being calculated as the average of the other two values. The debugger window shows the values of all the variables as they are collected and/or calculated.

Variables and Functions

Automatic Threshold Threshold Calculations (cont.)

6. Open your LineTrackTimer program.



6a. Open and Compile
Select File > Open and Compile to be prompted to open a file.

6b. Select the program
Select LineTrackTimer from your previously saved programs, then double click to open it.

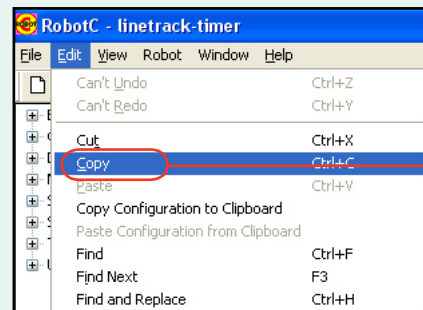
7. Copy the code highlighted below, from lines 5 to 29 of the LineTrackTimer program. Be careful to copy exactly this portion of the program.

```

4
5  ClearTimer(T1);
6
7  while (time1[T1] < 3000)
8  {
9
10     if (SensorValue(lightSensor) < 43)
11     {
12
13         motor[motorC]=0;
14         motor[motorB]=80;
15
16     }
17
18     else
19     {
20
21         motor[motorC]=80;
22         motor[motorB]=0;
23
24     }
25
26 }
27
28     motor[motorC]=0;
29     motor[motorB]=0;
30

```

7a. Highlight code
Highlight exactly this section of code in the LineTrackTimer program.

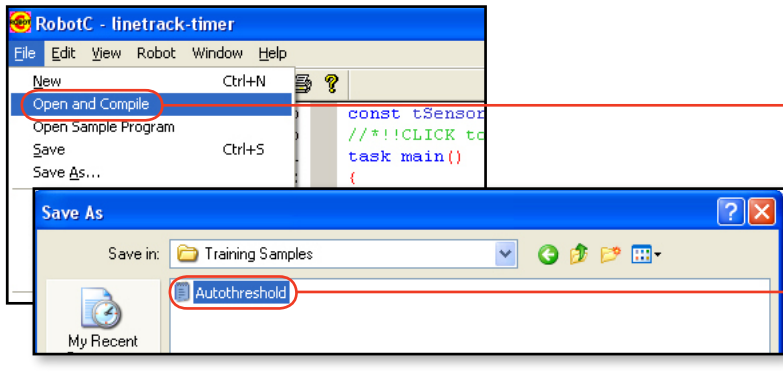


7b. Select Copy
Select Edit > Copy to copy the highlighted code.

Variables and Functions

Automatic Threshold Threshold Calculations (cont.)

8. Reopen the autothreshold program.



8a. Open and compile
Select File > Open and Compile to open a file.

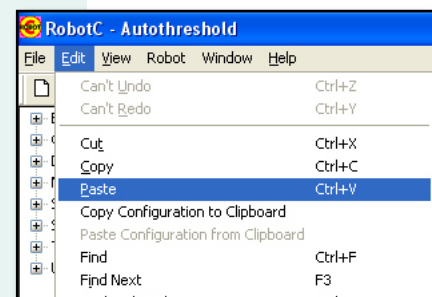
8b. Select the program
Select the autothreshold program from the previous saved programs.

9. Paste the code you copied between "sumValue/2;" and the concluding curly brace.

```

2  task main()
3  {
4
5  int lightValue;
6  int darkValue;
7  int sumValue;
8  int thresholdValue;
9
10 while (SensorValue(touchSensor)==0)
11 {
12 }
13
14 lightValue=SensorValue(lightSensor);
15
16 wait1Msec(1000);
17
18 while (SensorValue(touchSensor)==0)
19 {
20 }
21
22 darkValue=SensorValue(lightSensor);
23
24 sumValue = lightValue + darkValue;
25 thresholdValue = sumValue/2;
26
27 |
28 }

```



9. Paste the copied code
Place the cursor right before the last curly brace and select Edit > Paste to paste the code.

Variables and Functions

Automatic Threshold **Threshold Calculations** (cont.)

10. Change the condition of the “borrowed” if-else statement so that instead of comparing the light sensor value to a set number, it checks it against the “thresholdValue” variable calculated in the Autothreshold program.

```
26
27 ClearTimer(T1);
28
29 while(time1[T1] < 3000)
30 {
31
32     if(SensorValue(lightSensor) < thresholdValue)
33     {
34
35         motor[motorC] = 0;
36         motor[motorB] = 80;
37
38     }
39
40     else
41     {
42
43         motor[motorC] = 80;
44         motor[motorB] = 0;
45
46     }
47
48 }
```

10. Modify code

Replace the condition, which had contained a number, with the variable “thresholdValue”, that holds the calculated threshold value.

Variables and Functions

Automatic Threshold **Threshold Calculations** (cont.)

Checkpoint

Your final program should look like the one below, and on the following page.

```
2 task main()
3 {
4
5   int lightValue;
6   int darkValue;
7   int sumValue;
8   int thresholdValue;
9
10  while (SensorValue(touchSensor)==0)
11  {
12  }
13
14  lightValue=SensorValue(lightSensor);
15
16  wait1Msec(1000);
17
18  while (SensorValue(touchSensor)==0)
19  {
20  }
21
22  darkValue=SensorValue(lightSensor);
23
24  sumValue = lightValue + darkValue;
25  thresholdValue = sumValue/2;
26
27  ClearTimer(T1);
28
```

Variables and Functions

Automatic Threshold **Threshold Calculations** (cont.)

Checkpoint

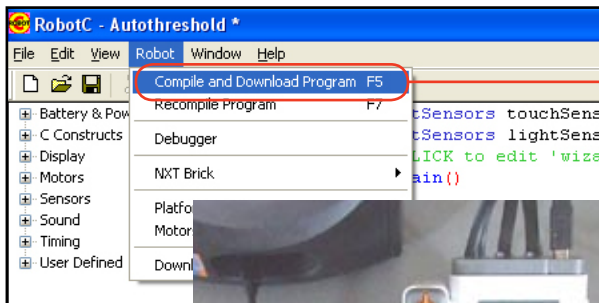
Your final program should look like the one below. (continued)

```
28
29 while (time1[T1] < 3000)
30 {
31
32     if (SensorValue(lightSensor) < thresholdValue)
33     {
34
35         motor[motorC]=0;
36         motor[motorB]=80;
37
38     }
39
40     else
41     {
42
43         motor[motorC]=80;
44         motor[motorB]=0;
45
46     }
47
48 }
49
50 motor[motorC]=0;
51 motor[motorB]=0;
52
53 }
```


Variables and Functions

Automatic Threshold Threshold Calculations (cont.)

11. Compile and Download to your robot.



11. Compile and download

Select Robot > Compile and Download Program to run your robot.



Checkpoint

Test your program. Find a line you can track in a place where you can turn the lights on and off. Run your program and press the Touch Sensor once with the Light Sensor over light, to read the value of the light surface. Move the robot so that it is in line tracking position, with the Light Sensor over the line.

Pressing the Touch Sensor for the second time should not only read the dark value and calculate the threshold, but should also make the robot track the line for three seconds. Now turn the lights off, and run the program again. The robot should still be able to track the line!



Test program with lights on

Show the robot what the light surface looks like, then the dark one, and it should track the line for three seconds.



Test program with lights off

Change the light in the room and test the program again. The robot should again be able to track the line, demonstrating its ability to calculate a threshold in different conditions.

Variables and Functions

Automatic Threshold **Threshold Calculations** (cont.)

The program works, but does need to be made more user-friendly. Right now, the robot will not tell you what to do, or when. Place simple instructions in the code to solve this problem.

- 12.** While the robot is waiting for the Touch Sensor to be pushed, program the robot to display a message telling a user to press the button over a light surface. This command makes the NXT display, on its screen, the words "Read Light Now" at position 0, 31 (that's the left edge, about halfway down). Place a similar line in the second while() loop that does the same thing, but says "Read Dark Now".

```

2  task main()
3  {
4
5  int lightValue;
6  int darkValue;
7  int sumValue;
8  int thresholdValue;
9
10 while (SensorValue(touchSensor)==0)
11 {
12     nxtDisplayStringAt(0, 31, "Read Light Now");
13 }
14
15 lightValue=SensorValue(lightSensor);
16
17 wait1Msec(1000);
18
19 while (SensorValue(touchSensor)==0)
20 {
21     nxtDisplayStringAt(0, 31, "Read Dark Now");
22 }
23
24 darkValue=SensorValue(lightSensor);
25
26 sumValue = lightValue + darkValue;
27 thresholdValue = sumValue/2;
28

```

12a. Add this code

Tells the NXT to display, on its screen, the words "Read Light Now" at the beginning of the program.

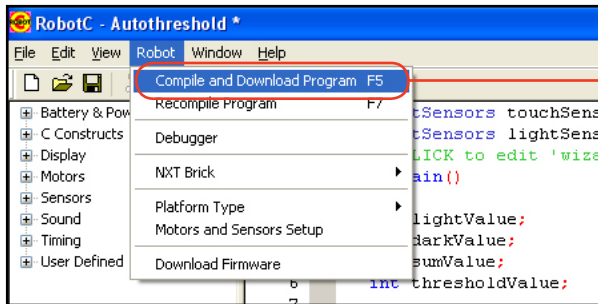
12b. Add this code

Tells the NXT to display, on its screen, the words "Read Dark Now" after the Touch Sensor has been pushed and released once.

Variables and Functions

Automatic Threshold Threshold Calculations (cont.)

13. Compile and Download your program to the robot.



13. Compile and Download
Select Robot > Compile and Download Program.

14. Test the program. After the program starts, the message, "Read Light Now" should appear on the NXT screen. After the Touch Sensor is pushed and released, the NXT screen should display the message, "Read Dark Now." As you did previously, place the robot so that its Light Sensor is directly over the line, and its chassis roughly parallel with the line so that it is in good position to track it. When you press the button, the threshold should be calculated, and the robot should track the line for three seconds.



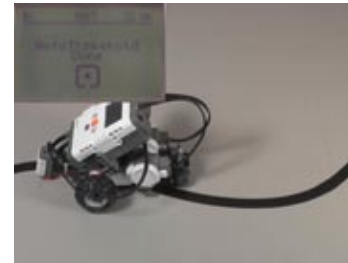
14a. Read light

When the NXT displays "Read Light Now", record the light surface value.



14b. Read dark

When the NXT displays "Read Dark Now", place the robot in position to track a line.



14c. Autothreshold line track

The robot should track a line for three seconds and end the program.

Variables and Functions

Automatic Threshold **Threshold Calculations** (cont.)

End of Section

This is the complete code for the Automatic Threshold program.

```
2  task main()
3  {
4
5  int lightValue;
6  int darkValue;
7  int sumValue;
8  int thresholdValue;
9
10 while (SensorValue(touchSensor)==0)
11 {
12     nxtDisplayStringAt(0, 31, "Read Light Now");
13 }
14
15 lightValue=SensorValue(lightSensor);
16
17 wait1Msec(1000);
18
19 while (SensorValue(touchSensor)==0)
20 {
21     nxtDisplayStringAt(0, 31, "Read Dark Now");
22 }
23
24 darkValue=SensorValue(lightSensor);
25
26 sumValue = lightValue + darkValue;
27 thresholdValue = sumValue/2;
28
29 ClearTimer(T1);
30
31 while (time1[T1] < 3000)
32 {
33
```

Variables and Functions

Automatic Threshold Threshold Calculations (cont.)

```
33
34   if (SensorValue(lightSensor) < thresholdValue)
35   {
36
37       motor[motorC]=0;
38       motor[motorB]=80;
39
40   }
41
42   else
43   {
44
45       motor[motorC]=80;
46       motor[motorB]=0;
47
48   }
49
50 }
51
52 motor[motorC]=0;
53 motor[motorB]=0;
54
55 }
```

The robot now tracks a line with its own calculated threshold, and can advise users what to do, and when.



Variables

Automatic Thresholds Quiz

NAME _____ DATE _____

1. The output of a sensor is always in the form of a:

- a. value.
 - b. decimal.
 - c. threshold.
 - d. frequency.
-

2. If we want to store a decimal value in a variable named my_variable, then the variable type we select should be a(n) _____.

3. What does it mean to “declare” a variable?

4. Cross out the names on the following list, which **cannot** be used as variable names.

- a. true
 - b. my_variable
 - c. var1x
 - d. ants go marching
 - e. 1_by_1
 - f. one_by_one
 - g. motor
 - h. PB&J
-

5. Using the following bit of code, write a line of code that will calculate the value of a times b and store it in the variable “product”. What value will be in “product” after the line is run?

```
1 int a;  
2 int b;  
3 int product;  
4 a = 10;  
5 b = 100;  
6
```

Variables

6. In the space below, identify all the variables used in the Automatic Thresholds program, and briefly describe what each one does.

```

1  const tSensors touchSensor           = (tSensors) S1;
2  const tSensors lightSensor          = (tSensors) S2;
3
4  task main()
5  {
6    int lightValue;
7    int darkValue;
8    int sumValue;
9    int thresholdValue;
10 while (SensorValue(touchSensor) == 0)
11 {
12     nxtDisplayStringAt(0,31,"Read Light Now");
13 }
14 lightValue = SensorValue(lightSensor);
15 wait1Msec(1000);
16 while (SensorValue(touchSensor) == 0)
17 {
18     nxtDisplayStringAt(0,31,"Read Dark Now");
19 }
20 darkValue = SensorValue(lightSensor);
21 sumValue = lightValue + darkValue;
22 thresholdValue = sumValue/2;
23 ClearTimer(T1);
24 while (time1[T1] < 3000)
25 {
26     if (SensorValue(lightSensor) < thresholdValue)
27     {
28         motor[motorC] = 0;
29         motor[motorB] = 80;
30     }
31     else
32     {
33         motor[motorC] = 80;
34         motor[motorB] = 0;
35     }
36 }
37 motor[motorC] = 0;
38 motor[motorB] = 0;
39 }

```

Variables and Functions

Line Counting **Counting**

In this lesson, we're going to investigate the meaning of this curious line of code:

$$n = n + 1;$$

Reading it in the normal mathematical sense, this is a contradiction... an impossibility. There's no number out there that can be one more than itself. Of course, that would be misreading what the line says entirely. In fact, this is not an equation, but a command in ROBOTC, and a perfectly sensible one, when you understand what it's really saying.

Variables and Functions

Line Counting **Counting** (cont.)

In this lesson, we're going to learn how to use a robot to count. Using code we have already discussed, along with some new stuff, we will find out what we can do with this "line counting" concept.

Let's back up a step. Where have we seen something like this before?

motor[motorC] = 50;

...sets a motor power setting to the numeric value 50.

motor[motorB] = SensorValue(soundSensor);

...sets a motor power to match the value of a sensor reading.

thresholdValue = sumValue/2;

...sets one variable to be equal to another variable divided by two.

In all of these situations, the command is to **set a value to something**. To the **left** of the equal sign, is the variable or other quantity that is set. To the **right** of the equal sign, is the value that it will be set to.

$n = n + 1$; is part of the same family of commands. It is clearly not meant to say that "n is equal to n plus one," but rather that the program should set n equal to n plus one. How does that work? Well, if n starts at zero, then running this command sets n to be equal to 1. Let's substitute 0 for the n on the right side and see what happens.

n starts at 0, so...

$n = n + 1$; becomes **$n = 0 + 1$** ;

n is set to 1, so now...

$n = n + 1$; becomes **$n = 1 + 1$** ;

... and n is set to 2, so now...

$n = n + 1$; becomes **$n = 2 + 1$** ;

And so on! Each time you run the command $n = n + 1$; the value in the variable n is increased by 1!

Variables and Functions

Line Counting **Counting** (cont.)

When would this be useful? Let's examine the warehouse task in more detail.



To get around the warehouse, the robot needs to count lines. Every time you reach a new line, you add one to the number of lines that you've seen. In command form, that looks like:

count = count + 1;

The new count equals the current count plus one. Commands of the form $n = n + 1$, like this `count = count + 1`, add one to the value of the variable each time the command is executed, and can be used over and over to count upwards, leaving the current count in the variable each time. By running this line once each time you spot an object that you want to count, you can keep a running tally in your program, always stored in the same variable. **Your robot can count lines!** This will come in quite handy for this project.

Variables and Functions

Line Counting Line Counting (Part 1)

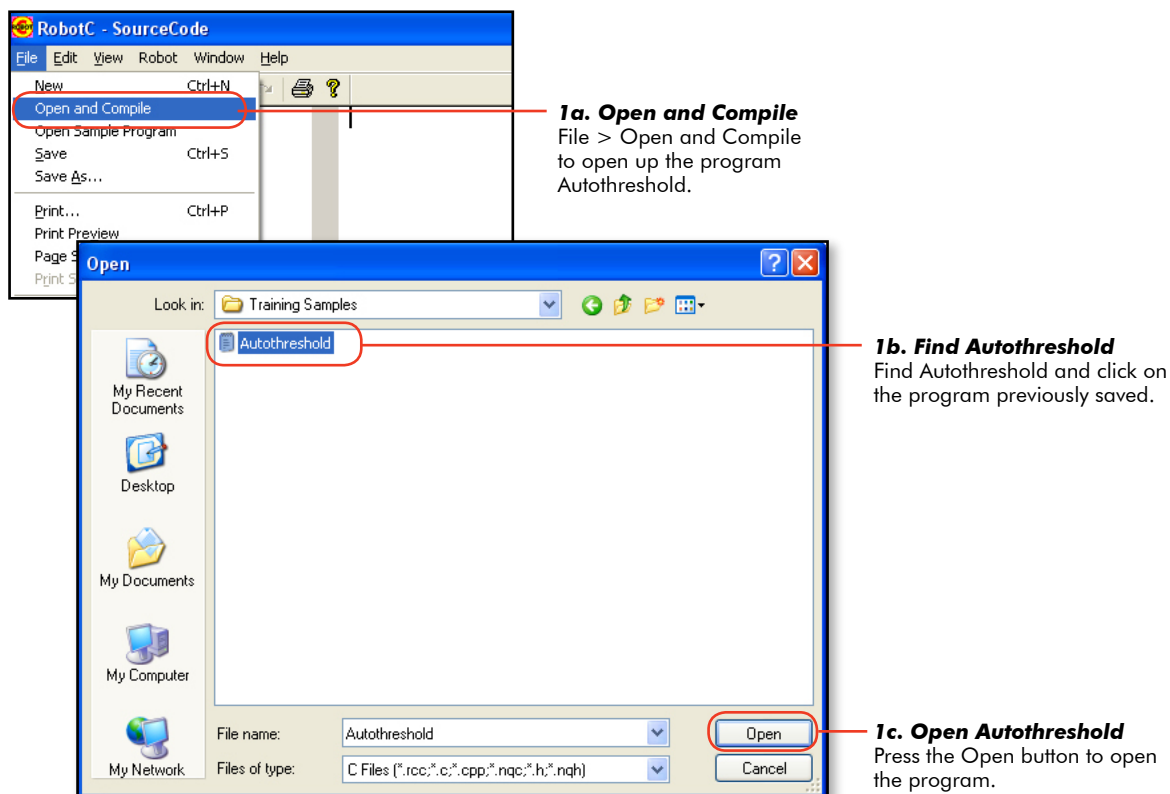
In this lesson, we're going to start teaching the robot to count lines. The eventual goal of this robot is to have it travel to a certain destination by counting special navigation markers on the floor.

We have one piece of the puzzle now, we know how to count.

What we still need to figure out are:

- When to count
- When NOT to count
- How to stop, based on the count

1. Start with your automatic threshold calculation program, the one that asks you to push the button over light and dark, and then tracks the line.



1a. Open and Compile
File > Open and Compile to open up the program Autothreshold.

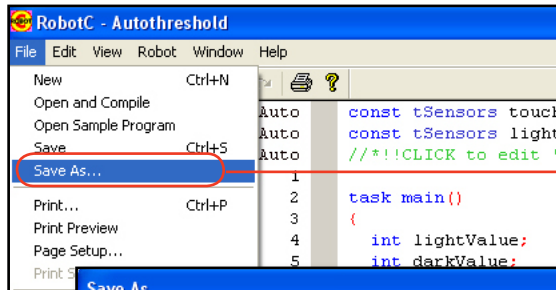
1b. Find Autothreshold
Find Autothreshold and click on the program previously saved.

1c. Open Autothreshold
Press the Open button to open the program.

Variables and Functions

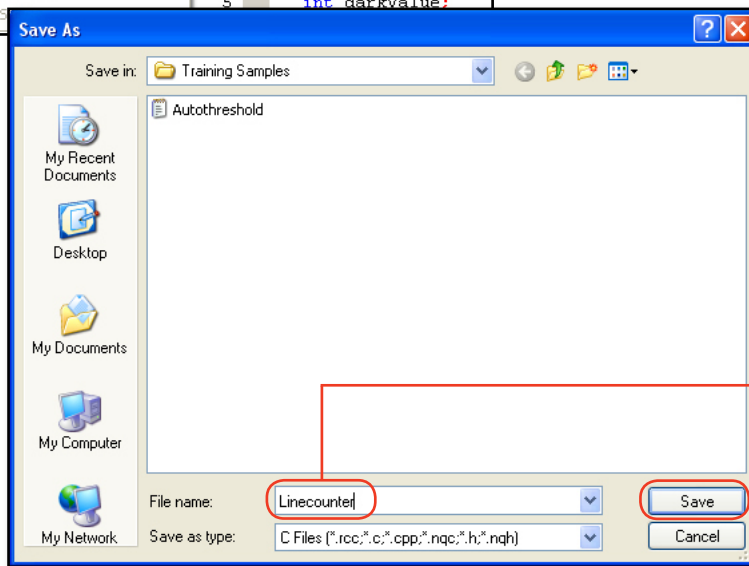
Line Counting **Line Counting (Part 1)** (cont.)

2. For this lesson, this program will be saved as, "Linecounter".



2a. Save As

Select File > Save As to save your existing code to a new file, with a new name.



2b. Name the program

Name the new program file "Linecounter".

2c. Save the program

Press the Save button to save the new program.

Variables and Functions

Line Counting **Line Counting (Part 1)** (cont.)

Checkpoint

This is what the program should look like before modifications.

```
2  task main()
3  {
4
5    int lightValue;
6    int darkValue;
7    int sumValue;
8    int thresholdValue;
9
10   while (SensorValue(touchSensor)==0)
11   {
12     nxtDisplayStringAt(0, 31, "Read Light Now");
13   }
14
15   lightValue=SensorValue(lightSensor);
16
17   wait1Msec(1000);
18
19   while (SensorValue(touchSensor)==0)
20   {
21     nxtDisplayStringAt(0, 31, "Read Dark Now");
22   }
23
24   darkValue=SensorValue(lightSensor);
25
26   sumValue = lightValue + darkValue;
27   thresholdValue = sumValue/2;
28
29   ClearTimer(T1);
30
31   while (time1[T1] < 3000)
32   {
33
34     if (SensorValue(lightSensor) < thresholdValue)
35     {
36
37       motor[motorC]=0;
38       motor[motorB]=80;
39
40     }
41
42     else
43     {
44
45       motor[motorC]=80;
46       motor[motorB]=0;
47
48     }
49
50   }
51
52   motor[motorC]=0;
53   motor[motorB]=0;
54
55 }
```

Variables and Functions

Line Counting **Line Counting (Part 1)** (cont.)

The existing program already has the sensors configured, and finds a nice threshold value so we don't have to worry about either of those. The task at hand, counting lines, involves looking for light or dark just like the Line Tracker did. But unlike the line tracker, our robot only needs to move straight forward, so let's convert over the parts of the code that do steering.

3. Change the first movement portion of the Line Tracking if-else statement to just make the robot go straight instead. Remove the other movement-related commands.

```

28
29 ClearTimer (T1);
30
31 while (time1[T1] < 3000)
32 {
33
34     if (SensorValue(lightSensor) < thresholdValue)
35     {
36
37         motor[motorC]=50;
38         motor[motorB]=50;
39
40     }
41
42     else
43     {
44
45         motor[motorC]=80;
46         motor[motorB]=0;
47
48     }
49
50 }
51
52     motor[motorC]=0;
53     motor[motorB]=0;
54
55 }
```

3a. Modify this code

Change both motorB and motorC to equal power levels. We will use 50.

3b. Delete this code

Delete both these sections of code, which steer the robot in the original line tracking program.

Variables and Functions

Line Counting **Line Counting (Part 1)** (cont.)

Checkpoint

This is what the program should look like after modifying the steering.

```
2  task main()
3  {
4
5    int lightValue;
6    int darkValue;
7    int sumValue;
8    int thresholdValue;
9
10   while (SensorValue(touchSensor)==0)
11   {
12     nxtDisplayStringAt(0, 31, "Read Light Now");
13   }
14
15   lightValue=SensorValue(lightSensor);
16
17   wait1Msec(1000);
18
19   while (SensorValue(touchSensor)==0)
20   {
21     nxtDisplayStringAt(0, 31, "Read Dark Now");
22   }
23
24   darkValue=SensorValue(lightSensor);
25
26   sumValue = lightValue + darkValue;
27   thresholdValue = sumValue/2;
28
29   ClearTimer(T1);
30
31   while (time1[T1] < 3000)
32   {
33
34     if (SensorValue(lightSensor) < thresholdValue)
35     {
36
37       motor[motorC]=50;
38       motor[motorB]=50;
39
40     }
41
42     else
43     {
44
45     }
46
47   }
48
49 }
```

Variables and Functions

Line Counting **Line Counting (Part 1)** (cont.)

4. Now let's add the lines to turn on PID control for both motors to help keep the robot moving in a straight line. If you need a refresher you can review PID in the improved movement section.

```

2  task main()
3  {
4
5      int lightValue;
6      int darkValue;
7      int sumValue;
8      int thresholdValue;
9
10     nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
11     nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;
12
13     while (SensorValue(touchSensor)==0)
14     {
15         nxtDisplayStringAt(0, 31, "Read Light Now");
16     }
17

```

4. Add this code

Add these two lines to turn on PID control for both motors.

5. Because we do want to look at light and dark for counting purposes, let's keep the light sensor if-else statement in place.

```

31
32     ClearTimer(T1);
33
34     while (time1[T1] < 3000)
35     {
36
37         if (SensorValue(lightSensor) < thresholdValue)
38         {
39
40             motor[motorC]=50;
41             motor[motorB]=50;
42
43         }
44
45         else
46         {
47
48         }
49
50     }
51
52 }

```


Variables and Functions

Line Counting **Line Counting (Part 1)** (cont.)

6. We're definitely going to be **counting** (lines), and we don't have a counter variable, so let's create one. It has to be an integer – it's a numeric value, and it won't have decimals – and we'll call it "countValue". After the name, add "= 0" before the semicolon. This statement declares an integer named "countValue" and assigns it an initial value of 0.

```
2  task main()
3  {
4
5     int lightValue;
6     int darkValue;
7     int sumValue;
8     int thresholdValue;
9     int countValue = 0;
10
11    nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
12    nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;
13
14    while (SensorValue(touchSensor)==0)
15    {
16        nxtDisplayStringAt(0, 31, "Read Light Now");
17    }
18
19    lightValue=SensorValue(lightSensor);
20
21    wait1Msec(1000);
22
23    while (SensorValue(touchSensor)==0)
24    {
25        nxtDisplayStringAt(0, 31, "Read Dark Now");
26    }
```

6. Add this code

Declare an integer variable named "countValue", with a value of 0. This variable will be used to count the number of lines we have passed.

Variables and Functions

Line Counting **Line Counting (Part 1)** (cont.)

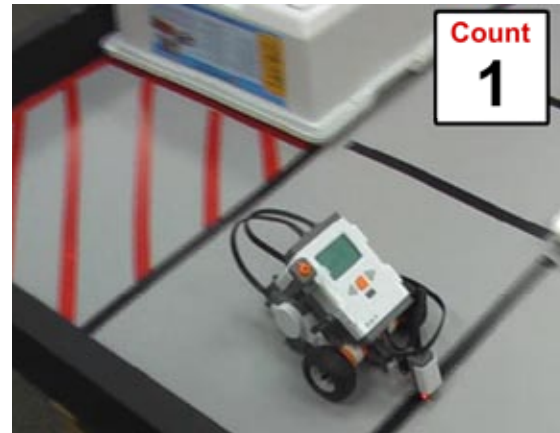
Checkpoint

Now, let's see what we should be doing in terms of counting. Suppose the robot starts running, and...



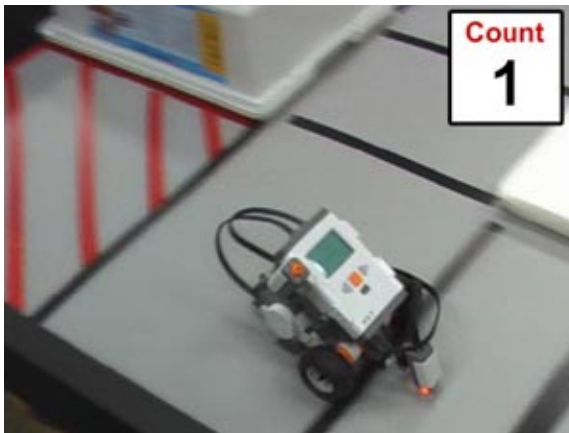
Light Sensor runs over light

Since the robot is running over light, it hasn't reached a line, and the line count should remain at zero.



Robot crosses first line

The robot has crossed its first line, so the line count should now increase to one.



Light Sensor runs over light

The robot is running over light again, and the line count remains at one.



Robot crosses second line

The robot has crossed its second line, so the line count increases again to two.

And so on. It looks like **dark means a line**, and that's when we want to count.

Count when dark... "If the light sensor value is lower than the threshold, count." The adding-one code should go in the part of the code that is run **when the value of the Light Sensor is below the threshold**: inside the {body} of the if-else statement starting on line 38.

Variables and Functions

Line Counting **Line Counting (Part 1)** (cont.)

7. Put the add-one code `countValue = countValue + 1;` in the “seeing dark” part of the if-else statement. The “else” block of code should remain empty so that the robot does nothing when it’s over a light area, just like we want.

```
32
33 ClearTimer(T1);
34
35 while (time1[T1] < 3000)
36 {
37
38     if (SensorValue(lightSensor) < thresholdValue)
39     {
40
41         motor[motorC]=50;
42         motor[motorB]=50;
43         countValue = countValue + 1;
44     }
45
46
47     else
48     {
49
50     }
51
52 }
53
54 }
```

7. Add this code

Insert the add-one code here, so that the robot adds one to the line count whenever it sees dark.

Variables and Functions

Line Counting **Line Counting (Part 1)** (cont.)

Checkpoint

This is what the program should look like after adding the add-one code.

```

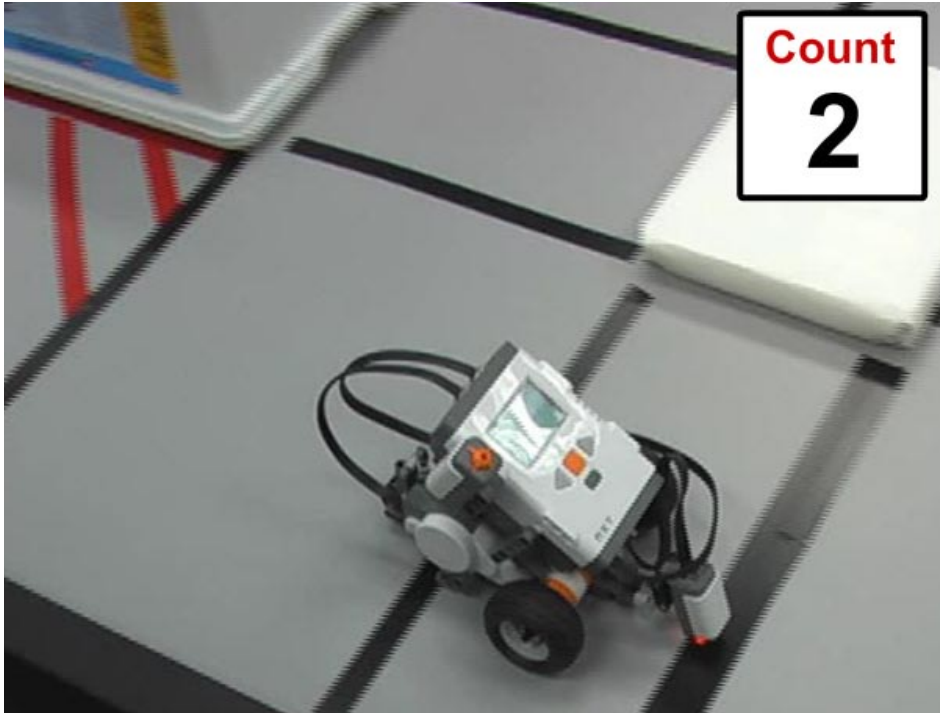
2  task main()
3  {
4
5     int lightValue;
6     int darkValue;
7     int sumValue;
8     int thresholdValue;
9     int countValue = 0;
10
11    nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
12    nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;
13
14    while (SensorValue(touchSensor)==0)
15    {
16        nxtDisplayStringAt(0, 31, "Read Light Now");
17    }
18
19    lightValue=SensorValue(lightSensor);
20
21    wait1Msec(1000);
22
23    while (SensorValue(touchSensor)==0)
24    {
25        nxtDisplayStringAt(0, 31, "Read Dark Now");
26    }
27
28    darkValue=SensorValue(lightSensor);
29
30    sumValue = lightValue + darkValue;
31    thresholdValue = sumValue/2;
32
33    ClearTimer(T1);
34
35    while (time1[T1] < 3000)
36    {
37
38        if (SensorValue(lightSensor) < thresholdValue)
39        {
40
41            motor[motorC]=50;
42            motor[motorB]=50;
43            countValue = countValue + 1;
44
45        }
46
47        else
48        {
49
50        }
51
52    }
53
54 }
```

Variables and Functions

Line Counting **Line Counting (Part 1)** (cont.)

End of Section

We've written code that tells the robot when to count.
Still to come: testing and debugging the program.

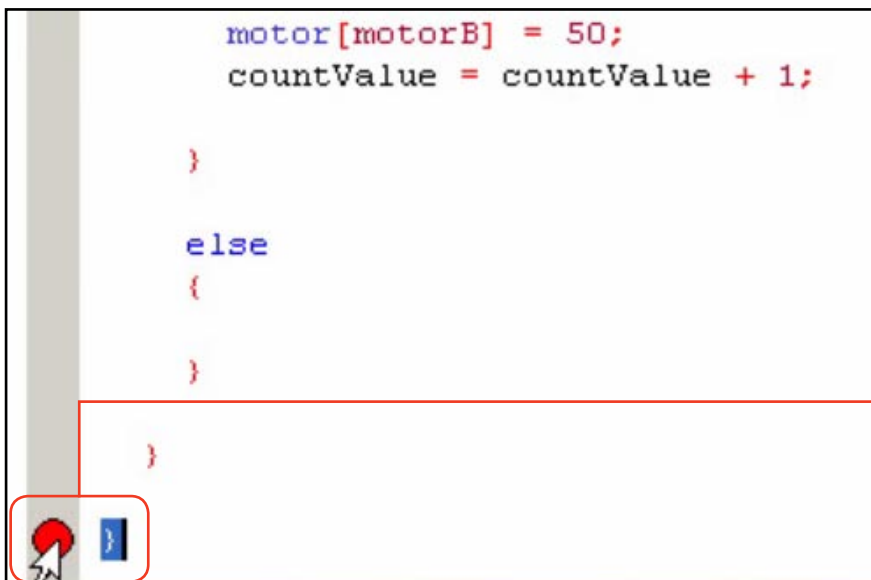


Variables and Functions

Line Counting Line Counting (Part 2)

In this lesson, we're going to learn how to use a breakpoint to debug the line counting program.

1. Before we test, we need to add something that will help us determine whether the program is working or not, before it just reaches the end and erases all the data. Since we have the **debugger** on our side, there's a trick we can use. On the very last line of your program, click once in the grey bar between the code and the line number. A **red circle** will appear, marking the creation of a **breakpoint**.

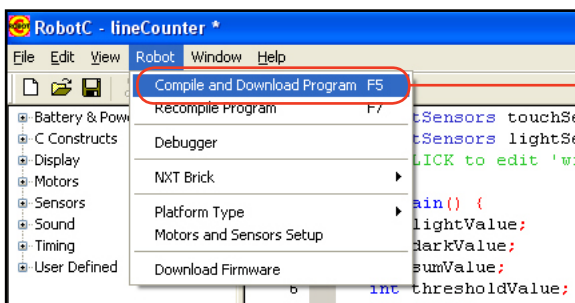


1. Add breakpoint

Place your cursor next to the last curly brace, then click in the grey bar to create a breakpoint, marked by a red circle.

A breakpoint is a spot you designate in your code where the robot should automatically go into a time-stop state, as it did while using the step command. The advantage to using a "breakpoint" rather than the "step" approach allows your robot to run the program at normal speed until the program reaches the break point.

2. Compile and Download the program to the robot to begin the test!



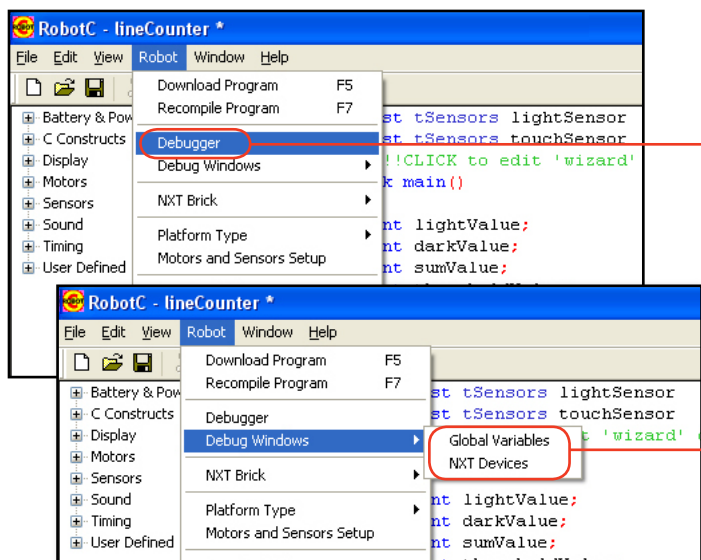
2. Compile and Download

Robot > Compile and Download Program

Variables and Functions

Line Counting **Line Counting (Part 2)** (cont.)

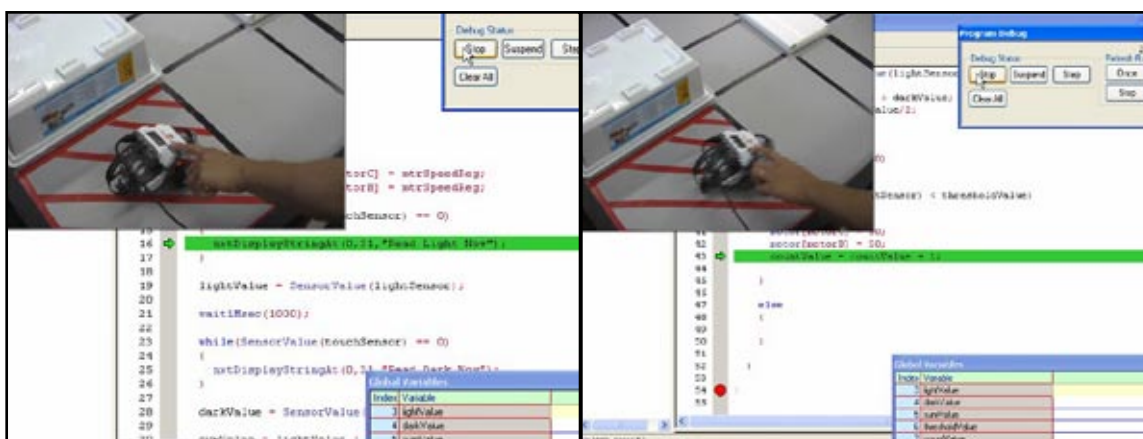
3. Open up the Debugger, then select both the Global Variables and the NXT Devices options so both these windows are visible.



3a. View Debugger
Select Robot > Debugger to open up the Program Debug window.

3b. View Debugger Windows
Select Robot > Debug Windows and select **both** Global Variables and NXT Devices.

4. Start the program, then follow the prompts on the NXT screen to press the Touch Sensor to store the values of light and dark surfaces in the variables lightValue and darkValue. The second time you press the Touch Sensor, the robot should begin moving forward.



4a. Find lightValue

Push the Touch Sensor while the robot's Light Sensor is over a light area.

4b. Find darkValue

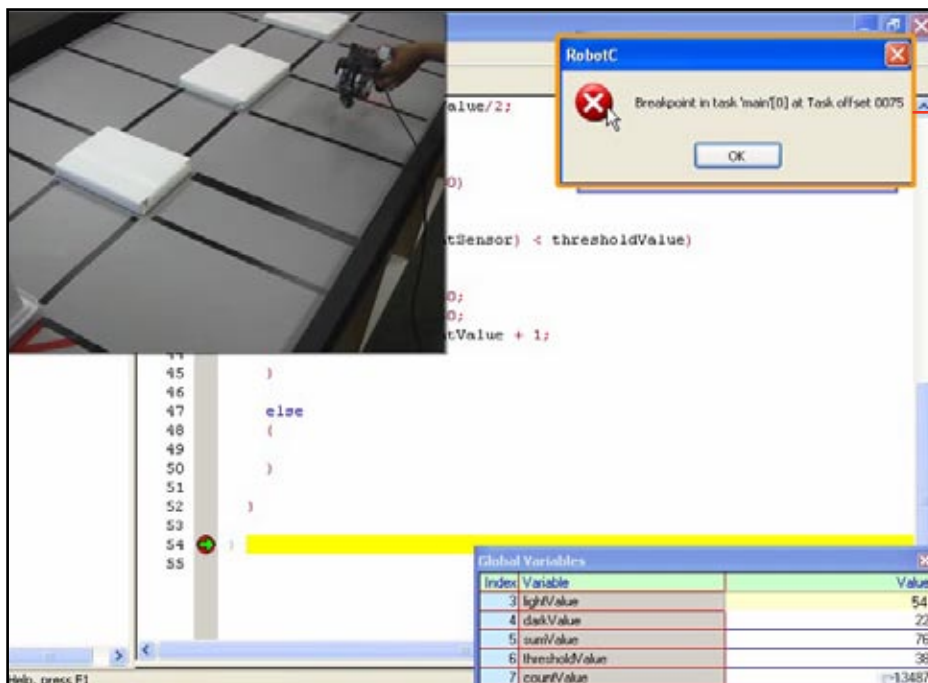
Wait 1 second, position the robot with the Light Sensor over the first line and positioned to go forward, and press the Touch Sensor again. As soon as the Touch Sensor is pressed, the robot will begin to move forward, counting lines as it goes.

Variables and Functions

Line Counting **Line Counting (Part 2)** (cont.)

Checkpoint

Since the line tracker we originally borrowed the code from was the Timer version, this behavior should run for a set amount of time, then hit the breakpoint. The program state freezes when it hits the breakpoint, so the motors keep running – they were running when we froze the program, so they'll keep running because there's nothing to tell them to stop.



Breakpoint

This dialog tells you that your program has reached the breakpoint you set.

- Observe the variables window, and find the value of your variable "countValue", which should be the number of lines your robot passes over. The number of lines the robot has passed appears to be... negative 13,487.

| Index | Variable | Value |
|-------|----------------|--------|
| 3 | lightValue | 54 |
| 4 | darkValue | 22 |
| 5 | sumValue | 76 |
| 6 | thresholdValue | 38 |
| 7 | countValue | -13487 |

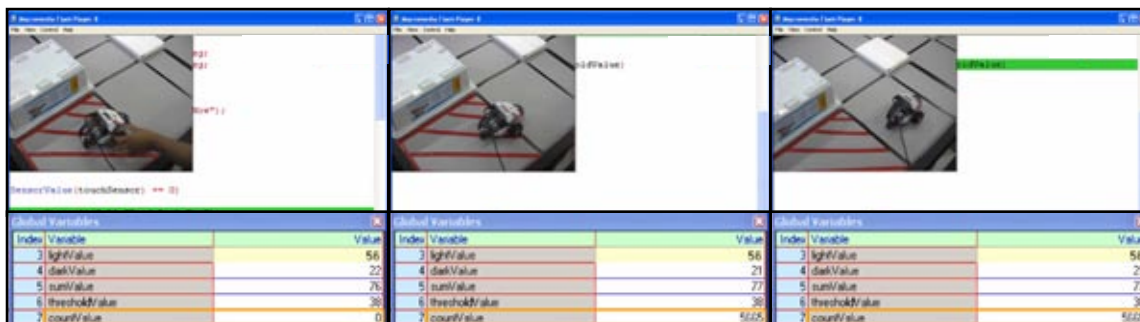
5. Observe "countValue"

Observe the value of the last variable, "countValue" in the Global Variables window.

Variables and Functions

Line Counting **Line Counting (Part 2)** (cont.)

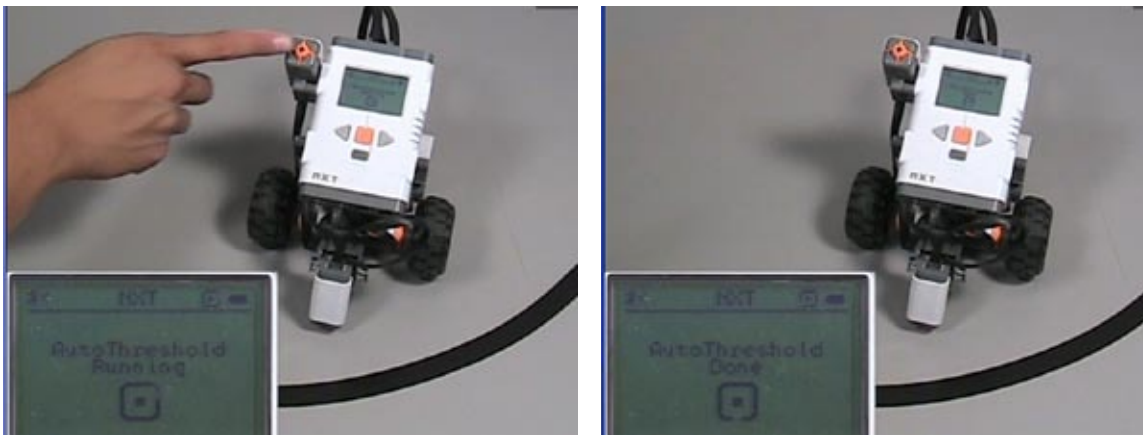
- Run the program again with the robot connected, this time watching to the value of the variable "countValue" in the variable window as the robot runs.



Checkpoint

Look what happens to the variable "countValue." It doesn't move when we're over light, which it shouldn't. But when you place it over the dark line and press the Touch Sensor it counts more than once – thousands of times, actually! The number gets so big that it confuses ROBOTC and wraps around into the negative numbers!

Counting more than once is the same problem we had when we were trying to detect the Touch Sensor press to read Thresholds! Remember back when the program zipped through both readings too quickly because the Touch Sensor was still held when the program reached the second check?



Flashback: Counting too fast

Back in the Automatic Thresholds section, you had another situation where the robot counted too many times, too quickly, and did not work correctly as a result.

Variables and Functions

Line Counting **Line Counting (Part 2)** (cont.)

It looks like the same thing is happening here, but about 10,000 times worse. Per second.

| Index | Variable | Value |
|-------|----------------|-------|
| 3 | lightValue | 56 |
| 4 | darkValue | 21 |
| 5 | sumValue | 77 |
| 6 | thresholdValue | 38 |
| 7 | countValue | 5665 |

Let's look at what happens when the robot crosses a dark line. It checks the if-condition, which is true, and adds one to the variable "countValue".

```

32
33 ClearTimer (T1);
34
35 while (time1[T1] < 3000)
36 {
37
38     if (SensorValue(lightSensor) < thresholdValue)
39     {
40
41         motor[motorC]=50;
42         motor[motorB]=50;
43         countValue = countValue + 1;
44     }
45
46     else
47     {
48     }
49
50 }
51
52 }
53
54 }

```

If (condition)
The robot checks the condition.
When it is true, it adds one to the
value of "countValue."

Variables and Functions

Line Counting **Line Counting (Part 2)** (cont.)

Then it skips the else, and moves back up to the top of the while() loop.

```

32
33 ClearTimer (T1);
34
35 while (time1[T1] < 3000)
36 {
37
38   if (SensorValue(lightSensor) < thresholdValue)
39   {
40
41     motor[motorC]=50;
42     motor[motorB]=50;
43     countValue = countValue + 1;
44
45   }
46
47   else
48   {
49
50   }
51
52 }
53
54

```

Top of the while() loop

After adding one to "countValue", the robot moves back to the top of the while() loop.

Then it does what it did before: it checks the condition... **which is still true even on this second pass because the sensor is still over the line**, and adds *another* 1 to "countValue".

```

32
33 ClearTimer (T1);
34
35 while (time1[T1] < 3000)
36 {
37
38   if (SensorValue(lightSensor) < thresholdValue)
39   {
40
41     motor[motorC]=50;
42     motor[motorB]=50;
43     countValue = countValue + 1;
44
45   }
46
47   else
48   {
49
50   }
51
52 }
53
54

```

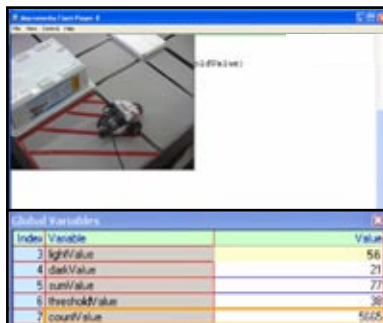
If (condition) again...

The robot again checks the condition, which is still true, and adds one to the value of "countValue."

Variables and Functions

Line Counting Line Counting (Part 2) (cont.)

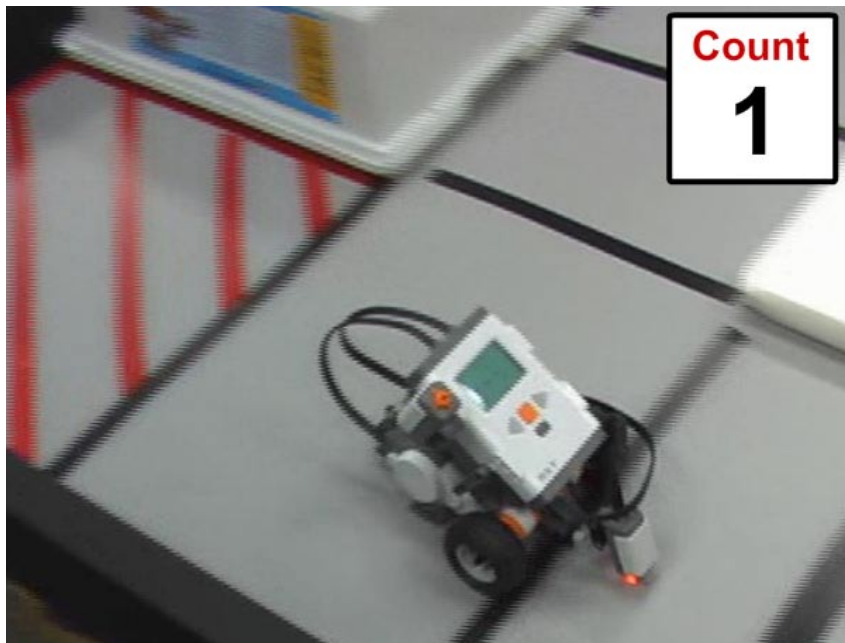
The robot keeps cycling through the while loop over and over again, and keeps adding one to "countValue" every time it does. And this is the problem: the robot is seeing, and hence **counting**, the **same black line** for what seems to be **thousands of cycles** in the amount of time it takes to pass over it.



| Index | Variable | Value |
|-------|----------------|-------|
| 3 | lightValue | 56 |
| 4 | darkValue | 21 |
| 5 | sumValue | 77 |
| 6 | thresholdValue | 30 |
| 7 | countValue | 5555 |

End of Section

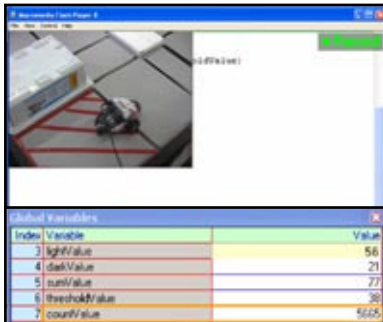
We've found the problem: the robot counts one black line thousands of times when we only want to count the line only once. In the next lesson you will use a variable to put a stop to the double counting.



Variables and Functions

Line Counting Line Counting (Part 3)

We need to find some way to make the robot count the line only once.



In the Autothreshold program, we solved the problem by putting in a one second delay to allow you to take your finger off the button before the program moves to the next line of code.

```

2  task main()
3  {
4
5    int lightValue;
6    int darkValue;
7    int sumValue;
8    int thresholdValue;
9
10   while (SensorValue(touchSensor)==0)
11   {
12     nxtDisplayStringAt(0, 31, "Read Light Now");
13   }
14
15   lightValue=SensorValue(lightSensor);
16
17   wait1Msec(1000);
18
19   while (SensorValue(touchSensor)==0)
20   {
21     nxtDisplayStringAt(0, 31, "Read Dark Now");
22   }
23
24   darkValue=SensorValue(lightSensor);
25
26   sumValue = lightValue + darkValue;
27   thresholdValue = sumValue/2;
28

```

Variables and Functions

Line Counting **Line Counting (Part 3)** (cont.)

A one-second pause worked well for the button-pushing situation, but is it really appropriate here? What if the lines are close together? The robot could miss a lot of lines in that one second gap. Or what if the line is really huge? It would still count more than once.



Multiple close lines

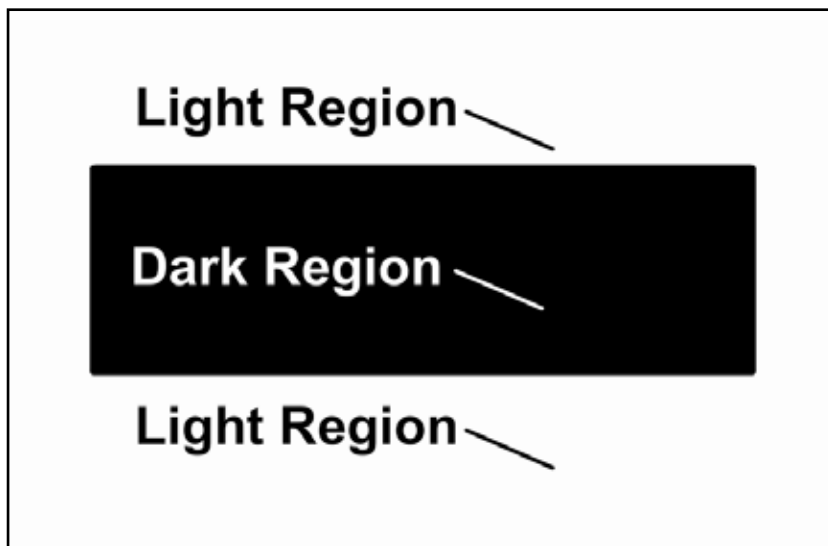
The robot should count all of these lines separately, but could potentially drive over all of them during the “don’t count again” period, and end up counting them as only one line.



Single thick line

If a line is thick enough for whatever reason, the robot may still not get past it before counting again, and it would be counted twice.

It looks like we’ll have to come up with something more creative. We could look at this line as being **made up of several distinct regions**: one light region where you come in from, a dark region, and then another light region.



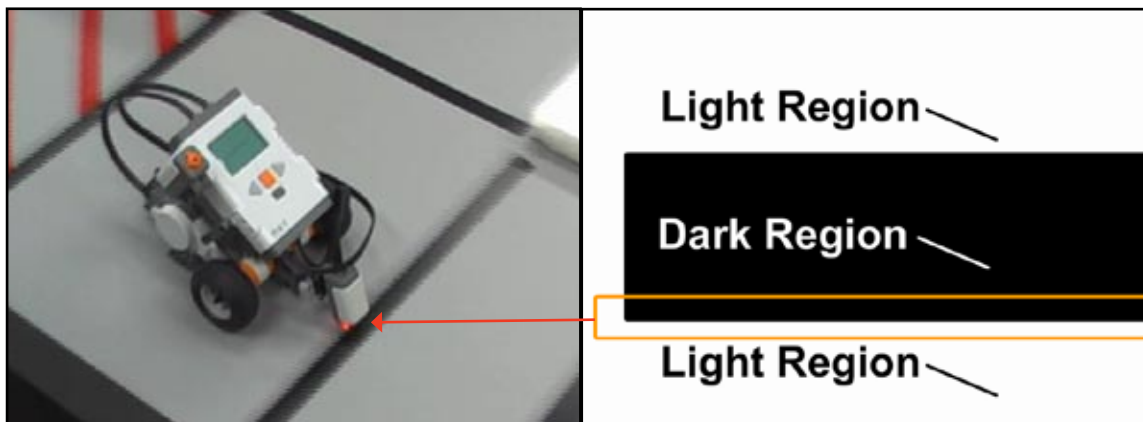
Anatomy of a Line

A line is composed of a light region, followed by a dark region, followed by another light region.

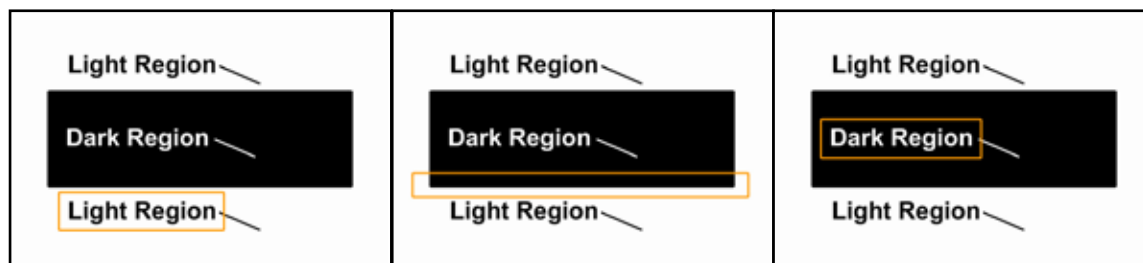
Variables and Functions

Line Counting **Line Counting (Part 3)** (cont.)

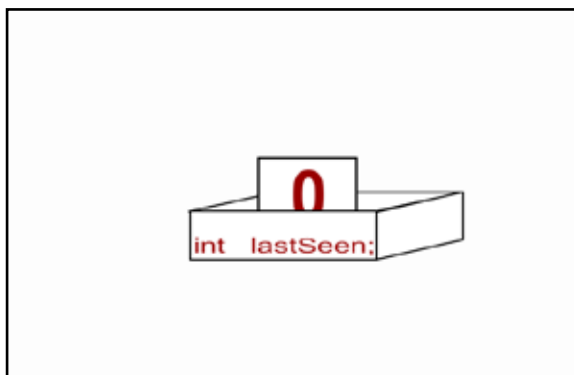
Why not let the robot count only on the **transition** from light to dark? If you look at this picture there is only one light to dark transition per line. And exactly one. So you can count every line and never count the same line twice. What we want to count is not “seeing dark”, but “**seeing the transition to dark.**”



What does this transition look like? The transition is when you **used to** be seeing light, as in the picture below left, and **now** are seeing dark, as in the picture below center.



But how do we keep track of that? What we need is **a variable** to store **the color of the region that we saw last.**



Variables and Functions

Line Counting **Line Counting (Part 3)** (cont.)

In this lesson, we're going to learn how to make our line counter count a line only once, by counting only the transition to dark. A variable will be used to keep track of the previously seen color.

1. Declare a new integer variable, int, and call it "lastSeen".

```
2 task main()
3 {
4
5     int lightValue;
6     int darkValue;
7     int sumValue;
8     int thresholdValue;
9     int countValue = 0;
10    int lastSeen;
```

1. Modify code

Declare a new integer variable called "lastSeen".

Checkpoint

We'll decide now that "lastSeen" is going to have 0 in it if the last thing it saw was dark, and a 1 in it if the last thing it saw was light. This is an arbitrary choice, but one that must be kept consistent after this point!

0 = dark

1 = light

Variables and Functions

Line Counting Line Counting (Part 3) (cont.)

2. Just before the while() loop, start the value of lastSeen at 1 (which is "light") so that we are able to count the first line

```

30
31  sumValue = lightValue + darkValue;
32  thresholdValue = sumValue/2;
33
34  ClearTimer(T1);
35  lastSeen = 1;
36
37  while (time1[T1] < 3000)
38  {
39
40      if (SensorValue(lightSensor) < thresholdValue)
41      {
42
43          motor[motorC]=50;
44          motor[motorB]=50;
45          countValue = countValue + 1;
46
47      }

```

2. Modify code

Assign the variable "lastSeen" the value of 1 just before the while() loop.

3. The rest of the program needs to make sure this variable stays up to date. In the block of code corresponding to the "dark" area of the if-else loop, add the line "lastSeen = 0;" And in the block for the "light" area (inside the else block), add the line "lastSeen = 1;".

```

30
31  sumValue = lightValue + darkValue;
32  thresholdValue = sumValue/2;
33
34  ClearTimer(T1);
35  lastSeen = 1;
36
37  while (time1[T1] < 3000)
38  {
39
40      if (SensorValue(lightSensor) < thresholdValue)
41      {
42
43          motor[motorC]=50;
44          motor[motorB]=50;
45          countValue = countValue + 1;
46          lastSeen = 0;
47      }
48
49
50      else
51      {
52          lastSeen = 1;
53      }
54
55  }
56
57 }

```

2. Modify code

Assign the variable "lastSeen" the value of 1 just before the while() loop.

3a. Modify code

Assign the variable "lastSeen" the value of 0 in the "dark" area of the if-else structure.

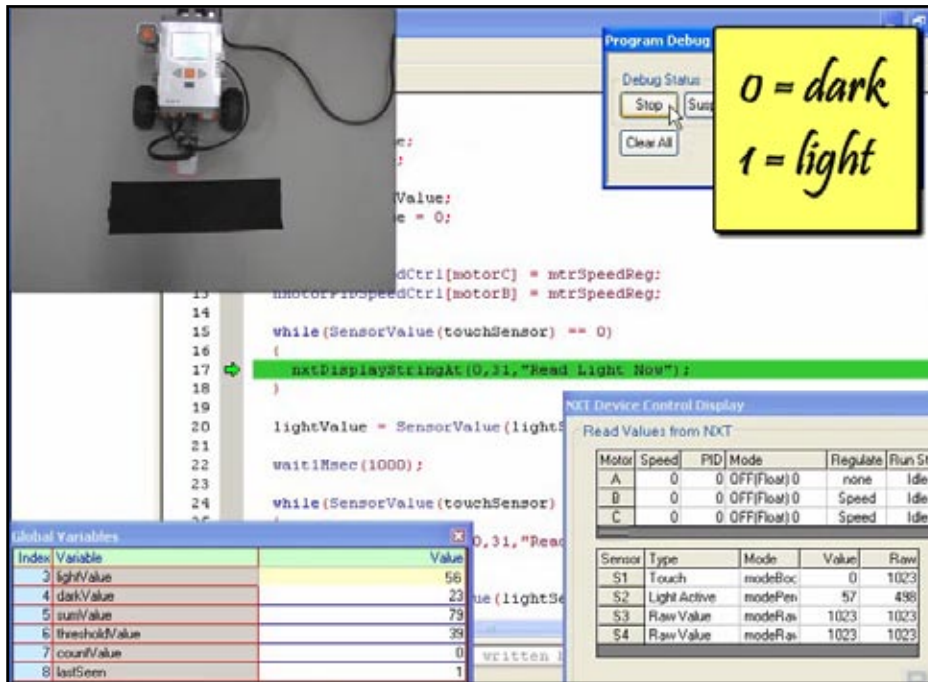
3b. Modify code

Assign the variable "lastSeen" the value of 1 in the "light" area of the if-else structure.

Variables and Functions

Line Counting **Line Counting (Part 3)** (cont.)

- Save, compile and download the program to the robot to see how we are doing. Bring up the **Debugger**, the **Global Variables Window** and the **NXT Device Control Display**.

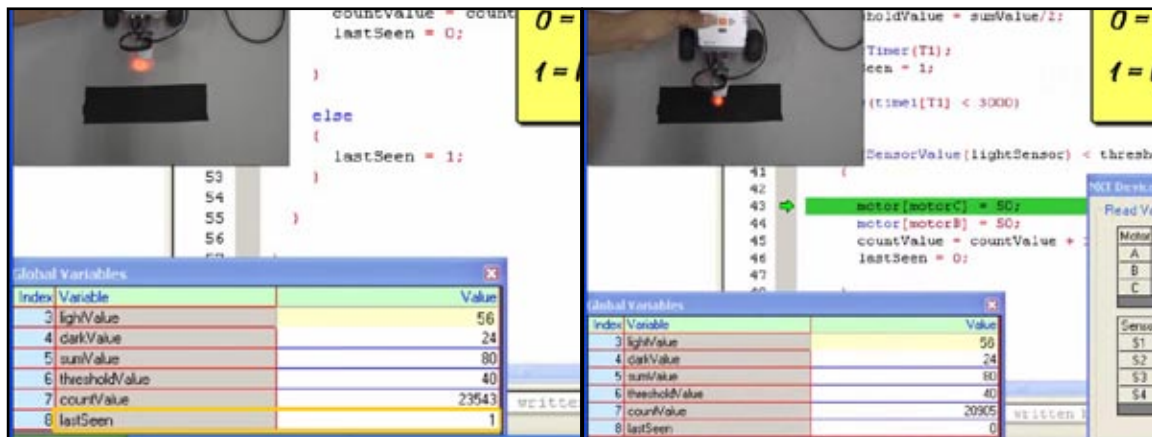


Debuggers

Compile and Download the program, then make sure the debugger windows are still open.

Checkpoint

Run the robot, but **pick it up and hold it over either the dark or light areas**. Whenever it's over the dark area, "lastSeen" should be 0. Whenever it's over the light area, "lastSeen" should be 1.



Robot held over Light

When the Light Sensor is held over the light area, the lastSeen variable in the Global Variables window should be 1.

Robot held over Dark

When the Light Sensor is held over the dark line, the lastSeen variable in the Global Variables window should be 0.

Variables and Functions

Line Counting **Line Counting (Part 3)** (cont.)

5. A light-to-dark transition will be marked by a “**last seen**” color of light, and a “**currently seeing**” color of dark. Therefore, the counting must be in the seeing-dark portion of the code, but should **also check** that the “lastSeen” value is light, a value of 1.

Create an if-else structure (beginning with the line “if (lastSeen == 1)” **around** the existing code. The “else” portion is actually optional, and is left out here to save space.

```

30
31  sumValue = lightValue + darkValue;
32  thresholdValue = sumValue/2;
33
34  ClearTimer(T1);
35  lastSeen = 1;
36
37  while (time1[T1] < 3000)
38  {
39
40    if (SensorValue(lightSensor) < thresholdValue)
41    {
42
43      motor[motorC]=50;
44      motor[motorB]=50;
45
46      if (lastSeen == 1)
47      {
48        countValue = countValue + 1;
49        lastSeen = 0;
50      }
51    }
52  }
53
54  else
55  {
56    lastSeen = 1;
57  }
58
59  }
60
61  }

```

5. Modify code
Create an if structure which checks if the variable “lastSeen” is equal to 1. The add-one code should become its {body}.

Variables and Functions

Line Counting **Line Counting (Part 3)** (cont.)

6. Save, download, and run. Watch the count variable in your program as the robot travels over lines, and with any luck, the count will match the number of lines!

```

ClearTimer(T1);
lastSeen = 1;

while(time1[T1] < 3000)
{
  if (SensorValue (light
  {
    motor[motorC] = 50;
    motor[motorB] = 50;

    if (lastSeen == 1)
    {

```

| Variable | Value |
|----------------|-------|
| lightValue | 55 |
| darkValue | 23 |
| sumValue | 78 |
| thresholdValue | 39 |
| countValue | 1 |
| lastSeen | 0 |

Success

The robot now travels for 3 seconds, counting appropriately only when it has reached a new line (a light-to-dark transition).

Observe the value of “countValue” in the debug window for each position of the robot shown below.

```

ClearTimer(T1);
lastSeen = 1;

while(time1[T1] < 3000)
{
  if (SensorValue (light
  {
    motor[motorC] = 50;
    motor[motorB] = 50;

    if (lastSeen == 1)
    {

```

| Variable | Value |
|----------------|-------|
| lightValue | 55 |
| darkValue | 23 |
| sumValue | 78 |
| thresholdValue | 39 |
| countValue | 2 |
| lastSeen | 0 |

```

ClearTimer(T1);
lastSeen = 1;

while(time1[T1] < 3000)
{
  if (SensorValue (light
  {
    motor[motorC] = 50;
    motor[motorB] = 50;

    if (lastSeen == 1)
    {

```

| Variable | Value |
|----------------|-------|
| lightValue | 55 |
| darkValue | 23 |
| sumValue | 78 |
| thresholdValue | 39 |
| countValue | 3 |
| lastSeen | 0 |

Variables and Functions

Line Counting **Line Counting (Part 3)** (cont.)

Checkpoint. This is what the program should look like after all your modifications.

```

2  task main()
3  {
4
5     int lightValue;
6     int darkValue;
7     int sumValue;
8     int thresholdValue;
9     int countValue = 0;
10    int lastSeen;
11
12    nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
13    nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;
14
15    while (SensorValue(touchSensor)==0)
16    {
17        nxtDisplayStringAt(0, 31, "Read Light Now");
18    }
19
20    lightValue=SensorValue(lightSensor);
21
22    wait1Msec(1000);
23
24    while (SensorValue(touchSensor)==0)
25    {
26        nxtDisplayStringAt(0, 31, "Read Dark Now");
27    }
28
29    darkValue=SensorValue(lightSensor);
30
31    sumValue = lightValue + darkValue;
32    thresholdValue = sumValue/2;
33
34    ClearTimer(T1);
35    lastSeen = 1;
36
37    while (time1[T1] < 3000)
38    {
39
40        if (SensorValue(lightSensor) < thresholdValue)
41        {
42
43            motor[motorC]=50;
44            motor[motorB]=50;
45
46            if (lastSeen == 1)
47            {
48                countValue = countValue + 1;
49                lastSeen = 0;
50            }
51
52        }
53
54        else
55        {
56            lastSeen = 1;
57        }
58
59    }
60
61 }

```

Variables and Functions

Line Counting **Line Counting (Part 3)** (cont.)

End of Section

We've covered the first two items we need for our line counting program. In the next section, we'll learn how to stop.

Still to come...

- When to count
- When NOT to count
- How to stop

Variables and Functions

Line Counting Line Counting (Part 4)

In this lesson, we're going to learn how to make our line counter stop when it has passed over a specific number of lines, instead of stopping after a specific amount of time has elapsed.

The Line Tracking code we originally borrowed was the Timer version, which works by running while the elapsed time value is **less than the time limit**. Right now it loops until 3000 milliseconds have passed. What we really want is for this robot to move until it has **passed 7 lines**.

```

33
34   ClearTimer(T1);
35   lastSeen = 1;
36
37   while (time1[T1] < 3000)
38   {
39
40       if (SensorValue(lightSensor) < thresholdValue)
41       {
42
43           motor[motorC]=50;
44           motor[motorB]=50;
45
46           if (lastSeen == 1)
47           {
48               countValue = countValue + 1;
49               lastSeen = 0;
50           }
51
52       }
53
54       else
55       {
56           lastSeen = 1;
57       }
58
59   }
60
61 }
```

While loop

The (condition) in the while loop determines whether the move-and-count behavior continues, or whether the program moves on to the next behavior.

Variables and Functions

Line Counting **Line Counting (Part 4)** (cont.)

1. Replace the "Timer < 3000" condition with "countValue < 7".

```

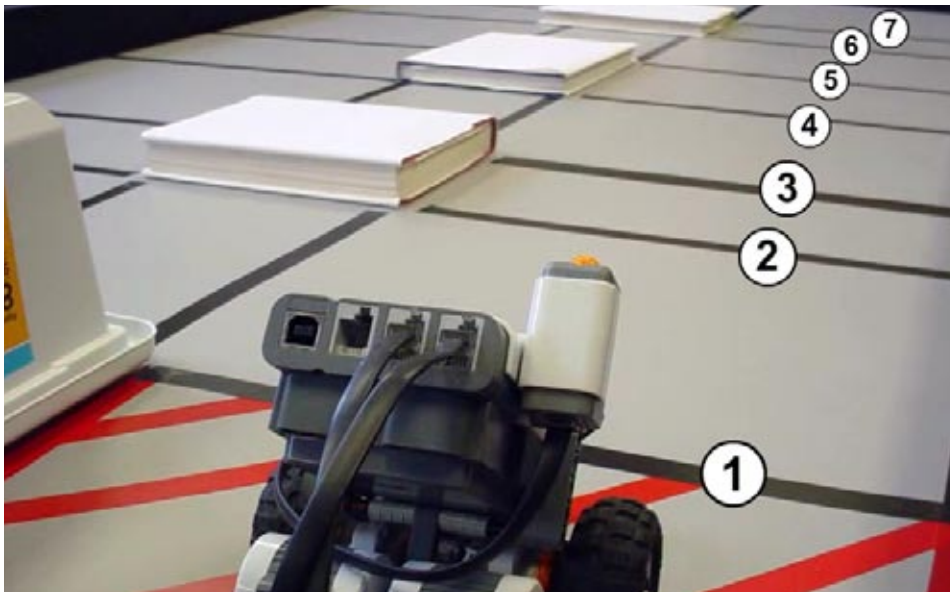
33
34   ClearTimer(T1);
35   lastSeen = 1;
36
37   while (countValue < 7)
38   {
39
40       if (SensorValue(lightSensor) < thresholdValue)
41       {
42
43           motor[motorC]=50;
44           motor[motorB]=50;
45
46           if (lastSeen == 1)
47           {
48               countValue = countValue + 1;
49               lastSeen = 0;
50           }
51
52       }
53
54   else
55   {
56       lastSeen = 1;
57   }
58

```

1. **Modify code**

Change the condition the while() loop checks from "Timer < 3000" to "countValue < 7".

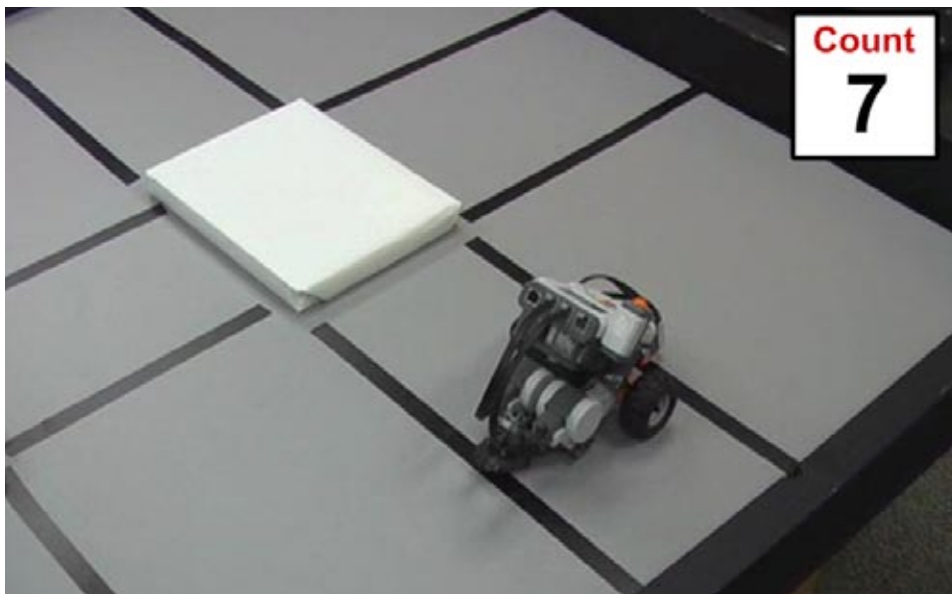
2. Make sure your table has at least seven lines on it.



Variables and Functions

Line Counting **Line Counting (Part 4)** (cont.)

3. Save, download and run.



Variables and Functions

Line Counting Line Counting (Part 4) (cont.)

End of Section If this is what your program looks like, you've finished the line counting program!

```

2  task main()
3  {
4
5  int lightValue;
6  int darkValue;
7  int sumValue;
8  int thresholdValue;
9  int countValue = 0;
10 int lastSeen;
11
12 nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
13 nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;
14
15 while (SensorValue(touchSensor)==0)
16 {
17     nxtDisplayStringAt(0, 31, "Read Light Now");
18 }
19
20 lightValue=SensorValue(lightSensor);
21
22 wait1Msec(1000);
23
24 while (SensorValue(touchSensor)==0)
25 {
26     nxtDisplayStringAt(0, 31, "Read Dark Now");
27 }
28
29 darkValue=SensorValue(lightSensor);
30
31 sumValue = lightValue + darkValue;
32 thresholdValue = sumValue/2;
33
34 ClearTimer(T1);
35 lastSeen = 1;
36
37 while (countValue < 7)
38 {
39
40     if (SensorValue(lightSensor) < thresholdValue)
41     {
42
43         motor[motorC]=50;
44         motor[motorB]=50;
45
46         if (lastSeen == 1)
47         {
48             countValue = countValue + 1;
49             lastSeen = 0;
50         }
51     }
52 }
53
54 else
55 {
56     lastSeen = 1;
57 }
58 }
59 }
60 }
61 }

```

Variables

Line Counting Quiz

NAME _____ DATE _____

```

1  int lastSeen;
2  void forward4Lines()
3  {
4    lastSeen = 1;
5    while(countValue < 4)
6    {
7      if(SensorValue(lightSensor) < thresholdValue)
8      {
9        motor[motorC] = 50;
10       motor[motorB] = 50;
11       if(lastSeen == 1)
12       {
13         countValue = countValue + 1;
14         lastSeen = 0;
15       }
16     }
17     else
18     {
19       lastSeen = 1;
20     }
21   }
22 }
```

1. Explain, in your own words, how the “lastSeen” variable prevents double-counting of lines in the program above.

```
1  n = n + 1;
```

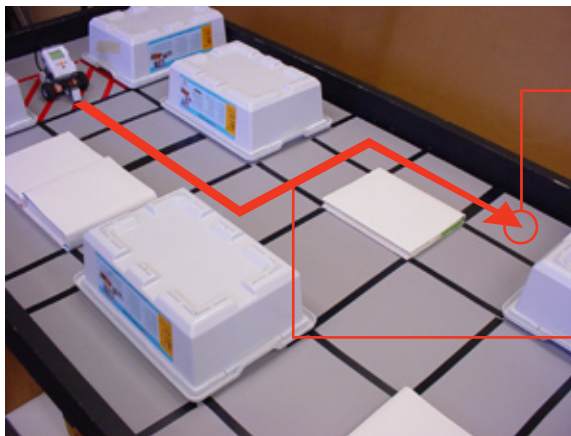
2. Explain what the line of code above does.

Variables and Functions

Patterns of Behavior Behaviors

In this lesson, you will learn how the various ways of navigating the warehouse environment break down into a common set of sub-behaviors.

A typical task for the inventory robot may be to retrieve the object in aisle five. How can the robot get there? The robot is not yet advanced enough to determine its own path, so it will require human assistance to find a route. For example:



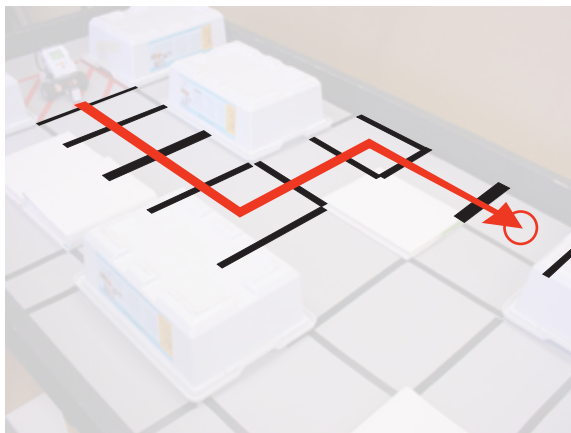
Destination

The robot must navigate to this location to retrieve an item.

Path

A human programmer chose this path for the robot (others were also possible).

In order to follow the path above, the robot needs a way to orient itself in the warehouse environment. With the irregular spacing between shelves, distance may not be reliable. Instead, the robot must rely on the floor markings.



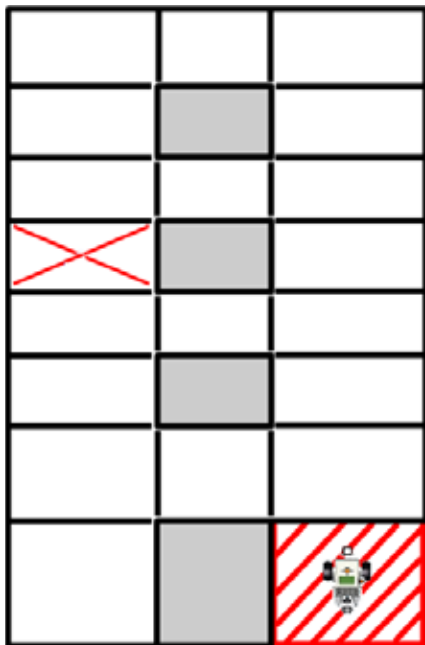
Landmarks

This robot will rely on floor markings to help it find its way along the path.

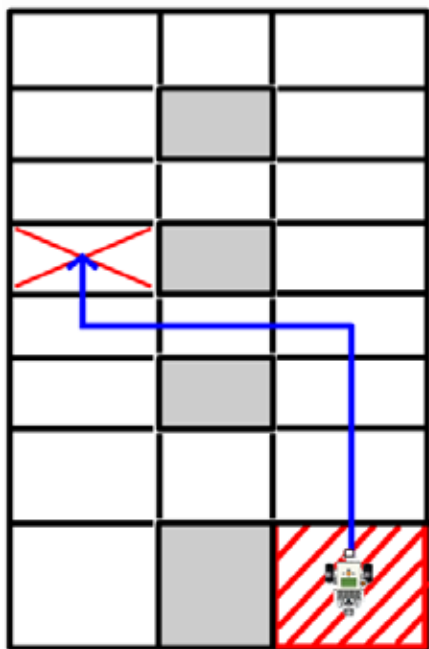
Variables and Functions

Patterns of Behavior **Behaviors** (cont.)

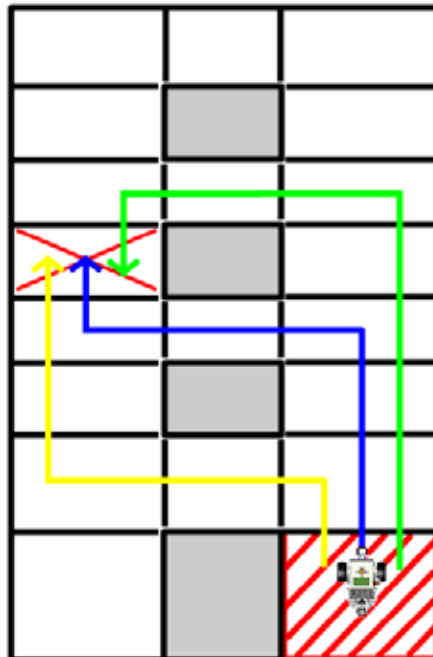
Let's view a typical task for the robot. Suppose the object it needs to get to is at the X on the map below. How can the robot get there?



We could get there by following this blue route ...



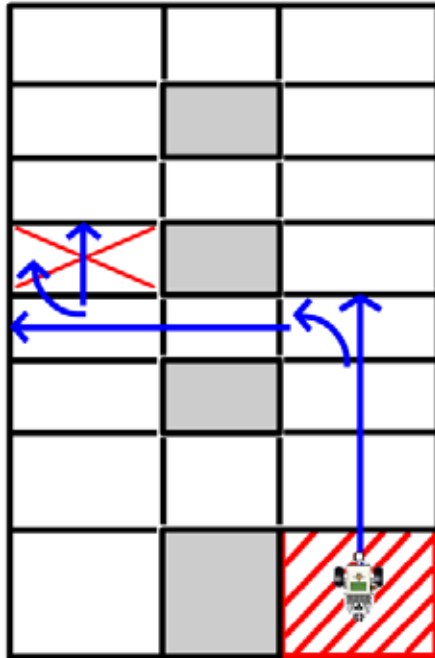
... or perhaps this yellow one, or this green one?



Variables and Functions

Patterns of Behavior Behaviors (cont.)

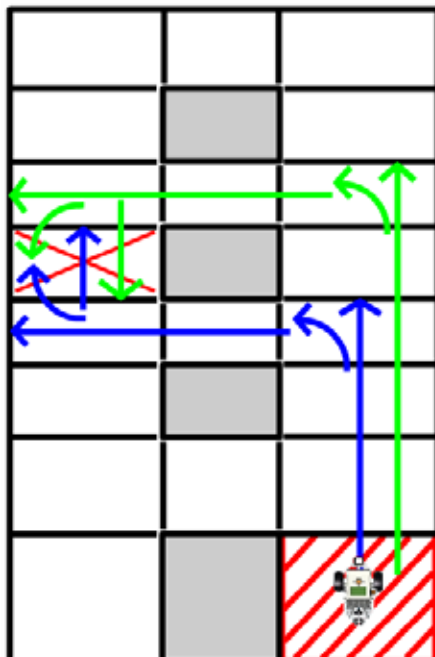
Let's focus on the first (blue) path. What does the robot need to do in order to follow this path?
The large behavior breaks down nicely into **smaller behaviors**.



Blue Path

Forward 4 Lines
Turn Left
Forward 3 Lines
Turn Right
Forward 2 Lines

The green path can also be broken down easily into smaller behaviors.



Blue Path

Forward 4 Lines
Turn Left
Forward 3 Lines
Turn Right
Forward 2 Lines

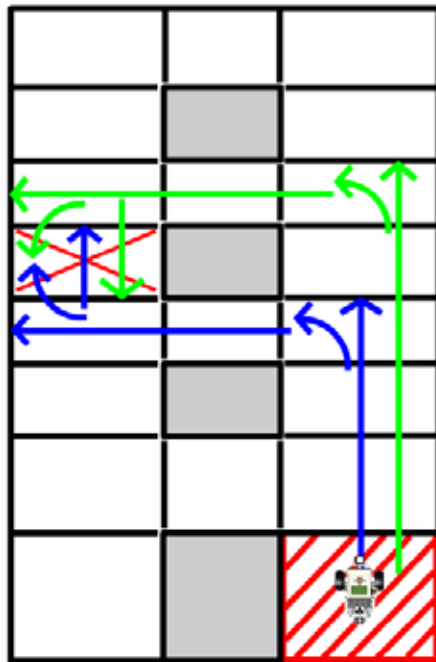
Green Path

Forward 6 Lines
Turn Left
Forward 3 Lines
Turn Left
Forward 2 Lines

Variables and Functions

Patterns of Behavior **Behaviors** (cont.)

Perhaps most interestingly, these two paths seem to share some common sub-behaviors...



| <u>Blue Path</u> | <u>Green Path</u> |
|------------------|-------------------|
| Forward 4 Lines | Forward 6 Lines |
| Turn Left | Turn Left |
| Forward 3 Lines | Forward 3 Lines |
| Turn Right | Turn Left |
| Forward 2 Lines | Forward 2 Lines |

Shared behaviors
These sub-behaviors appear in **both** of the larger behaviors.

End of Section

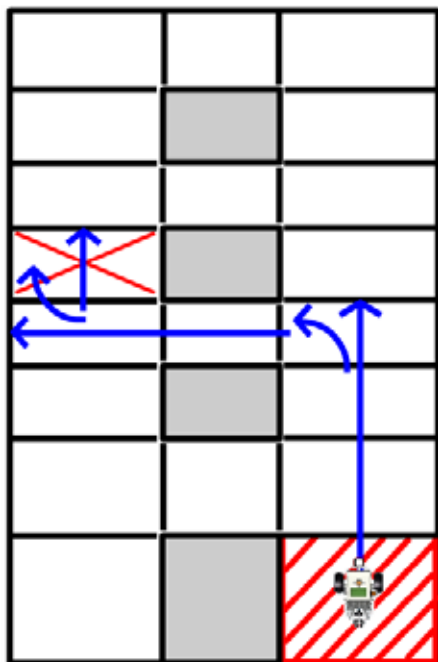
This repeating of sub-behaviors is no coincidence. The smaller behaviors actually represent **common actions** in the warehouse environment, and so they will likely show up in any number of tasks there. C languages like ROBOTC include structures called "**functions**" that are made to capitalize on exactly this kind of patterned reuse of commands to make your code more adaptable, readable, and reusable. In the next few sections, you will learn to build and use a set of functions to allow rapid construction and reorganization of behaviors to get around the warehouse.

Variables and Functions

Patterns of Behavior **Creating and Using Functions**

In this lesson, you will learn how to create and use functions for two of the simplest behaviors.

In the last video, we identified several key simple behaviors that, when combined, will make up the complex behavior of moving to the destination shown here.



Blue Path

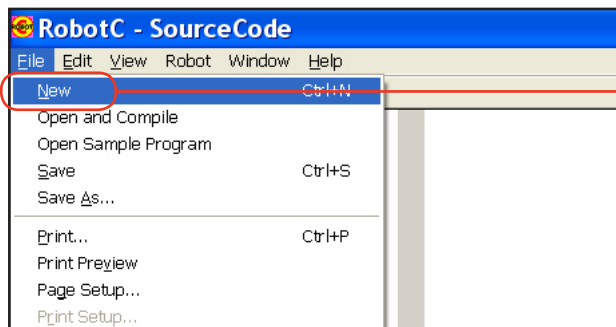
Forward 4 Lines
Turn Left
Forward 3 Lines
Turn Right
Forward 2 Lines

For each of these **simple behaviors**, we are going to create a function which encapsulates the behavior in a single, reusable package. Declaring a function basically means you're **creating your own command in the language of ROBOTC**, so you can already begin to see how powerful this technique will be once you master it...

Variables and Functions

Patterns of Behavior **Creating and Using Functions** (cont.)

1. Open ROBOTC and start a new program.



1. Create new program

Select File > New to create a new program.

2. Create the familiar task main, but don't put anything in it yet.

```

1  task main()
2  {
3
4
5  }
```

2. Add this code

Add a task main() {}.

3. At the top of your program, before task main(), make some space for your functions.

```

1
2
3  task main()
4  {
5
6
7  }
```

3. Create space

Add a few blank lines above task main where your functions will go.

Variables and Functions

Patterns of Behavior **Creating and Using Functions** (cont.)

4. Create the basic skeleton of a function called “turnLeft”. “**void**” is a keyword used to begin the declaration of the function, much like “task” in task main, and similarly, the function includes a pair of curly braces that will contain the commands in the function body.

```

1 void turnLeft ()
2 {
3
4
5
6 }
7
8 task main ()
9 {
10
11
12 }

```

4. Add this code

This code creates a new function called turnLeft(), and leaves a space between the curly braces to put its commands.

5. Place the commands for a left turn behavior between the function’s { } braces. This version of the left turn uses the **rotation sensor** to determine when the robot has turned far enough. See Sensing > Line Tracking > Line Tracking (Rotation, Pt. 1 and Pt. 2) for a review of this sensor.

```

1 void turnLeft ()
2 {
3
4     nMotorEncoder[motorB] = 0;
5     while(nMotorEncoder[motorB] < 160)
6     {
7         motor[motorC] = -50;
8         motor[motorB] = 50;
9     }
10
11     motor[motorC] = 0;
12     motor[motorB] = 0;
13 }
14
15
16 task main ()
17 {
18
19
20 }

```

5. Add this code

Add the commands for a left turn behavior, between the curly braces of the new function.

The function will run the commands between its braces when it is called, just as task main runs the commands between its braces when the main program is run.

The left turn itself resets the rotation sensor, then turns until the wheel has rotated a set amount, then stops both motors.

Variables and Functions

Patterns of Behavior **Creating and Using Functions** (cont.)

6. You have created the function `turnLeft`, and specified the commands that it should run when called (a Rotation Sensor-controlled left turn). To use the function, simply call it by name in the main task.

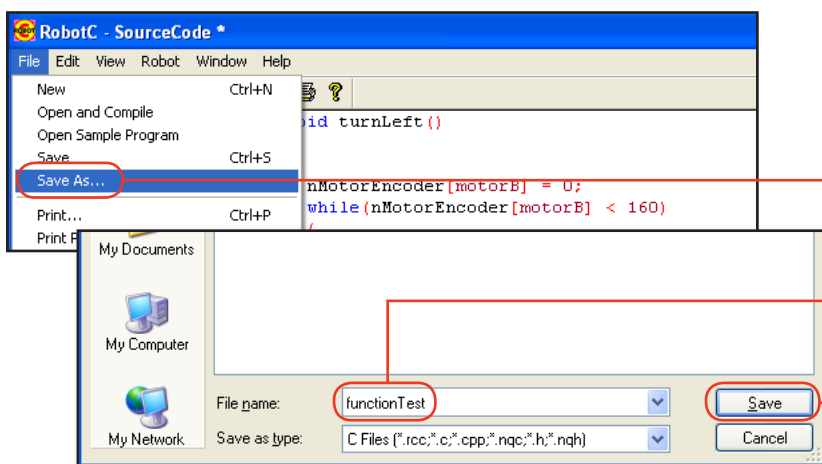
```

1 void turnLeft()
2 {
3
4     nMotorEncoder[motorB] = 0;
5     while(nMotorEncoder[motorB] < 160)
6     {
7         motor[motorC] = -50;
8         motor[motorB] = 50;
9     }
10
11    motor[motorC] = 0;
12    motor[motorB] = 0;
13
14 }
15
16 task main()
17 {
18
19     turnLeft();
20
21 }

```

6. Add this code
Call the function `turnLeft()` by name, followed by a semicolon, to run it.

7. Save your program, download, and run.



7a. Save As
Go to the File menu and select "Save As..."

7b. Name the program
Give this program the name "functionTest".

7c. Save the program
Press Save to save the program with the new name.

Variables and Functions

Patterns of Behavior **Creating and Using Functions** (cont.)

Checkpoint

You have created the function `turnLeft` and told your program to run it in the main task. Does the robot do what you wanted?



Robot running the leftTurn() function

The robot seems to do what we wanted...

7. We said that one of the major advantages of functions was their reusability. Let's see it in action. Add another left turn, separated from the first one by a 1 second wait.

```

1 void turnLeft()
2 {
3
4     nMotorEncoder[motorB] = 0;
5     while(nMotorEncoder[motorB] < 160)
6     {
7         motor[motorC] = -50;
8         motor[motorB] = 50;
9     }
10
11     motor[motorC] = 0;
12     motor[motorB] = 0;
13
14 }
15
16 task main()
17 {
18
19     turnLeft();
20     wait1Msec(1000);
21     turnLeft();
22
23 }
```

6. Add this code
Add another call to `turnLeft()`, separated from the first one by a 1 second delay.

Variables and Functions

Patterns of Behavior **Creating and Using Functions** (cont.)

9. Download and run again.



Robot running two leftTurn() functions

The robot turns once, then waits, and makes a second 90 degree turn.

10. The use of the turnLeft() function to encapsulate the turning behavior in a custom command seems to work well! Now, create one for the right turn, right below the turnLeft() function.

```

1 void turnLeft()
2 {
3
4     nMotorEncoder[motorB] = 0;
5     while(nMotorEncoder[motorB] < 160)
6     {
7         motor[motorC] = -50;
8         motor[motorB] = 50;
9     }
10
11    motor[motorC] = 0;
12    motor[motorB] = 0;
13
14 }
15
16 void turnRight()
17 {
18
19    nMotorEncoder[motorC] = 0;
20    while(nMotorEncoder[motorC] < 160)
21    {
22        motor[motorC] = 50;
23        motor[motorB] = -50;
24    }
25
26    motor[motorC] = 0;
27    motor[motorB] = 0;
28
29 }
```

10. Add this code

Create a function called turnRight(), below turnLeft(). It should be almost identical, but with a right-turn behavior inside it instead.

Variables and Functions

Patterns of Behavior **Creating and Using Functions** (cont.)

11. Change the second left turn to a right turn instead. What should the robot do now?

```
1 void turnLeft()
2 {
3
4     nMotorEncoder[motorB] = 0;
5     while(nMotorEncoder[motorB] < 160)
6     {
7         motor[motorC] = -50;
8         motor[motorB] = 50;
9     }
10
11    motor[motorC] = 0;
12    motor[motorB] = 0;
13
14 }
15
16 void turnRight()
17 {
18
19    nMotorEncoder[motorC] = 0;
20    while(nMotorEncoder[motorC] < 160)
21    {
22        motor[motorC] = 50;
23        motor[motorB] = -50;
24    }
25
26    motor[motorC] = 0;
27    motor[motorB] = 0;
28
29 }
30
31 task main()
32 {
33
34    turnLeft();
35    wait1Msec(1000);
36    turnRight();
37
38 }
```

11. Modify this code
Change the second
leftTurn() call to a
rightTurn() call instead.

Variables and Functions

Patterns of Behavior **Creating and Using Functions** (cont.)

12. Download and run again.



Robot running *leftTurn()* then *rightTurn()*

The robot turns once, then waits, and turns the opposite direction back to the place where it started.

End of Section

You now have two of the most common warehouse (and movement, in general) behaviors written as functions. You have also seen the ease with which these functions can be treated as commands in the ROBOTC language to allow their rapid reuse in a situation like the warehouse where they will be seen over and over again.

In the next lessons, you will move these two functions from their current location in the test program (`functionTest`) into the main program, and complete the remaining behaviors.

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 1)**

In this lesson, you will transfer your two turning behaviors into the program from earlier Warehouse activities, and create functions for the remaining behaviors in the program.

The two turning behaviors you've created are useful, but disconnected from the rest of the Warehouse program we've been working on. Functions only work in the programs they are declared in, and right now, ours are in a test program called `functionTest`. Let's start this lesson by moving them into the main program file, and then we'll work on turning the other behaviors in the program into functions.

1. Highlight and copy the two functions in your "functionTest" program.

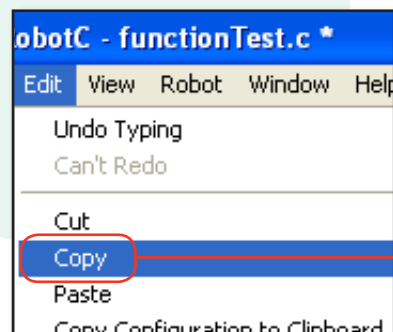
```

1 void turnLeft ()
2 {
3
4     nMotorEncoder[motorB] = 0;
5     while (nMotorEncoder[motorB] < 160)
6     {
7         motor[motorC] = -50;
8         motor[motorB] = 50;
9     }
10
11     motor[motorC] = 0;
12     motor[motorB] = 0;
13
14 }
15
16 void turnRight ()
17 {
18
19     nMotorEncoder[motorC] = 0;
20     while (nMotorEncoder[motorC] < 160)
21     {
22         motor[motorC] = 50;
23         motor[motorB] = -50;
24     }
25
26     motor[motorC] = 0;
27     motor[motorB] = 0;
28
29 }
30

```

1a. Highlight code

Highlight both functions and all their associated code as shown.



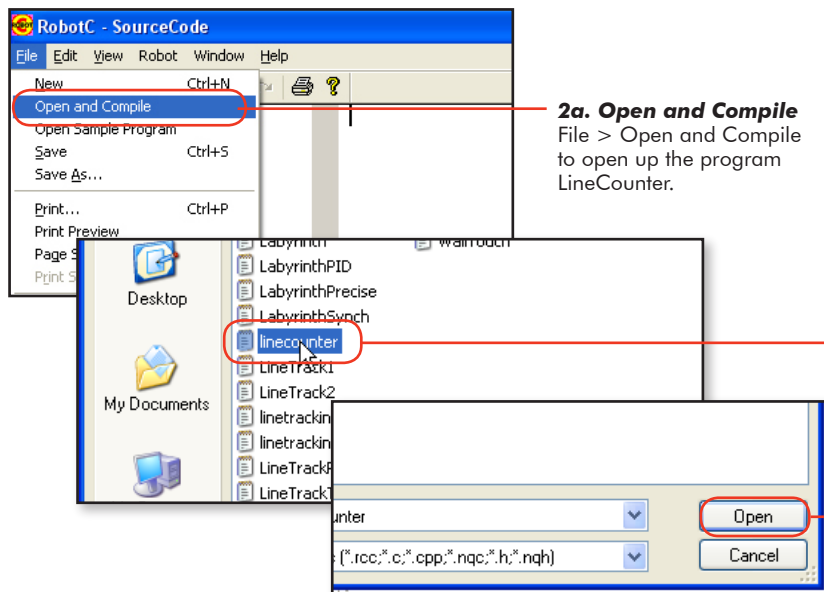
1b. Copy

Select Edit > Copy to put the highlighted code on the clipboard.

Variables and Functions

Patterns of Behavior Variables and Functions (Part 1) (cont.)

2. Open your LineCounter program.



2a. Open and Compile
File > Open and Compile to open up the program LineCounter.

2b. Find LineCounter
Find LineCounter and click on the program previously saved.

2c. Open LineCounter
Press the Open button to open the program.

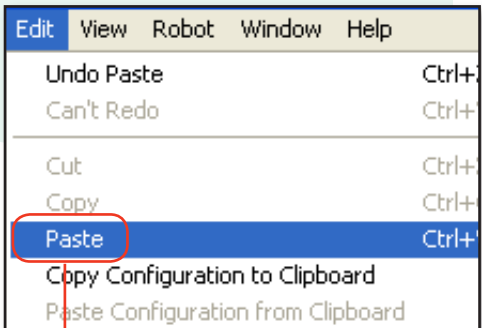
3. Paste your functions just above the task main code in the LineCounter program.

```

Auto  const tSensors touchSensor      = (tSensors) S1;
Auto  const tSensors lightSensor     = (tSensors) S2;
1      
2
3
4     task main()
5     {
6
7         int lightValue;
8         int darkValue;
9         int sumValue;
10        int thresholdValue;
11        int countValue = 0;

```

3a. Place cursor here
Place your cursor on the line just above task main so your pasted code will go there.



3b. Paste
Select Edit > Paste to put the copied code into this program.

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 1)** (cont.)

Checkpoint

And just like that, your program has access to both functions. The rest of the task main could use some cleaning via functions, though, so let's do that next.

The other two functions we'll need to create are:

- Threshold calculation
- Moving forward for specific numbers of lines

4. Create the structure for the **findThreshold()** function. Put it at the top of the program, above the newly-pasted **turnLeft()** function.

```

Auto  const tSensors touchSensor      = (tSensors) S1;
Auto  const tSensors lightSensor     = (tSensors) S2;
1     void findThreshold()
2     {
3
4
5
6     }
7
8     void turnLeft()
9     {
10
11     nMotorEncoder[motorB] = 0;
12     while (nMotorEncoder[motorB] < 160)
13     {
14         motor[motorC] = -50;
15         motor[motorB] = 50;

```

4. Add this code

Add the basic structure for the **findThreshold()** function that we are about to create.

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 1)** (cont.)

5. Highlight all the lines currently in task main that have to do with **automatic threshold calculation**, and cut them to the clipboard using the Edit > Cut command.

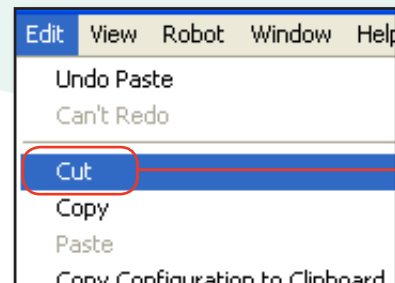
```

38 task main()
39 {
40
41     int lightValue;
42     int darkValue;
43     int sumValue;
44     int thresholdValue;
45     int countValue = 0;
46     int lastSeen;
47
48     nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
49     nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;
50
51     while (SensorValue(touchSensor)==0)
52     {
53         nxtDisplayStringAt(0, 31, "Read Light Now");
54     }
55
56     lightValue=SensorValue(lightSensor);
57
58     wait1Msec(1000);
59
60     while (SensorValue(touchSensor)==0)
61     {
62         nxtDisplayStringAt(0, 31, "Read Dark Now");
63     }
64
65     darkValue=SensorValue(lightSensor);
66
67     sumValue = lightValue+darkValue;
68     thresholdValue = sumValue/2;
69
70     ClearTimer(T1);
71     lastSeen = 1;
72

```

5a. Highlight code

Highlight the code that performs the automatic threshold measurement and calculation.



5b. Cut

Select Edit > Cut to remove the highlighted code from the program and put it on the clipboard.

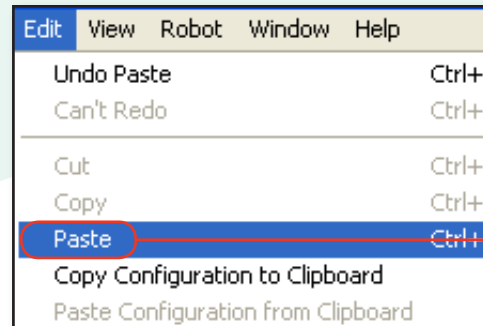
Variables and Functions

Patterns of Behavior **Variables and Functions (Part 1)** (cont.)

6. Paste the lines into the {body} section of the findThreshold() function.

```

Auto  const tSensors touchSensor      = (tSensors) S1;
Auto  const tSensors lightSensor     = (tSensors) S2;
1     void findThreshold()
2     {
3
4     |
5
6     }
7
8     void turnLeft()
  
```



6a. Place cursor here

Place your cursor on the line between findThreshold()'s curly braces so your pasted code will go there.

6b. Paste

Select Edit > Paste to put the copied code into this program.

Checkpoint. Finding a threshold is now as simple as telling the program to `findThreshold()`; . But first, let's finish writing the other functions.

```

Auto  const tSensors touchSensor      = (tSensors) S1;
Auto  const tSensors lightSensor     = (tSensors) S2;
1     void findThreshold()
2     {
3
4     while (SensorValue(touchSensor)==0)
5     {
6         nxtDisplayStringAt(0, 31, "Read Light Now");
7     }
8
9     lightValue=SensorValue(lightSensor);
10
11    wait1Msec(1000);
12
13    while (SensorValue(touchSensor)==0)
14    {
15        nxtDisplayStringAt(0, 31, "Read Dark Now");
16    }
17
18    darkValue=SensorValue(lightSensor);
19
20    sumValue = lightValue+darkValue;
21    thresholdValue = sumValue/2;
22
23 }
  
```

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 1)** (cont.)

7. Create the structure for the **forward7Lines()** function. Put it just under the findThreshold() function, outside its last closing brace.

```

Auto const tSensors touchSensor      = (tSensors) S1;
Auto const tSensors lightSensor      = (tSensors) S2;
1 void findThreshold()
2 {
3
4     while (SensorValue(touchSensor)==0)
5     {
6         nxtDisplayStringAt(0, 31, "Read Light Now");
7     }
8
9     lightValue=SensorValue(lightSensor);
10
11    wait1Msec(1000);
12
13    while (SensorValue(touchSensor)==0)
14    {
15        nxtDisplayStringAt(0, 31, "Read Dark Now");
16    }
17
18    darkValue=SensorValue(lightSensor);
19
20    sumValue = lightValue+darkValue;
21    thresholdValue = sumValue/2;
22
23 }
24
25 void forward7Lines()
26 {
27
28
29
30 }
31
32 void turnLeft()

```

7. Add this code
Add the basic structure for the findThreshold() function that we are about to create.

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 1)** (cont.)

8. Highlight all the lines currently in task main that have to do with **moving forward for a given number of lines**, and cut them to the clipboard using the Edit > Cut command. Delete the unneeded ClearTimer command that's still in this portion of the code.

```

76
77 ClearTimer(T1);
78 lastSeen = 1;
79
80 while (countValue < 7)
81 {
82
83     if (SensorValue(lightSensor) < thresholdValue)
84     {
85
86         motor[motorC]=50;
87         motor[motorB]=50;
88
89         if (lastSeen == 1)
90         {
91             countValue = countValue + 1;
92             lastSeen = 0;
93         }
94     }
95 }
96
97 else
98 {
99     lastSeen = 1;
100 }
101
102 }
103
104 }

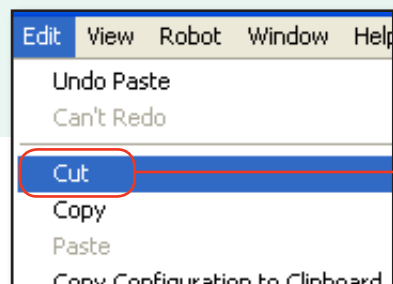
```

8a. Delete this code

Remove the leftover ClearTimer command that is in this section.

8b. Highlight code

Highlight the code that performs the line counting and forward movement based on lines crossed.



8c. Cut

Select Edit > Cut to remove the highlighted code from the program and put it on the clipboard.

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 1)** (cont.)

9. Paste the lines into the {body} section of the forward7Lines() function.

```

23 }
24
25 void forward7Lines ()
26 {
27 |
28
29
30 }
31
32 void turnLeft ()
  
```

| Edit | View | Robot | Window | Help |
|------------------------------------|------|-------|--------|-------|
| Undo Paste | | | | Ctrl+ |
| Can't Redo | | | | Ctrl+ |
| <hr/> | | | | |
| Cut | | | | Ctrl+ |
| Copy | | | | Ctrl+ |
| Paste | | | | Ctrl+ |
| Copy Configuration to Clipboard | | | | |
| Paste Configuration from Clipboard | | | | |

9a. Place cursor here

Place your cursor on the line between forward7Lines()'s curly braces so your pasted code will go there.

9b. Paste

Select Edit > Paste to put the copied code into this program.

Checkpoint

You now have a function that lets you move forward for 7 lines at a time. Save your program, but don't download yet.

```

24
25 void forward7Lines()
26 {
27
28     lastSeen = 1;
29
30     while (countValue < 7)
31     {
32
33         if (SensorValue(lightSensor) < thresholdValue)
34         {
35
36             motor[motorC]=50;
37             motor[motorB]=50;
38
39             if (lastSeen == 1)
40             {
41                 countValue = countValue + 1;
42                 lastSeen = 0;
43             }
44
45         }
46
47         else
48         {
49             lastSeen = 1;
50         }
51
52     }
53
54
  
```

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 1)** (cont.)

Your program now has access to two additional behaviors: **findThreshold()** and **forward7Lines()**, which we have just extracted into separate functions. In addition, we have the two behaviors we imported from our test file, **turnLeft()** and **turnRight()**. All that remains now is to tell the task main to run them in the desired order... right?

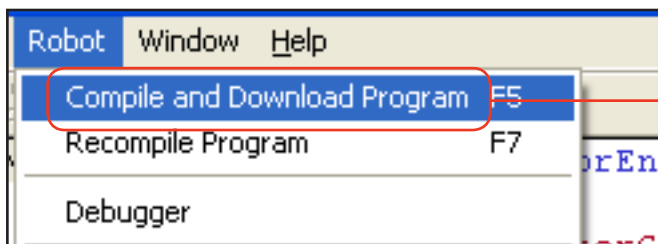
- 10.** Call the new functions in **task main**. Finding the threshold comes first, followed by the movement forward for 7 lines. We'll wait to see if that works before we put in the turns.

```

86 task main()
87 {
88
89     int lightValue;
90     int darkValue;
91     int sumValue;
92     int thresholdValue;
93     int countValue = 0;
94     int lastSeen;
95
96     nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
97     nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;
98
99     findThreshold();
100    forward7Lines();
101
102
103 }
```

10. Add this code
Tell your robot to run the `findThreshold()` and `forward7Lines()` functions as part of its main program.

- 11.** Save, download, and run. An error message will appear, indicating that something is not right... let's see if we can find what's going wrong.

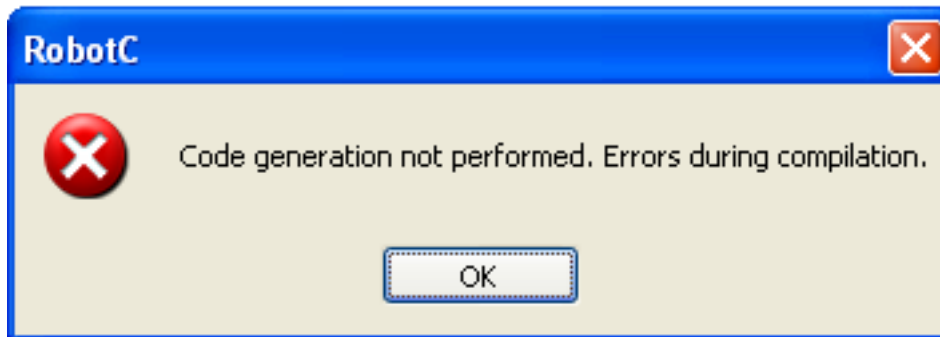


11. Compile and Download
Compile and download your program, but be ready for unusual results... continue on to the next step.

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 1)** (cont.)

Checkpoint



Error!
Something is wrong
with the program.

Scope: How broadly applicable a value should be

The problem with your program has to do with a property of variables called **scope**. Scope determines **how broadly applicable a value should be**. The variables in your program are all **declared in task main**. But the actual code that's trying to use them is outside task main, in **separate functions**. They cannot "see" the variables because they are only accessible within task main. It seems silly to us now that this should be the case, but scope actually plays a vital role in letting functions run without interfering with each other.

```

1 void findThreshold()
2 {
3
4     while (SensorValue(touchSensor)==0)
5     {
6         nxtDisplayStringAt(0, 31, "Read Light Now");
7     }
8
9     lightValue=SensorValue(lightSensor);

```

Scope

The variable lightValue is declared in task main, so the function findThreshold() cannot see it to use it. This causes an error when you try to compile and download the program.

```

85
86 task main()
87 {
88
89     int lightValue;

```

Nevertheless, for now, we're going to take a very heavy-handed approach to solving this problem. We're going to move the variables so that they are visible to all functions and tasks by making them **global**. This has advantages and disadvantages, but for now, we're going with it.

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 1)** (cont.)

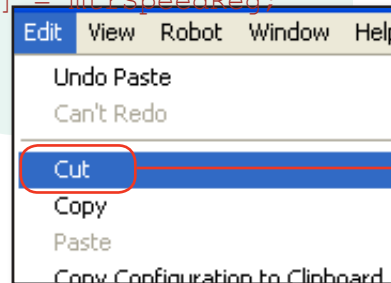
12. Highlight all the lines currently in task main that **declare variables needed by the functions**, and cut them to the clipboard using the Edit > Cut command.

```

85
86 task main()
87 {
88
89     int lightValue;
90     int darkValue;
91     int sumValue;
92     int thresholdValue;
93     int countValue = 0;
94     int lastSeen;
95
96     nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
97     nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;
98
99     findThreshold();
100    forward7Lines();

```

12a. Highlight code
Highlight the code that declares variables in task main.



12b. Cut
Select Edit > Cut to remove the highlighted code from the program and put it on the clipboard.

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 1)** (cont.)

13. Paste the lines at the top of the program, outside all the functions, but just below the Motor and Sensors auto-generated lines.

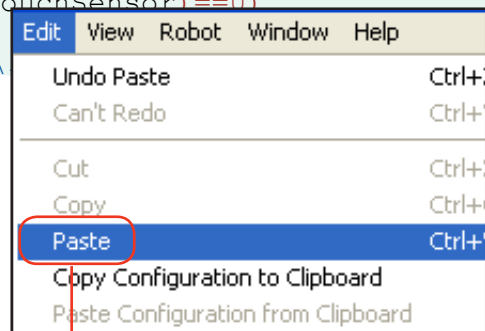
```

Auto  const tSensors touchSensor      = (tSensors) S1;
Auto  const tSensors lightSensor     = (tSensors) S2;
1
2  |
3
4  void findThreshold()
5  {
6      while (SensorValue(touchSensor) == 0)
7      {
8          nxtDisplayStringA

```

9a. Place cursor here

Place your cursor on the line above the **findThreshold()** declaration so your pasted code will go there.



9b. Paste

Select Edit > Paste to put the copied code into this program.

Checkpoint

All your variables are now declared “globally”, and therefore will be visible to all of the functions and tasks in the program. This will have side effects down the line, but for now, it will get us the result we want.

```

Auto  const tSensors touchSensor      = (tSensors) S1;
Auto  const tSensors lightSensor     = (tSensors) S2;
1
2  int lightValue;
3  int darkValue;
4  int sumValue;
5  int thresholdValue;
6  int countValue = 0;
7  int lastSeen;
8
9  void findThreshold()
10 {
11
12     while (SensorValue(touchSensor) == 0)

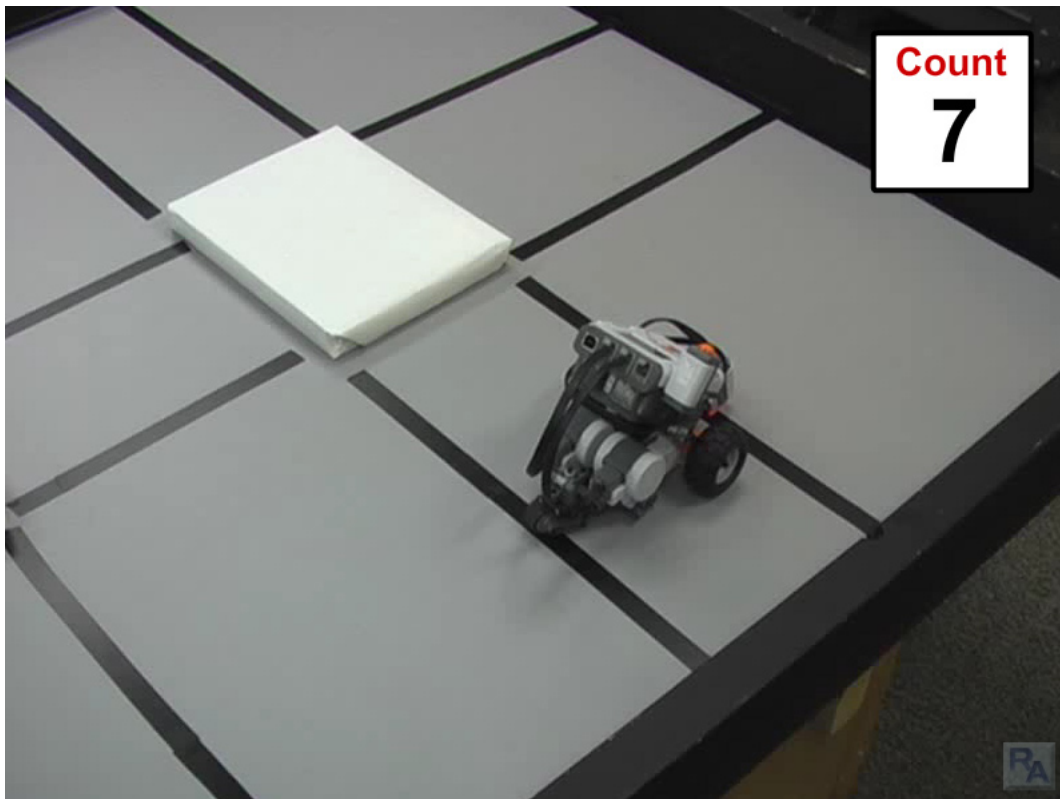
```

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 1)** (cont.)

End of Section

Download and run your program. The robot should now run exactly seven lines, then stop, using functions. The result isn't any different from what you've seen before, but you know that under the hood, your program is much more powerful and expandible now, and you are now ready to finish solving the warehouse problem. In the next lesson, you will program the remaining necessary functions for the warehouse.

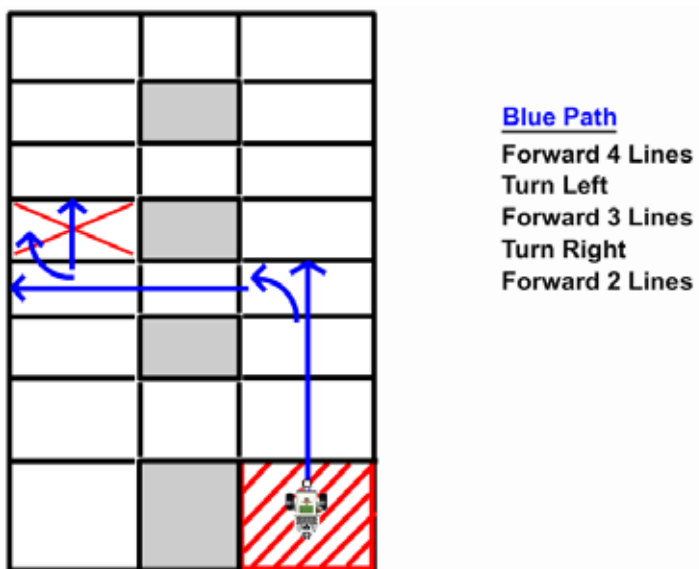


Variables and Functions

Patterns of Behavior **Variables and Functions (Part 2)**

In this lesson, you will learn to adjust the behaviors to fit the actual path you want to take in the warehouse, and make a few additional refinements as necessary.

And now, let's return to the path we want to take through the warehouse. The needed behaviors (in addition to finding the threshold, which isn't shown) are:



Our currently programmed behaviors are:

- `findThreshold()`, which finds a threshold
- `forward7Lines()`, which travels forward for 7 lines
- `turnLeft()`, which turns the robot 90 degrees to the left
- `turnRight()`, which turns the robot 90 degrees to the right

It looks like we have a fair number of changes to make, so let's get started.

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 2)** (cont.)

1. The current forward-for-lines goes for 7 lines, but the path requires a 4, a 3, and a 2. Start by modifying the 7-line behavior to be a 4-line behavior.

```

32
33 void forward4Lines ()
34 {
35
36     lastSeen = 1;
37
38     while (countValue < 4)
39     {
40
41         if (SensorValue(lightSensor) < thresholdValue)
42         {
43
44             motor[motorC]=50;
45             motor[motorB]=50;
46
47             if (lastSeen == 1)
48             {
49                 countValue = countValue + 1;
50                 lastSeen = 0;
51             }
52
53         }
54
55     else
56     {
57         lastSeen = 1;
58     }
59
60 }
61
62 }

```

1a. Modify this code

Change the name of the function to indicate its new behavior: going forward for 4 lines, instead of 7.

1b. Modify this code

The number of line counts in the while loop's (condition) is what determines how many lines the robot watches for. Change this number from 7 lines to 4 lines.

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 2)** (cont.)

2. Adjust your task main to run the new forward-for-4-lines function, and add all the other behaviors which we will need, even if they haven't been written yet. Note those for later.

```

93
94 task main()
95 {
96
97
98
99     nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
100    nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;
101
102    findThreshold();
103    forward4Lines();
104    turnLeft();
105    forward3Lines();
106    turnRight();
107    forward2Lines();
108
109 }

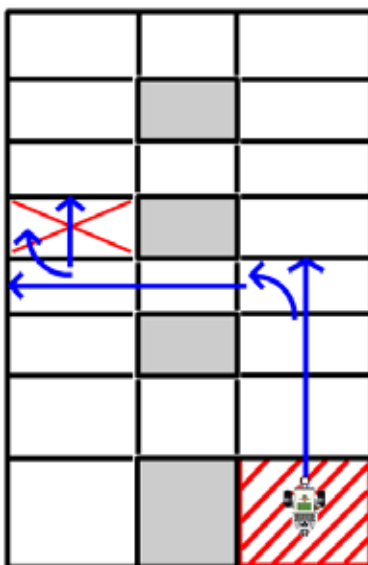
```

2a. Modify this code
Change the old forward-for-7-lines command to the new forward-for-4-lines one.

2b. Add this code
Add the appropriate function calls for the remaining behaviors, even the ones where we haven't written the actual functions yet.

Checkpoint

Our functions are in place to perform each of the behaviors we identified in our initial plan. Three of them, forward4Lines, turnLeft, and turnRight are already written. Let's finish up the others.



```

101
102 findThreshold();
103 forward4Lines();
104 turnLeft();
105 forward3Lines();
106 turnRight();
107 forward2Lines();
108

```

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 2)** (cont.)

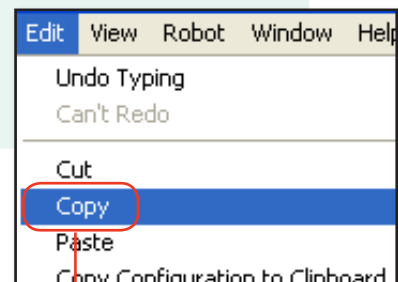
3. The remaining two behaviors, forward3Lines() and forward2Lines() are very close relatives of the existing forward4Lines(). Copy the forward4Lines() function, and paste two copies of it, which we will turn into the 3-line and 2-line behaviors in the next step.

```

32
33 void forward4Lines ()
34 {
35
36     lastSeen = 1;
37
38     while (countValue < 4)
39     {
40
41         if (SensorValue(lightSensor) < thresholdValue)
42         {
43
44             motor[motorC]=50;
45             motor[motorB]=50;
46
47             if (lastSeen == 1)
48             {
49                 countValue = countValue + 1;
50                 lastSeen = 0;
51             }
52         }
53
54     }
55     else
56     {
57         lastSeen = 1;
58     }
59
60 }
61
62 }
63

```

3a. Highlight code
Highlight the forward4Lines() function, including its curly braces and everything between them.



3b. Copy
Select Edit > Copy to put the highlighted code on the clipboard.

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 2)** (cont.)

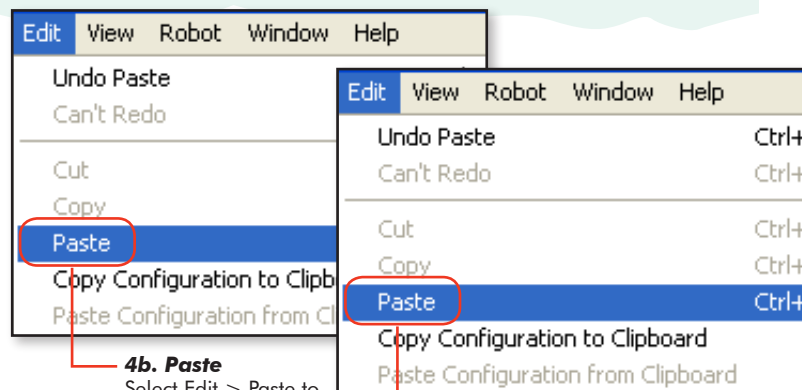
4. Paste **two** copies of the behavior right after the original.

```

25     if (SensorValue(lightSensor) < thresholdValue)
26     {
27         motor[motorC]=50;
28         motor[motorB]=50;
29
30         if (lastSeen == 1)
31         {
32             countValue = countValue + 1;
33             lastSeen = 0;
34         }
35
36     }
37
38     else
39     {
40         lastSeen = 1;
41     }
42
43 }
44
45 }
46
47 |
48
49 void turnLeft()
50 {

```

4a. Place cursor here
Place your cursor on the line below the **forward4Lines()** declaration so your pasted code will go there.



4b. Paste
Select Edit > Paste to put the copied code into this program.

4c. Paste again
Paste a second copy of the same code right after the first one.

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 2)** (cont.)

5. You have three copies of the same behavior. Change two of them to 3-line and 2-line versions.

```

32
33 void forward4Lines()
34 {
35     lastSeen = 1;
36
37     while (countValue < 4)
38     {
39         if (SensorValue(lightSensor) < thresholdValue)
40         {
41             motor[motorC]=50;
42             motor[motorB]=50;
43
44             if (lastSeen == 1)
45             {
46                 countValue = countValue + 1;
47                 lastSeen = 0;
48             }
49         }
50     }
51     else
52     {
53         lastSeen = 1;
54     }
55 }
56
57 void forward3Lines()
58 {
59     lastSeen = 1;
60
61     while (countValue < 3)
62     {
63         if (SensorValue(lightSensor) < thresholdValue)
64         {
65             motor[motorC]=50;
66             motor[motorB]=50;
67
68             if (lastSeen == 1)
69             {
70                 countValue = countValue + 1;
71                 lastSeen = 0;
72             }
73         }
74     }
75     else
76     {
77         lastSeen = 1;
78     }
79 }
80
81 void forward2Lines()
82 {
83     lastSeen = 1;
84
85     while (countValue < 2)
86     {
87         if (SensorValue(lightSensor) < thresholdValue)
88         {
89             motor[motorC]=50;
90             motor[motorB]=50;
91
92             if (lastSeen == 1)
93             {
94                 countValue = countValue + 1;
95                 lastSeen = 0;
96             }
97         }
98     }
99     else
100    {
101        lastSeen = 1;
102    }
103 }
104
105

```

```

32
33 void forward4Lines()
34 {
35     lastSeen = 1;
36
37     while (countValue < 4)
38     {
39
40

```

```

63
64 void forward3Lines()
65 {
66     lastSeen = 1;
67
68     while (countValue < 3)
69     {
70
71

```

```

74
75 void forward2Lines()
76 {
77     lastSeen = 1;
78
79     while (countValue < 2)
80     {
81
82

```

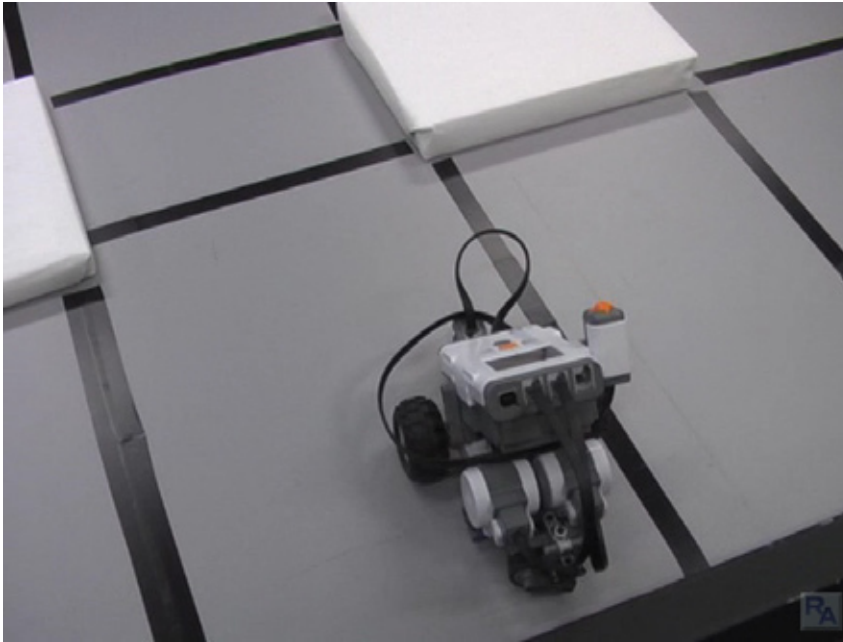
Modify this code
Change the second forward-for-lines behavior to do 3 lines, and the third behavior to do 2 lines.

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 2)** (cont.)

Checkpoint

Save, download, and run. The robot will scoot along for 4 lines and turn, just as planned... and then, mysteriously, stop.



Stuck?

The robot seems to stop after the second command. What's it waiting for?

The program has a few bugs. This is normal, programs seldom work perfectly on the first try, especially after making big changes like the ones we just did. Continue on to begin fixing them!

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 2)** (cont.)

6. The problem seems to have occurred in the forward3Lines() function, but remember that errors in this function will need to be corrected in its two twins as well. It turns out there are two things keeping this robot from moving on.

```

64 void forward3Lines ()
65 {
66
67     lastSeen = 1;
68
69     while (countValue < 3)
70     {
71         if (SensorValue(lightSensor) < thresholdValue)
72         {
73
74             motor[motorC]=50;
75             motor[motorB]=50;
76
77             if (lastSeen == 1)

```

Missing reset

An ugly side effect of using global variables is that they are shared between functions even when you don't want them to be.

This means that countValue is still 4 from the 4-line command that is run earlier in the program. The while loop will immediately kick out without running any additional lines!

Move only on dark?

We didn't really give this much thought, but this bug has been here the whole time. The robot only starts moving if it's seeing dark, because it doesn't reach the motor commands otherwise.

```

64 void forward3Lines ()
65 {
66
67     lastSeen = 1;
68     countValue = 0;
69
70     while (countValue < 3)
71     {
72         motor[motorC]=50;
73         motor[motorB]=50;
74
75         if (SensorValue(lightSensor) < thresholdValue)
76         {
77

```

6a. Add this code

This line resets the value of countValue to a fresh count of 0 lines for this new movement.

6b. Modify this code

Move the motor lines out of the "dark" portion of the code and put them just outside the if-else statement so they run regardless of whether the robot is seeing light or dark.

```

33 void forward4Lines ()

```

```

96 void forward2Lines ()

```

6c. Repeat

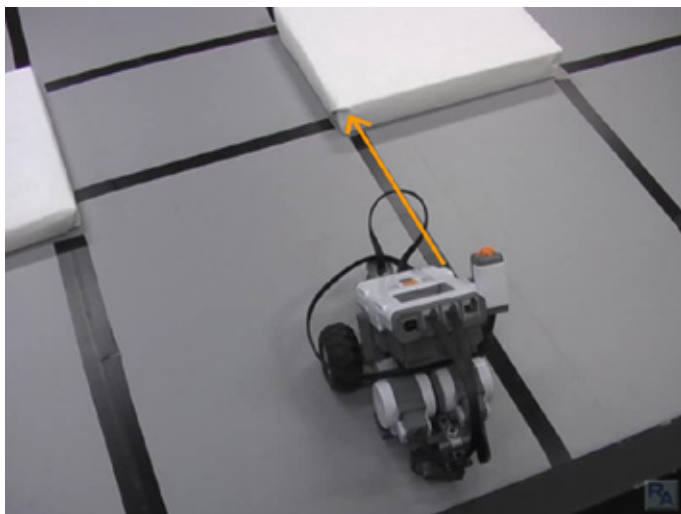
Make the same changes in the 4-line and 2-line versions of the function.

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 2)** (cont.)

Checkpoint

That should solve the stopping problem. Now, let's look at one other issue that you may have seen.



Clearance

The robot is clearly biased toward the side of the corridor. If it proceeds along this path, it will hit the wall.

7. The robot needs to back up a little before each turn. Add the appropriate movement code to both turning functions.

```

132 void turnLeft()
133 {
134     nMotorEncoder[motorB] = 0;
135     while(nMotorEncoder[motorB] < -100)
136     {
137         motor[motorC] = -50;
138         motor[motorB] = -50;
139     }
140
141     nMotorEncoder[motorB] = 0;
142     while(nMotorEncoder[motorB] < 160)
143     {
144         motor[motorC] = -50;
145         motor[motorB] = 50;
146     }
147
148     motor[motorC] = 0;
149     motor[motorB] = 0;
150
151 }
152
153 void turnRight()

```

7a. Add this code

Make the robot back a little away from the line before turning.

7b. Repeat

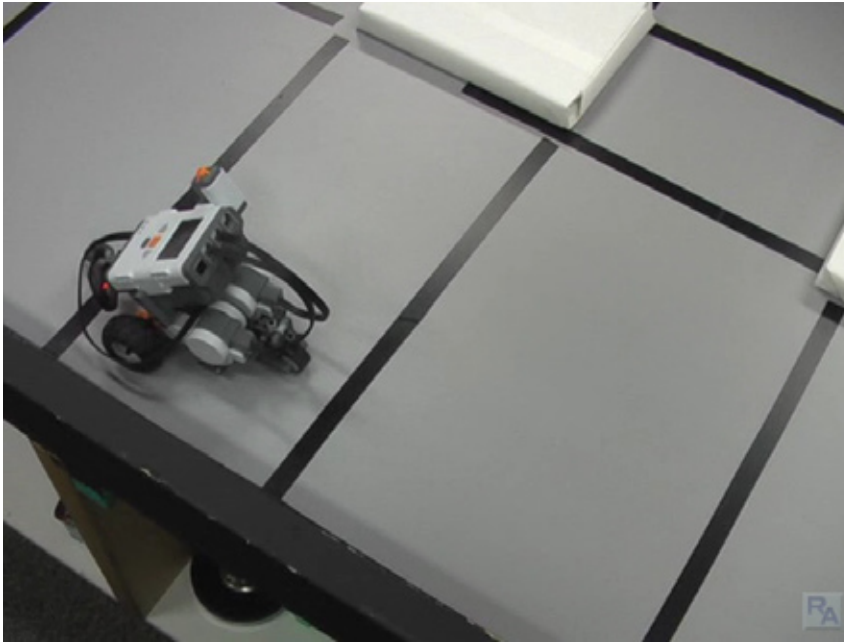
Make the same change in the right turn function.

Variables and Functions

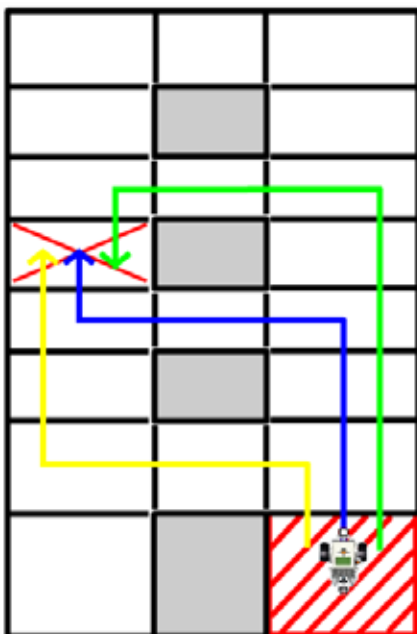
Patterns of Behavior **Variables and Functions (Part 2)** (cont.)

End of Section

Save, download, and run your program. At long last, the robot should complete its path from start to finish.



So why did we go through all this extra trouble to write functions instead of just putting all the code in the main task? Ask yourself for a moment what changes it would take to your program to use the **green** or **yellow** paths instead: the simplicity of **reuse** will speak for itself... to use the yellow path, all you would have to do is switch two lines in task main!



```
findThreshold();
forward4Lines();
turnLeft();
forward3Lines();
turnRight();
forward2Lines();
```

```
findThreshold();
forward6Lines();
turnLeft();
forward3Lines();
turnLeft();
forward2Lines();
```

```
findThreshold();
forward2Lines();
turnLeft();
forward3Lines();
turnRight();
forward4Lines();
```

Variables and Functions

Patterns of Behavior Variables and Functions (Part 3)

In this lesson, you will learn how to use functions with parameters to expand their reusability beyond the level of simple copy-and-paste.

There's still one thing about these functions that could stand to be improved. As it is right now, you have to write a new function every time you want to go a different distance. There is a better way. Consider first, what the actual difference in code is between the three functions below:

```
void forward4Lines()
{
  countValue = 0;
  lastSeen = 1;
  while (countValue < 4)
  {
    motor[motorC]=50;
    motor[motorB]=50;
    if (SensorValue(lightSensor) < thresholdValue)
    {
      if (lastSeen == 1)
      {
        countValue = countValue + 1;
        lastSeen = 0;
      }
    }
    else
    {
      lastSeen = 1;
    }
  }
}

void forward3Lines()
{
  countValue = 0;
  lastSeen = 1;
  while (countValue < 3)
  {
    motor[motorC]=50;
    motor[motorB]=50;
    if (SensorValue(lightSensor) < thresholdValue)
    {
      if (lastSeen == 1)
      {
        countValue = countValue + 1;
        lastSeen = 0;
      }
    }
    else
    {
      lastSeen = 1;
    }
  }
}

void forward2Lines()
{
  countValue = 0;
  lastSeen = 1;
  while (countValue < 2)
  {
    motor[motorC]=50;
    motor[motorB]=50;
    if (SensorValue(lightSensor) < thresholdValue)
    {
      if (lastSeen == 1)
      {
        countValue = countValue + 1;
        lastSeen = 0;
      }
    }
    else
    {
      lastSeen = 1;
    }
  }
}
```

The answer: **one number.**

```
while (countValue < 4)
```

```
while (countValue < 3)
```

```
while (countValue < 2)
```

The difference

These three huge functions differ only in one place: a single number that they use to check how many lines they should run for.

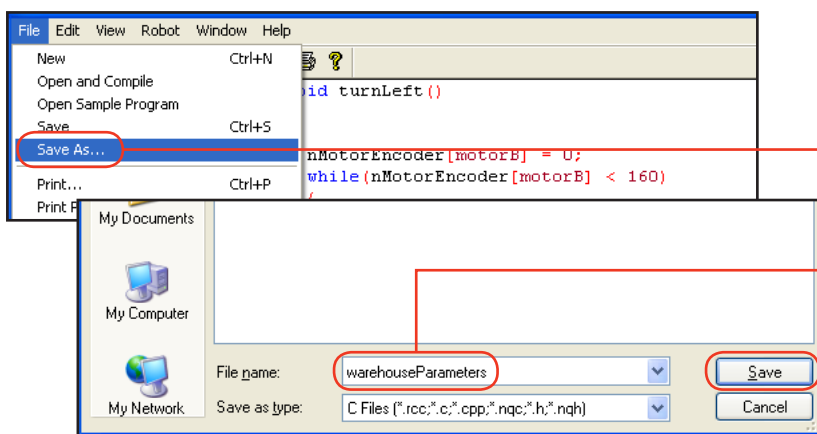
Variables and Functions

Patterns of Behavior **Variables and Functions (Part 3)** (cont.)

We need to take advantage of this somehow. We can do it using a feature of functions called **parameters**. A parameter is a “placeholder value” that you can use in a function’s **declaration** to stand for a value that you will specify in the function **call**. Because you call a function separately every time you want it to run, this means you can specify a different value for the placeholder parameter every time!

```
while (countValue < (your value here!))
```

1. Save your program as “warehouseParameters”.



1a. Save As
Go to the File menu and select “Save As...”

7b. Name the program
Give this program the name “warehouseParameters”.

7c. Save the program
Press Save to save the program with the new name.

2. Delete two of your forward-for-lines functions.

```
66 void forward3Lines ()
67 {
68
69     countValue = 0;
70     lastSeen = 1;
71
72     while (countValue < 3)
```

```
99 void forward2Lines ()
100 {
101
102     countValue = 0;
103     lastSeen = 1;
104
105     while (countValue < 2)
```

2a. Delete these functions
Delete both the **forward3Lines()** and **forward2Lines()** functions.

Make sure you catch all the code inside them, and the closing braces at the ends.

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 3)** (cont.)

3. Modify your remaining forward-for-lines function to be a general-purpose parameter version.

```

32
33 void forwardLines(int numLines)
34 {
35
36     lastSeen = 1;
37     countValue = 0;
38
39     while (countValue < numLines)
40     {
41
42         if (SensorValue(lightSensor) < thresholdValue)
43         {
44
45             motor[motorC]=50;
46             motor[motorB]=50;
47
48             if (lastSeen == 1)
49             {
50                 countValue = countValue + 1;
51                 lastSeen = 0;
52             }
53
54         }
55

```

3a. Modify this code
Rename the remaining function to have a more general name.

3b. Modify this code
Creating a **parameter** looks a lot like a variable declaration, placed between the parentheses that follow the function name.

The parameter "numLines" is created here as an integer, and can be used as a placeholder anywhere in the function {body}.

Its value is not specified here at all. It will (and must) be provided by the task that calls this function.

3c. Modify this code
Put the placeholder parameter "numLines" here in place of the value that we want to be able to fill in.

Placeholder using Parameters

Parameters are like **temporary placeholder variables** that give the programmer the ability to "substitute" a value inside the function, without actually rewriting the function each time. They require attention in two places: the **function declaration**, and the **function call**.

Function declaration:

In the function declaration, the presence of a parameter is announced by declaring it, **variable-style**, between the (parentheses) following the function name. The parameter can then be used like a value in the rest of the function.

```
void forwardLines(int numLines)
```

Parameter declared
"numLines" is now usable as a placeholder in the function.

(continued on next page...)

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 3)** (cont.)

4. Modify your main task to take advantage of the new parameter.

```

110
111
112
113     nMotorPIDSpeedCtrl[motorC] = mtrSpeedReg;
114     nMotorPIDSpeedCtrl[motorB] = mtrSpeedReg;
115
116     findThreshold();
117     forwardLines(4);
118     turnLeft();
119     forwardLines(3);
120     turnRight();
121     forwardLines(2);
122
123 }

```

4. Modify this code

Change the function names to simply forwardLines to match your new function.

In the (parentheses), place the value that you want the parameter to use for that run.

Checkpoint. Visualize the substitution that is happening with your parameter.

```

108 task main()
109 {
...
...
117     forwardLines(4);

```



```

33 void forwardLines(int numLines)
34 {
35
36     lastSeen = 1;
37     while (countValue < numLines)
38     {

```

Substitution

The value 4 is placed in the (parentheses) when the function is called, so the value 4 takes the place of placeholder "numLines" everywhere it appears in the function.

Placeholder using Parameters (cont.)

Function call:

The value of the parameter is specified separately each time the function is run. A value is included in the (parentheses) following the function name when called, and becomes the value of the placeholder in the function's {body} code!

```
forwardLines(4);
```

Parameter supplied

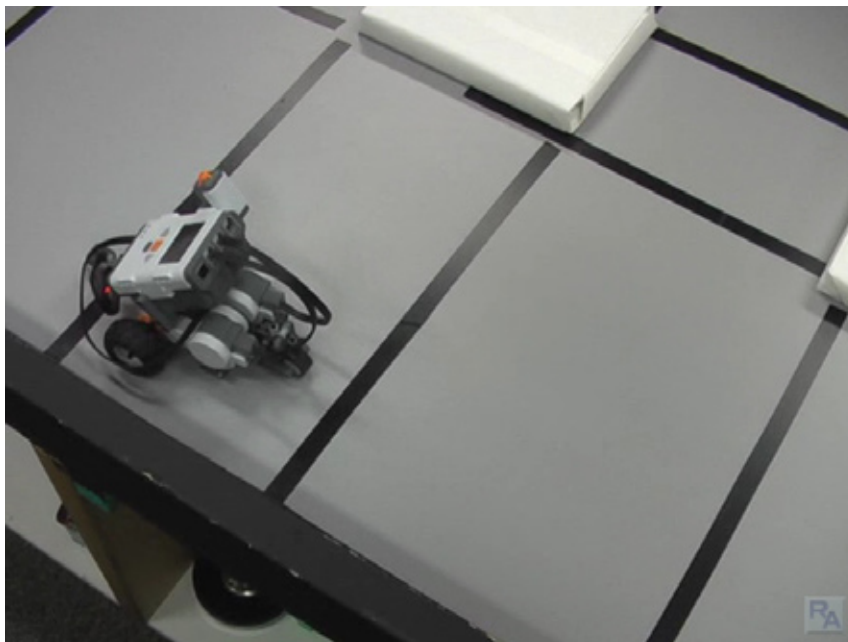
The numeric value 4 will take the place of "numLines"

Variables and Functions

Patterns of Behavior **Variables and Functions (Part 3)**

End of Section

Save, download, and run your program. The robot should complete its path from start to finish.



Take a moment to reflect on what you have done here. You haven't solved a simple problem using complex tools. You've solved a whole **family of problems**, and **created easy-to-use tools** that will make it simple to follow **any** of the paths your robot might need to take through the warehouse.

Your robot is beginning to reach a higher level. You are no longer limited to simply performing single tasks. Your programs, through the use of sensor information and the reuse of their own code in parameterized functions, are beginning to solve the actual problems that underlie the tasks, instead of just the single cases. This approach is many times more powerful, and your understanding of it marks your entry into the real world of programming. Congratulations.

Variables

Patterns of Behavior Quiz

NAME _____ DATE _____

1. In your own words, define a behavior.

2. Which of the following keywords begins a function declaration?

- a. task
- b. int
- c. main
- d. func

3. Use the following code to answer the questions below.

```
1 void specialFunction(int p)
2 {
3     motor[motorC] = p/2;
4     motor[motorB] = p/2;
5     wait1Msec(100*p);
6 }
7
8 task main()
9 {
10     specialFunction(50);
11 }
```

a. What behavior would running this program cause your robot to exhibit?

a. How could you get specialFunction to move for 1 second?