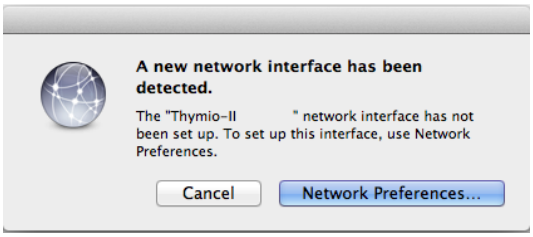
1 The Programming Environment and the Thymio II Robot
-Becoming Familiar with the System

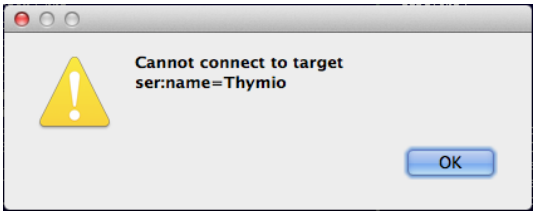
In this tutorial you will familiarize yourself with the Aseba Studio programming environment and build some small programs for the Thymio II robot. These programs are designed to facilitate experimentation with and calibration of the sensors and actuators used by the robot. Recall that one of the challenges in robotics is to define and specify a set of assumptions about the environment in which your robot operates. These assumptions must include the capabilities and tolerances of the robot's sensors and actuators. For example, it would be difficult to program a robot to pick up an egg without knowing how much force is exerted by the actuators—too little, and the egg will fall; too much, and the egg will be crushed.

1.1 Aseba Studio Programming Environment

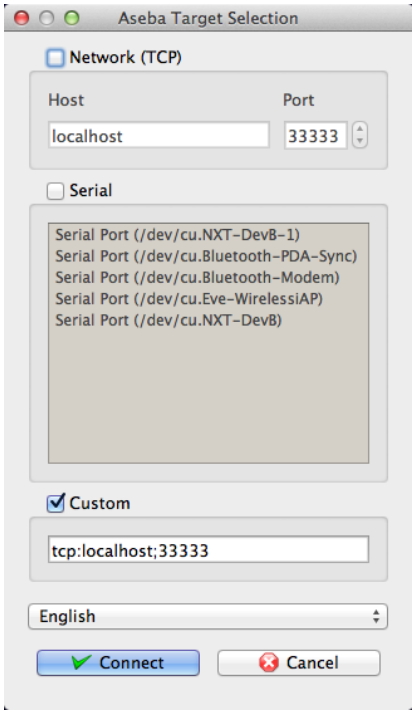
1. Log in into the computer.
2. Plug in the Thymio II robot and turn it on by holding down the center button until the robot turns on. If a dialog, such as the one below, appears, just click "`Cancel`".



(a) Just click "`Cancel`".

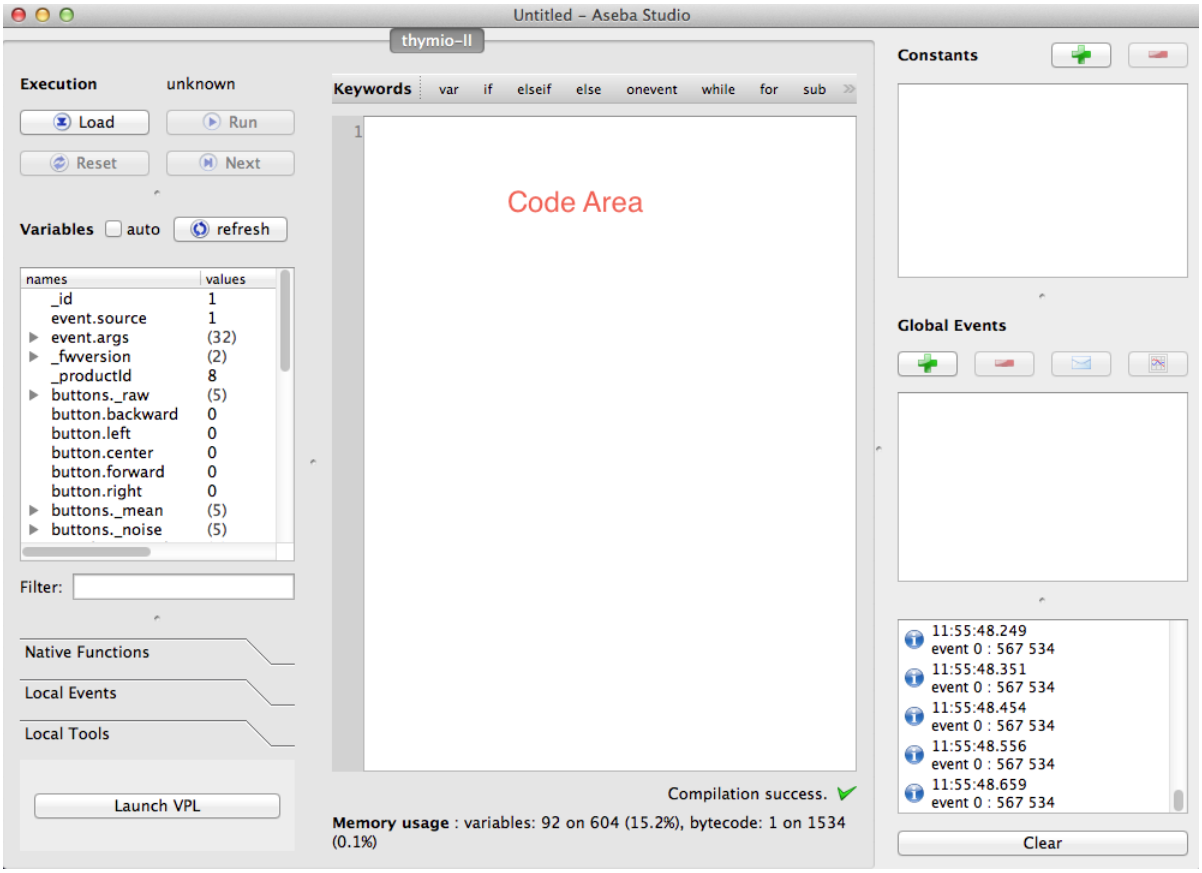


(b) Just click "`OK`".



(c) Just click "`Cancel`".

3. Run the "`Aseba Studio for Thymio II`" from "`Applications`" folder (Mac OS X) or the "`Start`" menu (Windows). If the robot was not plugged in, you will get warning about not being able to connect to the robot. In this case, click on "`OK`". A second dialog will appear. Click on "`Cancel`" to close the application, plug in the robot and reload the application. The software will take a few seconds to load; you will then see the application window.



4. You should become familiar with the layout of this development environment. The center area contains the Code Area, where you will add code to control your robot. The left list-box lists all the variables: both system and programmer-defined—variable names and variable values are displayed in this box. By default the values are not automatically updated. However, in most instances automatic update is preferred.

5. Check the "`auto`" box above the Variable List, which causes automatic update of the variables.

6. Scroll down the Variable List and expand the "`prox.horizontal`". A 7-element array, *prox.horizontal*, is displayed. The elements store the values measured by the proximity sensors on the front (5) and back (2) of the robot. place your hand in front of each of the proximity (**prox**) sensors and observe how the values change.

7. **Questions for Section 1.1:**

   (a) By placing your finger in front of each of the seven proximity sensors, identify the corresponding number of the sensor. I.e., the sensor numbers should range between 0 and 6. Draw a small diagram indicating the sensor numbers.

   (b) What value indicates that something is in close proximity to the sensor? What value indicates that there is nothing in the proximity of the sensor?

   (c) Are there other proximity sensors on the robot? If so, where are they?

Additional documentation can be found at: `https://aseba.wikidot.com/en:thymioprogram`

1.2 Moving in a Square

We are now ready to write our first program. In this case, we will do something simple like moving the robot in a square (approximately). A program comprises three parts:

Variable Declarations comprising a list of programmer defined variables, one per line. Each declaration begins with a *var* keyword followed by the variable name and an optional initializer, e.g., `var answer = 42`. There are no programmer defined variables in this example.

Initialization Code comprises the part of a program that is executed when the program starts running. This code initializes variables to initial values and prepares the program for execution. In the example below, the first three lines represent initialization code.

Event Handlers and subroutines comprise code for all the event handlers and programmer-defined subroutines. Each event handler begins with the keyword *onevent* and each subroutine begins with the keyword *sub*. In this example, there are three event handlers.

As a general rule, variable declarations must come first, followed by initialization code, and lastly, event handlers and subroutines.

1. Start a new program and enter the program in Figure 1 into the Code Area.

```
motor.left.target = 0                # reset motors amd timers
motor.right.target = 0
timer.period[0] = 0

onevent button.forward               # on forward button start
  motor.left.target = SPEED          # motor and timer
  motor.right.target = SPEED
  timer.period[0] = FWD_PERIOD

onevent button.backward              # on reverse button turn
  motor.left.target = 0              # off motors and timer
  motor.right.target = 0
  timer.period[0] = 0

onevent timer0                              # on timer event swicth
  motor.left.target = -motor.left.target    # between turn and fwd
  if motor.left.target < 0 then             # if we are turning
    timer.period[0] = TURN_PERIOD           # set turn time to 1sec
  else                                      # else
    timer.period[0] = FWD_PERIOD            # set turn time to 2 secs
  end
```

Figure 1: The Square Program

The `#` marks identify *comments*. A comment is ignored by the system, but is used by the programmer to document what her program is doing in a high-level easy to understand way. For example, the first statements of the program reset the motors and timer of the robot, the next piece of code specifies what should happen when the **Forward Button** is pressed (the motors are started as well as the timer), the third piece of code specifies what happens when the Back button is pressed, and the last part specifies what happens when the timer goes off.

This program works in the following manner:

(a) To move in a square, we need to make the robot move forward for a short period of time, say two seconds, and then make a 90 degree turn, then move forward another 2 seconds, make another 90 degree turn, and so on.

(b) The first three lines are executed when the program starts running. These lines reset the motors and the timer to 0. The **motor** devices are controlled by setting the *motor.left.target* and *motor.right.target* variables. The maximum value (speed) for the motors is 500 (forward) and −500 (reverse). To turn motors off, the variables are set to 0. To go straight, run both the left and right motors at the same speed. To turn the robot, run one of the motors forward and the other in reverse.

The robot has two timers, **timer0** and **timer1**. To use the timers, set the variable *timer.period*[*0*] and *timer.period*[*1*] to a period (in milliseconds) between 0 and 32767. If the period is nonzero, the timers will generate events `timer0` and `timer1` every so many milliseconds, as specified by the period. for example, if *timer.period*[*0*] is set to 1000., then **timer0** will generate a `timer0` event every second (1000 milliseconds).

By setting the motor and timer variables to 0, we ensure that the robot starts operation in a known state. The remainder of the program comprises three event handlers, which run when a corresponding event is generated.

(c) The `button.forward` event handler is executed whenever the **Forward Button** is pressed. This event handler starts the robot's motion. It runs both **motors** forward at the same speed by setting *motor.left.target* and *motor.right.target* to the constant *SPEED*. Lastly, the handler sets the period of **timer0** to 2 seconds, by setting the variable *timer.period*[*0*] to the constant *FWD_PERIOD*.

(d) The `button.reverse` event handler is executed whenever the **Reverse Button** is pressed. This event handler stops the robot's motion. It turns off both **motors** and **timer0** by setting *motor.left.target*, *motor.right.target*, and *timer.period*[*0*] to 0.

(e) The `timer0` event handler is executed whenever the period of the timer expires. The handler toggles the motion of the from forward to turn and vice versa by reversing the direction of one of the **motors**, and then toggling the period of **timer0** between *FWD_PERIOD* and *TURN_PERIOD*.

The program determines which period to use by comparing the speed of the afore mentioned **motor**. If the **motor** is rotating forward, then the robot is moving forward and the corresponding period is used. If the **motor** is rotating in reverse, then the robot is turning and the corresponding period is used.

2. Observe the message immediately below the Code Area, depicted in Figure 2. This indicates

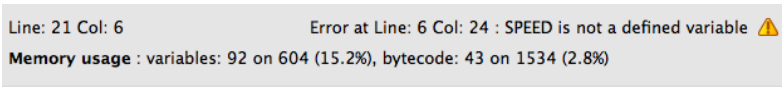


Figure 2: There is an error in your program.

that there is a problem with your program. Whenever you add code, the system will automatically check whether your program is correct and will notify you if it finds any errors by displaying the first error it discovers. In this case, the error message is:

```
Error at Line: 6  Col: 24 : SPEED is not a defined variable
```

Take a look at that line. Notice that we try to assign *SPEED* to *motor.left.target*, but we have never defined *SPEED*. In this case, we want to define a *constant* called *SPEED*. A *constant* is a named value that does not change, i.e., it's a value that remains constant.

3. Click on the [+] button in top right corner. A dialog will appear to enter the name and value of the constant. Enter *SPEED* as the name and 200 as the value, as shown below. Then,

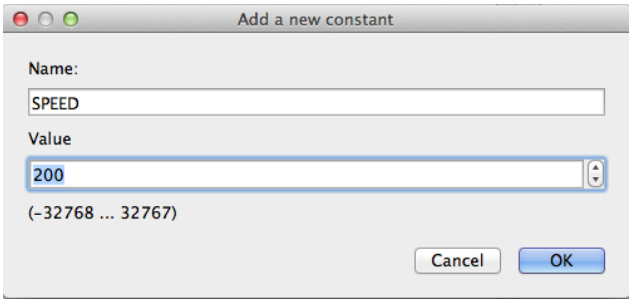


Figure 3: Declare a constant be entering the name and value.

click "`OK`". Notice that the error message changes, because the first error has been fixed.

4. Fix the remaining two error by adding the *FWD_PERIOD* and *TURN_PERIOD* constants. The *FWD_PERIOD* constant should be 2000, i.e., we want the robot to move forward for 2 seconds (2000 milliseconds). The *TURN_PERIOD* constant should be set to 1000, i.e., the right angle turn takes approximately 1 second (1000 milliseconds) to perform.

5. Notice that your program should now compile correctly, i.e., it has no obvious errors. At the bottom, you should see the message like in Figure 4. We now load and run the program!

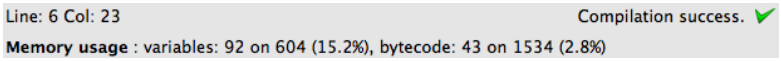


Figure 4: Your program is syntactically correct.

6. Save your program, by clicking on the "`Save`" option of the "`File`" menu. As in the previous module, it is important to save your work on a regular basis. Call your program "`Square`"

and save it to your shared drive. The resulting file will be called "`Square.aesl`". Be sure each group member has a copy of it as well by saving it on their USB key. This will ensure that everyone has copy of their work.

7. Load the program on to the robot by clicking on the "`Load`" button in the top left corner.

8. Run the program by clicking on the "`Run`" button. Note: the robot will not start moving because it is waiting for the **Forward Button** to be pressed.

9. Unplug the USB cable from the robot. Even though the USB cable is long and light, it will still affect the robot's motion. Hence, in most cases, it is recommended that you unplug the robot before hitting "Go". A dialog box will appear stating that the connection has been lost and that it is attempting to require the connection (Figure 5). **Do NOT press "`Cancel`".** If you do, you will need to exit the application and restart it in order to reconnect the robot.

Figure 5: **Do NOT press "`Cancel`".**

Once the robot is reconnected, everything will return to as it was prior to the disconnect. However, you will need to enable auto-update of the variables by clicking on the "`auto`" check-box. This is an annoying feature of the program.

10. You can now run the robot in a square. Find an empty surface, place the robot on it and press the **Forward Button**. The robot should start moving in a square. Note, that its turns may be slightly off, i.e., as time progresses, the square starts shifting.

11. Press the **Back button** to stop the robot. You can make it move again by pressing the **Forward Button** once more.

12. Stop the robot and reconnect it to the USB cable.

13. **Questions for Section 1.2:**

    (a) Suppose we changed the *SPEED* constant to a different value, say 400. How would this affect the behaviour of the robot?

    (b) Given the change above, would we need to change any other constants so that the robot continues to move in a square.

    (c) As you may notice, a 1 second turn, at a speed of 200, does not create a perfect 90 degree turn. Try adjusting the *TURN_PERIOD* constant to get as close as possible to the optimum value for a 90 degree turn. What is this value?

    Record your results in a log book—you will need them later.

Congratulations! You completed your first robot programming task.

1.3 The Ground Proximity Sensors: Staying within the Lines

Your next task is to learn about the ground proximity sensors that the robot has and how to use them to build a program that stays within an area demarcated by a black line.

1. Start a new program.
2. Create two constants: *SPEED* with a value of 200, and *EDGE* with a value of 400.
3. Enter the program in Figure 6 in to the Code Area.

```
motor.left.target = 0                        # reset motors (turn them off)
motor.right.target = 0

onevent button.forward                       # when forward button is pressed
  motor.left.target = SPEED                  # start motors forward
  motor.right.target = SPEED

onevent button.backward                      # when backward button is pressed
  motor.left.target = 0                      # stop the motors.
  motor.right.target = 0

onevent prox                                 # check proximity sensors
  if motor.left.target != 0 then             # only if motors are rotating

    # check if one of the ground sensors has recently detected a dark line
    when prox.ground.delta[0] < EDGE or prox.ground.delta[1] < EDGE do
      if prox.ground.delta[0] > EDGE then    # if line is not on the left
        motor.left.target = -SPEED           #   start turning right
      else                                   # else
        motor.right.target = -SPEED          #   start turning left
      end
    end

    # when neigther sensor detects a black line
    when prox.ground.delta[0] > EDGE and prox.ground.delta[1] > EDGE do
      motor.left.target = SPEED              # start moving straight again
      motor.right.target = SPEED
    end
  end
```

Figure 6: Boundary Avoidance Program

This program works in the following manner:

(a) To avoid the edges, we need to turn away from an edge whenever the robot detects that it is near one. The robot uses its two proximity ground sensors to detect the edge, which

is a thick black line. The sensors work by detecting a change in brightness of the light from the surface at the front of the robot.

(b) The first three parts of the program are nearly identical to those of the preceding program. The first two lines reset the motors to 0 (no motion). The `button.forward` and `button.backward` event handlers start and stop the robot's motors. The only difference is that the timer is not used in this program and hence its period does not need to be initialized.

(c) The program has a `prox` event handler that is executed every tenth of a second (10 times per second). The event handler first checks whether the robot is moving; otherwise there is no need to do anything. The handler then uses the *when ... do* statement to determine when the robot encounters a line, and when it is no longer on the line.

(d) The *when ... do* statement is similar to *if ... then* statement but differs in one important way. Recall that the *if ... then* statement evaluates its condition, and if the condition is true, executes the body of the statement (the part after the *then*). The *when ... do* statement is similar except that the body of the statement is executed only when the condition becomes true. For example, consider the code snippets in Figure 7. Suppose x is initially 0, since the condition above is false neither y or z is incremented. After

```
when x != 0 do
  y = y + 1
end
```

```
if x != 0 then
  z = z + 1
end
```

Figure 7: *when ... do* versus *if ... then*

x becomes nonzero, the next time the *when ... do* statement is executed, y will be incremented and so will z. Repeatedly executing the *if ... then* statement while x is not zero will cause z to be incremented each time. However, repeatedly executing the *when ... do* statement will not increment y each time. That is, the *when ... do* statement executes its body the first time the condition is evaluated as true. The body will not be executed again until the condition first becomes false and then becomes true again. As an analogy suppose you are driving a car. When the light turn red, you stop the car. But, while the light remains red, you do not keep stopping the car. The next time you stop the car will be only after the light turns green and then red again. Whereas the *if ... then* statement would have you repeatedly stopping the car, even though you may not be moving.

(e) The first *when ... do* statement checks whether either of the two proximity ground sensors senses a dark line. That is, if $prox.ground.delta[0]$ is below *EDGE* that means that sufficiently little brightness is being sensed, which indicates that the sensors is over a black line.

(f) When one of the proximity ground sensors encounters a dark line, the program determines which way to turn. If only the right sensor encounters the line, the sensor robot turns left. Otherwise, the robot turns right.

(g) The second *when ... do* statement returns the robot to its forward motion. When neither of the two sensors encounter a black line, both motors rotate in the forward direction.

4. Save the program under the name "`BoundaryAvoider`".
5. Load the program on your robot.
6. Create a small arena using black electrical tape provided by the lab facilitator.
7. Run the program and unplug the robot from the USB cable.
8. Place the robot into the arena and press the **Forward Button**.
9. Observe the robot's behaviour. The robot should remain within the boundary of the arena,
10. Stop the robot and reattach the USB cable.
11. **Questions for Section 1.3:**
    (a) What would happen if we had used the *if ... then* statement instead of the *when ... do* statement? Would it make a difference? Why?
    (b) Does the speed at which the robot travels matter? Try increasing the speed to maximum (500) and see if it makes a difference.
    (c) Suppose instead of the line being black, it was a lighter colour. What would you need to adjust? Why?

1.4 The Horizontal Proximity Sensors: Staying within the Walls

Your last task is to learn about the horizontal proximity sensors that the robot has and how to use them to build a program that avoids objects (walls) in front of the robot. You will also use programmer-defined variables and a math function provided by the system.

1. Start a new program.
2. Create two constants: *SPEED* with a value of 200, and *THRESHOLD* with a value of 0.
3. Enter the program in Figure 8 in to the Code Area.

   This program works in the following manner:

   (a) To avoid objects, we need to turn away from an object whenever the robot detects that it is near one. To detect nearby objects the robot can use its **horizontal proximity sensors**. The robot has seven (7) horizontal proximity sensors: five in the front and two in the rear. You can observe them in action by placing your hand or an object in front of a sensor—the closer the object is the brighter the associated LED lights up.

   The values measured by the sensors are stored in an array *prox.horizontal*[*0...6*]. The first five elements [0 . . . 4] store values from the front sensors (left to right) and the last two elements store the values from the rear sensors (left and right). The greater the value, the closer an object is to the sensor.

   (b) The first part of the program declares three programmer-defined variables. These are used to store statistics for the values measured by the horizontal proximity sensors.

```
var min                              # min over all sensor readings
var max                              # max over all sensor readings
var mean                             # average over all sensor readings

motor.left.target = 0                # reset motors
motor.right.target = 0

onevent button.forward               # when forward button is pressed
  motor.left.target = SPEED          # start moving forward
  motor.right.target = SPEED

onevent button.backward              # when backward button is pressed
  motor.left.target = 0              # stop motors
  motor.right.target = 0

onevent prox                         # check proximity sensors
  if motor.right.target != 0 then    # only if we are moving

    # compute min, max, and mean over the current sensor readings
    call math.stat( prox.horizontal[0:4], min, max, mean )

    when max > THRESHOLD do          # when a sensor is above threshold
                                     # if object is closer on the left
      if prox.horizontal[0] > prox.horizontal[4] then
        motor.right.target = -SPEED        # start turning right
      else                                 # else
        motor.left.target = -SPEED         # start turning left
      end
    end

    when max <= THRESHOLD do         # when all sensors are below threshold
      motor.left.target = SPEED      # start moving straight again
      motor.right.target = SPEED
    end
  end
```

Figure 8: Object Avoidance Program

(c) The next three parts of the program are identical to those of the preceding program. The first two lines reset the motors to 0 (no motion). The `button.forward` and `button.backward` event handlers start and stop the robot's motors.

(d) The program has a `prox` event handler that is executed every tenth of a second (10 times per second). The event handler first checks whether the robot is moving; otherwise there is no need to do anything.

(e) If the robot is moving, the `prox` event handler computes the maximum value reported by the five horizontal proximity sensors ($prox.horizontal[0:4]$). This is accomplished by calling the built in `math.stat()` function. The result is stored in variable *max*.

(f) The first *when ... do* statement checks whether there is an object in front of the robot by comparing the value in *max* to the *THRESHOLD*. If the value is greater than *THRESHOLD*, then there is an object in front of the robot.

(g) When an object is encountered, the program determines if it is on the left side or right side of the robot, by comparing the values of the leftmost and rightmost sensors (the larger value) indicates a closer object). If the object is on the left side, the robot starts turning right, otherwise it starts turning left.

(h) The second *when ... do* statement returns the robot to its forward motion. When the value in *max* is below *THRESHOLD*, this indicates that none of the horizontal proximity sensors are reporting an object in front of it. In this case, both motors should rotate forward.

4. Save the program under the name "`ObjectAvoider`".
5. Load the program on your robot.
6. Create a small enclosure using Duplo based walls provided by the lab facilitator (3×2).
7. Run the program and unplug the robot from the USB cable.
8. Place the robot into the enclosure and press the **Forward Button**.
9. Observe the robot's behaviour. The robot should avoid bumping into any of the walls.
10. Stop the robot and reattach the USB cable.
11. **Questions for Section 1.4:**

    (a) What would happen if we had used the *if ... then* statement instead of the *when ... do* statement? Would it make a difference? Why?

    (b) Does the speed at which the robot travels matter? Try increasing the speed to maximum (500) and see if it makes a difference.

    (c) Suppose you wanted to make the robot move closer to an object before turning away. What would you need to adjust? Why?

    (d) Suppose you were to use this program to run your Roomba, a robot that sweeps your floor while you are out. Would this program work well? Why or why not?

1.5 If You have Time

Create a program that searches for an object inside an arena bounded by a black line (electrical tape). Once an object is found, the robot should stop and start beeping until the object is removed from the arena. The robot should then start searching for the next object in the arena. For every object that the robot finds, an LED around the buttons should light up (up to 8 objects).

2 Characterizing Sensors
-Creating Sensor Models

In this tutorial you will empirically evaluate your robot's sensors and create simple models of how the sensors behave. A *model* is a simplified description of a system (sensor) that we are trying to understand and use. We use the model to predict how the sensor behaves and interpret its response.

You will be provided with programs that you will use to measure the accuracy of the sensors. This is called *characterizing* the sensors, i.e., determining the sensors' characteristics. This tutorial builds on the previous one by having you empirically evaluate the sensors you will be using.

Once you have gathered the data, you will then construct a simple model of the sensor, which you will then be able to use when programming your robot in the future.

2.1 Characterizing the Ground Proximity Sensors

In this section, we will investigate the response of the ground proximity sensors in typical lighting conditions. We will then construct a simple model of sensors' response.

1. Start a new program.
2. Add three constants: *SAMPLES* with a value of 10, *SAMPLE_PERIOD* with a value of 2000 and *SENSOR* with a value of 0. The first constant represents the number of times that we want the measurement performed, i.e., 10 measurements. The second constant represents the delay between measurements (two seconds). The third constant represents the sensor number; in this case, either 0 or 1.
3. Enter the code listed in Figure 9 into the Code Area.

```
var v[SAMPLES]                              # array of samples
var min                                     # stats variables
var max
var mean
var i = 0                                   # counter

call math.fill( v, 0 )                      # initialize arrray
timer.period[0] = SAMPLE_PERIOD             # turn on timer

onevent timer0                              # when time goes off
  v[i] = prox.ground.delta[SENSOR]          # record sensor reading
  i++                                       # increment counter
  if i >= SAMPLES then                      # if we are done
    timer.period[0] = 0                     #  disable timer and
    call math.stat( v, min, max, mean )     #  compute stats
  end
  call sound.freq( 440, 10)                 # beep to indicate progress
```

Figure 9: Sensor Sampling Program

This program works in the following manner:

(a) The program takes a measurement of the ground proximity sensor, one every 2 seconds for 20 seconds. It stores the measurements in an array, and after completing the measurements computes the minimum, maximum, and mean of the values in the array. These values can be viewed from the Aseba Studio, by examining the variables in question in the Variable Area on the left.

(b) Each of the ground proximity sensors measures three values:

ambient light is the light intensity at ground level (the surrounding light level)

reflected light is the mount of light received while the sensor emits an infrared light

delta light is the difference between the reflected and the ambient light

To use an analogy, if you are out in the forest at night, there is certain amount of light around, even when your flashlight is off. This is the *ambient* light. If you turn your flashlight on, you see all the light emanating from the flashlight that *reflects* from the ground. The difference between your flashlight being on and off is the *delta* light. We are typically interested in the *delta* or *reflected* light because the ambient light has little effect on the sensors.

(c) The program first declares and initializes the variables it uses to store the sensor measurements and compute the various statistics.

(d) The program then initializes the period of **timer0** to $SAMPLE\_PERIOD$, which is 2000 ms (2 seconds). This causes the `timer0` event to occur every 2 seconds.

(e) The `timer0` event handler reads the current value from one of the ground proximity sensors $prox.ground.delta[SENSOR]$ and stores it in the ith element of the array, where i is then incremented by the event handler.

(f) If i is greater or equal to the number of $SAMPLES$, then the sampling is completed. The handler then disables the timer by setting the timer period to 0 and computes the statistics.

(g) Lastly, the handler emits a beep, indicating that progress is being made.

(h) Once the beeping stops, the values in the array and the other variables can be examined in the Variable Area, located on the left side of the main pane of Aseba Studio.

4. Save the program under the name "`GroundSensorSampler`".

5. Load the program on the robot.

6. Create a table with six columns entitled "Response of Ground Sensor 0". Each of the six columns should be labeled with a gray scale value, $0\%, 20\%, 40\%, 60\%, 80\%, 100\%$. Each column will contain 13 values: 10 measurements, the minimum, the maximum, and the mean (see Table 1). **This is to be included with your lab report (the lab sheet that has several additional questions for you to answer).**

7. For each band in the gray-scale in Figure 10 (located at the end of this tutorial):

(a) Place the robot so that proximity ground sensor 0 is directly over the band.

(b) Run the program.

(c) Once the program finishes (no more beeps) examine the array of recorded values and statistics and record these in the corresponding column of the table.

Note, to speed things up, you can view the array and variables as the program runs and transcribe the values on the fly.

8. Change the constant *SENSOR* in the program above from 0 to 1.

9. Load and run the modified program.

10. Repeat the above measurement procedure for ground sensor 1, but this time only record the minimum, maximum and mean in the table for each grayscale value (see Table 2).

11. Create a line graph of "Grayscale Value" on the x-axis and "Ground Sensor Readings" on y-axis. Plot two lines, one for each of the ground sensors, using the computed mean for each of the grey bands. **This is to be included with your lab report as well.**

12. Let the other groups know of your results by publishing a table of the computed means on the whiteboard at the front of the lab. Note: All ten groups will need to put their tables up, so don't take too much space. Your table should look like this:

Group #	**Group Members:** Alice, Bob, Carol					
Real Values	0%	20%	40%	60%	80%	100%
Mean Values (Sensor 0)						
Mean Values (Sensor 1)						

13. We can construct a linear model for each of the sensors. For each sensor:

    (a) Using a ruler, draw a line that best approximates the plotted values. Mark the start and end points of the line on the graph, and determine their coordinates. E.g., The start point s may be at $(0, 95)$ and the end point e at $(100, 700)$. This line represents the *linear* model of the sensor. From this we can derive an equation that models the sensor.

    (b) Compute the slope m of the line by dividing the rise by the run. I.e.,

$$m = \frac{e_y - s_y}{e_x - s_x}$$

    In the example above, the slope would be:

$$m = \frac{700 - 95}{100 - 0} = 6.05$$

    (c) Compute the y-intercept of the line, denoted by b, by noting where it crosses the y axis. In this example, the initial point $(0, 95)$ is on the y axis, so the y-intercept is $b = 95$.

    (d) We now insert the slope m and y-intercept b into the equation of a line

$$y = m \times x + b$$

to yield the equation for our model, where x is the actual value and y is the measured value. That is, if we measure a grey colour at brightness x, then y is what the sensor in question should return. We can use this equation to model (predict) how our sensor behaves. Typically, it is more useful to determine what the brightness is, given the value from the sensor. Thus, we need to re-arrange the equation:

(e) Re-arrange this equation by solving for x:

$$x = \frac{y - b}{m} = \frac{y}{m} - \frac{b}{m}$$

Thus, if we measure y, plug it into the equation above, the result (approximately) is the shade of grey that the sensor is sensing.

Congratulations! You have created a linear model of your robot's ground proximity sensors.

14. Stick a piece of electrical tape onto your wooden table top.

15. Repeat the above procedure, i.e., create a table for the following two conditions (instead of the grey scale values):

    (a) the robot is sensing the desk

    (b) the robot is sensing the tape

    Hence, you will need to a new table (see Table 3) with four columns (two per sensor) and three rows each. **Include your the table as part of the report.**

16. Let the other groups know of your results, by publishing a summary table. The summary table on the whiteboard should look like this:

Group #	Alice, Bob, Carol	
Surface	Desk	Tape
Mean Values (Sensor 0)		
Mean Values (Sensor 1)		

17. **Questions for Section 2.1:**

    (a) What variables were fixed and what variables varied in the characterization that you performed?

    (b) Would the lighting conditions affect the readings? For example, would shadows or additional overhead lights change your measurements?

    (c) Do both ground proximity sensors have the same response (behaviour)? If not, what are the differences?

    (d) Suppose that you implemented your program on one robot but then had to run your program on a different robot (of the same type)? What problems would you encounter? How could these problems be addressed?

(e) Would it be possible to create a single model (a single equation) for both sensors? Why or why not?

(f) Given your measurements with the tape and table top, what threshold(s) would you use to distinguish between tape versus the table top? Can the same threshold be used for both sensors? Why or why not?

(g) How do your values (in both tables) compare to that of the other groups? Note: you may need to answer this question near the end of the lab.

2.2 Characterizing the Horizontal Proximity Sensors

In this section you will modify the program that you used in the previous section to characterize all seven of the horizontal proximity sensors.

1. Load the program "`GroundSensorSampler`", which you used in Section 2.1.
2. Save the program under the name "`HorizontalSensorSampler`".
3. Modify the program in the following ways:
   (a) Change the *SENSOR* constant back to 0.
   (b) On line 11, replace the variable access *prox.ground.delta*[*SENSOR*] with the variable access *prox.horizontal*[*SENSOR*].
4. Save the program.
5. Load the program on the robot.
6. Using a blank piece of paper and a ruler, create a 15cm ruler on the paper, marked off in 1cm increments. E.g., see Figure 11. Note: Some tape measures are available, but you should bring your own ruler to this lab.
7. Borrow a Duplo wall from the lab facilitator. The wall should be three bricks high.
8. Create a table with 7 columns, each column labeled by the corresponding distance: 1cm, 2cm, 4cm, 7cm, 10cm, 12cm, 13cm (see Table 4). Title the table "Response of Horizontal Sensors".
9. For each of the distances (1cm ... 13cm):
   (a) Place the robot on the ruler paper so that the horizontal sensor is located at centimeter 0 and the ruled line emanates directly from the center of the sensor and is perpendicular to the surface of the robot.
   (b) Place the Duplo wall at the corresponding distance such that it is directly parallel to the sensor (perpendicular to the ruler line).
   (c) Run the program.
   (d) Once the program finishes (no more beeps) examine the array of recorded values and record the mean statistic in the first row of the table, labeled as '0'.

10. Repeat the measurement procedure for at least two of the remaining six horizontal proximity sensors. You will need to:

    (a) Modify the program by changing the *SENSOR* constant to the corresponding identifier $(1 \ldots 6)$.

    (b) Adjust the position of the robot so that the correct sensor is located at centimeter 0.

    (c) Adjust the orientation of the robot so that the correct sensor is face-on with the Duplo wall and centered on the ruler.

    (d) Run the program to perform the measurements and record the results in subsequent rows of the table.

11. Plot a line graph of "Distance" on the x-axis and "Horizontal Sensor Readings" on y-axis. For each of the horizontal sensors plot the mean for each of the distances on this single graph. **This is to be included with your lab report as well.**

12. Using the same procedure as in Section 2.1 derive a single linear model for the sensors and derive the corresponding equation. That is:

    (a) For each distance, compute the average of the means (this can be done visually from the plot), and mark it on the plot.

    (b) Draw a line that best matches the plotted averages.

    (c) Compute the slope (m) and y-intercept (b) of the line.

    (d) Plug the values into the standard equation of a line.

13. Get together with another group and repeat the procedure at steps 8 and 9 for horizontal sensor 2 (the one in the center), when two robots are facing each other. That is, each group should take measurements of the response from their robot's sensor while the robots are 1cm, 2cm, ..., 13cm apart. The results should be recorded in a table (see Table 5), as in other cases.

14. **Questions for Section 2.2:**

    (a) What variables were fixed and what variables varied in the characterization that you performed?

    (b) Would the lighting conditions affect the readings? For example, would shadows or additional overhead lights change your measurements?

    (c) Do all the horizontal proximity sensors have the same response (behaviour)? If not, what are the differences?

    (d) Would it be possible to create a single model (a single equation) for all 7 sensors? Why or why not?

    (e) Compare your plots with a couple other groups. How do your robot's horizontal proximity sensors compare to that of other groups? Note: you may need to answer this question near the end of the lab.

    (f) Is the response of the center horizontal sensor significantly different when facing another robot versus a Duplo wall? If so, what are the differences?

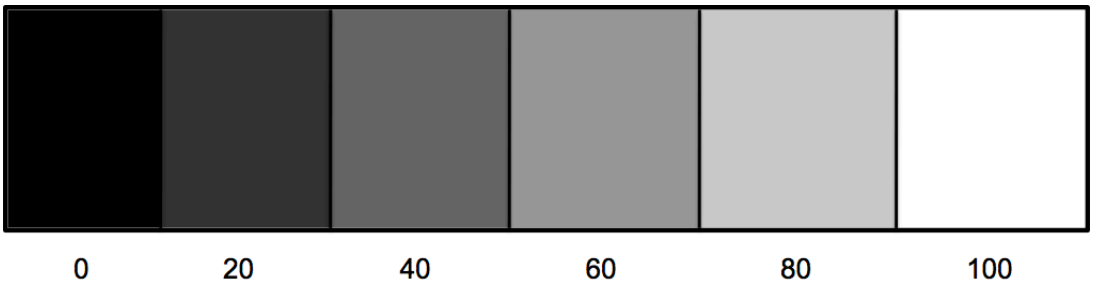


Figure 10: Gray-scale for characterizing the ground proximity sensors.

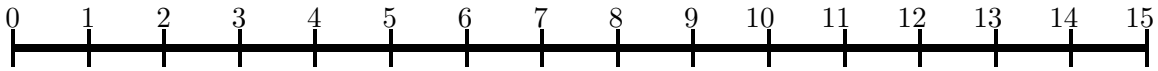


Figure 11: Ruler for characterizing the horizontal proximity sensors.

Table 1: Ground Sensor 0 Measurements

	0%	20%	40%	60%	80%	100%
0						
1						
2						
3						
4						
5						
6						
7						
8						
9						
min						
max						
mean						

Table 2: Ground Sensor 1 Measurements Summary

	0%	20%	40%	60%	80%	100%
min						
max						
mean						

Table 3: Ground Sensor Application Measurements Summary

	Sensor 0		Sensor 1	
	tape	desk	tape	desk
min				
max				
mean				

Table 4: Mean Horizontal Sensor Distance Measurements

Sensor	1cm	2cm	4cm	7cm	10cm	12cm	13cm
0							
1							
2							
3							
4							
5							
6							

Table 5: Horizontal Sensor 2 Distance Measurements to Other Robot Summary

	1cm	2cm	4cm	7cm	10cm	12cm	13cm
min							
max							
mean							

3 Characterizing Drive Actuators
-Handling Divergence

In this tutorial you will empirically evaluate the robot's actuators. For example, suppose that you wanted your robot to make a 90 degree right turn. In this case, the left wheel has to turn forward, and the right wheel in reverse. Furthermore, for each degree that the robot turns, the motors need to run at the given speed for a specific duration. Without knowing the duration needed to turn the robot some fixed number of degrees, it is not possible to program the robot to make specific maneuvers, In fact, as we shall see, the duration depends not only on the amount of turning required, but also on the speed and possibly other factors as well.

You will also evaluate movement accuracy. Initially you would expect that the robot will move or turn by an exact amount and that the robot should behave identically for a rerun of the same program. However, this may not be the case, and knowing the accuracy of the robot's movement will be critical when deciding how much to rely on the robot's expected position in a given task.

3.1 Calibrating and Measuring Divergence

In this experiment you will measure the amount of divergence from the straight path that robot undergoes as it moves in a straight line. Since the motors are imperfect, they may run at slightly different speeds, resulting in the robot veering to one side or another. The severity of this effect tends to be correlated with the speed of the robot. In this tutorial, you will simultaneously calibrate the forward movement of your robot and measure the divergence that occurs over a fixed distance that a robot travels at various speeds. You will then use this data to create a histogram that shows the expected amount of divergence of your robot, using the calibrated forward movements.

1. Start a new program.

2. Build a simple program that drives the robot forward when the **Forward Button** is pressed. The robot should stop *when* either of its ground proximity sensors encounters a black line (electrical tape).

   You will not need any variables, but you should create several constants: *SPEEDLEFT* and *SPEEDRIGHT*, initially set to 100, which you can use to independently set the speeds of the robot motors, *SPEEDMULTIPLIER*, initially set to 1, which is multiplied by the speed of each motor to increase the overall forward speed of the robot, and *THRESHOLD*, set to 300, which is to be used to identify when the robot has encountered the black line. You will need two event handlers: One for the `button.forward` event, to start the robot, and the other for the `prox` event, which should use the *when ... do* statement to check whether either of the two ground sensors is registering the black line, and if so, stops both motors. As we learned in the last tutorial, a ground proximity sensor registers a dark patch on the ground if it returns low values that are typically less then 300, which is what our *THRESHOLD* is set to.

3. Save the new program as "`DivergenceTest`".

4. Using black tape, demarcate a start and finish line on the table, such that the lines are parallel, and exactly 61cm apart. The robot itself is 11cm deep, so when its rear is placed on the start line, it will need to travel exactly 0.5m (50cm) before stopping. Note: The finish line should be about 20cm to 30cm long. Using a straight edge and two small pieces of tape

mark two points, representing a straight line that is perpendicular to the start and finish lines. (See Figure 12.) The first point should be on the finish line. The second, should be 11cm from the start line, when the nose of the robot will be when the rear of the robot is aligned with the start line. Note: the tape marks should be as small as possible.

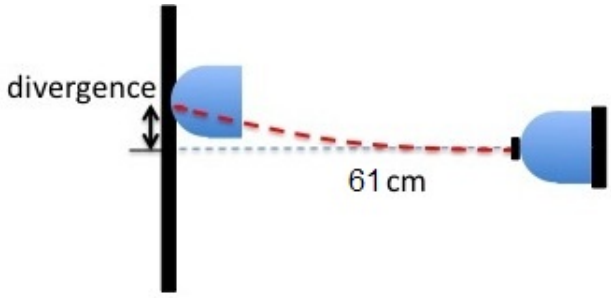


Figure 12: Setup for measuring divergence per speed multiplier.

5. Create a table with five (5) columns entitled "Divergence per Speed Multiplier" (see Table 6). Each of the five columns should be labeled with a motor speed multiplier setting: 1, 2, 3, 4, and 5. Each column will contain 15 values: the values for *SPEEDLEFT* and *SPEEDRIGHT*, 10 measurements, the minimum, the maximum, and the mean. **This is to be included with your lab report.**

6. For each of the five speed multiplier settings:

   (a) Set the *SPEEDMULTIPLIER* constant to the target setting.

   (b) Set the *SPEEDLEFT* and *SPEEDRIGHT* constants to 100.

   (c) Save the program and load it on the robot.

   (d) Run the program and unplug the USB cable.

   (e) Place the robot so that its rear edge is aligned with the start line and its front is aligned with the tape mark.

   (f) Press the **Forward Button**. The robot will move forward until it encounters the finish line.

   (g) Rerun this process a couple of times. If the robot appears to have a strong curve to the right or left, rather than moving straight, reconnect the USB cable and decrease either the *SPEEDLEFT* or *SPEEDRIGHT* accordingly, load and rerun your program. Do not increase these values above 100 since the maximum speed is 500 and you will be using a *SPEEDMULTIPLIER* of 5 shortly. Reduce the curving effect as much as possible.

   (h) Record the values of *SPEEDLEFT* and *SPEEDRIGHT* in the table under the corresponding column.

   (i) With the calibration completed (i.e., curving has been mostly eliminated), rerun the robot 10 times, and measure the distance along the finish line between the nose of the robot and the mark on the finish line, denoting where the robot should end up if there is no divergence. Record the distance in the table under the corresponding column.

   (j) Plug the USB cable back into the robot

(k) Manually compute the minimum, maximum and mean for the column of measurements and record them at the bottom of the column. To compute the mean (average), sum the 10 measurements and divide the total by 10.

7. Plot a line graph of "Multiplier Setting" on the x-axis and "Speed Ratio (Left/Right)" on y-axis. The ratio of $\frac{SPEEDLEFT}{SPEEDRIGHT}$ allows us to use a single point to represent the relationship of these two values.

8. Create a linear model, which tells us the appropriate speed ratio to use for a given speed multiplier. **This is to be included with your lab report as well.** Recall from the previous tutorial, that we construct a model by:

   (a) Using a ruler, draw a straight line that best approximates the plotted points.

   (b) Compute the slope (m) of the line.

   (c) Find the y-intercept (b).

   (d) Plug these into the equation of the line:

$$y = m \times x + b$$

9. Let the other groups know of your results by publishing a table of the computed means on the whiteboard at the front of the lab. Note: All ten groups will need to put their tables up, so don't take too much space. Your table should look like this:

Group #	**Group:** Alice, Bob, Carol				
Speed Multiplier	1	2	3	4	5
Speed of Left divided by Right					

10. Create a histogram showing the divergence of your robot. A histogram has vertical bars or bins, whose height indicates how frequently something occurs. For example, your histogram should have 10 bins, where each bin covers a range of divergence values (e.g., -0.5cm to 0.5cm) spanning the extent of the divergence (say, somewhere between -5cm and 5cm). For every divergence measurement that falls in the range of a particular bin, you increase its height by 1 unit. After slotting your 50 measurements in their appropriate bins, your histogram may have a bell-shaped (or Normal or Gaussian) distribution, which expresses how much your robot is expected to deviate for a traveling distance of 0.5m.

11. **Questions for Section 3.1:**

   (a) What variables were fixed and what variables varied in the characterization that you performed?

   (b) Suppose you wanted your robot's right motor to have a speed of 250. At what speed should you set your left motor so that the robot will go straight, given the linear model you constructed?

   (c) Does your histogram's highest bin have a range that surrounds a divergence of 0? If not, do you think that adjusting the slower motor speed would do this? Why or why not?

(d) How do your measurements compare to that of the other groups? Note: you may need to answer this question near the end of the lab.

(e) Describe how you would design an experiment to find a linear model relating the distance the robot travels and its speed, given a fixed time duration. Do you think it would experience some form of divergence as well? How could you check?

Table 6: Divergence per Speed Multiplier

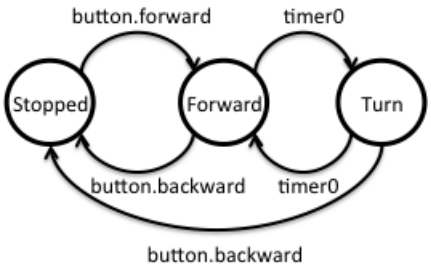
SPEEDMULTIPLIER	1	2	3	4	5
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					
min					
max					
mean					
SPEEDLEFT					
SPEEDRIGHT					

4 Modeling the Real World
-States and Transitions

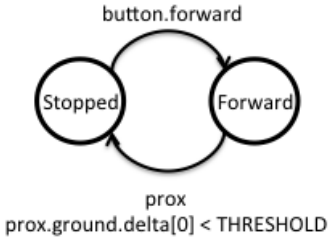
In this tutorial you will investigate methods for modeling the robot's behaviour and environment. You will begin by writing a small program that pauses the robot when it encounters an object ahead of it and another program that makes the robot to follow a thick black line (electrical tape). You will then experiment with the program, extend it in various ways, and derive a general approach for modeling robot behaviour in terms of states and transitions.

Robot behaviour in general can become very complex, particularly as the complexity of the robot's environment and the complexity of the tasks it must perform increases. Programming complex behaviour can quickly become overwhelming without some ability to manage this complexity. One approach is to model a robot's behaviour in terms of states and transitions. A state refers to a situation where a unique set of assumptions hold.

For example, consider the "Square" program from the previous tutorials. Initially the robot is in a Stopped state where it does not do anything. Once the `button.forward` event occurs, it starts moving forward and transitions to the Forward state. Once the `timer0` event occurs, the robot starts turning and transitions to the Turn state. Once the `timer0` event occurs again, the robot starts moving straight once more and transitions back to the Forward state. If the `button.backward` event occurs, the robot stops and transitions to the Stopped state.



In another example, consider the "DivergenceTest" program from the previous tutorial. In this program the robot is initially in a Stopped state. Once the `button.forward` event occurs, the robot starts moving forward and enters the Forward state. Once the robot detects a black line during the `prox` event it stops, transitioning to the Stopped state.



There are also problems with this approach. The number of states can quickly become very large and the number of transitions even greater. Furthermore, the state-transition approach may not be appropriate for certain tasks that involve counters. In this tutorial you will investigate the state transition approach.

4.1 A Simple Stop and Go Program

1. Start a new program.

2. Enter the code listed in Figure 13.

```
var min                          # min over all sensor readings
var max                          # max over all sensor readings
var mean                         # average over all sensor readings
var state = STOPPED              # state of the robot

motor.left.target = 0            # reset motors
motor.right.target = 0

onevent button.forward           # when forward button is pressed
  state = FORWARD                # transition to FORWARD state
  motor.left.target = TARGET     # start moving forward
  motor.right.target = TARGET

onevent button.backward          # when backward button is pressed
  state = STOPPED                # transition to STOPPED state
  motor.left.target = 0          # stop motors
  motor.right.target = 0

onevent prox                     # check proximity sensors
  if state != STOPPED then       # only if we are not STOPPED
    # compute min, max, and mean over the current sensor readings
    call math.stat( prox.horizontal[0:4], min, max, mean )

    when max > THRESHOLD do      # when a sensor is above threshold
      state = BLOCKED            # transition to BLOCKED state
      motor.left.target = 0      # stop motors
      motor.right.target = 0
    end

    when max <= THRESHOLD do      # when all sensors are below threshold
      state = FORWARD             # transition to FORWARD state
      motor.left.target = TARGET # start moving forward
      motor.right.target = TARGET
    end
  end
```

Figure 13: A Simple Stop and Go Program

3. Add the following constants:

TARGET the target setting for the motors when the robot is moving, set to a value like 200.

THRESHOLD the threshold used to determine whether the horizontal sensors are sensing an object ahead, set to 0.

STOPPED the Stopped state identifier, set to 0.

FORWARD the Forward state identifier, set to 1.

BLOCKED the Blocked state identifier, set to 2.

Note that the identifiers of the states need to be unique. The easiest way to ensure this is to number the states consecutively, starting from 0. The state transition diagram for the program is depicted in Figure 14.

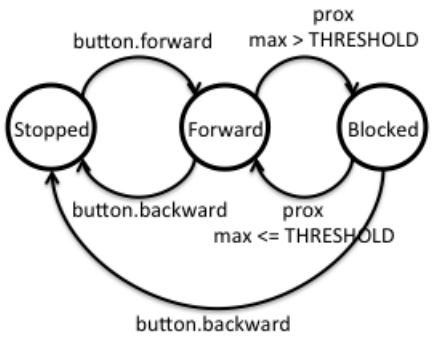


Figure 14: State transition diagram for the Stop and Go program.

4. Save the program as "`StopNGo`".

5. Load the program on your robot and give it a try. You should not need to unplug it.

6. **Questions for Section 4.1:**

   (a) Briefly explain how this program works.

   (b) How does the program encode its current state?

   (c) How did the programs that we developed previously encode their state? (Hint: Compare the `prox` event handler in this program to the `prox` event handler in the previous programs.)

   (d) Identify where the transitions depicted by the state-transition diagram occur in the program. Briefly describe the cause of each of the transitions.

4.2 A Simple Line Following Program

We now implement a line following program. That will allow the robot to follow a black line (electrical tape). Ideally we would first derive a state-transition diagram and then implement it. However, it is instructive to first do the reverse: derive a state-transition diagram from the program.

1. Start a new program.

2. Enter the code listed in Figure 15.

```
var state = STOPPED                                # variable declarations

motor.left.target = 0                              # reset motors
motor.right.target = 0

onevent button.forward                             # on forward button press
  state = LEFT                                     # transition to LEFT state

onevent button.backward                            # on backward button press
  state = STOPPED                                  # transition to STOPPED state
  motor.left.target = 0                            # stop motors
  motor.right.target = 0

onevent prox                                        # on prox event
  if state != STOPPED then                          # if robot is moving

    when prox.ground.delta[0] >= THRESHOLD do # when not on the line
      state = RIGHT                                #  transition to RIGHT state
      motor.left.target = TARGET                   #  move right
      motor.right.target = 0
    end

    when prox.ground.delta[0] < THRESHOLD do  # when on the line
      state = LEFT                                 #  transition to LEFT state
      motor.left.target = 0                        #  move left
      motor.right.target = TARGET
    end
  end
```

Figure 15: Line Following Program

3. Add the following constants:

   TARGET the target setting for the motors when the robot is moving, set to a value like 300.

   THRESHOLD the threshold used to determine whether the horizontal sensors are sensing an object ahead, set to 300, for now.

   STOPPED the Stopped state identifier, set to 0.

   LEFT the Left state identifier, set to 1.

   RIGHT the Right state identifier, set to 2.

4. Save the program as "`LineFollow`".

5. Load the program on your robot.

6. Using black tape, create a closed loop on the table to be used as a track by the robot. It should have a couple curves in it and one or two straightaways. (Don't make the corners too sharp.

7. Run the robot on the track to see how it performs.

8. Try adjusting the *TARGET* constant (speed of the motors) and the *THRESHOLD* used to distinguish tape from table.

9. Find the optimal settings for your line-follower so that it moves as quickly as possible without losing track of the line.

10. **Questions for Section 4.2:**

    (a) Briefly explain how this program works.

    (b) Draw the state-transition diagram for the program. Be sure to label all the states and transitions.

    (c) Compare this state-transition diagram to the one from the previous section. How similar are they? Why?

    (d) Identify where the transitions depicted by the state-transition diagram occur in the program. Briefly describe the cause of each of the transitions.

    (e) Consequently, why do we use *when ... do* statements rather than *if ... then* statements?

    (f) What is the sharpest corner that this program can negotiate in both directions? I.e., the robot should be able negotiate the corner regardless of which direction (along the line) it is traveling.

    (g) Under what conditions will the robot go off-track? Why?

4.3 A Stop and Go Line Follower

One problem with our current line follower is that the robot will bump into things as it is following the line. Ideally, we would like to make use of the horizontal proximity sensors to detect objects in the robot's path and pause the robot until the object is moved. Our next goal is to extend the line-following program to achieve this requirement.

1. Using the preceding state-transition diagrams as possible starting points, derive a new state-transition diagram for a program that follows the line but pauses the robot when an object appears in its path. Hint: The state-transition diagram requires at most four or five states. **This should be submitted with your lab report.**

2. Load the "`LineFollow`" program.

3. Save it as a new program called "`StopNFollow`".

4. Using the state-transition diagram you developed as a guideline, and the code from both the "`StopNGo`" program and the "`LineFollow`" program as guidelines, extend this copy of the "`LineFollow`" program to stop the robot whenever it is blocked. That is:

    - The robot should start following the line when the **Forward Button** is pressed.

- The robot should stop when the **Backward Button** is pressed.
- If the robot is moving and it encounters an object, The robot should pause while its path is blocked.
- If the robot is paused because it is blocked and the path becomes unblocked, the robot should resume following the line.

Note: Your program will have two different thresholds, one for the ground proximity sensors and the other for the horizontal proximity sensors. Hence you will need distinct names for the threshold constants. Be sure that all the states have distinct values.

5. Save your program and load it on the robot.
6. Try running your program. Fix any problems you encounter.
7. **Questions for Section 4.3:**
   (a) Briefly explain how this program works.
   (b) Identify where the transitions depicted by the state-transition diagram occur in the program. Briefly describe the cause of each of the transitions.
   (c) When the robot becomes unblocked, how does your program determine whether to transition to the Left or Right state?

4.4 Building a Better Line Follower

Interestingly, the line follower we have developed only uses one of its ground proximity sensors (the left one). The robot also wastes a lot of time and energy swinging from side to side when it could be going straight. The question you should ask is: Can we do better?

1. Load the "LineFollow" program.
2. Save it as a new program called "BetterLineFollow"
3. Think about how you can improve on the design of this program. As mentioned already, two possible things to consider is using the other sensor and allowing the robot to travel in a straight line, when it can.
4. Plan your program extension by designing the state transition diagram for the extension. The state transition diagram for the "LineFollow" program may be a good starting point.
5. Implement your extensions.
6. Test your extensions. Are your extensions an improvement?
7. Iterate over your design and improve it as much as possible.
8. Compare your design to the original by timing how long each of the programs takes to race around your track. Repeat the races several times. Note: If your design loses the line much more often than the original design, this may be a drawback.

9. Record the outcomes of your comparison in a table and determine how much faster your design is compared to the original.

10. **Questions for Section 4.4:**

    (a) Describe the improvements you made to the original program and justify your decisions.

    (b) Include the state-transition diagram of your final design.

    (c) What is the performance improvement of your design? Include the timings you gathered when comparing your design to the original.

    (d) Are there any drawback or flaws in your design that you have noticed? If yes, describe them. If no, justify why.

5 Dealing with an Imperfect World
-Fault Identification and Recovery

In this tutorial you will investigate methods for dealing with the harsh reality that the world is not perfect, that sensors fail, and that almost nothing goes exactly as planned. You will begin with a line following program that we developed in the last tutorial. We will then "mess" with the program, and then investigate how to compensate for the "messing".

To deal with the imperfections of the real world, we must do two things. First, we must identify when something has gone wrong, i.e., the failure mode. For example, you identify that you have missed the turn-off because you have driven for too long or that the furnace is broken because it's suddenly very cold in the house. Note, in most cases, the failure modes are explicitly enumerated and hence form part of the underlying assumptions in the system. In many cases, systems fail because the designers have not taken all failure modes into account, i.e., their assumptions are either incorrect or incomplete.

Once the identification of failure modes is accomplished, recovery mechanisms must be designed to deal with the failure modes. In some cases, the only solution is to shut the system down and wait for a repair-person. However, if the system itself is in an inconvenient location, say Mars, this is not a workable solution. For example, in the case of missing the turn-off, we find a way to reverse directions and return to the turn-off. In the case of a furnace failure we call the landlord (and hope for the best). Of course, if a failure occurs in the recovery mechanism, we may need a recovery mechanism for the recovery mechanism—especially if you are the landlord. In the language of states and transitions, a failure mode occurs when the system enters a state it's not supposed to be in. The goal of recovery is to return the system to a state that it is supposed to be in.

In this tutorial you will investigate how to deal with common failures of sensors and actuators.

5.1 Determining Failure Modes

1. Load the "`LineFollow`" program that was implemented in the last tutorial.
2. Load the program on your robot.
3. Using the tape track that you created in the previous tutorial, test how well the robot follows the line. If the track is too simple, feel free to make it more complex, with sharper corners. Note, there is asymmetry in the program in the sense that the robot only uses its left ground proximity sensor and uses it to detect the left edge of the line. Consequently, the robot may have more difficulty traversing the track in one direction than another. Specifically, the robot may be more likely to lose track of the line during a sharp left turn than a sharp right turn.
4. Try increasing the likelihood of the robot losing the line by increasing its speed.
5. Place a piece of paper on the track and observe what happens when the robot runs over it.
6. **Questions for Section 5.1:**
   (a) At what point do you know (from watching the robot) that it has lost the line?
   (b) In what direction (left or right) is the robot turning when it loses the line?
   (c) Consequently, what state is the robot in when it loses the line?

(d) Can the robot (running this specific program) lose the line while turning in the other direction? Why or why not?

(e) What would cause the robot to lose the line? Justify your answer.

(f) Suppose we were sitting inside the robot. How could we identify that this failure has occurred?

5.2 Failure Identification

We will now extend the "`LineFollow`" program by adding in a simple mechanism for detecting that the line has been lost. The failure that we need to identify occurs when the robot's ground proximity sensor either swings over the line too quickly to detect it, or is confused by other debris covering the line. In both cases the sensor does not detect the line and the robot continues to turn in the same direction, assuming incorrectly, that its sensor has not yet reached the line. A close analogy is a driver missing the turn-off either because she was driving too fast or because the road is very foggy. The driver would identify such a failure mode by noting that she has been driving too long without seeing the turn-off and concludes that she has missed the turn-off.

1. Save a copy of the "`LineFollow`" program as "`FailFollow`".

2. Update the state-transition diagram for the program by adding an additional state (Lost). As you have already observed, the robot can only lose the line when turning right. I.e., it is in the Right state. Thus, there must be a transition from the Right state to the Lost state. The question is, when does this transition occur?

   Just like the driver, the robot can assume that it has lost the line, if it has been turning right (in the Right state) for too long. That is, a `timer*` event has occurred. Thus, the transition from Right to Lost will happen as a result of a `timer*` event. Once the robot enters the Lost state, it should, at least for now, beep to alert the user.

   We are now ready to make the modifications to our program based on our new state-transition diagram. **Be sure to include the diagram with your lab report.**

3. Make the following modifications to the program:

   (a) In the initialization code, after the variable declarations, initialize the period of **timer0** to 0. (For examples, see programs from previous tutorials.)

   (b) Add a new constant *LOST* with a value of 3. I.e., it should be different from all other state identifiers.

   (c) At the end of the program add a `timer0` event handler that

      i. Sets *state* to *LOST*,

      ii. Resets the period for **timer0** to 0, and

      iii. Generates a beep.

   (d) Immediately after the line that sets *state* to *LEFT*, add a line that sets the period of **timer0** to 0.

   This line turns the identification mechanism off, because it is not needed when the robot is turning left, and in fact could adversely affect the program if triggered, i.e., like a false fire alarm.

(e) Similarly, in the `button.backward`, set the period of **timer0** to 0 as well.

(f) Immediately after the line that sets *state* to *RIGHT*, add a line that sets the period of **timer0** to 1 second (1000ms).

This line is essential because it turns the timer on and begins the countdown. Once two seconds has elapsed, the `timer0` event is triggered, and its handler transitions the system to the Lost state,

4. Save the program and load it onto the robot.

5. Try out this program on the track and see if it beeps when the line is lost. Note, the program makes no attempt, yet, to recover from the failure, all it does is identify that a failure has occurred.

6. **Questions for Section 5.2:**

(a) According to the state transition diagram, once the robot enters the Lost state, it can never leave.[1] Is this actually the case? There are two missing transitions in the state-transition, both emanating from the Lost state.

i. Identify the transitions and add them to the state-transition diagram.

ii. Justify these transitions by referring to the program to explain how they could occur.

(b) Once we identify that the line is lost, how could the robot recover from this failure?

5.3 Failure Recovery

We will now complete the "`FailFollow`" program by adding in a simple recovery mechanism for returning to line that was lost. Recall, that our failure occurs when the robot's sensor misses the line because it was moving too quickly, or there was some debris on the line. We can deal with this in the same manner that a driver would deal with the failure of missing her turn-off: Simply backtrack and try again. This approach works remarkably well for many simple failure modes.

1. Until the robot recovers, it remains in the Lost state. Recovery ensues when the robot detects the line once again. Once it does, it can transition into the Left state, and continue as before. Consequently, our state-transition diagram should already be up to date. All we need to do is to implement the recovery behaviour associated with the Lost state, which in this case is simply reversing direction and backtracking to to the line.

2. Add code to the end of the `timer0` event handler to move in reverse of the present trajectory. That is, both the left and right motors should be assigned target settings that are negative of their current ones. That's all! Once the robot finds the line and transitions to the Left state, the motors' target settings will again be set to forward motion during the transition.

3. Save the program and load it onto the robot.

4. Try out this program on the track and see if it recovers from its blunders.

5. **Questions for Section 5.3:**

[1] Like Hotel California.

(a) Suppose the robot reaches the end of the line (not a loop). What happens?

(b) Is it possible for the robot to get stuck in the same place and not make any progress? Why or why not?

(c) Under what conditions could this recovery mechanism fail?

(d) Could these identification and recovery mechanisms be used with your "`StopNFollow`" program? Why or why not? If yes, are there modifications that you would need to make to your program in addition to the ones described in your tutorial? If so, what are they? If not, why not?

5.4 Acknowledging Defeat

Suppose your robot comes to the end of the line. The robot will assume it lost the line and will attempt to recover. However, if the robot has arrived at the end of the line, then it will continue to search for the lost line that is no longer there, forever. This is not desirable. Ideally, the robot should attempt to locate the line, but if it does not succeed after a couple attempts, it should let the user know, stop, and save its batteries.

There are two ways to accomplish this. The first is to use a timer. For example, if the robot remains in the Lost state for more than ten seconds, the search is hopeless. The other approach is to use a counter. For example, if the robot enters the Lost state too many times in a given period of time, then there is clearly something wrong and the robot should respond to this. Typically, both approaches are used because they are complementary.

You will extend your "`FailFollow`" program by implementing one or both of these methods.

1. Save a copy of the "`FailFollow`" program as "`TryFollow`".

2. Update the state-transition diagram for the program by adding another transition from the Lost state to the Stopped state. The transition will be triggered by a timer expiring. That is, when the Lost state is entered, a timer will be set. If the timer expires while the systems is still in the Lost state, then the robot can transition into the Stopped state.

3. Implement the additions to the state-transition diagram by modifying the `timer0` event handler. Instead of changing the state of the system to the Lost state, the handler should check, using an *if ... then ... else* statement if the system is already in the Lost state. If not, then the handler should transition to the Lost state, emit a beep, and set the motor speeds as before, but instead of disabling **timer0** by setting its period to 0, the period should be set to ten seconds, the amount of time after which the robot is expected to give up.

   If the `timer0` event handler executes and the robot is in the Lost state, then the handler, should disable the timer and the motors, and transition to the Stopped state.

4. Save your program, load it on the robot, and give it a try. Try running your robot on a blank table top, with no tape. See if the robot stops after ten seconds.

5. We will now add a mechanism to stop the robot if it loses the line too many times within a specified period.

   Update the state-transition diagram for the program by adding another transition from the Lost state to the Stopped state. The transition will be triggered by a timer expiring in conjunction with a counter variable being too high.

6. Program the changes in the state-transition diagram by adding a variable, adding a second timer and modifying the `timer0` event handler. First, add a variable called *LostCount*. In the `button.forward` handler initialize it to 0 and set the period of **timer1** to ten seconds. This will cause the second timer to go off every ten seconds.

   Second, whenever the robot transitions into the Lost state, increment the *LostCount* variable. Hence, we are counting the number of times that the robot loses the line.

   Third, add a `timer1` event handler and a constant called MAX_LOST, which should be set to 3. Using an *if ... then ... else* statement, the handler should compare *LostCount* to MAX_LOST. If *LostCount* is less than MAX_LOST, the handler should simply reset the variable to 0. Otherwise, the handler should transition the robot to the Stopped state and disable all motors and timers.

   Lastly, you may wish to disable **timer1** whenever the robot transitions to the Stopped state by setting its period to 0.

7. Save your program, load it on the robot, and give it a try. Try running your robot on a track and use a blank piece of paper to make the robot lose the line several times in quick succession. See if the robot stops.

8. **Questions for Section 5.4:**

   (a) Suppose the robot only had one timer instead of two. Is it possible to implement the same behaviour as above? If yes, how? If no, why not?

   (b) Currently, the timeout periods are set to 10 seconds.

      i. Why is this not necessarily a good idea?

      ii. Could you suggest a better way of setting the timeout period? Hint: What is the timeout period a function of?

6 Programming Techniques
-Subroutines and Development Cycle

In this tutorial you will learn about subroutines, a useful programming feature, and go through the process of solving a problem by analysis, design, implementation and testing. Subroutines provide a simple mechanism for making your programs simpler and avoiding replication of code. Solving a problem in robotics, as in any other area can be challenging. However, having a process to follow does make the task easier, but still requires significant work on your behalf.

6.1 Subroutines

A subroutine is a piece of code that can be called by other parts of the program. When the subroutine is called, it executes its code and then returns to the caller, from where it was called. As an analogy, think about how you check your email. That is, every time you check your email you do the same thing: stop what you are doing, get your phone, enter the access code, run the mail application, check the messages, lock the phone, put the phone back, go back to whatever you were doing. Because you check your email or text messages quite often, you most likely have a routine (subroutine) that you perform each time you do it. When a program needs to perform a specific task, it calls (executes) the subroutine, and when the subroutine completes, it returns and the program continues running from where it left off. Subroutines can be called from multiple locations in a program, which means that they can be used to organize your program into modules or parts, making it easier to both understand and to modify.

1. Load the "`TryFollow`" program that was implemented in the last tutorial.
2. Save a copy of the program as "`TryFollowSub`".
3. Observe that whenever the program transitions into the Stopped state, the program performs the same operations: perform transition, disable motors, and disable timers (see Figure 16).

```
state = STOPPED
motor.left.target = 0
motor.right.target = 0
timer.period[0] = 0
timer.period[1] = 0
```

Figure 16: Operations performed on transition to Stopped state.

The transition to the Stopped state occurs at least three times in the program, and whenever it occurs, the same code is repeated. This is a problem!

There are actually a couple problems. First, your program is longer than it needs to be, taking up more memory and seeming more complex than it needs to be. Second, adding new transitions to the Stopped state will require you to duplicate the code for each new transition. Third, any changes to the code associated with transitions into the Stopped state will need to be duplicated in multiple locations in the code. For example, recall from the last tutorial, that you had to update the code for all transitions into the Stopped state when you started

using the second timer. This adds further complexity to the program and potential for bugs. Ideally, there should be a single instance of the code to transition into the Stopped state. We can use a subroutine to solve these problems.

4. Immediately prior to the first event handler in your program define a subroutine called "to_stop" that performs the statements in Figure 16. A subroutine comprises three parts:

```
sub to_stop
  state = STOPPED
  motor.left.target = 0
  motor.right.target = 0
  timer.period[0] = 0
  timer.period[1] = 0
```

Figure 17: Operations performed on transition to Stopped state.

   (i) the `sub` keyword, (ii) the subroutine name (a single alphanumeric string), and (iii) the subroutine body, comprising the statements to be executed when the subroutine is called.

   It is strongly recommended that all subroutines be defined *before* any of the event handlers because subroutines must be defined before they can be called. However, subroutines must be defined *after* the variable declarations and initialization code. Consequently, the initialization code cannot call any subroutines.

5. In the event handlers, replace the code for transitioning into the Stopped state (see Figure 16) with a call to the subroutine. A call consists of the keyword `callsub` followed by the name of the subroutine, e.g., `callsub to_stop`

   Observe that your code has become shorter, because the event handlers are shorter and easier to read, and if you do need to make changes to the transition into the Stopped state, you only need to make them in one place.

6. Save, load, and test the program. It should behave exactly the same way as the original program did.

7. **Questions for Section 6.1:**

   (a) What other programs that you have written over the past tutorials could be improved through the use of subroutines. Briefly justify your answer.

   (b) One use for subroutines is to make a subroutine for each state transition in your program. Is this a good idea? Why or why not?

6.2 A Spin through the Development Cycle

Developing a program for the robot to solve a given task can be a challenging experience. As we discussed previously, the world is uncertain and many things can go wrong, Furthermore, the robot has to make decisions and perform actions in a given order which can be complicated in its own right. Fortunately, we can follow a process to make the task more manageable.

The process involves several steps: analyzing the problem, developing a strategy, designing a solution, implementing the solution, and testing the solution. For the remainder of this tutorial, we will run through this process.

First the problem. Suppose we wanted our robot to negotiate a maze composed of walls of Duplo blocks. Furthermore, we do not know the layout of the maze. We may be told that all passageways will be 20cm wide and that there are no cycles in the maze, but we do not have a map of the maze. How do we solve this problem?

1. We begin with an analysis of the problem. First, since the walls of the maze will be composed of Duplo blocks, we can use the robot's horizontal proximity sensors to sense the walls. Second, since we do not know the shape or layout of the maze, the robot will need to search the maze for the exit. Third, since all passage ways will be at least 20cm wide, the robot should have plenty of space to maneuver. Thus, we need a strategy for finding the maze exit.

2. Next, we develop a strategy. Our analysis reveals that our robot will need to search the maze for an exit. We now consider several strategies: the first one would be the random strategy: drive forward until encountering a wall, turn a random number of degrees, and drive forward again. Although this strategy may take a very long time to succeed, it will, with high probability, eventually succeed. Furthermore, the strategy is easy to implement—we did this in the first tutorial. Can we do better?

   Another strategy that would yield a more systematic (and faster search) is to use the right-hand rule. The idea is that the robot follows the wall on its right-hand size. For example, suppose we were in an unlit building. We walk until we walk into a wall; place our right hand on the wall; and start walking forward, keeping our hand on the wall. Eventually, assuming there are no cycles in the building, we will find the exit. Our robot can use the same strategy. This strategy is guaranteed to work in a bounded amount of time, because our robot will traverse each passageway in the maze at most twice (once in each direction). So, we will use this strategy.

3. We now need to design a solution based on this strategy. How do we implement a right-hand rule? What tactics should the robot use? Well, the robot must track the wall on its right.

   - If the robot senses no wall in front of it or on its right, the robot should move rightward, until it senses a wall.
   - If the robot senses no wall in front of it but a wall on its right, it should move along the wall, and perhaps slightly leftward to avoid crashing into the wall—once it stops sensing the wall on its right, it can start moving in a rightward direction again.
   - If the robot senses a wall in front of it, but no wall on its right, the robot should make a sharp right turn until it no longer senses the wall in front—the right-hand rule dictates that the robot must always turn right if it can.
   - If the robot senses a wall in front and on its right, then its only option is to make a sharp left turn.

   Note how these four rules essentially define the right-hand rule strategy. We are getting closer to our solution, because if you observe carefully, the above four rules define the robot's states that it will use to implement the strategy.

We should also suggest that robot begin in a nonmoving state, and start moving rightward when the **Forward Button** is pressed. It should stop when the **Backward Button** is pressed.

4. We now derive the corresponding state-transition diagram. There will be five (5) states:

   Stopped Robot is motionless.

   Rightward Robot is moving in a rightward direction.

   Leftward Robot is moving in a leftward direction.

   RightTurn Robot is making a sharp right turn.

   LeftTurn Robot is making a sharp left turn.

   The rules in the previous step define the transitions. Note, that the robot can transition from any of the latter four states to any of the states, and it transitions from the Stopped state to the Rightward state.

   Submit this State transition diagram as part of your lab report.

5. Start a new program.

6. We now implement the solution, represented by the state-transition diagram as a program. We will only need to use two horizontal proximity sensors: *prox.horizontal*[*2*] (front) and *prox.horizontal*[*4*] (right). Based on the description of the transitions, the program will need to have three event handlers: `prox`, `button.forward`, and `button.backward`. As with our previous programs, your program will need to have a *state* variable, and several constants representing the states: *STOPPED*, *RIGHTWARD LEFTWARD*, *RIGHTTURN*, and *LEFTTURN*, all with unique values. You will also need a *TARGET* constant and a *THRESHOLD* constant to define the speed of the robot and to compare the values returned by the horizontal proximity sensors. Use programs that you implemented previously as templates. Hint: To implement the transitions into the latter four states, your `prox` event handler will have four *when ... do* statements, one for each of states.

   For the left turn and right turn settings, use the sharpest turn possible, i.e., one motor should be set to *TARGET* and the other to $-TARGET$. For the Leftward and *Rightward* states the turns must be wider. In fact, the rightward turn should be wider than the leftward turn. You will need to try out a variety of values for the target settings. Hint: To make things consistent, the target setting of one motor should be a fraction of the other. For example, for the rightward turn, if the target setting for the left motor is *TARGET*, then the target setting for the right motor should be $TARGET \times \alpha$ where α is a fraction between 0 and 1, e.g., 0, $\frac{1}{2}$, 1, etc.

   Save your program as "`MazeRunner`".

7. Lastly, test and refine your program. Using the Duplo blocks provided by the lab facilitator, create a maze. Note that one wall (2 by 20 pips) is approximately 20cm wide. Run your robot through the maze. You may need to adjust the settings for your leftward and rightward fractions of *TARGET*, as well as possibly the *TARGET* and *THRESHOLD* as well. This process may take some time.

8. Congratulations! You have completed a program that solves a specific task.

9. **Questions for Section 6.2:**

   (a) Give the state-transition diagram for the "Random" strategy described in Step 2.

   (b) For each of the *when ... do* statements in your program give the conditions under which the statement executes its body and identify the transitions associated with it.

   (c) What target settings did you use for the leftward and rightward motion?

   (d) Suppose the robot is placed in the middle of a large area in the maze such that there are no walls near by. What would happen once the **Forward Button** was pressed?

   (e) How long (how many trials) did it take you to determine the optimal settings?

   (f) Suppose we wanted the robot to stop at the exit to the maze, which is demarcated by black tape. How would this change the state-transition diagram and your program?