

First Steps in Robotics with the Thymio Robot and the Aseba/VPL Environment

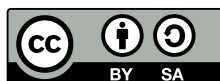
Moti Ben-Ari and other contributors

see authors.txt for details

Version 1.5.1 for Aseba 1.5.2

© 2013–16 by Moti Ben-Ari and other contributors.

This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/> or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.



Contents

| | |
|-------------------------------------|-----------|
| I Tutorial | 9 |
| 1 Your First Robotics Project | 10 |
| 2 Changing Colors | 17 |
| 3 Let's Get Moving | 20 |
| 4 A Pet Robot | 23 |
| 5 The Robot Finds Its Way by Itself | 27 |
| 6 Bells and Whistles | 31 |
| 7 A Time to Like (Advanced Mode) | 34 |
| 8 States (Advanced Mode) | 36 |
| 9 Counting (Advanced Mode) | 43 |
| 10 Accelerometers (Advanced Mode) | 47 |
| II Parsons Puzzles | 49 |
| 11 Parsons Puzzles for VPL | 50 |
| III Projects | 55 |
| 12 Braitenberg Creatures | 56 |
| 13 The Rabbit and the Fox | 58 |
| 14 Reading Barcodes | 59 |
| 15 Sweeping the Floor | 60 |
| 16 Measuring Speed | 61 |
| 17 Catch the Speeders | 62 |
| 18 Finite Automata | 63 |
| 19 Multiple Sensor Thresholds | 66 |

| | |
|--|-----------|
| 20 Multiple Thymios | 68 |
| IV From visual to textual programming | 69 |
| 21 Learning AESL from VPL programs | 70 |
| V Appendices | 83 |
| A The VPL User Interface | 84 |
| B Summary of VPL Blocks | 86 |
| C Tips for Programming with VPL | 89 |
| D Techniques for Using the Sliders | 91 |

Preface

What is a robot?

You are riding your bicycle and suddenly you see that the street starts to go uphill. You pedal faster to supply more power to the wheels so that the bicycle won't slow down. When you reach the top of the hill and start to go downhill, you squeeze the brake lever. This causes a rubber pad to be pressed against the wheel and the bicycle slows down. When you ride a bicycle, your eyes are *sensors* that sense what is going on in the world. When these sensors—your eyes—detect an *event* such as a curve in the street, you perform an *action*, such as moving the handlebar left or right.

In a car, there are sensors that *measure* what is going on in the world. The speedometer measures how fast the car is going; if you see it measuring a speed higher than the limit, you might tell the driver that he is going too fast. In response, he can perform an action, such as stepping on the brake pedal to slow the car down. The fuel meter measures how much fuel remains in the car; if you see that it is too low, you can tell the driver to find a gas station. In response, he can perform an action: raise the turn-signal lever to indicate a right turn and turn the steering wheel in order to drive into the station.

The rider of the bicycle and the driver of the car receive data from the sensors, decide what actions to take and cause the actions to be performed. A *robot* is a system where this process is carried out by a computerized system, usually without the participation of a human.

The Thymio robot and the Aseba VPL environment

The Thymio is a small robot intended for educational purposes (Figure 1.1). The robot includes sensors that can measure light, sound and distance, and can detect when buttons are touched and when the robot's body is tapped. The most important action that it can perform is to move using two wheels, each powered by its own motor. Other actions include generating sound and turning lights on and off.

The name Thymio is used in this document to refer to the Thymio II robot.

Aseba is a programming environment for small mobile robots such as the Thymio. VPL is a component of Aseba for *visual programming* that was designed to program Thymio in an easy way through event and action blocks.

Overview of the tutorial

For each chapter, we give the main topic, as well as lists of the event and action blocks that are introduced. I suggest that you start with the tutorial chapters on VPL basic mode. Then, you

can study the tutorial on advanced mode, or try some of the projects. The Parsons puzzles can be tried whenever you feel the need to evaluate your knowledge of VPL. Read the Chapter 21 when you are ready to leave VPL and work with the more advanced Aseba Studio environment. The appendices contain reference material that can be read as needed.

Part I: Tutorial

Chapters 1 and 2 are an essential introduction to the robot, the VPL environment and its principal programming construct: the event-actions pair.

Events: Buttons



Actions: Top colors, bottom colors



Chapters 3 to 5 present the events, actions and algorithms for constructing autonomous mobile robots and should be the core of any activity using Thymio and VPL.

Events: Buttons, front sensors, bottom sensors



Actions: Motors



Chapter 6 describes features of the robot that can be fun to use but are not essential: sounds and shocks.

Events: Tap, clap



Actions: Music, top colors, bottom colors



Advanced mode

VPL has a basic mode which supports elementary events and actions that are easy for beginners to master. The advanced mode of VPL supports more events and actions that require experience to use. Explanations of the features of advanced mode start in Chapter 7.

Chapter 7 presents timed events. There is an action to set a timer and when the timer expires, an event occurs.

Events: Timer expired



Actions: Set timer



Chapters 8 and 9 explain state machines, which enable the robot to perform different operations at different times. States can also be used to perform elementary arithmetic like counting.

Events: State associated with an event

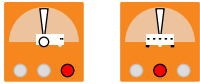


Actions: Change state



Chapter 10 describes how to use the accelerometers in the Thymio robot.

Events: Accelerometer events



Part II: Parsons puzzles

Chapter 11 presents Parsons puzzles which are exercises that you can use to check your knowledge of VPL.

Part III: Projects

Chapters 12 to 20 specify projects that you can design and implement on your own. The VPL source is available in the archive, but I suggest that you work on the projects before looking at the solutions.

Part IV: From visual to textual programming

Chapter 21 points to the next step: using the textual Studio environment which offers significantly more support for developing robots than does the VPL environment.

Part V: Appendices

Appendix A contains a description of the user interface—the buttons on the toolbar.

Appendix B is a list of the event and action blocks in both basic and advanced modes.

Appendix C provides guidance for teachers and mentors of students. The first section suggests ways to encourage exploration and experimentation. The next section focuses on good programming practices. The final section lists some pitfalls that may be encountered and offers hints on how to overcome them.

Appendix D describes techniques for working with the sliders of the sensor and motor blocks.



Reference cards

You will find it useful to print out one or both of the VPL reference cards, which are in the same zip file as this document and also available at <https://www.thymio.org/en:visualprogramming>.

- A single page that summarizes the event and action blocks.
- A two-sided page that can be folded to form a handy card. It summarizes the VPL interface, the event and action blocks, and includes example programs.

Installing Aseba

To install Aseba, including VPL, go to <https://www.thymio.org/en:start> and click on the icon for your system (Windows, Mac OS, etc.). Following the instructions to download and install the software. The Aseba installation includes both the VPL and the Studio (see Chapter 21) development environments.

VPL Tutorial Version History

Version 1.5

- Events are now implemented using the AESL programming language structure when instead of `if`. This means that events will occur only when the event initially occurs and not repeatedly, as explained on page 74. This change might cause unexpected behavior in some programs described in this tutorial.
- Dynamic feedback of the execution is implemented (page 19).

Version 1.4

- The graphic design of the buttons for the blocks has been changed, primarily to support additional features.
- In 1.3, a *red* box in the event block for a horizontal sensor caused an event when the sensor detected an object, while a *white* box caused an event when there was no object in front of the sensor. In 1.4, a *white* box causes an event when a lot of reflected light is detected from an object, while a *black* box causes an event when little or no reflected light is detected because there is no object in front of the sensor (pages 21, 23). The ground sensors also use white and black, instead of white and red, but the behavior in 1.4 is the same as in 1.3 except that black is used instead of red.
- In advanced mode, the thresholds of sensors can be set (page 92).
- In advanced mode, an event can be associated with ranges of values of the forward/backward and left/right accelerometers (page 47).
- Multiple actions associated with an event (page 19).
- Blocks and event-actions pairs can be copied (page 16).
- Screenshots of VPL programs can be exported in several graphics formats (page 85).
- Undo/Redo buttons have been added (page 84).
- The Run button blinks green when the program has been changed (page 16).
- It is no longer possible to change the color scheme of VPL.

Part I

Tutorial

Chapter 1

Your First Robotics Project


Getting to know your Thymio

Figure 1.1 shows the front and top of Thymio. On the top you can see the center circular button (A) and four directional buttons (B). Behind the buttons, the green light (C) shows how much charge remains in the battery. At the back are the top lights (D), which have been set to red in this picture. There are similar lights on the bottom (see Figure 3.1). The small black rectangles (E) are sensors which you will learn about in Chapter 4.



Figure 1.1: The top and front of the Thymio robot

Connect the robot and run VPL

Connect your Thymio robot to your computer with a USB cable; the robot will play a sequence of tones. If the robot is turned off, turn it on by touching the center button for a few seconds until you hear the tones. Run VPL by double-clicking on the icon  on your computer.



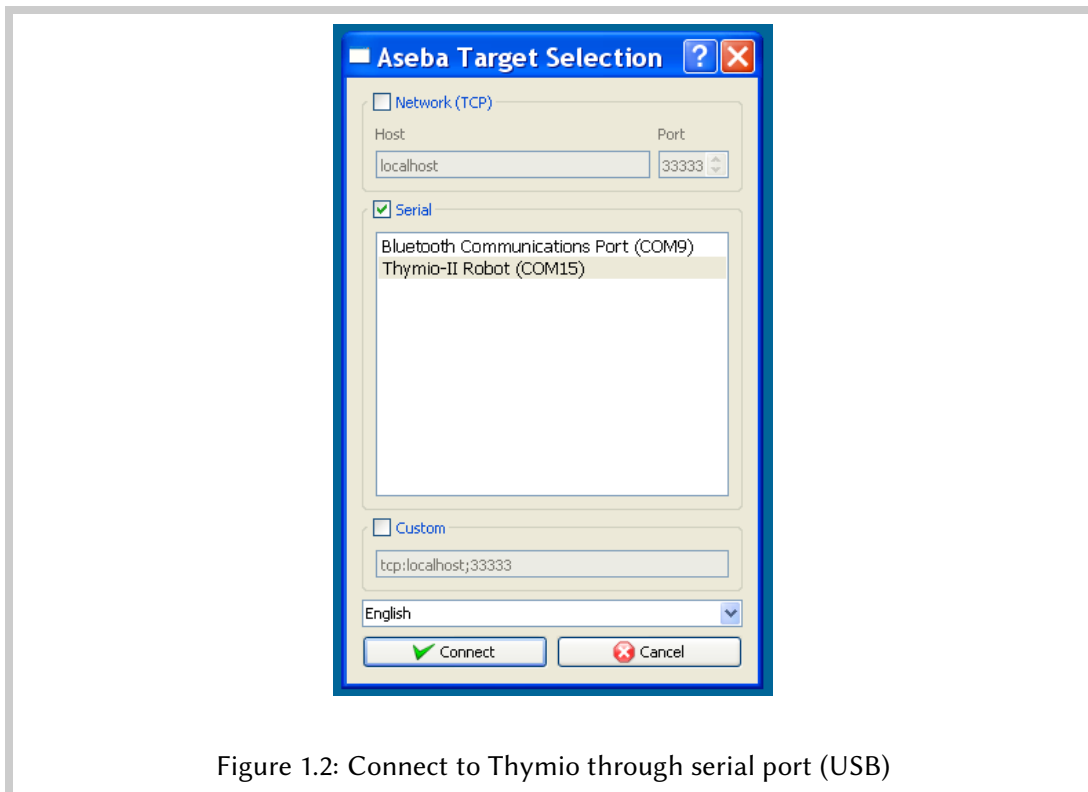


Figure 1.2: Connect to Thymio through serial port (USB)

Small images

When a small image appears in the text, a larger image is displayed in the margin.

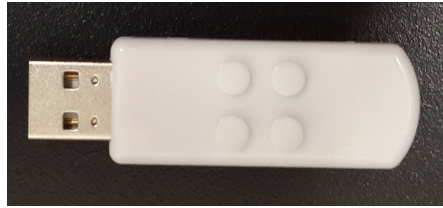
VPL may connect automatically to your robot. If not, the window shown in Figure 1.2 will be displayed. Check the box next to **Serial**, click on **Thymio Robot** below it, select a language, and then click **Connect**. Depending on the configuration of your computer and operating system, there may be several entries in this table and the data following **Thymio Robot** may be different from what is shown in the Figure.

Trick

It is also possible to access VPL from Aseba Studio, the text-based programming environment through the VPL plugin found in the *Tool* area at the bottom left of the screen.

Wireless Thymio

The Thymio robot comes in a wireless version that does not need to be connected to a computer in order to load programs. The wireless robot comes with a small object called a *dongle*:



Plug the dongle into a USB socket on your computer, turn the robot on and start VPL, as described above. When you see the connection window (Figure 1.2) select the line **Thymio-II Wireless (COM15)**.

Important information

You need to use the USB cable to charge the Thymio (Figure 1.4).

The VPL user interface

The user interface of VPL is shown in Figure 1.3. There are six areas in the interface:

1. A toolbar with buttons for opening, saving, running a program, etc.
2. A program area where programs for controlling the robot are constructed.
3. A message area which displays error messages if the program is not well-formed.
4. A column with event blocks for constructing your program.
5. A column with action blocks for constructing your program.
6. The translation of the program into AESL, the textual language of Aseba.

The VPL toolbar

Appendix A contains a description of all the buttons in the VPL toolbar. Look at it occasionally until you have learned how to use them.

To go further

When you construct a program using VPL, the translation of the program into the textual programming language AESL appears in the right part of the window. It is the AESL program that is actually run by the robot. Chapter 21 explains these translations. The webpage <https://www.thymio.org/en:asebausermanual> contains learning and reference materials on AESL and its Studio environment.

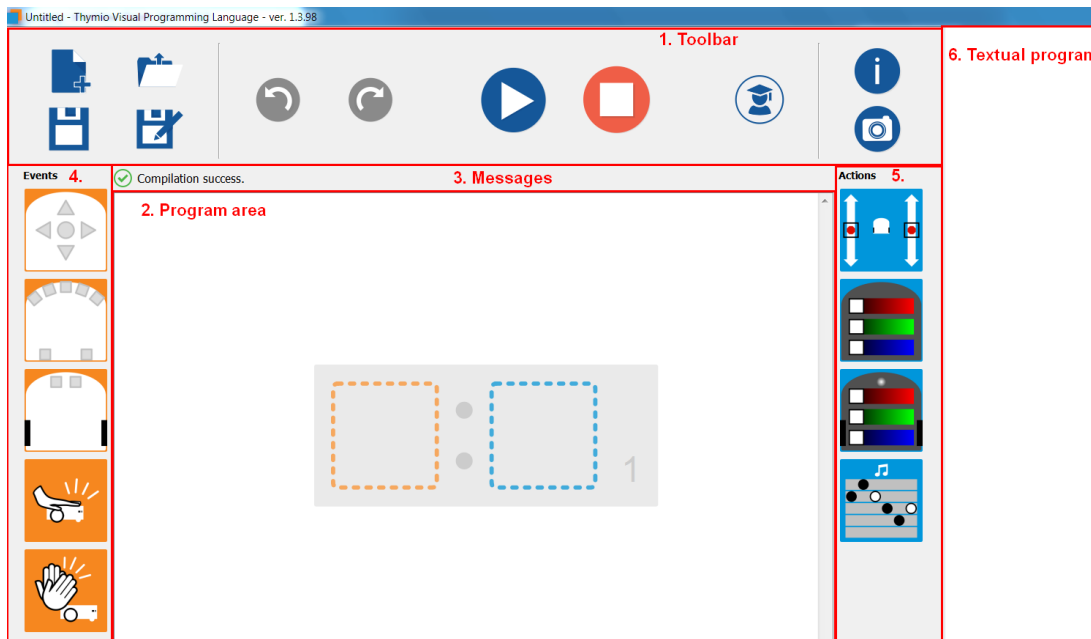



Figure 1.3: The VPL window

Write a program

When you start VPL, a blank program area is displayed.

If, after having built a piece of program, you wish to clear the content of the program area, click  (New).

A program in VPL consists of *event-actions pairs*, each constructed from an event block and one or more action blocks. For example, the pair:

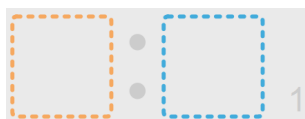


causes the top light of the robot to display red when the front button is touched.

Meaning of an event-actions pair

When the event occurs, the associated actions are run.



The program area will initially contain an empty frame for an event-actions pair:



To bring a block to the program area from the columns (areas 4 and 5 of Figure 1.3), press and hold the left mouse button and drag the block to a dashed square. When the block is over the square, release the mouse button, dropping the block into its place.

Important information

The technique just described is called *drag-and-drop* and is widely used in the user interface of programs.

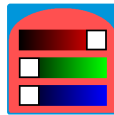
Start by bringing the button event block  into the left side of the empty frame. You will get a message inviting you to add an action block. Drag the top color action block  and drop it into the right side of the frame. You have constructed an event-actions pair!

Next, we have to modify the event and the action to do what we want. For the event, click on the front button (the top triangle); it will turn red:




This specifies that an event will occur when the *front button* of Thymio is touched.

The color action block contains three *sliders*—colored bars with a white square—one for each of the primary colors red, green, blue. Drag a white square to the right and then back to the left, and you will see that the background color of the block changes. All colors can be made by mixing these three primary colors: red, green and blue. Move the red slider until the square is at the far right, and move the green and blue sliders until they are at the far left. The color will be all red with no blue nor green:




Save the program

Before running the program, save it on your computer. Click on the button  (**Save**) in the toolbar. You will be asked to give the program a name; choose a name that will help you remember what the program does, perhaps, **display-red**. Choose the location where you want to save your program and click on **Save**.

Save frequently

When you modify a program, click **Save** frequently so that you don't lose your work if something happens to the computer.

Run the program

To run the program, click on  (**Run**) in the toolbar. Touch the front button on the robot; the light on top of the robot should change to red.

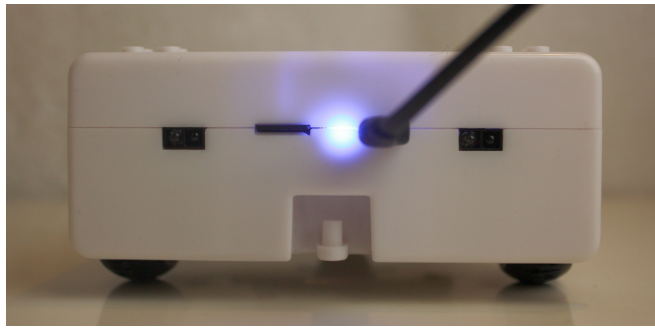



Figure 1.4: The back of the Thymio showing the USB cable and the charging light

★ **Congratulations!**

You have created and run your first program. Its behavior is:


When you touch the forward button of the Thymio, it becomes red.

If you need to stop the VPL program, click  (**Stop**). This is important when you run a program that causes the robot to move, but the program does not have an event-actions pair to stop the motors.



Turn the robot off


When you have finished working with the Thymio robot, turn it off by touching and holding the center button for a few seconds until you hear a sequence of tones. The battery will continue charging as long as it is connected to a working computer. A red light next to the USB cable connector means that the robot is charging; it turns blue when the charging is completed (Figure 1.4). You can disconnect the cable when you are not using the robot.

 **Trick**


To charge the robot faster, use a mobile phone charger with a micro-USB connector.

Should the USB cable disconnect during programming, VPL will wait for the connection to be made again. Check both ends of the cable, reconnect and see if VPL is working. If you have a problem, you can always close VPL, reconnect the robot and open VPL again.

Modify a program

- To delete an event-actions pair, click  at the top-right of the pair.




- To add an event-actions pair, click  available below an existing pair.
- To move an event-actions pair to another position in the program, drag and drop it at the desired location.
- To copy an event-actions pair to another position in the program, press and hold the **Ctrl** and then use the mouse to drag and drop the pair at the desired location.¹




★ The blinking Run button

When you modify a program, the **Run** button blinks blue and green to remind you that you need to click the button to load the modified program into the Thymio robot.

If you want to experiment with a modification but not lose an existing program, you can create a copy of the existing program by clicking  (**Save as**) and giving a new file name.



Open an existing program

Suppose that you have saved your program and turned off the robot and the computer, but later you wish to continue to work on the program. Connect the robot and run VPL as described previously. Click on  (**Open**) and select the program you want to open, for example, **display-red**. The event-actions pairs of the program will be displayed in the program area, and you can continue working on it.



The current event-actions pair

When you click on an event-actions pair, it will be displayed with a yellow background. This will also occur when you enter an event or action block in an empty pair:



The left gold-colored square is the space for the event; the right blue-colored square is for the first (or only) action. The pair with the yellow background is called the *current* pair.

★ Quick entering of a block

If you click on an event or action block, it will be automatically placed in the program area in the current event-actions pair.

¹On Mac OS, the **Command** button is used instead of **ctrl**.



Chapter 2

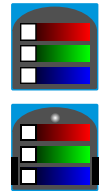
Changing Colors

Display colors

Create a program that causes two different colors to be displayed on the top of the Thymio robot when the front and back buttons are touched, and two other colors to be displayed on the bottom of the robot when the left and right buttons are touched.


Program file **colors.aesl**

We need four event-actions pairs. There are four events—touching the four buttons—and a color action is associated with each event. Note the difference between the action blocks  and . The first block changes the color displayed on the top of the robot, while the second changes the color on the bottom of the robot. The block for the bottom light has two black marks that represent the wheels and a white dot representing the support at the front of the robot (Figure 3.1).



The program is shown in Figure 2.1.

What colors are displayed? In the first three actions, the slider for one color is moved to the right edge, while the sliders for the other two colors are moved to the left edge. The colors are not mixed so these actions display pure red, blue and green, respectively. The action associated with the left button mixes red and green giving yellow. You can see that the background of the block changes when the sliders are moved and shows which color the robot will display.

Run the program  and touch the buttons to change the robot's colors. Figure 1.1 shows the Thymio displaying red on the top and Figure 3.1 shows it displaying green on the bottom.



Exercise 2.1

Experiment with the sliders to see which colors can be displayed.



Information

By mixing together red, green and blue, you can make any color (Figure 2.2)!



Figure 2.1: Changing colors when a button is touched

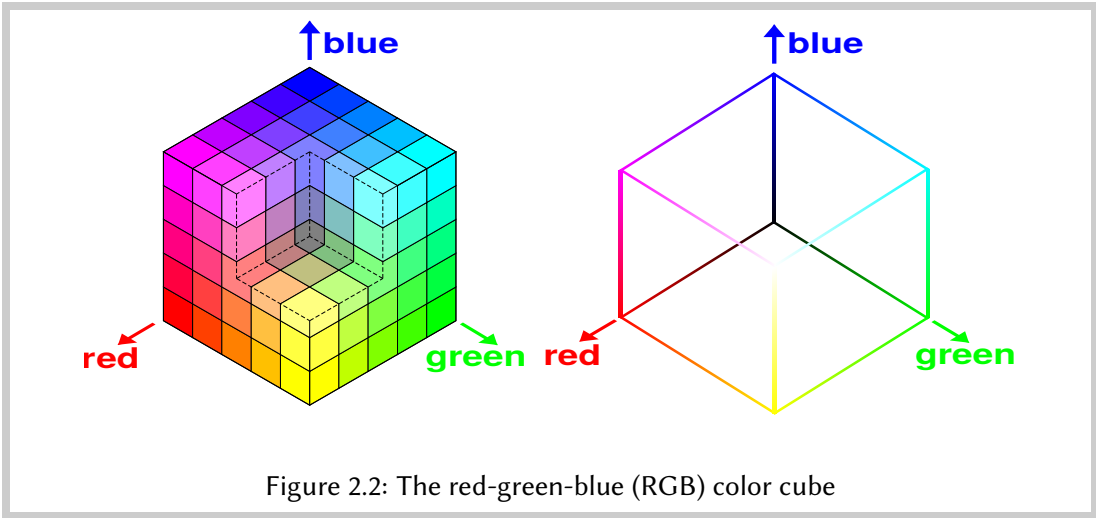
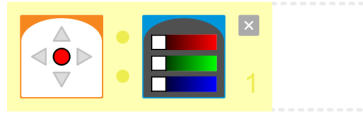


Figure 2.2: The red-green-blue (RGB) color cube

Multiple actions associated with one event


Let us modify the program so that the lights are turned off when the center button is touched. We need two actions to occur when a single event—touching the center button—occurs. We can associate *two* actions with one event in an event-actions pair. After inserting the event and the first action (the top color action), a gray outline will appear to the right of the action:



You can now drag and drop the bottom color action into this outline, giving a pair with one event and two actions:



Program file **colors-multiple.aesl**

Don't forget to click  to run the program. In the future, we will not remind you to click this button to run a program.

Rules for event-actions pairs

- When a program is run, all the event-actions pairs in the program are run.
- It is possible for several event-actions pairs to have the same event as long as their parameters are different. For example, you can have several pairs with the button event, if different sets of buttons are required for different events.
- If the event is exactly the same in two or more pairs, VPL will display an error message (in area 3 in Figure 1.3). You will not be able to run the program as long as there are errors.

Dynamic feedback

Whenever a button is touched, an event is generated and the event-actions pair associated with that event is run. VPL provides dynamic feedback so that you can see exactly which pair is run. The pair is emphasized with a yellow frame and a yellow arrow to its left:




The feedback will appear briefly when the event occurs and then it is removed. For example, if you touch a button, an event is generated when the button is first touched. If you continue to touch the button, no additional events are generated, so the feedback will be removed. It will re-appear only if you release the button and then touch it again.

Chapter 3

Let's Get Moving

Move forwards and backwards

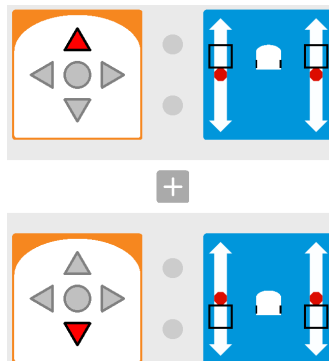
The Thymio robot has two motors, one connected to each wheel. The motors can be run forwards and backwards, causing the robot to move forwards and backwards, and to make turns. Let us start with a simple project to learn about the motors.

The motor action block  displays a small image of the robot in the center together with two sliders. The sliders control the speed of the motors, one slider for the left motor and one for the right motor. When the black frame is centered on the red dot in the slider, the corresponding motor is off. You can drag the frame up above the red dot to increase the forward speed and down below the red dot to increase the backwards speed. Let us write a program to move the robot forward when the front button is touched and backwards when the back button is touched.



Program file **moving.aesl**

We need two event-actions pairs:



Drag and drop the event and action blocks and set the sliders equally for the left and right motors, half-way up for forward and half-way down for backwards.

Run the program and touch the buttons to make the robot go forwards and backwards.

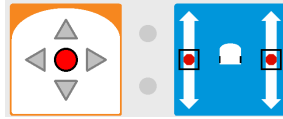
Stop the robot

Help! I can't stop the robot's motors!

Click on the button  to stop the robot.



Let us fix this problem by adding an event-actions pair that will stop the motors when the center button is touched:



When you drag the motor action block into the program area, it is already set with the sliders in the middle to turn off the motors.

Don't fall off the table

If your robot moves on the floor, at worst it might hit a wall, but if you place your robot on a table, it might fall off, crash and break! Let us arrange for the robot to stop when it reaches the end of a table.


Warning!

Whenever the robot moves on a table, be ready to catch it in case it does fall off.

Turn the Thymio on its back. You will see at the front two small black rectangles with optical elements inside; they are displayed at the top of Figure 3.1. These are the *ground sensors*. They send a pulse of infrared light and measure the amount of light that is reflected. On a light-colored table, there is a lot of reflected light, but when the front of the robot goes past the end of the table, there will be much less reflected light. When this is detected we want the robot to stop.

Trick

Use a table colored with a light color or tape a sheet of white paper on the table. Don't use a glass table, as it will likely not reflect the light and Thymio will believe that it is not on a table!

Drag and drop the ground sensor event  into the program. There are two small squares at the top of the block. Clicking the squares changes them from gray to red to black and finally back to gray. For this block, the meanings of these colors are:



- **Gray:** The sensor is not used.
- **White:** An event occurs when there is a lot of reflected light. Next to the white square a small red dot will be displayed; this dot corresponds to the small red light next to each sensor that is turned on when the sensor detects something.¹

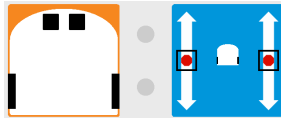
¹The white square has a red border to remind you that the event will occur when the lights next to the sensor itself are red.



Figure 3.1: The bottom of the Thymio with two ground sensors at the front

- **Black:** An event occurs when there is little reflected light.

To cause the robot to stop at the border of the table (when there is little reflected light), click both squares until they are black. Create the following event-actions pair:



Place the robot near the edge of the table, facing the edge, and touch the front button. The robot should move forward and stop before falling off the table.

Exercise 3.1

Experiment with the speed of the robot. At maximum speed, is the robot still able to stop and not fall off the table? If not, at what speed does the robot start to fall off? Can you stop the robot from falling off when it is going backwards?

Warning!

When I ran the program, the robot *did* fall off. The reason was that my desk has a rounded edge; by the time that the robot detected a low level of reflected light, it was no longer stable and tipped over. My solution was to place a strip of black tape close to the edge of the desk.

Chapter 4

A Pet Robot


Autonomous robots display independent behavior that is normally associated with living things like cats and dogs. The behavior is achieved by *feedback*: the robot will sense that something occurs in the world and modify its behavior accordingly.

The robot obeys you

We will program the robot to obey: the robot stays in place without moving, but when it detects your hand in front of it, it moves towards your hand.

Program file **obeys.aes1**

There are five horizontal distance sensors on the front of the Thymio robot and two on the rear. They are similar to the ones under Thymio that we used in Chapter 3. Bring your hand slowly towards the sensors; when it gets close, red lights will appear around the sensors that detect your hands (Figure 4.1).

The block  is used to sense if something is close to a sensor or not. In either case it causes an event to occur. The small squares in the block (five on the front and two on the rear) are used to specify when an event occurs. Clicking on a square changes it from gray to white to black and back to gray. The meaning of these colors is:



- **Gray**: The sensor is not used.
- **White**: An event occurs when there is a lot of reflected light. The white square has a red border to remind you that the event will occur when the lights next to the sensor itself are red.
- **Black**: An event occurs when there is little reflected light.



Figure 4.1: The front of the Thymio. Two sensors detect the fingers.

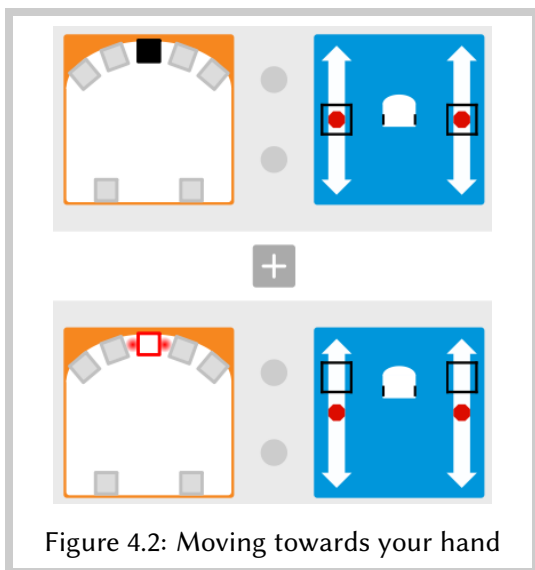


Figure 4.2: Moving towards your hand

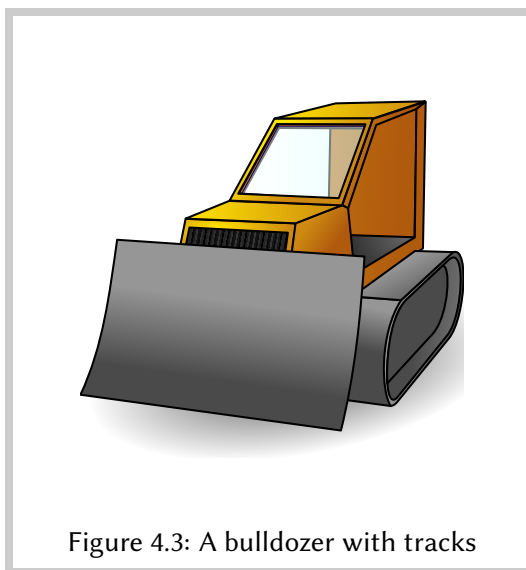



Figure 4.3: A bulldozer with tracks

If you wish an event to occur when an object is close to the sensor, use a white square because the object will reflect a lot of light. If you wish an event to occur when no object is close to the sensor, use a black square because little light will be reflected.

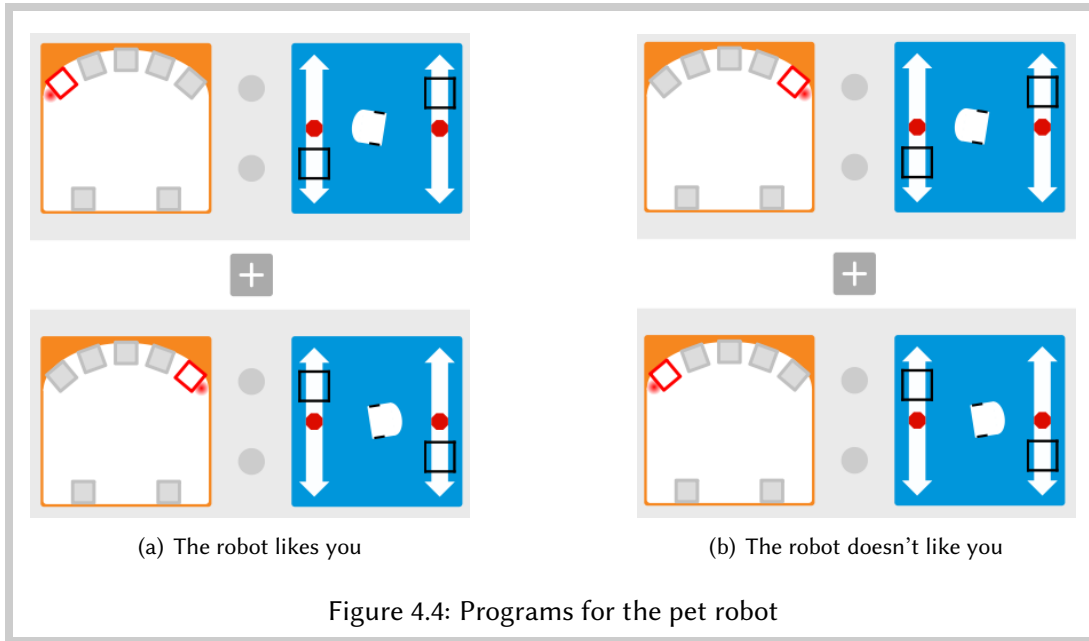
To implement the required behavior, we need two event-actions pairs (Figure 4.2). In the first pair, the center front sensor is black and the associated action is that the motors are off. Therefore, when the robot does not detect an object, it will not move, and it will stop if it had been moving. In the second pair, the center front sensor is white and the sliders of the motor block are at the top. Therefore, when you bring your hand near the front of the robot, an event occurs that causes both motors to run quite fast and the robot to move forward.

Steering the Thymio robot

The Thymio robot does not have a steering wheel like a car or a handlebar like a bicycle. So how can it turn? The robot uses *differential drive*, which is familiar from tracked vehicles like the bulldozer (Figure 4.3). Instead of turning a handlebar a desired direction, the left and right tracks or wheels are driven by individual motors at *different* speeds. If the right track moves faster than the left one, the vehicle turns left, and if the left track moves faster than the right one, the vehicle turns right.


Differential drive for the Thymio robot is implemented by setting the left and right sliders of a motor action block—and therefore the wheel speeds—to different values. The greater the difference between the speeds, the tighter the turn. To achieve a large difference of speeds, drive one track forward and one track backwards. In fact, if one track moves forward at a certain speed, while the other track moves backwards at the same speed, the Thymio turns in place. For example, in the motor action block , the left slider has been set for fast speed backwards, while the right slider has been set for fast speed forwards. The result is that the robot will turn to the left.





Experiment with an event-actions pair such as:



Set the left and right sliders, run the program and touch the center button; to stop the robot click on . Now you can change the sliders and try again.



Trick

The small image of Thymio in the center of the motor action block shows an animation of the movement of the robot when you move the sliders. When the animation stops, the image shows the direction in which the robot will move when this action block is run.

The robot likes you

A real pet follows you around. To make the robot follow your hand, add two additional event-actions pairs: if the robot detects an object in front of its left-most sensor, it turns to the left, while if it detects an object in front of its right-most sensor, it turns to the right.

Program file **likes.aesl**

The program consists of two event-actions pairs (Figure 4.4(a)). Experiment with the sliders on the motor action blocks.

Exercise 4.1

Modify the behaviour of the robot so that it moves forward when the program is run and stops when it detects the edge of a table (or a strip of black tape).

Exercise 4.2

What happens if you change the order of the event-actions pairs that you used in the previous exercise?

The robot doesn't like you

Sometimes your pet may be in a bad mood and turn away from your hand. Write a program that causes this behavior in the robot.

Program file **does-not-like.aesl**

Open the program for the pet that likes you and exchange the association of the events with the actions. Detection of an object by the left sensor causes the robot to turn right, while detection of an object by the right sensor causes the robot to turn left (Figure 4.4(b)).

Exercise 4.3

The front horizontal sensors are numbered 0, 1, 2, 3, 4 from the left of the robot to its right. The rear sensors are numbered 5 for the left one and 6 for the right one. Modify the programs in Figure 4.4 so that instead of using sensors 0 and 4:

- Use sensors 1 and 3 to turn the robot left and right, respectively.
- Use both sensors 0 and 1 to turn the robot left and both sensors 3 and 4 to turn the robot right.
- Add event-actions pairs for the rear sensors 5 and 6.

Trick

Appendix D explains how to set the sliders to precise motor speeds.

Sensors in advanced mode

In advanced mode (Chapter 7), there is a fourth mode for specifying when sensors cause events, in addition to modes indicated by gray, white and black. See Appendix D.

Chapter 5



The Robot Finds Its Way by Itself

Consider a warehouse with robotic carts that bring objects to a central dispatching area. There are lines painted on the floor of the warehouse and the robot receives instructions to follow certain lines until it reaches the storage bin of the desired object. Let us write a program that causes the robot to follow a line on the floor.

Program file **follow-line.aesl**

The line-following task brings out all the uncertainty of constructing robots in the real world. The line might not be perfectly straight, dust may obscure part of the line, or dirt may cause one wheel to move more slowly than the other one. To follow a line, the robot must use a *controller* that decides how much power to apply to each motor depending on the data received from the sensors.

The line and the robot

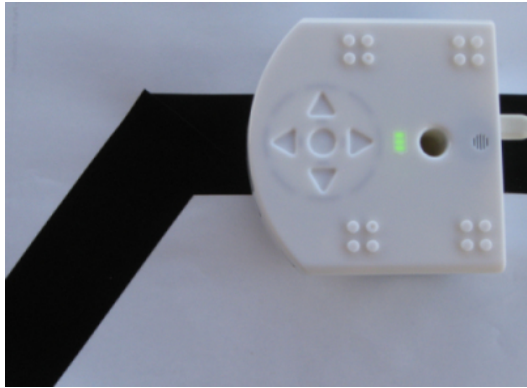
To follow a line, we use the ground sensors (Chapter 3). Remember that these work by sending infrared light (which is invisible to human eye) and measuring how much is reflected back. If the floor is light-colored, the sensor will detect a lot of reflected light and the event  will occur. We need a line that will cause an event to occur when there is little reflected light . This is easy to do by printing a black line on paper and taping it to the floor or by attaching black electrician's tape on the floor (Figure 5.1(a)). The line must be wide enough so that both ground sensors will sense black when the robot is successfully following the line. A width of 5 centimeters is sufficient for the robot to follow the line even if there are small deviations.



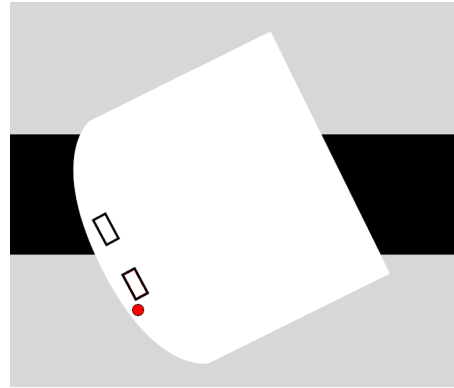
To implement line-following, first, we cause the robot to move forward whenever *both* sensors detect a dark surface—it is on the line—and to stop whenever *both* sensors detect a light surface—it is not on the line. See Figure 5.2(a).

Trick

Make sure that you use a USB cable that is long enough (say, two meters), so that the Thymio can stay connected to the computer even as it moves. You can find extension cables in any computer shop.

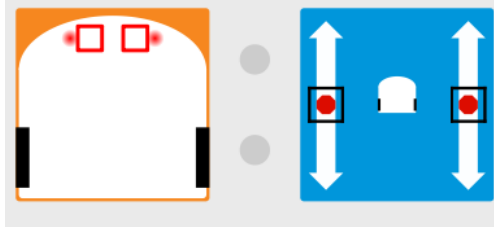


(a) Thymio following a line of tape

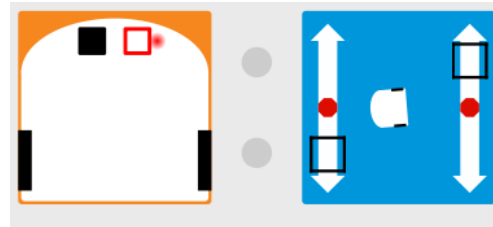


(b) The left sensor is off the tape and the right sensor is on the tape. The red dot indicates that the left sensor detects a lot of reflected light

Figure 5.1: Thymio on a black tape



(a) Start and stop the robot



(b) Correcting deviations

Figure 5.2: A program for line following

★ Wireless Thymio

If you have a wireless Thymio it can move freely without being limited by the length of the cable.

Your first controller

The next step is to program the controller that follows the line. Two event-actions pairs are needed (Figure 5.2(b)).

- If the robot moves off the tape to the *left* (Figure 5.1(b)), the *left* sensor will detect the floor while the *right* sensor is still detecting the tape; the robot must turn slightly to the *right*.
- If the robot moves off the tape to the *right*, the *right* sensor will detect the floor while the *left* sensor is still detecting the tape; the robot must turn slightly to the *left*.

Setting the parameters

It is easy to see that if the robot runs off the left edge of the tape, it has to turn to the right (Figure 5.1(b)). The question is how tight should the turn be? If the turn is too gentle, the right sensor might *also* run off the tape before the robot turns back; if the turn is too sharp, it might cause the robot to run off the other end of the tape. In any case, sharp turns can be dangerous to the robot and cause whatever it is carrying to fall off.

You will need to experiment with the speeds of the left and right motors in each motor action block until the robot runs *reliably*. Here, reliably means that the robot can successfully follow the line several times. Since each time you place the robot on the line you might place it at a slightly different position and point it in a slightly different direction, you need to run several tests to make sure that the program works.

The forward speed of the robot on the line is also an important parameter. If it is too fast, the robot can run off the line before the turning actions can affect its direction. If the speed is too slow, no one will buy your robot to use in a warehouse.

Exercise 5.1

The robot stops when both ground sensors detect that they are off the tape. Modify the program so that the robot makes a gentle left turn in an attempt to find the tape again. Try it on a tape with a left turn like the one shown in Figure 5.1(a). Try increasing the forward speed of the robot. What happens when the robot gets to the end of the tape?

Exercise 5.2

Modify the program from the previous exercise so that the robot turns right when it runs off the tape. What happens?

It would be nice if we could *remember* which sensor was the last one to lose contact with the tape in order to cause the robot turn in the correct direction to find the tape again. In Chapter 8 we will learn how Thymio can remember information.

Exercise 5.3

Experiment with different arrangements of the lines of tapes:

- Gentle turns;
- Sharp turns;
- Zigzagging lines;
- Wider lines;
- Narrow lines.

Run competitions with your friends: Whose robot successfully follows the most lines? For each line, whose robot follows it in the shortest time?

Exercise 5.4

Discuss what effect the following modifications to the Thymio would have on the ability of the robot to follow a line:

- Ground sensing events occur more often or less often.
- The sensors are further apart or closer together.
- There are more than two ground sensors on the bottom of the robot.

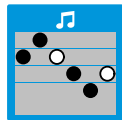
Chapter 6

Bells and Whistles

Let's have some fun with the robot. We show how the Thymio can play music, respond to a sound or react when it is tapped.

Playing music

The Thymio robot contains a sound synthesizer and you can program it to play simple tunes using the music action block:



Program file **bells.aesl**

You won't become a new Beethoven—you can only play six notes using five tones of two different lengths—but you can compose a short tune. Figure 6.1 shows two event-actions pairs that causes a tune to be played when the front or the back button is touched. A different tune associated with each event: twice long-short-rest or twice short-long-rest.

The six small circles are notes. A black circle is a short note, a white circle is a long note and a blank is a rest. To change from one length to the other, click on the circle. There are five gray horizontal bars, representing five tones. To move a circle to one of bars, click on the bar above or below the circle, or drag-and-drop the note to a bar.

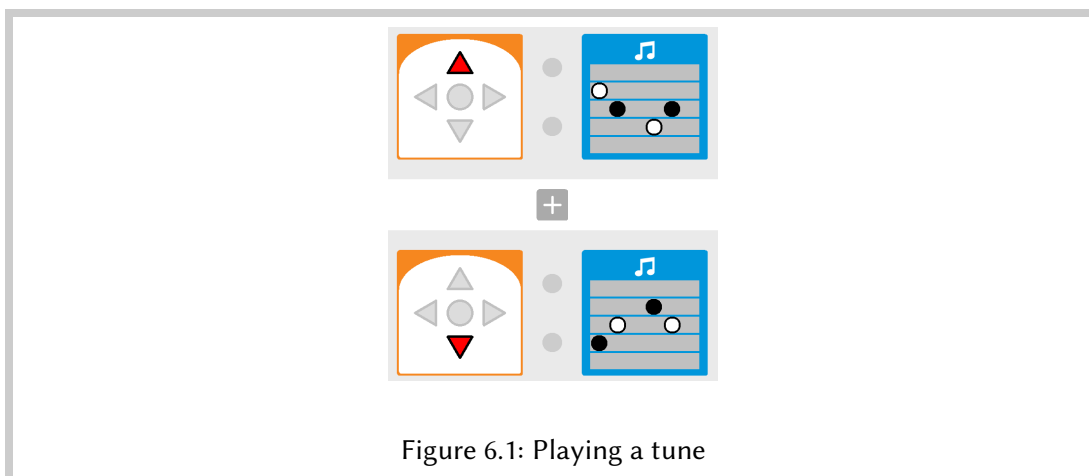



Figure 6.1: Playing a tune

Exercise 6.1

Write a program that will enable you to send a message in **Morse code**. Letters in Morse code are encoded in sequences of long tones (*dashes*) and short tones (*dots*). For example, the letter *V* is encoded by three dots followed by one dash.

Controlling your robot by sound

The Thymio has a microphone. The event  occurs when the microphone senses a loud noise, for example, from clapping your hands. The following event-actions pair will turn on the bottom lights when you clap your hands:



Information


In a noisy environment you may not be able to use this event, because the sound level will always be high and cause repeated events.

Exercise 6.2

Write a program that causes the robot to move when you clap your hands and to stop when you touch a button.

Write a program that does the opposite: the robot starts moving when you touch a button and stops when you clap your hands.

Good job, robot

Pets don't always do what we ask them to do. Sometimes they need a pat on the head to encourage them. You can do the same with your robot. The Thymio contains a tap sensor that causes the event  to occur in response to a quick tap on the top of the robot. The following event-actions pair causes the top lights to turn on when you tap the top of the robot:



Construct a program from this event-actions pair and the following pair that turns on the bottom lights when you clap your hands:



Program file **whistles.aesl**

Can you turn on just the top lights? This is difficult to do: a tap causes a sound that can be loud enough to cause the bottom lights to be turned on as well. With a little practice I was able to tap the robot gently enough so that the sound of the tap was not considered an event.

Exercise 6.3

Write a program that causes the robot to move forward until it hits a wall.

Make sure that the robot **moves slowly** so that it doesn't damage itself.

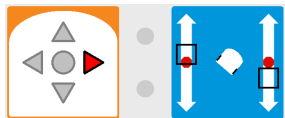
Chapter 7

A Time to Like (Advanced Mode)

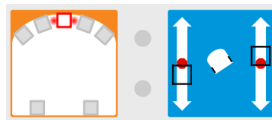
In Chapter 4 we programmed a pet robot which either did or did not like us. Let us consider a more advanced behavior: a shy pet who can't make up its mind whether it likes us or dislikes us. Initially, the pet will turn towards our outreached hand, but then it will turn away. After a while, it will reconsider and turn back in the direction of our hand.

Program file **shy.aes1**

When the right button is touched the robot turns to the right:




When it detects your hand, it turns to the left:



The behavior of turning back “after a while” can be divided into two parts:

- *When* the robot starts to turn away → *start a timer* for two seconds.
- *When* the timer runs down to zero → *turn* to the right.


We need a new *action* for the first part and a new *event* for the second part.


The action to set a *timer* is like an alarm clock . Normally, we set an alarm clock to an absolute time, but when I set the alarm clock in my smartphone to an absolute time like 07:00, it tells me the relative time: “Alarm set for 11 hours and 23 minutes from now.” You can set the timer action block for a certain number of seconds; when the timer has *expired*—that is, when the number of seconds has passed from the setting of the timer—a timer *event* occurs.



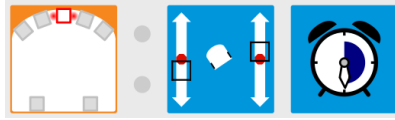
The timer can be set for up to four seconds, where each second is represented by one quarter of the clock face. Click anywhere within the white circle; there will be a short animation and then the appropriate part of the clock face will be colored dark blue.

★ **Advanced mode**


Timers are supported in *advanced mode*. Click on  to enter advanced mode.

The icon will change to  and you can click on it to change back to *basic mode*.

The event-actions pair for this first part of the behavior is:



When the event of detecting your hand occurs, there will be two actions: turning the robot to the left and setting the timer to two seconds.

The second part of the behavior uses a ringing-alarm-clock timer event  that occurs when the amount of time set on the timer expires.



Here is the event-actions pair to turn the robot to the right when the timer expires:



Exercise 7.1

Write a program that causes the robot to move forward at top speed for three seconds when the forward button is touched; then it runs backwards. Add an event-actions pair to stop the robot by touching the center button.

Chapter 8

States (Advanced Mode)

A program in VPL is a list of event-actions pairs. *All* the events are checked periodically and the appropriate actions are taken. This limits the programs that we can create. To develop more complex programs, we need a way to specify that some event-actions pairs are active, while others are not.

For example, in the line-following program in Chapter 5, when the robot runs off the tape, we want it to turn left or right to search for the tape with the direction depending on which side it ran off. There will be two event-actions pairs: one to turn left when the robot runs off the right of the tape and one to turn right when it runs off the left of the tape.

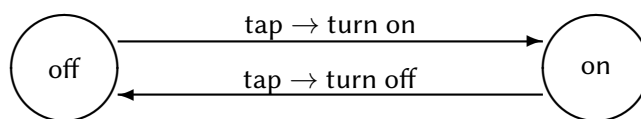
Tap, tap

In many programs, we used one button to start the robot's behavior and another to stop it. Consider, though, the power switch on a computer. The same switch is used to turn the computer on and off; the computer *remembers* whether it is in the state **on** or the state **off**.

Write a program that turns the robot's lights on when it is tapped and turns them off when tapped again.

Program file **tap-on-off.aesl**

It is convenient to display the required behavior in a *state diagram*:

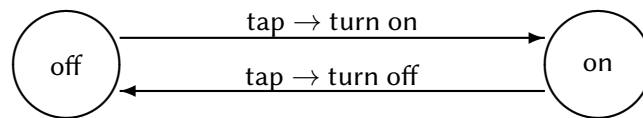


In the diagram there are two states indicated by circles labeled with the names of the states **off** and **on**. From state **off** the robot can go to state **on** and back, but only by following the instructions on the arrows. The instructions describe when a transition from one state to another can occur and what happens when it does occur:

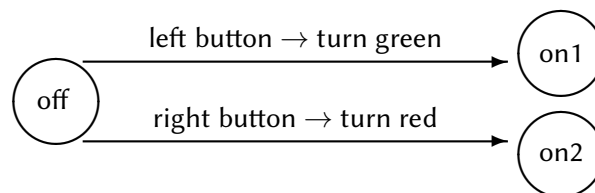
- When the robot is in state **off** *and* the *tap* event occurs → turn the lights *on* *and* go to state **on**.
- When the robot is in state **on** *and* the *tap* event occurs → turn the lights *off* *and* go to state **off**.

The emphasized word **and** before the arrow \rightarrow means that there are *two conditions* that must be true in order for the transition to be taken. (a) The robot must be in a certain state and (b) the event must occur. When both conditions are true, the transition is taken, causing both the state to change and the action written after the arrow \rightarrow to be performed.


It is important to realize that the two parts of the condition are independent. In the above diagram (repeated here), the event *tap* appears twice, but the action caused by the occurrence of this event *depends on which state the robot is in*.



In a single state, different events can cause different actions and different transitions. In the following diagram, touching the left button in the state **off** causes the green light to be turned on and a change to state **on1**, while touching the right button *in the same state* causes a different action, the red light is turned on, and a change to a different state, **on2**.




Implementing state diagrams with event-actions pairs

Figure 8.1 shows the implementation of the behavior described in the state machine above. The left circle in the block  is checked (and is displayed in red) to indicate that this is a block for the tap event.



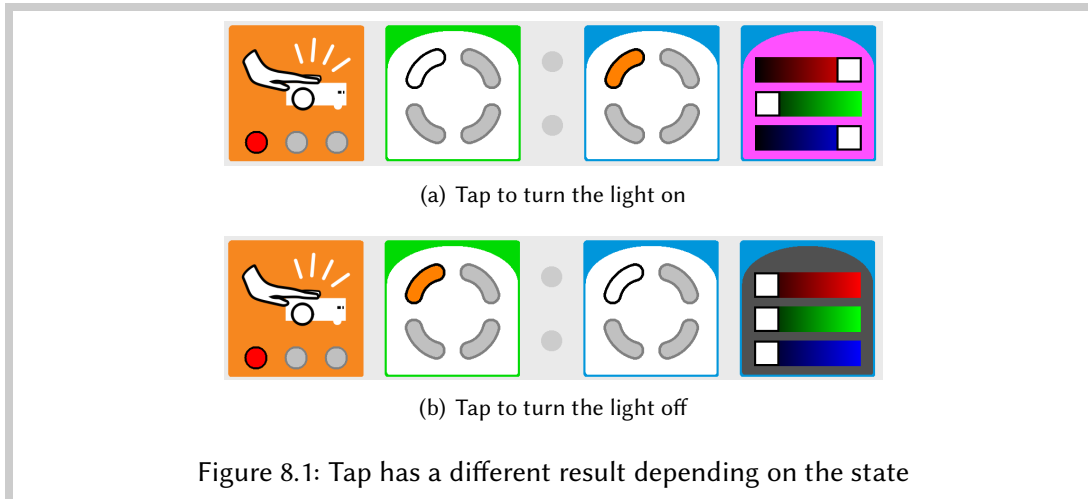
The tap block in advanced mode


The block for the tap event is different in advanced mode, because it is also used for accelerometer events as described in Chapter 10.

In the first event-actions pair (Figure 8.1(a)), the event is composed of the tap block together with an indication of the state . A state is indicated by four quarters of a circle, each of which can be either on (orange) or off (white). In this program, we will use the upper-left quarter to indicate whether the robot's top light is off or on. In Figure 8.1(a), this quarter is colored white, meaning that the robot's light is off. Therefore, the meaning of this pair is: **if** the robot is tapped **and** the light is off, **then** turn the light on.



For the second event-actions pair (Figure 8.1(b)), the quarter is colored orange, indicating that the robot's light is on. The meaning of pair is: **if** the robot is tapped **and** the light is on, **then** turn it off.



If you look again at the state diagram, you will see that only half the job is done. When turning the light on or off, we also have to change the state of the robot from **off** to **on** or from **on** to **off**. Therefore, we need to add a *state* action block  to each pair. This block changes the state as indicated by the quarters that are white or orange.



We can summarize the meaning of the program in Figure 8.1 as follows:

*When the robot is tapped and the state is **off**, change the state to **on** and turn the top light **on**.*



*When the robot is tapped and the state is **on**, change the state to **off** and turn the top light **off**.*

Each event causes both an action on the light and a change of the state of the robot. The actions depend on the *current* state of the robot.

How many states can the robot be in?

When used in an event state block or in the action state block, each quarter can be:

- **White:** the quarter is *off*;
- **Orange:** the quarter is *on*;
- **Gray:** the quarter is ignored.

For example, in , the upper-left and lower-right quarters are on, the upper-right one is off and the lower-left one is not taken into account, meaning that if  is associated with an event block, the event will occur if the state is either set to:



Since each of the four quarters can be either on or off, there are $2 \times 2 \times 2 \times 2 = 16$ states:

(off, off, off, off), (off, off, off, on), (off, off, on, off),
...
(on, on, off, on), (on, on, on, off), (on, on, on, on).

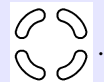
Figure 8.2(a) displays these 16 possible states.

Important information

The current state of the robot is displayed in the four diagonal segments of the light circle on the top of the robot. Figure 8.2(b) shows the robot in the state (on, on, on, on).

Information

When a program is run, the initial state is (off, off, off, off):



Trick

If you do not use all possible 16 states, but only 2 or 4, for example, you are free to decide which quarters you use to represent your state. In addition, if you have two different things you want to encode, and each of them has two possible values, you can use two quarters independently. That is why the ability to *ignore* a quarter is very useful!

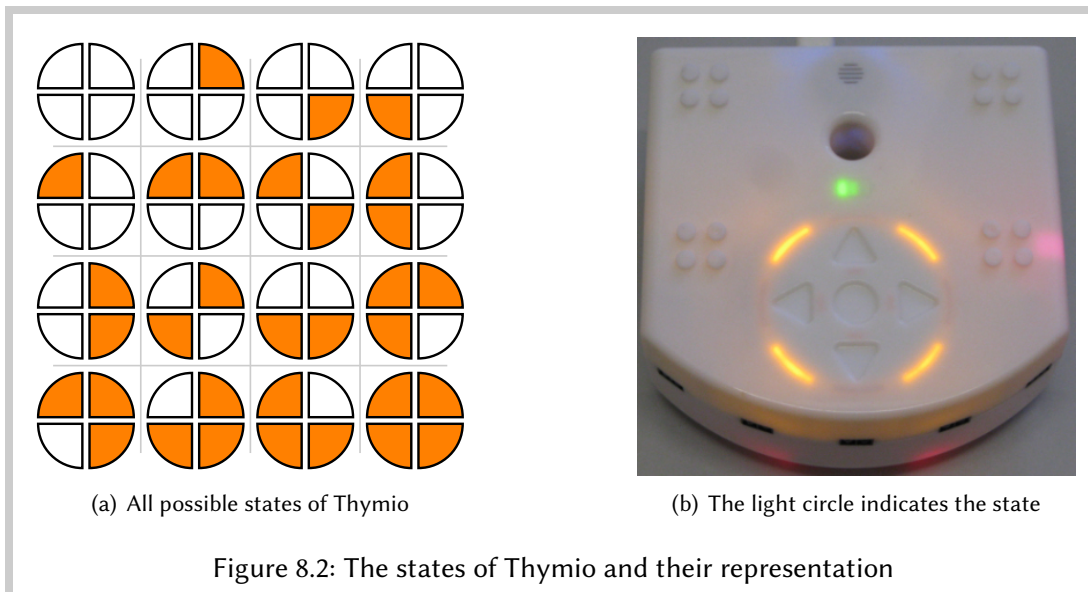
Get the mouse

Write a program to implement the behavior of a cat searching for a mouse: When the center button is touched, the robot turns counterclockwise (from right to left), searching for a mouse. If the robot detects a mouse with its rightmost sensor, it turns clockwise (from left to right) until the mouse is detected by its center sensor, at which point it stops (Figure 8.3).

Program file **mouse.aesl**

The following state diagram describes the behavior of the robot:





1. When the center button is touched, the robot enters the state **search left** and moves from right to left.
2. When the robot is in state **search left** and it detects the mouse in the rightmost sensor, it changes to state **search right** and moves from left to right.
3. When the robot is in state **search right** and it detects the mouse in the center sensor, it changes to state **found** and stops.

The important point to notice is that when the mouse is detected by the center sensor, it stops *only if* the robot is in state **search right**. Otherwise (if the mouse is detected by the center sensor when the robot is in state **search left**), nothing happens.

Let us now implement this behavior. We represent the state of the robot by the upper-left quarter of the state indicator. We choose white for the state **search left** and orange for the state **search right**. Since the program ends when the mouse is detected in state **search right**, we don't need to represent state **found** explicitly. Initially, all the quarters are **off** (white).

The following event-actions pair implements the behavior in step 1:



When the center button is touched, the state changes to **search left** and the robot turns to the left.

The event-actions pair that implements step 2 is:



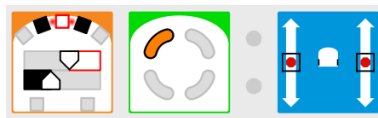
When the mouse is detected by the rightmost sensor while in state **search left**, the state changes to **search right** and the robot turns to the right.




Figure 8.3: The robot cat is looking for the mouse

The small square next to the rightmost sensor is set to black so that the event occurs only when the rightmost sensor alone detects the mouse.

Step 3 is implemented by the following event-actions pair:



When the mouse is detected by the center sensor while in state **search right**, the robot stops.

 **Trick**

You will have to experiment with the distance of the mouse to the robot. If it is too close to the robot, the sensors on either side of the central sensor will also detect the mouse, while the event requires that they *not* detect it.

 **Exercise 8.1**

Write a program that causes the robot to dance: it turns left in place for two seconds and then turns right in place for three seconds. These movements are repeated indefinitely.



Exercise 8.2 (Difficult)


Modify the line-following program from Chapter 5 so that the robot turns left when it leaves the right-hand side of the line and turns right when it leaves the left-hand side of the line.

Chapter 9

Counting (Advanced Mode)

In this chapter we show how states of the Thymio robot can be used to count numbers and even perform simple arithmetic.

The design and implementation of the projects will not be presented in detail. We assume that you have enough experience by now to develop them yourself. The source code of working programs is included in the archive, but don't look at them unless you really have difficulties solving a problem.

These projects use the clap event  to change states and the default behaviour of the circle lights to display the state. Feel free to change either of these behaviours.



Important information

By default, the current state of the robot is displayed in the circle lights on the top of the robot. Figure 8.2(b) shows the state (**on, on, on, on**).

Odd and even

Program

Choose one of the quarters of the state. It will be **off** (white) if the number of claps is even and **on** (orange) if the number of claps is odd. Touching the center button will reset the count to even (since zero is an even number).

Program file **count-to-two.aesl**

Even and odd are terms from *modulo 2 arithmetic*, where we count from 0 (even) to 1 (odd) and then back to 0. The term *modulo* is like the term *remainder*: if there have been 7 claps, then dividing 7 by 2 gives 3 and remainder 1. We only keep the remainder 1.

Another term for the same concept is *cyclic arithmetic*. Instead of counting from 0 to 1 and then from 1 to 2, we *cycle* back to the beginning: 0, 1, 0, 1,

These concepts are familiar from counting time: minutes and seconds are computed modulo 60 and hours are computed modulo 12 or 24. The second after 59 is not 60; instead, we cycle around and start counting from 0 again. Similarly, the hour after 23 is not 24, but 0. If the time is 23:00 and we agree to meet after 3 hours, the time set for the meeting is $(23+3) \text{ modulo } 24 = 26 \text{ modulo } 24 = 02:00$ in the morning.

Counting in unary

Modify the program to count modulo 4. There are four possible remainders, 0, 1, 2, 3. Choose three quarters, one each to represent the values 1, 2 and 3; the value 0 will be represented by setting all quarters to **off**.

This method of representing numbers is called *unary representation* because different elements of a state represent different numbers. We often use unary representation to keep track of the count of some objects; for example, |||| | represents 6.

Program file **count-to-four.aesl**



Exercise 9.1

How high can we count on the Thymio using unary representation?

Counting in binary

We are familiar with *based representation*, in particular base 10 (decimal) representation. The symbols 256 in base 10 don't represent three unrelated objects. Instead, the 6 represents the number of 1's, the 5 represents the number of 10's, and the 2 represents the number of $10 \times 10 = 100$'s. Adding these factors gives the number two hundred and fifty-six. Using base 10 representation, we can write very large numbers in a compact representation. Furthermore, arithmetic on large numbers is relatively easy using the methods we learned at school.

We use base 10 because we have 10 fingers so arithmetic in base 10 is easy to learn. Computers, however, have two "fingers" (**off** and **on**) so base 2 arithmetic is used in computation. Base 2 arithmetic looks strange at first; while we use the familiar symbols 0 and 1 also used in base 10, the rules for counting are cyclic at 2 instead of cyclic at 10:

0, 1, 10, 11, 100, 101, 110, 111, 1000, ...



Given a base 2 number such as 1101, we compute its value from right to left just as in base 10. The rightmost digit represents the number of 1's, the next digit represents the number of 2's, the third digit represents the number of $2 \times 2 = 4$'s, and the leftmost digit represents the numbers of $2 \times 2 \times 2 = 8$'s. Therefore, 1101 represents $1 + 0 + 4 + 8$, which is thirteen.

Program

Modify the program for counting modulo 4 to use binary representation.

Program file **count-to-four-binary.aesl**

We need only *two* quarters of the state to represent the numbers 0, 1, 2, 3 in base 2. Let the upper-right quarter represent the number of 1's—**off** (white) for none and **on** (orange) for

one—and let the upper-left quarter represent the number of 2's. For example,  represents the number 1 because the upper-left quarter is white and the upper-right quarter is orange. If both quarters are white, the state represents 0, and if both quarters are orange, the state represents 3. The number 2 is represented by , where the upper-left quarter is orange and the upper-right quarter is white.



There are four transitions $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 0$, so four event-actions pairs are needed, in addition to a pair to reset the program when the center button is touched.

★ **Ignore unused quarters of the state**

The two bottom quarters are not used, so they are left gray and are ignored by the program.

 **Exercise 9.2**

Extend the program so that it counts modulo 8. The lower-left quarter will represent the number of 4's.

 **Exercise 9.3**

How high can we count on the Thymio using binary representation?

Adding and subtracting

Writing the program to count to 8 is quite tedious because you had to program 8 event-actions pairs, one for each transition from n to $n + 1$ (modulo 8). Of course, that is not how we count in a based representation; instead, we have methods for performing addition by adding the digits in each place and carrying to the next place. In base 10 representation:

$$\begin{array}{r} 387 \\ +426 \\ \hline 813 \end{array}$$

and similarly in base 2 notation:

$$\begin{array}{r} 0011 \\ +1011 \\ \hline 1110 \end{array}$$

When adding 1 to 1, instead of 2, we get 10. The 0 is written in the same column and we carry the 1 to the next column to the left. The example above shows the addition of 3 (=0011) and 11 (=1011) to obtain 14 (=1110).

Program

Write a program that starts with a representation of 0. Each clap adds 1 to the number. The addition is modulo 16, so adding 1 to 15 results in 0.

Program file **addition.aesl**

Guidance

- Starting from the upper-right corner and continuing counter-clockwise, the quarters will represent the number of 1's, 2's, 4's and 8's in the number. Thus, the lower-right quarter represents the number of 8's.
- If the upper-right quarter representing the number of 1's shows 0 (white), simply change it to 1 (orange). Do this regardless of what the other quarters show.
- If the upper-right quarter representing the number of 1's shows 1 (orange), change it to 0 (white) and then carry the 1. There will be three event-action pairs, depending on the location of the *next* quarter showing 0 (white).
- If all quarters show 1 (orange), the value of 15 is represented. Adding 1 to 15 modulo 16 results in 0, represented by all quarters showing 0 (white).



Exercise 9.4

Modify the program so that it starts with the value 15 and subtracts one at each clap down to zero, and then cyclically back to 15.



Exercise 9.5

Place a sequence of short segments of black tape on a light surface. Write a program that causes the Thymio to move forward and stop at the fourth tape.

This exercise is not easy: the strips of tape have to be sufficiently wide so that the robot detects them, but not so wide that more than one event occurs per strip. You will also have to experiment with the speed of the robot.

Chapter 10



Accelerometers (Advanced Mode)

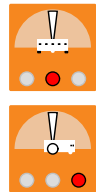
We are all familiar with *acceleration*, the rate of change of speed, for example, when a car speeds up or slows down. An *accelerometer* is a device for measuring acceleration. An airbag in a car uses an accelerometer to detect if the speed of the car is decreasing “too fast” because the car has crashed; if so, the airbag is inflated.

The Thymio robot has three accelerometers, one for each direction: forward / backwards, left / right, and up / down.

It is hard to achieve measurable accelerations, except for the case of *gravity* which is an acceleration towards the center of the earth. In this project, we use the accelerometers to measure the angle at which the robot is tilted.

There are two events that can detect the angle of the robot relative to the earth:

- : An event occurs when the left / right angle of the robot is within the white angle segment in the half-circle. (The technical term is *roll*)
- : An event occurs when the forward / backwards angle of the robot is within the white angle segment of the half-circle. (The technical term is *pitch*)



Initially, these blocks have the white segment pointing upwards from the top of the image of the Thymio, so that an event occurs when the robot is placed on a level surface such as a table or the floor. By dragging the segment with the mouse, you can select other angles; for example, the following block causes an event to occur when the robot is tilted left roughly half-way from vertical to horizontal:



Program


Hold the robot so that it is facing you and tilt it left and right. The top light of the robot will display a different color for each range of the angle of the tilt.

Program file **measure-angles.aesl**


Construct a set of event-actions pairs where each event is a left-right accelerometer event and the corresponding action changes the top color:



Make a list relating colors to angles so that you can translate any color to a specific angle.
The quarters of the event-state block are gray so the event causes the action in any state.

 **Exercise 10.1**

Can two events use the same white segment of angles?
How many events with different angles and you can construct?

 **Exercise 10.2**

Write a program that causes the robot to move forwards when a button is touched and to stop when it starts to tip forwards.
To avoid damaging the robot, test the program by having the robot fall off a magazine or two placed on a table!

Program file **acc-stop.aesl**

Part II

Parsons Puzzles

Chapter 11

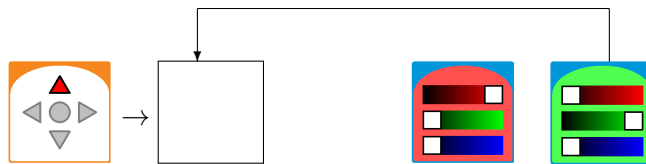
Parsons Puzzles for VPL

What are Parsons puzzles?

Parsons puzzles are a form of exercise that can help students learn how to program.¹ A Parsons puzzle consists of a specification of a program together with a set of statements in a programming language. Your task is to place the statements in the correct order so that they form a program that implements what is required. A Parsons puzzle may also include *distractors*, which are incorrect statements or extra statements that are not needed in the solution. The advantage of Parsons puzzles is that all the statements needed for the solution are visible to the student and have the correct syntax.

In VPL, there is almost no meaning to the order of the set of event-actions pairs in a program. Therefore, the puzzles will be programs where one or more pairs are missing the event block, the action block or both. To the right of each event-actions pair will appear two or more blocks; select the correct block and draw an arrow from it to the empty block.

Example When the forwards button is touched, the top green light is turned on.



The puzzles

1. When the right button is touched the bottom red light is turned on.



2. When the right button is touched the top red light is turned on.

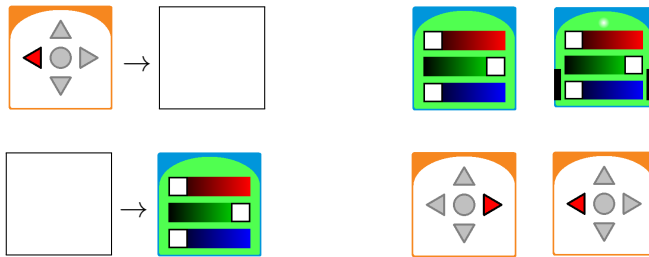


¹Parsons, D. and Haden, P. Parson's programming puzzles: A fun and effective learning tool for first programming courses. *Proceedings of the 8th Australian Conference on Computing Education*, Darlinghurst, Australia, 2006, 157-163.

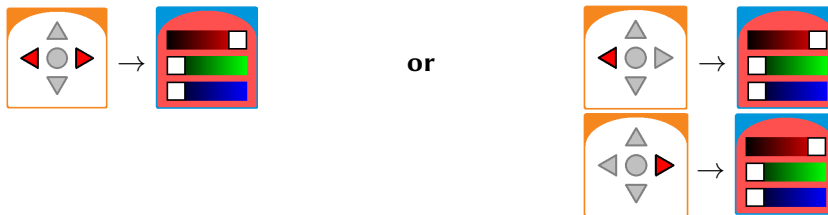
3. When the left button is touched the bottom green light is turned on.



4. When the left button **or** the right button is touched, the top green light is turned on.



5. When **both** the left button **and** the right button are touched, the top red light is turned on. Select one of the following two programs:



6. If an object is detected **only** by the leftmost sensor, turn left.



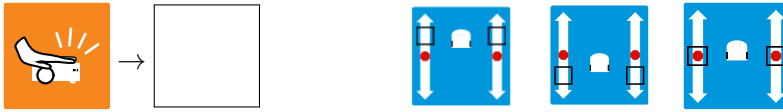
7. Stop the robot when the end of the table has been reached.



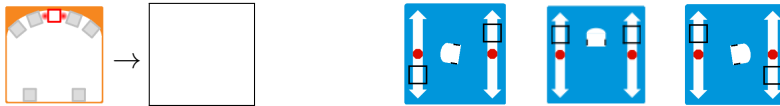
8. When the robot detects a wall, the top red light is turned on.



9. When the robot hits the wall, the motors are turned off.



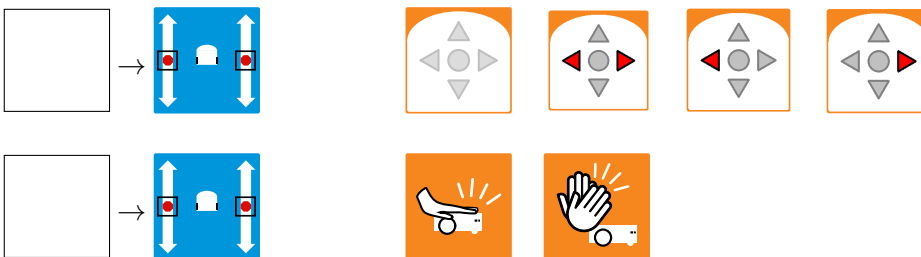
10. The robot turns to the left if there is an object in front of the center sensor.



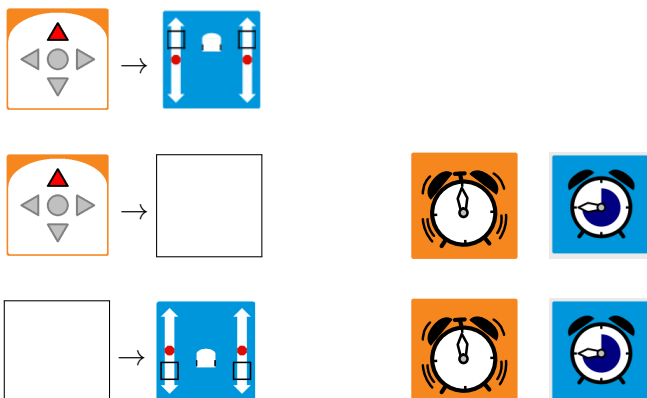
11. The robot turns to the right if there is **no** object in front of the center sensor.



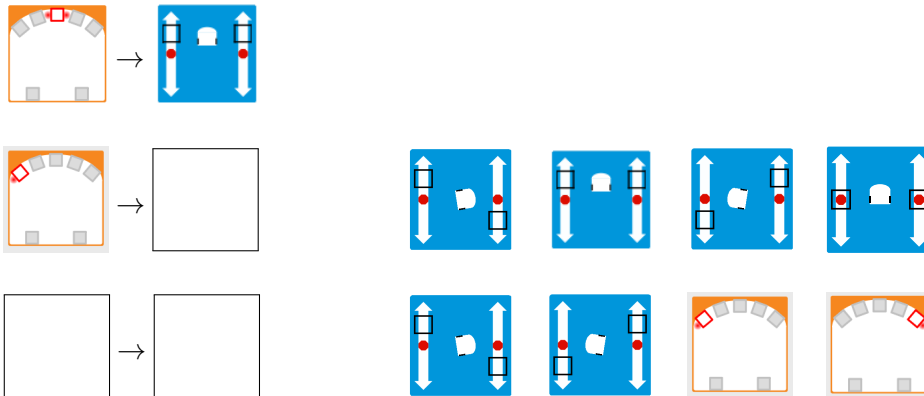
12. The motors are turned off when the left button is touched **or** if the robot is tapped.



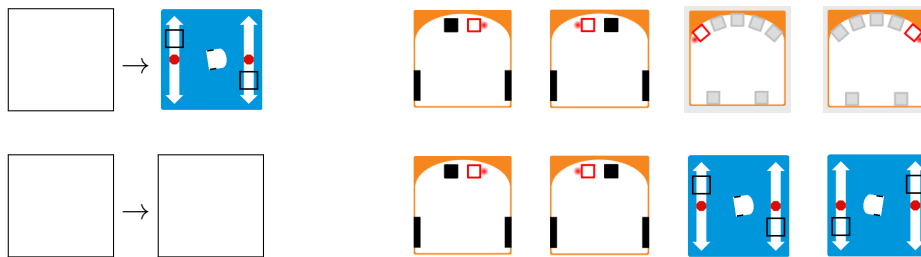
13. When the forwards button is touched, the robot moves forward for three seconds and then moves backwards.



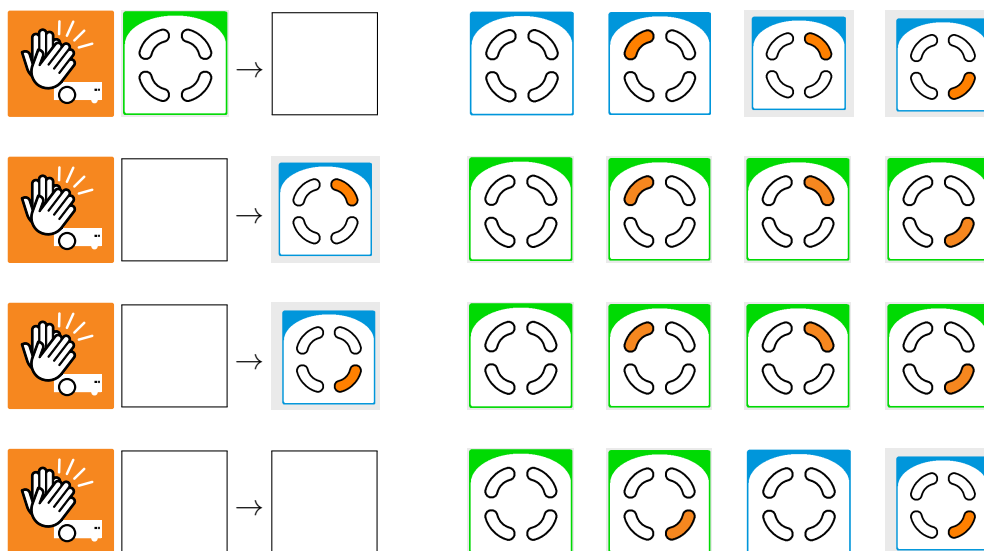
14. The robot moves towards an object that is detected by its left, right or center sensor.



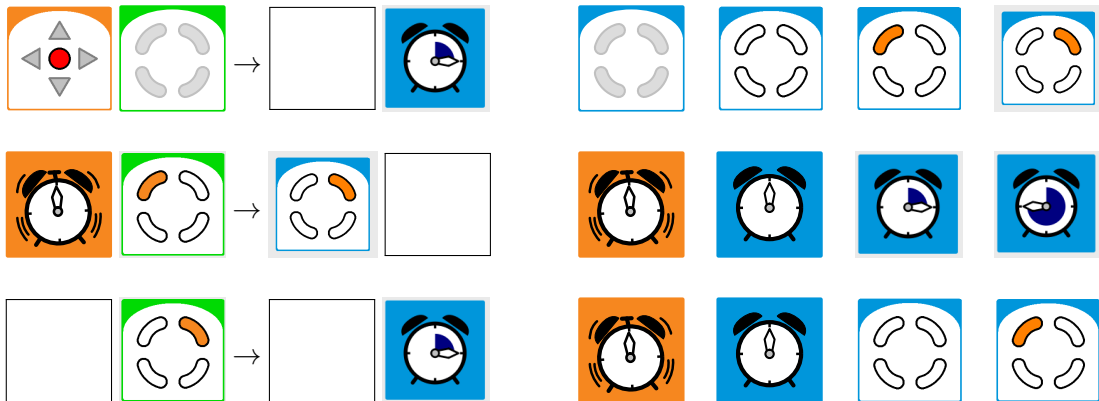
15. The robot is following a line on the floor. It turns left if it no longer detects the line in its right sensor and it turns right if it no longer detects the line in its left sensor,



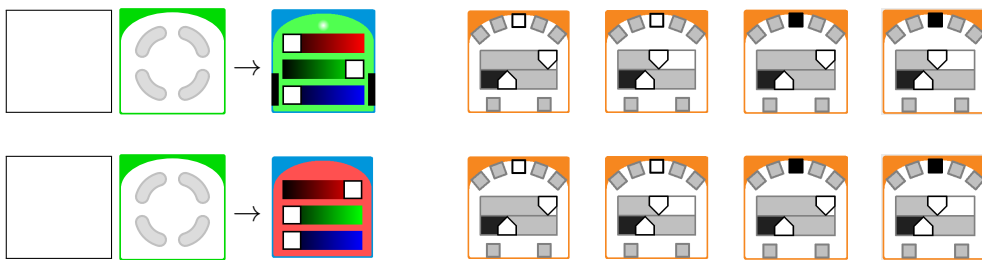
16. The robot counts 0,1,2,3,0,1,2,3, ..., whenever it detects a clap event.



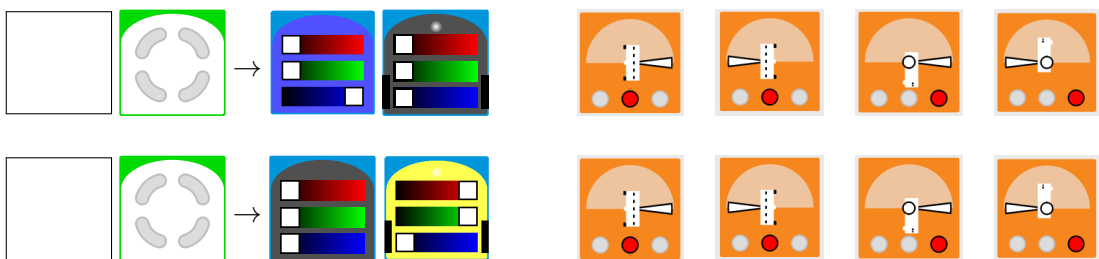
17. When the center button is touched, the right front and left front circle lights turn on and off alternately at one-second intervals.



18. The bottom light of the robot turns green when it detects an object far away from it and the top light of the robot turns red when it detects an object close to it.



19. Tilt the robot on its left side; the top light turns blue and the bottom light is turned off. Tilt the robot on its back; the top light is turned off and the bottom light turns yellow.



Part III

Projects

Chapter 12

Braitenberg Creatures

What are Braitenberg creatures?

Valentino Braitenberg was a neuroscientist who wrote a book describing the design of virtual vehicles which exhibited surprisingly complex behavior.¹ Braitenberg's vehicles have been widely used in educational robotics. Researchers at the MIT Media Lab created hardware implementations of the vehicles called *Braitenberg creatures*.² The vehicles were built from *programmable bricks* that were the forerunner of the LEGO Mindstorms robotics kits.

This chapter describes an implementation of most of the Braitenberg creatures from the MIT report adapted for the Thymio robot with VPL. The MIT hardware used light and touch sensors, while the Thymio robot relies primarily on infrared proximity sensors. To enable comparison with the MIT report, the names of the creatures used there have been retained, even though they may not be appropriate for the Thymio implementations. The order of presentation from the report has also been retained, although this does not correspond to the difficulty of implementation in VPL.

In the descriptions, the phrase “detects an object” is used. Unless otherwise indicated, this means that an object is detected by the front center sensor. The easiest way to do this is to place your hand so that it is detected by a sensor.

The VPL source code is available in the archive. The file names are the same as the names of the creatures with the extension `.aesl`. For some creatures, additional behaviors are suggested as exercises and their implementations also appear in the archive.

Specification of the creatures

Timid When the robot does not detect an object, it moves forwards. When it detects an object, it stops.

Indecisive When the robot does not detect an object, it moves forwards. When it detects an object, it moves backwards. At just the right distance, the robot will *oscillate*, that is, it will move forwards and backwards in quick succession.

Paranoid When the robot detects an object, it moves forwards. When it does not detect an object, it turns to the left.

¹V. Braitenberg. *Vehicles: Experiments in Synthetic Psychology* (MIT Press, 1984).

²David W. Hogg, Fred Martin, Mitchel Resnick. *Braitenberg Creatures*. MIT Media Laboratory, E&L Memo 13, 1991. http://cosmo.nyu.edu/hogg/lego/braitenberg_vehicles.pdf.

Exercise (Paranoid1) When an object is detected by the center sensor of the robot, it moves forwards. When an object is detected by the right sensor (but not by the center sensor), the robot turns right. When an object is detected by the left sensor (but not by the center sensor), the robot turns left.

Exercise (Paranoid2, advanced mode) As in **Paranoid**, but the robot alternates the direction of its turn every second. **Hint:** Use states to keep track of the direction and a timer to change states.

Dogged When the robot detects an object in front, it moves backwards. When the robot detects an object in back, it moves forwards.

Exercise (Dogged1) As in **Dogged**, but when an object is not detected, the robot stops.

Insecure If an object is not detected by the left sensor, turn the robot's right motor on and the left motor off. If an object is detected by the left sensor, turn the right motor off and the left motor on. The robot should follow a wall placed to its left. **Hint:** See the note about turning the robot in Appendix B.

Driven If an object is detected by the left sensor, the robot turns the right motor on and the left motor off. If an object is detected by the right sensor, the robot turns the left motor on and the right motor off. The robot should approach the object in a zigzag.

Persistent (advanced mode) The robot moves forwards until it detects an object. It then moves backwards for one second and reverses to move forwards again.

Attractive and repulsive When an object approaches the robot from behind, the robot runs away until it is out of range.

Consistent (advanced mode) The robot cycles through four states when it is tapped: moving forwards, turning left, turning right, moving backwards.

Frantic (advanced mode) The top light flashes red. **Hint:** You can use the sensor event block with all sensors gray as explained in Appendix B.

Exercise (Frantic1, advanced mode) Implement the flashing light using the button event block instead of the sensor event block. Is there a difference in the behavior of the robot? If so, what causes it?

Observant (advanced mode) The robot turns the top light green when the right sensor detects an object. The robot turns the top light red when the left sensor detects an object. Once a light is turned on, it waits three seconds before turning off; during this period, the light does not change.

Chapter 13

The Rabbit and the Fox

This chapter contains the specification of a large project (my program uses 7 event-actions pairs, each with 2–3 actions). You should have enough experience by now designing and implementing VPL programs in order to write it yourself. We will give the specification of the behavior of the robot as a list of tasks and suggest that you develop the program by implementing each task in turn.

Story¹ The robot is a rabbit, walking in the forest. A fox chases the the rabbit to catch it from behind. The rabbit senses the fox, turns around and catches the fox.

Specification

For each event, we specify a top color to be displayed when the event occurs.

1. Touch the forwards button: the robot moves forwards (blue).
2. Touch the backwards button: the robot stops (off).
3. If the robot detects the edge of the table it stops (off).
4. If the left rear sensor detects an object, the robot quickly turns left (counterclockwise) until the object is detected by the front center sensor (red).
5. If the right rear sensor detects an object, the robot quickly turns right (clockwise) until the object is detected by the front center sensor (green).
6. When the object is detected by the front center sensor, the robot moves forward quickly for one second (yellow) and then stops (off).

Program file **rabbit-fox.aesl**

¹The story is loosely inspired by a [joke](#) well-known to PhD students.

Chapter 14

Reading Barcodes

Barcodes are universally used in supermarkets and elsewhere to identify objects. The identification is a number or a sequence of symbols that is different for each type of object. The identification is used to access a database containing information about an object, such as its price. Let us build a barcode reader from the Thymio robot.

Specification

1. Carefully measure the distance between two front horizontal sensors and the width of a single sensor. Using a piece of flexible cardboard, black tape and strips of aluminum foil, construct an object with various arrangements of the strips of foil:



2. Each configuration of the three center horizontal sensors will represent a different code. (How many codes can there be?) For some or all of these codes, implement event-actions pairs that display a top color depending on the code identified.

Guidance:

We will only use the three center sensors, so the squares for the outer sensors will be gray. For the center sensors, squares where the reflecting foil must appear in the code are white, while squares where the foil must not appear are black. For example, the following event-actions pair displays yellow for the code **on-off-on**:



The program in the archive identifies barcodes with foil opposite any two of the three center sensors, as well as the code with no foil.

Program file **barcode.aesl**

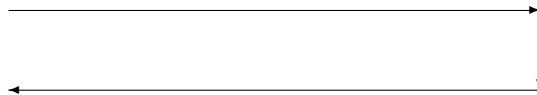
Chapter 15

Sweeping the Floor

Are you tired of cleaning your house? Now there are *robotic vacuum cleaners* that can do the job for you! The robot systematically moves over the floor of your apartment, navigating around furniture and other obstacles, while it vacuums the dirt.

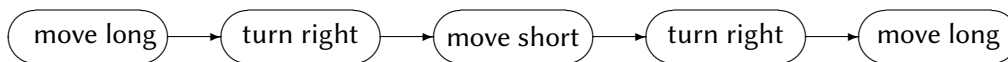
Specification

When the forwards button is touched, the Thymio robot travels from one side of the room to the opposite side, turns, moves a few steps and then returns to the first side:



Guidance

There are three subtasks that the robot must implement: (1) move the length of the room (to the right or to the left), (2) turn right, (3) move a few steps down. The subtasks are performed in the following order:



The robot will use states to keep track of which subtask it is performing. The direction and amount of movement for each subtask is determined by the speeds of the left and right motors, and by the length of time that the motors run. Therefore, each subtask will be implemented by an event-actions pair, where the event is the expiration of the timer for the previous subtask and the actions are to set the following parameters for the next subtask: (1) the state; (2) the left and right motor speeds; (3) the timer period. The program is initiated by a button event.

You will have to experiment with the speeds and the timer period to cause the robot to follow the required rectangular path.

Program file **sweep.aesl**

For fun, add colors to the top lights: green for straight, yellow for turn and red for stop.

Program file **sweep1.aesl**

Chapter 16

Measuring Speed

Specification

Measure the speed of the Thymio robot for different settings of the motors. Place a strip of black tape on a light-colored surface as you did for the line-following program (Chapter 5). Put the robot just before the one end of the tape. Implement the following behavior:

- The robot starts moving forward when the center button is touched.
- When the start of the tape is detected by the ground sensors, start a one-second timer.
- When the timer expires, change the top color and reset the timer to one second.
- When the end of the tape is detected, turn the motors off.

Run the program and count the number of times the color changes. This is the number of seconds that the robot took to move over the tape. Divide the length of the tape by the number of seconds to get the speed. For example, if the length of the tape is 30 centimeters and the color changes 6 times, the speed of the robot is $30/6=5$ centimeters per second.

Experiment with different motor settings and lengths of the tape.

Guidance

Write down a list of colors, say, 1=red, 2=blue, 3=green, 4=yellow, etc. Use the list to translate a color of the robot to a number of seconds.

Use states to keep track of the current and next colors. For example, in state 3, the color is green; when the timer expires *and* the state is 3, change the state to 4, change the color to yellow and reset the timer. There are three actions for each timer event.

Program file **measure-speed.aesl**

Chapter 17

Catch the Speeders

Specification

Help the police catch drivers who travel at high speed. Measure the speed by detecting how far the car travels in a fixed period of time.

The robot detects an object moving from its left side to its right side in front of the sensors. Turn the top light a different color to indicate how far the object has moved during one second from when it is first detected by the leftmost sensor.

Guidance

- In the initial state, when the leftmost sensor detects the object, start a one-second timer.
- When the timer expires, change the state to a new state; let us call it the *measure* state.
- Construct four event-actions pairs, one with an event for each of the other sensors; the event will only occur in the measure state. When a sensor detects the object, it turns the top light on to a color associated with that sensor.
- Ensure that an event-actions pair is run only when the object is detected by the corresponding sensor and not by the neighboring sensors.

Program file **speeders.aesl**

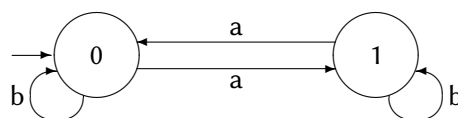
Chapter 18

Finite Automata

A *finite automaton (FA)*¹ is an abstract machine that is capable of performing computations. FA are extremely important in many areas of computer science.² Consider finite strings composed of two symbols a and b :

aabbbababbaba

The task is to read such strings and to decide if the number of a 's is odd or even. A FA to solve this problem has two states: it is in state 0 if the number of a 's read so far is even and it is in state 1 if the number of a 's read so far is odd. The FA is represented in the following diagram. It consists of two states, 0 and 1, and transitions from each state that are labeled a and b .

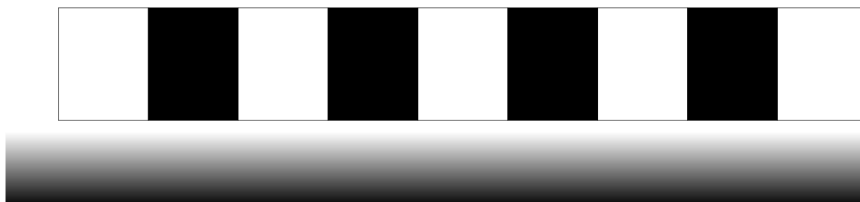


When the FA is in a state and reads a symbol from the string, it changes its state as indicated by the transitions. If the number of a 's read so far is even (state 0) and an a is read, the FA takes the transition to state 1, and conversely if the number of a 's read so far is odd (state 1), the transition to state 0 is taken. If a b is read, the state does not change, because the number of a 's read does not change.

The FA starts in state 0 since the initial number of a 's read is 0 which is even. This initial state is indicated by the small arrow.

Specification³

Print out the file `fa-path-alternate.pdf` containing the following image:⁴



¹The plural is *finite automata* and its acronym is also FA.

²FA are formally defined and developed in textbooks such as: J.E. Hopcroft, R. Motwani, J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, Pearson, 2013.

³The specification and implementation were inspired by the [light-painting project](#).

⁴The path files can be found in the directory **images** in this archive.

The shaded line is used to ensure that the robot moves forward and does not move off to the right or the left. The string is encoded by squares, where a is represented by a black square and b is represented by a white square. This image represents the string $babababab$.

Place the robot at the left of the image before the first square, facing right, with the right ground sensor in the middle of the shaded line. The behavior of the robot is as follows:

1. Touch the forward button to start the robot. The top light is turned off, the motors are started, the state is initialized (see below) and a timer is started.
2. Touch the center button to stop the robot.
3. The robot moves to the right. It uses the *right* ground sensor to detect if it starts to turn left or right. If the robot turns right, the sensor will detect a lower (black) light level and the robot turns left; if the robot turns left, the sensor will detect a higher (white) light level and the robot turns right.
4. When the timer expires, the value of the *left* ground sensor is checked. If the robot is over a white square (reading a b), the top light is turned red and the timer reset. If the robot is over a black square (reading a a), the top light is turned green, the timer reset and the state changed from even to odd or from odd to even.

Guidance

The robot will use three of the four quarters in the state events and actions:

- The upper-left quarter is used to indicate if the program is running or not.
- The upper-right quarter is used to indicate if the color (black or white) of a square should be detected.
- The lower-left quarter keeps track of whether the number of a 's read is even or odd.

Here is an example of one event-actions pair:



if the left sensor detects little light (a black square) *and*
the program is running *and* the color of the square should be detected *and*
an even number of a 's has been read so far, then
turn on the top red light
change the state: the number of a 's read is odd *and*
the color of the square should not be detected
reset the timer

Program file **fa.aesl**

You will have to experiment with the speed of the motors and the duration of the timers to reliably detect the black and white squares.

Exercise Print out the file `fa-path-blank.pdf` which contains an image where all the squares are white. Use a marker pen to blacken a different set of squares and test the program. In particular, check if the program works where two or more adjacent squares are blackened, representing a string with more than one consecutive a .

Exercise Modify the program so that it detects the remainder 0, 1 or 2 of the number of a 's read divided by 3.

Modifying the layout



Warning!

This section explains how to modify the path (the shaded line and the squares), but you have to be familiar with the \LaTeX document processing system to do so.

The files `fa-path-*.tex` in the archive contain the \LaTeX source code.

The following instruction draws a shaded line 2 cm wide by 23 cm long:⁵

```
\shade[left color=black,right color=white] (0,0) rectangle +(2,23);
```

The black and white squares are drawn using the following instructions:

```
\foreach \a in {1, 3, 5, 7}
  \filldraw[color=black] (\offset,\height*\a) rectangle +(\width,\height);

\foreach \a in {0, 2, 4, 6, 8}
  \draw (\offset,\height*\a) rectangle +(\width,\height);
```

You can change the list of numbers in the `foreach` instructions to specify which squares are black and which are white.

The `filldraw` and `draw` instructions use length parameters that can be easily changed:

```
\setlength{\height}{2.4cm} % Height of a square
\setlength{\width}{3cm}    % Width of a square
\setlength{\offset}{2.3cm} % Offset of the squares from the shaded line
```

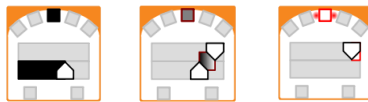
Format the file with `pdflatex` and print the page using software such as Adobe Reader or SumatraPDF. The image is created in portrait layout, but you can fix the page to a table in any orientation. Two or more images can be taped to a table in sequence to obtain a representation of a longer string.

⁵These instructions are from the *TikZ* graphics library.

Chapter 19

Multiple Sensor Thresholds

As explained in Appendix D, in advanced mode sensors events can be specified in three different ways: an event occurs when the reflected light is below a threshold (black), an event occurs when the reflected light is above a threshold (white), and an event occurs when the reflected light falls between two thresholds (dark gray):



Specification

Construct a program that causes the robot to approach an object, starting at a high speed, slowing down as it gets closer, and finally stopping when the robot is very close to the object.

Guidance

- Use three event-actions pairs, one with each type of sensor event.
- Carefully adjust the sliders (see Appendix D) so that the high value of one threshold is the same as the low value of the next threshold.
- Add a color block to each pair so that you can see the robot's speed setting.
- Use reflector tape to extend the range of the sensors as explained in Appendix B.

Program file **slow.aesl**

Specification

The line following program in Chapter 5 used two sensors to decide if the robot is moving off the line to the left or to the right. Implement a line following algorithm that uses one sensor.

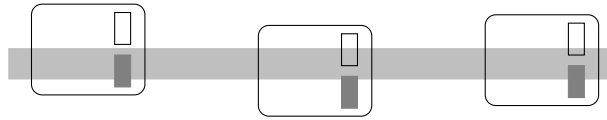
Guidance

The robot will follow the *edge* of the line, not its center. The ground sensors receive reflected light from a relatively wide area so the decision can be made using multiple thresholds. If the right sensor is used to follow the right edge of a (dark) line:

- If the sensor is to the left of the edge, little light will be sensed.

- If the sensor is to the right of the edge, a lot of light will be sensed.
- If the sensor is over the edge, the amount of light sensed will be midway between the two extreme values.

The three cases are shown in the following diagram:



Program file **line-one.aesl**

Chapter 20

Multiple Thymios

You can run two or more Thymio robots at the same time.

Specification

Place two robots T1 and T2 facing each other. T1 chases T2; when T1 detects that it is close to T2 it will stop. If T2 detects that T1 is close to it, T2 retreats until it no longer detects T1.

Guidance

- The programs for T1 and T2 have two event-action pairs: one whose event is the detection of an object by the center horizontal sensor, and another whose event is the non-detection of an object. However, the actions for T1 and T2 are different.
- Connect two Thymios T1 and T2 to the computer and turn them on. Run two copies of VPL. In the target-selection window (Figure 1.2), both T1 and T2 should appear; select T1 in one copy of VPL and T2 in the other. Open and run program chase in T1 and program retreat in T2.

Experiments

- What happens if you exchange the programs: T1 runs retreat and T2 runs chase? Explain.
- In advanced mode, experiment with different settings of the sensor thresholds.

Program file **chase.aesl**, **retreat.aesl**

★ Communications between robots

Multiple Thymio robots can send messages to each other. This capability is supported in the AESL language and Studio environment.

Part IV

From visual to textual programming

Chapter 21

Learning AESL from VPL programs

Congratulations! You are an expert in programming the Thymio robot using the *Visual Programming Environment (VPL)*. Now you want to move on and use the professional *Studio Programming Environment* (Figure 21.1) and its textual programming language, the *Aseba Event Scripting Language (AESL)*.

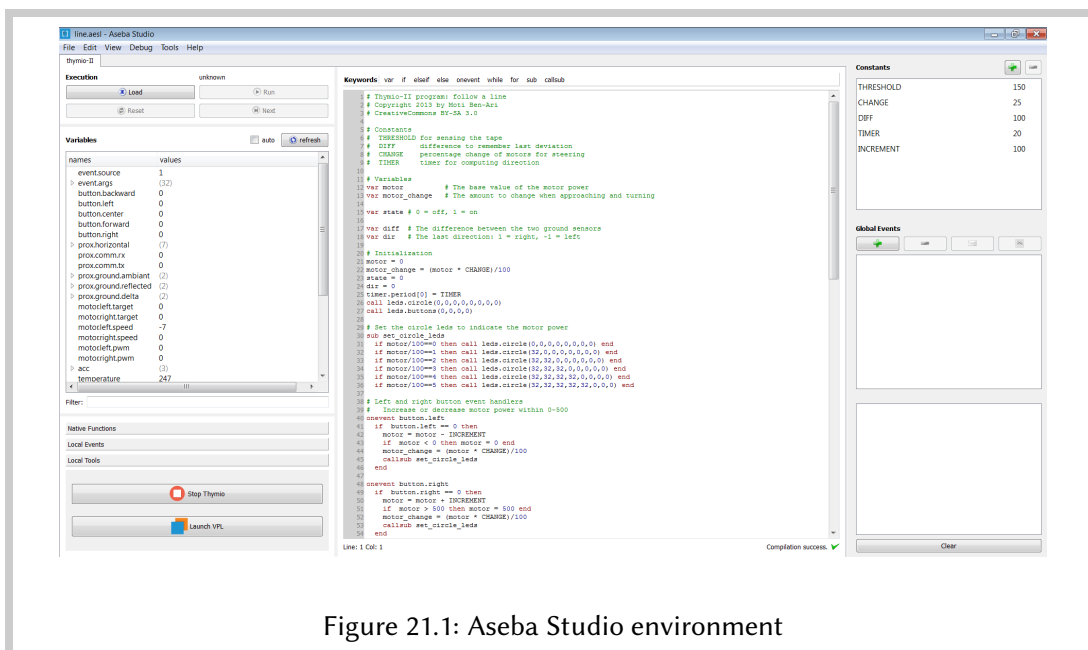


Figure 21.1: Aseba Studio environment

VPL translates graphical programs (event-actions pairs) into a textual AESL program, which is displayed in the right-hand panel of the VPL window (panel 6 in Figure 1.3 on page 13). This tutorial uses VPL programs from the previous chapters of this tutorial and explains the corresponding AESL program. You will be able to use your understanding of the VPL program to learn the fundamental concepts of AESL programming.

Programming in Aseba Studio is also based upon the concepts of events and actions. Since VPL programs are translated into AESL programs, everything you learned in this tutorial is supported in Studio, but now you have the flexibility of a full programming language with variables, expressions, and control statements.

When you are working with Aseba Studio, you can open VPL by clicking on the button **Launch VPL** in the *Tools* tab at the bottom left of the window. You can import VPL programs into Aseba Studio simply by opening its file.

Sections marked * present AESL programming concepts that go beyond what is found in the VPL projects. They can be skipped when you first read this tutorial.

Documentation

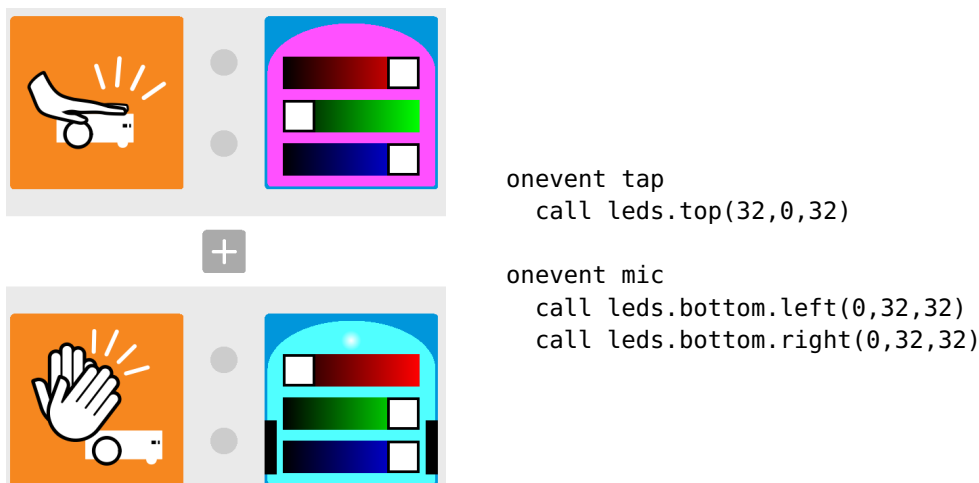
To learn about Aseba Studio and AESL, go to the *Programming Thymio* page at <https://www.thymio.org/en:asebausermanual>. You can find documentation of:

- The Studio programming environment.
- The AESL programming language.
- The interface to the Thymio robot. (There is a reference card for the interface).
- The native functions library supported in AESL.

There is also an archive describing interesting projects in AESL, together with the source of proposed solutions.

The Thymio Interface

Here is the program `whistles.aesl` from Chapter 6 together with part of the corresponding AESL program:



The diagram shows two Thymio interface blocks stacked vertically, separated by a plus sign. The top block has a 'tap' event icon (a hand tapping) and a light block with three sliders (red, green, blue) set to high red and blue, resulting in a magenta light. The bottom block has a 'clap' event icon (clapping hands) and a light block with three sliders (red, green, blue) set to high green and blue, resulting in a cyan light. To the right of the top block is the AESL code: `onevent tap` followed by `call leds.top(32,0,32)`. To the right of the bottom block is the AESL code: `onevent mic` followed by `call leds.bottom.left(0,32,32)` and `call leds.bottom.right(0,32,32)`.

Event handlers

When a tap event occurs, the top light is turned on with the color called *magenta*, and when the clap event occurs, the bottom light is turned on with the color called *cyan*. Corresponding to the event-actions pairs in VPL are *event handlers*, which are introduced by the keyword `onevent` (read this as two words: “on event”). You can find a list of events in the table at the bottom of the documentation for the Thymio programming interface.

The lines following `onevent` form the body of the event handler and correspond to the action blocks to the right of an event block in VPL.

When a tap event occurs, the *interface function* `leds.top` is called. The function takes three *parameters*, which specify the intensities of the red, green and blue components of the LED. Their values can range from 0 (off) to 32 (full). The combination of red and blue gives magenta.

The VPL clap event corresponds to the mic event (short for microphone). When the event occurs, the bottom LEDs are turned on. In VPL, one action block turns on both LEDs to the same color, whereas in AESL, the left and right LEDs can be set separately. Here, we set both of them to full intensity of green and blue, giving cyan.

Assigning a value to a variable

Look again at the AESL program in the VPL window. The first two lines are:

```
# setup threshold for detecting claps
mic.threshold = 250
```

A line beginning with # is called a *comment*. Comments do not affect the running of a program; they are used to give information to the reader of the program. Here, the comment notes that the clap event occurs when the intensity of the sound is greater than a *threshold*. The second line of the program specifies that the event occurs when the intensity of the sound (which can be in the range 0–255) is greater than 250.

In VPL, the threshold is built-in and cannot be changed, but in a textual program you can change it using an *assignment statement*:

```
mic.threshold = 180
```

Its meaning is that the *value* on the right-hand side of the = symbol is copied to the *variable* on the left-hand side. The variable mic.threshold is predefined for the Thymio robot.

Initialization of the Thymio

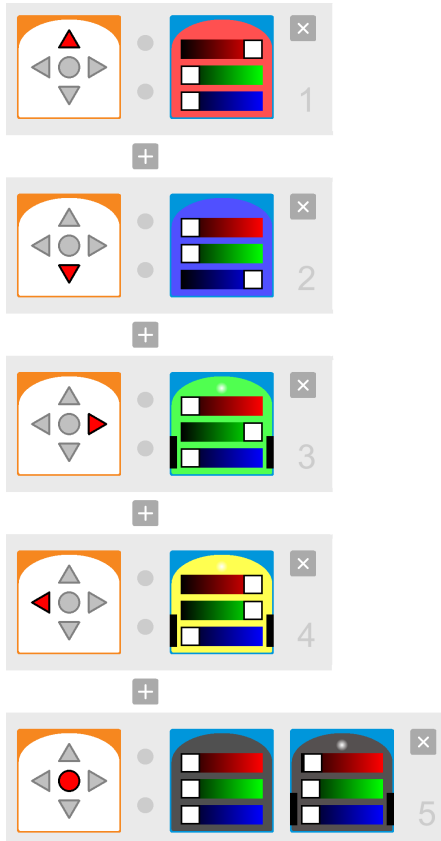
At the beginning of each program, VPL automatically inserts a sequence of statements that turns off all the LEDs and the sound:

```
# reset outputs
call sound.system(-1)
call leds.top(0,0,0)
call leds.bottom.left(0,0,0)
call leds.bottom.right(0,0,0)
call leds.circle(0,0,0,0,0,0,0,0)
```

This *initialization* is not visible in the VPL program. In a textual program, it is recommended that you include these statements, but it is not required.

Alternatives

The program `colors-multiple.aesl` from Chapter 2 changes the colors of the top and bottom LEDs when the buttons are touched:



```
onevent buttons
  when button.forward == 1 do
    call leds.top(32,0,0)
  end
  when button.backward == 1 do
    call leds.top(0,0,32)
  end
  when button.right == 1 do
    call leds.bottom.left(0,32,0)
    call leds.bottom.right(0,32,0)
  end
  when button.left == 1 do
    call leds.bottom.left(32,32,0)
    call leds.bottom.right(32,32,0)
  end
  when button.center == 1 do
    call leds.top(0,0,0)
    call leds.bottom.left(7,0,0)
    call leds.bottom.right(7,0,0)
  end
end
```

In the AESL program, a *single* event occurs when any of the five buttons is touched. The action of the event handler `onevent buttons` depends on which button is touched, so we check the value of the *button variables* in order to select an action. The statements:

```
when button.forward == 1 do
  call leds.top(32,0,0)
end
```

mean: *when* the value of the variable `button.forward` changes from some other value (here, 0) to 1, *then* perform the actions written on the lines between the keyword `do` and the keyword `end`. There are five `button` variables, one for each button. The value of a `button` variable is 1 if the button is touched and 0 if the button is released. In the program, there are five `when`-statements, one for each button. One or two actions are run if the expression in a `when`-statement *becomes* true.

One event or multiple events*

The Thymio interface includes separate events for each button, in addition to the `buttons` event that occurs if any button is touched or released. We could implement the program as follows, using multiple events without `when`-statements and `button` variables:

```

onevent button.forward
  call leds.top(32,0,0)

onevent button.backward
  call leds.top(0,0,32)

onevent button.right
  call leds.bottom.left(0,32,0)
  call leds.bottom.right(0,32,0)

onevent button.left
  call leds.bottom.left(32,32,0)
  call leds.bottom.right(32,32,0)

onevent button.center
  call leds.top(0,0,0)
  call leds.bottom.left(1,0,0)
  call leds.bottom.right(1,0,0)

```

The advantage of using separate events is that the program is easier to read and understand, but there are cases where you need to use the event buttons: (a) to distinguish between touching and releasing a button, and (b) to identify touching two buttons at once:

```

onevent buttons
  # Turn the top LEDs on when the forward button is released
  when button.forward == 0 do
    call leds.top(32,0,0)
  end

  # Turn the bottom LEDs on when
  # both the left and the right buttons are touched
  when button.left == 1 and button.right == 1 do
    call leds.bottom.left(0,32,0)
    call leds.bottom.right(0,32,0)
  end

```

Another difference is that the individual events occur when a button is touched or released, whereas the common event buttons occurs with a frequency of 20 Hz after updating the array of button variables (see page 76 for an explanation of these concepts).

if-statements

AESL supports two alternative statements:

```

when v == 1 do ... statements ... end

if v == 1 then ... statements ... end

```

that have different meanings:

when the value of *v* *becomes* 1, run the statements
if the value of *v* *is* 1, run the statements

when-statements are commonly used with variables representing events, because we usually want to run an event handler when something changes, not just because the value of a variable has a certain value. We could write a buttons event handler using an `if`-statement:

```
onevent buttons
  if button.forward == 1 then
    ... statements ...
  end
```

However, if we touch the forward button for a long period of time, the statements would be run several times. If the statements change the color of the LEDs, it wouldn't make a difference, but there are cases where it does matter and a `when`-statement is needed.

An `if`-statement is appropriate when we are interested the values of variables and not in their changes. The following statements set the value of the variable `max` to the maximum value returned by the two rear sensors:

```
if prox.horizontal[5] > prox.horizontal[6] then
  max = prox.horizontal[5]
else
  max = prox.horizontal[6]
end
```

Additional examples of `if`-statements appear in the next section and in [Figure 21.3](#).

Arrays

The program `likes.aesl` in Chapter 4 of the VPL tutorial causes the robot to follow your hand as you move it from side to side near the forward horizontal proximity sensors. When an object is not detected, the robot stops; when an object is detected in front of the center sensor, the robot moves forward; when an object is detected in front of the leftmost or rightmost sensor, the robot turns in that direction.

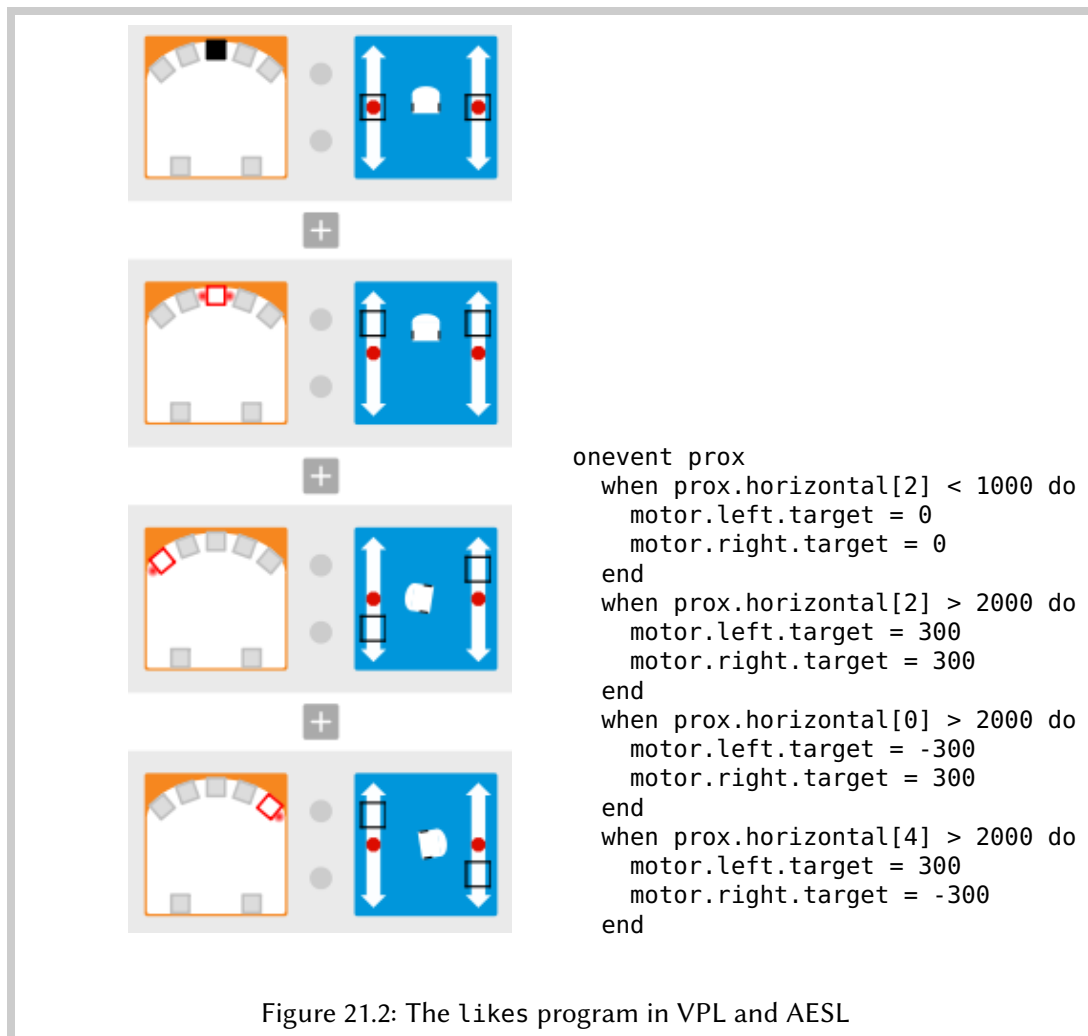


Figure 21.2: The likes program in VPL and AESL

The AESL program (Figure 21.2) is structured as an event handler with several when-statements. The values of the motor variables `motor.left.target` and `motor.right.target` are set to values corresponding to the positions of the sliders in the motor blocks. In VPL, the sliders change the values of the motor variables in increments of 50, but in AESL you can set them to any values in the range -500 to 500 .

The event is called `prox`. Unlike the button events, which occur when something “happens,” this event occurs *10 times every second*. Before the event occurs, the values of the `prox.horizontal` variables are set to values that depend on what the sensors are detecting. See the documentation of the Thymio programming interface for details.¹

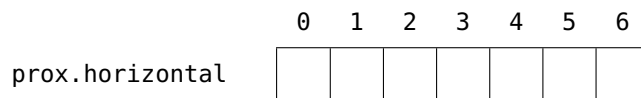
¹The unit for *frequency*, the number of times something happens per second, is called the *hertz*, abbreviated

Arrays as multiple variables

The Thymio robot has 7 horizontal proximity sensors, 5 in front and 2 in the back. To read the values detected by the sensors, it would be possible to define 7 different variables:

```
prox.horizontal.front.0
prox.horizontal.front.1
prox.horizontal.front.2
prox.horizontal.front.3
prox.horizontal.front.4
prox.horizontal.back.0
prox.horizontal.back.1
```

Instead, AESL enables you to define an *array*, which is a sequence of variables all with the same name. The different variables in the sequence are identified with a number. The array for the horizontal proximity sensors is predefined and is called `prox.horizontal`:



The first 5 components are for the front sensors from left to right, while the last two components are for the back sensors from left to right. If you don't remember the assignment of numbers to sensors, you can always look it up in the the documentation for the Thymio programming interface, even better, on the diagram in the reference card.

To access a specific component in an array, write its sequence number in square brackets after the name of the array variable. This number is called an *index* into the array. The following statement specifies that the motor variables will be set to 300 when the value of the *front center sensor* (index 2) becomes greater than 2000:

```
when prox.horizontal[2] > 2000 do
  motor.left.target = 300
  motor.right.target = 300
end
```

Later, we will see that array variables can have their values set in an assignment statement:

```
timer.period[0] = 1979
```

for-loops and index variables*

A natural generalization of arrays is to use a variable instead of a constant for the index.² The program `cats.aesl` contains the following statements:

Hz. The interface document specifies that the `prox` event occurs with frequency *10 Hz*.

²This is not used in translations of VPL programs into AESL, except in an advanced construct for constructing sounds; therefore, the simple example here is taken from the AESL projects.

```

var i

for i in 0:4 do
  if prox.horizontal[i] > DETECTION then
    state = 2
  end
end

```

Previously, we only used variables that are built into the Thymio interface; here, the first line *declares* a new variable called `i`. The next statement is a `for`-statement whose meaning is:

- Assign the values 0, 1, 2, 3, 4 in turn to the variable `i`;
- For each assignment, run the statements between `do` and `end`.

Here, there is a single `if`-statement between `do` and `end`. It checks the value of the horizontal proximity sensors and sets the value 2 in the variable `state` if the value read from a sensor is greater than the constant `DETECTION`.³

The variable `i` receives the values 0, 1, 2, 3, 4 in turn, so each time the `if`-statement is run, `prox.horizontal[i]` reads the value of each front sensor from left to right. The result of the `for`-statement is thus to set the value of the variable `state` to 2 *if any* of the front sensors detects an object.

Declaring an array

An array variable is declared by giving its size in brackets following the array name. The size can also be specified by providing an initial value:⁴

```

var state[4]           # An array with four components
var state[] = [0,0,0,0] # An array with four components

```

In the translation of the VPL program, both the size and the initial value are given. This is correct as long as the number of values is the same as the size:

```

var state[4] = [0,0,0,0] # OK, but redundant

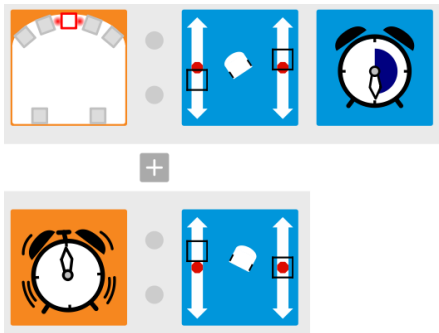
```

³See the documentation of the Aseba Studio environment for instructions on how to define constants.

⁴A comment need not start at the beginning of a line. Every character from the symbol `#` to the end of the line is considered to be a comment and is ignored by the computer.

Timers

Chapter 7 introduced *timers*. The program `shy.aesl` causes the robot to turn left when the front center sensor detects your hand; two seconds later, it turns right:



```
onevent prox
  when prox.horizontal[2] > 2000 do
    motor.left.target = -150
    motor.right.target = 100
    timer.period[0] = 2000
  end
```

```
onevent timer0
  timer.period[0] = 0
  motor.left.target = 200
  motor.right.target = 0
```

There are two timers in the Thymio robot. You set the duration of a timer by assigning a value to components 0 or 1 of the array `timer.period`. The value is in *milliseconds*, thousandths of a second. To set a duration of 2 seconds in timer 0, the value 2000 (milliseconds) should be assigned to `timer.period[0]`.

There are two events, `timer0` and `timer1`, one for each timer. When the duration has passed (we say that the timer has *expired*), the timer event occurs. In the handler for the event `timer0`, we set the timer duration to 0 so it won't occur again and change the motor settings.

As part of its initialization, the program sets the timer to 0 so that the event won't accidentally occur at the beginning of the program:

```
# stop timer 0
timer.period[0] = 0
```

States

Chapters 8 and 9 showed how to use *states*. The Thymio robot can be in one of 16 states and you can specify that an event causes an action only if the robot is in certain states. In the program `count-to-two.aesl` from Chapter 9, the state is set to 0 when the center button is touched and then it counts whether the number of claps is even or odd by alternating between state 0 and state 1. The VPL program and the AESL event handler for touching the center button are:⁵



```
var state[] = [0,0,0,0]

onevent buttons
  when button.center == 1 do
    state[0] = 0
    state[1] = 0
    state[2] = 0
    state[3] = 0
  end
```

The state is stored in an array `state[]` which has 4 components. Each component can be 0 or 1, so there are $2 \times 2 \times 2 \times 2 = 16$ possible values in the array. The components of the array are given initial values of 0 by assigning the four values `[0,0,0,0]`. The initial value of the array is also used to specify the number of components in an array; since there are 4 values in `[0,0,0,0]`, there are 4 components in the array.

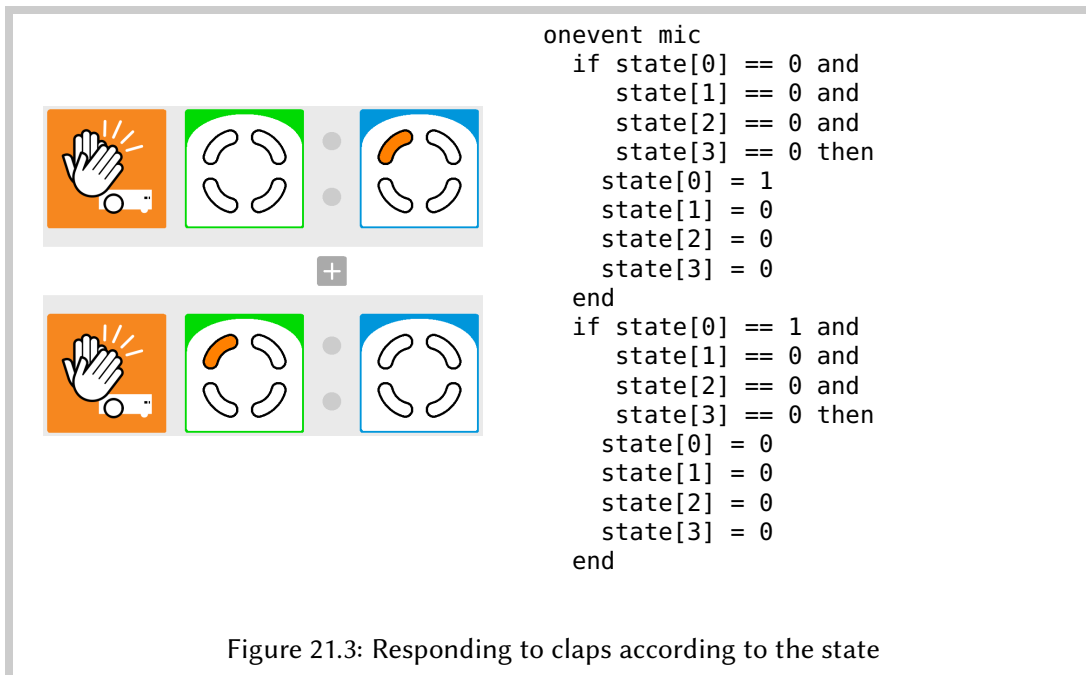
The state event block (green, next to the button event block) has all of its quarters set to gray; this means that the event can occur regardless of the current value of the state. Therefore, whenever the center button is touched, the state action block (blue) causes all the components of the array `state` to be set to 0, as indicated by the white quarters. In the corresponding AESL program, the `when`-statement checks if the button was touched, but does not check the value of the array `state`. If the button was touched, each component of the array is set to 0.

There are two clap event blocks, each one associated with a different state block. In the textual program a single `mic` event handler will be used. The statements to be run will depend on `if`-statements (Figure 21.3).

The meaning of the keyword `and` is that *all* the conditions in the `if`-statement must hold in order to run the statements between `then` and `end`. If all components of `state` are 0, the value of `state[0]` is set to 1, while the others are set to 0. This corresponds to the state event block having all quarters white and the state action block having the upper left quarter orange and the others white. Similarly, if value of `state[0]` is 1 and the values of the other components are 0, the values of all the components of `state` are set to 0.⁶

⁵The AESL program shown here is different from the one generated by VPL as explained later in this chapter.

⁶In the state action block, it would be sufficient just to change the upper left quadrant and gray the other quadrants since their values don't change. However, errors are less likely if we explicitly give values (orange or white) to each of the quadrants.



Subroutines

Quite often we need to run the same sequence of statements from many places within a program. We could write the statements once and copy them each time they are needed. A simpler solution is to use a *subroutine*, which assigns a name to a sequence of statements. In this program, the declaration `sub display_state` declares a subroutine that assigns the name `display_state` to a sequence of statements, here, the single statement `call leds.circle`:

```

# subroutine to display the current state
sub display_state
  call leds.circle(
    0, state[1]*32, 0, state[3]*32, 0, state[2]*32, 0, state[0]*32)

```

When the subroutine is *called*, it runs the statements assigned to the name of the subroutine:

```

callsub display_state

```

The interface function `leds.circle` sets the eight curved LEDs surrounding the buttons. You really do need to refer to the cheat sheet to learn which parameter sets which LED!

The intensity of each LED is set by giving the corresponding parameter a value between 0 (off) and 32 (full intensity). The front, back, left and right LEDs are set to 0 (off), whereas the diagonal LEDs are set to on if the corresponding state component is 1 and off if the state component is 0. This is achieved by the *arithmetic expressions* `state[...]*32` which multiply the value of the components of the array by 32. If one of the values is 0, the result is 0, while if it is 1, the result is 32.

Native functions

The program above has a problem. Since the components of the array state are set one-by-one, it is possible that an different event will occur when some, but not all, of the components have been set. To set all the components at once, the new values of the state are first set in a second array `new_state` and then they are copied to the first array state:

```
# variables for state
var state[4] = [0,0,0,0]
var new_state[4] = [0,0,0,0]

onevent buttons
  when button.center == 1 do
    new_state[0] = 0
    new_state[1] = 0
    new_state[2] = 0
    new_state[3] = 0
  end

  call math.copy(state, new_state)
  callsub display_state
```

The *native function* `math.copy` is used to copy the arrays. Native functions are built into the Thymio robot and are more efficient than sequences of statements in AESL. The native functions are described in the Aseba documentation.

The current version of AESL allows assignment of entire arrays, so that it would have been possible to use the assignment statement:⁷

```
state = new_state
```

⁷The array assignment translates into a sequence of individual assignment statements, one for each component, so nothing is actually gained by using an array assignment statement.

Part V


Appendices

Appendix A


The VPL User Interface

At the top of the VPL window is a toolbar:




New : Clears the current program and displays an empty program area.




Open : Click to open an existing program in VPL. A window will pop up and you can navigate to the directory where the program file (extension aesL) exists.




Save : Saves the current program. It is a good idea to click this button frequently so you don't lose your work if an fault occurs.




Save as : Saves the current program with a *different name*. Use it when you have a program and you want to try something new without changing the existing program.




Undo : Undo previous actions such as deleting an event-actions pair.




Redo : Redo an action that has been undone.




Run : Runs the current program. This button is only active if the compilation was successful. If you have changed the program after a previous run, the button will flash green to remind you that you must click it to load the modified program into the Thymio robot.



Stop : Stops the program that is running and sets the speed of the motors to zero. Use it when the program asks the robot to move but does not include an event-actions pair that can stop the motor.



Advanced mode : The advanced mode enables additional features: states, timers, accelerometers, setting sensor thresholds.



Basic mode: The above icon changes to  in advanced mode. Click it to return to basic mode.



Help ⓘ: Opens a browser window with the VPL documentation at:
<https://www.thymio.org/en:thymiovgl>.
An Internet connection is required.




Export 📷: Exports a graphical image of the program to a file. You can then import the graphics file into a document such as a textbook or worksheet. Several formats are available. svf will give the best quality, but PNG is more widely supported.




Appendix B

Summary of VPL Blocks

Event blocks

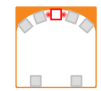
 **Buttons.** Click on one or more of the images of the buttons; they will turn red. An event will occur if the red buttons are touched.




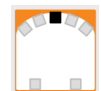
 **Horizontal sensors** (five at the front of the Thymio and two at the back). Click on one or more of the small squares and they will change color. Initially, all squares are gray, meaning that the reading of each sensor is ignored.



If the square is white with a red border , an event occurs if a lot of light is reflected.



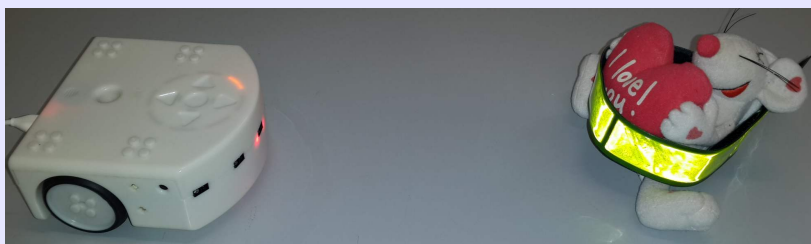
If the square is black , an event occurs if little light is reflected.




Trick

Ordinary objects need to be very close to the Thymio before they are detected by the horizontal sensors. You can greatly increase the range by attaching *reflector tape*, such as used on bicycles, to the objects.^a


Compare the following image with Figure 8.3:

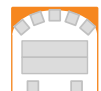


^aMy thanks to Francesco Mondada for showing this to me!


 **Ground sensors** (two on the bottom of the Thymio). Use like the block for the horizontal sensors.




 **Sensors, advanced mode.** Use like the previous blocks. The top slider sets the




threshold above which an object is detected and the bottom slider sets the threshold below which the absence of an object is detected.

There is an additional mode (the square is dark gray ) in which an event occurs if the value is between to upper and lower threshold.





 **Tap.** An event occurs when the Thymio is tapped.




 **Tap, advanced mode.** Use like the previous block. Click on the small center or right circle to change to an accelerometer event.




  **Accelerometer, advanced mode.** Drag the white angle segment on the half-circle left or right. An event will occur if the left / right or forwards / backwards angle (respectively) of the Thymio is within the segment.




 **Timer, advanced mode.** An event occurs when a timer has counted down to zero. The timer must have been set by a previous timer *action*.




 **State event, advanced mode.** The event occurs only if the components of the current state match the corresponding orange and white quarters of this block. The components corresponding to the gray quarters need not match.




Action blocks

 **Motors.** Move the left and right sliders up to increase the forwards rotation of the left and right motors, respectively, and move the sliders down to increase the backwards rotation.




 **Top lights.** Move the three sliders to the right to increase the red, green and blue components of the top light, respectively.




 **Bottom lights.** Turns the bottom lights on. Use like the previous block.




 **Music.** The six small circles are notes. A black circle is a short note, a white circle is a long note and a blank is a rest. Click on a circle to change the length. The five horizontal bars, represent tones. Click on the bar to move a circle to that bar.



 **Timer, advanced mode** The timer can be set for up to four seconds. Click anywhere within the white circle showing the face of the clock. There will be a short animation and then the amount of time until the alarm will be colored blue.



 **States, advanced mode** The four quarters in the block correspond to four components of a state. Click a quarter to turn it to gray, orange or white.



Notes on the VPL Blocks

★ Turning in place or a gentle turn

When the motors run at the same speed and in different directions, the robot turns in place. However, if only one motor is on, the robot goes both forwards and turns towards the side opposite of the running motor. You may need to set a higher power to overcome ground resistance.

★ Rapid repetitive sensor events

In most projects, you set a color (white, black, dark gray) in a square in a sensor block to indicate the the corresponding sensor participates in *filtering* the events. However, if *all* the sensors remain in the original gray, then no filtering is done. The event will occur 10 times per second no matter what values are read from the sensors.

★ Rapid repetitive button events

The same holds true for the button events, except that they occur 20 times per second.

Appendix C

Tips for Programming with VPL

Exploring and experimenting

Understand each event and action block For each event block and each action block, spend some time experimenting until you understand exactly how it works. To explore the behavior of an action block, construct a pair whose event is touching a button. This is a simple event that will let you concentrate on learning what the action block does. To explore the behavior of an event block, construct a pair whose action is changing the color of the robot.

Experiment with the sensor event blocks The small red lights next to each sensor show when that sensor detects an object. Move your fingers in front of the sensors and see which lights are turned on. Construct an event-actions pair consisting of the sensor event and the top color action, and experiment with the settings of the small squares in the event block (gray, white, black, and dark gray in advanced mode).


Experiment with the motors Experiment with the settings of the motors so that you get a general impression how fast the robot moves for particular settings. Experiment with different settings for the two motors in order to learn how the robot turns.

Constructing a program

Plan your program Before beginning to write a program, write down a description of how the program is supposed to work: a sentence for each event-actions pair.

Construct one event-actions pair at a time When you understand how each event-actions pair works, you can put them together in a program.

Test each addition to the program Test your program each time you add a new event-actions pair so that you can identify which pair causes an error.

Use Save as after modifying a program Before you change your program, click  to save the program under a different name. If the change makes things worse, it will be easy to go back to the previous version.



Display what happens Use colors to display what the program is doing. For example, if a sensor in one pair causes the robot to turn left and a sensor in another pair causes it to turn right, add an action to each pair that displays different top colors. You will be able to see if a problem is caused by the sensors or if the motors are not responding correctly to a sensor event.

Troubleshooting

Use a smooth surface Make sure that the surface on which the robot moves—a table or the floor—is very clean and smooth. Otherwise, the motors may not be able to move the robot or turns may be uneven.

Use a long cable Make sure that your cable is long enough. If the robot moves too far, the cable can cause the robot to slow down or stop.

Sensor events may not be detected Sensor events happen 10 times per second. If the robot is moving very fast, an event might not be detected.


For example, if the robot is supposed to detect the edge of the table and stop, and if the robot is moving very fast, it might fall off the table before the sensor event can stop the motor. When you run a program, start at a low speed and gradually increase it.


For another example, consider the line-following program in Chapter 5. Its algorithm depends on being able to detect when one ground sensor detects the line and the other doesn't. If the robot is moving too fast, the position where only one sensor detects the line does not cause an event.

Problems with the bottom sensors In programs like the line-following program, the sensor must distinguish between lots of reflected light and little reflected light. Make sure that there is a lot of contrast between the two. For example, if your table is not light enough, you can attach white paper to get better results.

Alternatively, you can adjust the thresholds in advanced mode.

Event-actions pairs are run sequentially In theory, the event-actions pairs are run *concurrently*—at the same time; in practice, they are run one after the other in the order they appear. As shown in Exercise 4.2, this can cause a problem, because the second action might conflict with what was done by the first action.

Problems with the clap event Do *not* use the clap event  when the motors are running. The motors make a lot of noise and can cause unexpected clap events.

Similarly, do *not* use the tap event  in the same program with the clap event. Tapping on the robot causes noise that can be interpreted as a clap.



Appendix D

Techniques for Using the Sliders

Setting the sliders in the motor blocks

It is difficult to set the sliders precisely so that, for example, both motors run at the same speed. We can use the translation of the event-actions pairs into an AESL textual program to improve the precision.



Trick

By moving the sliders on the motor action blocks, you will see that the target speeds of the motors (`motor.X.target`) jump by steps of 50 in the range -500 to 500 . By moving the sliders carefully, you can set the speeds to any of these values.

Figure D.1 shows the program from Figure 4.4 (where the pet likes you and follows you around) along with the text translation at the right of the VPL window. This text is modified automatically as you move the sliders.

onevent prox means: whenever the event of sampling the horizontal sensors (the *proximity* sensors, abbreviated *prox*) occurs, the instructions on the following lines before end will be run. The proximity event occurs 10 times a second.

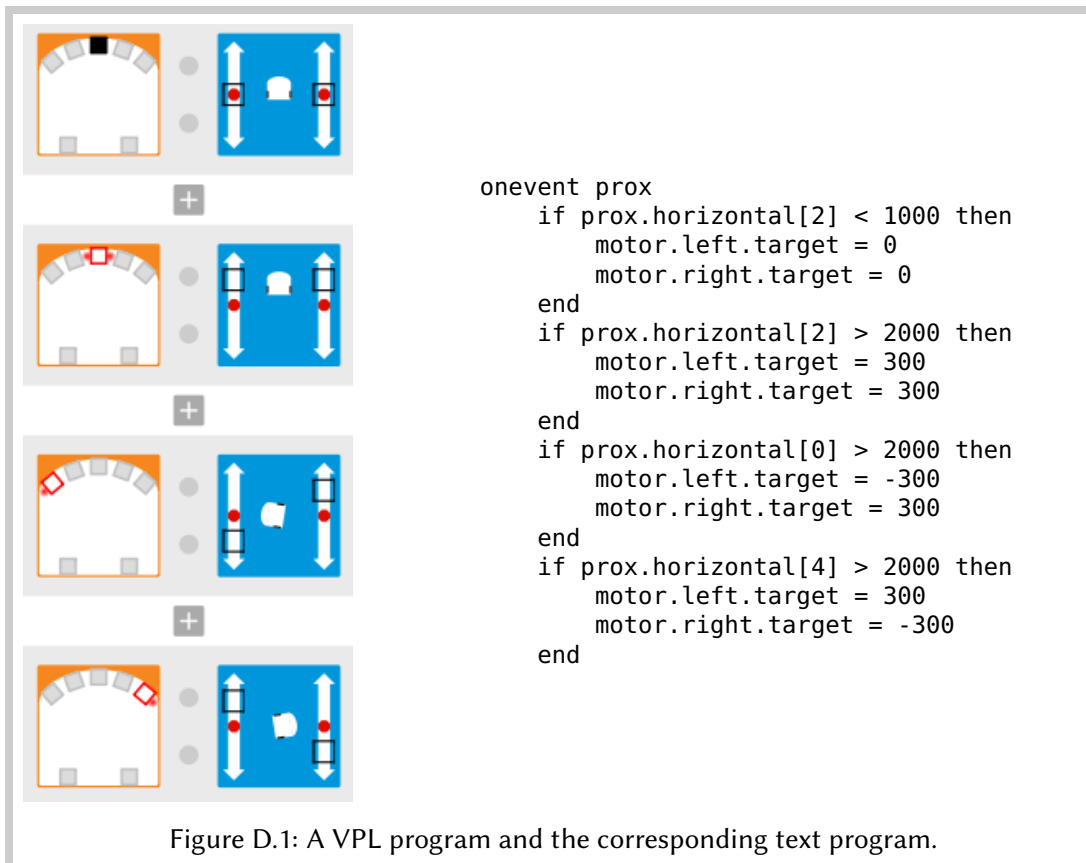
When the event happens, the values of the sensors are checked using a condition. Sensor number 2 (front center) is checked first: `prox.horizontal[2]`. If this value is lower than 1000, the speeds of the left and right motors are set to 0 using the instructions:

```
motor.left.target = 0
motor.right.target = 0
```

Every block `if ... then ... end` tests a specific sensor and runs the associated instructions if the result of test is true. The program runs the following algorithm:

0. Tests if nothing is in front; if that is true, the robot stops.
1. Tests if something is in front; if that is true, the robot goes forward.
2. Tests if there is something at left; if that is true, the robot turns left.
3. Tests if there is something at right; if that is true, the robot turns right.

Once all the sensors have been read and the appropriate action is performed, the program waits for the next event `prox` and starts these tests again. This happens again and again until the program is stopped.



Setting the duration of the timer

Trick

The duration in the time action block can be set in multiples of one-quarter second (250 milliseconds) up to four seconds.

Sensor event blocks in advanced mode

This section describes features of the sensor event blocks that are available in advanced mode.

Setting the thresholds of the sensors

In basic mode, the thresholds of the sensors are fixed. For the horizontal sensors, a value above 2000 means that a lot of light is reflected and an event will occur if the corresponding square is white, while a value below 1000 means that little light is reflected and an event will occur if the corresponding square is black. For the bottom sensors, the values are 450 and 400.

In advanced mode, the thresholds can be set. The top slider sets the threshold above which a white event occurs and the bottom slider sets the threshold below which a black event occurs:

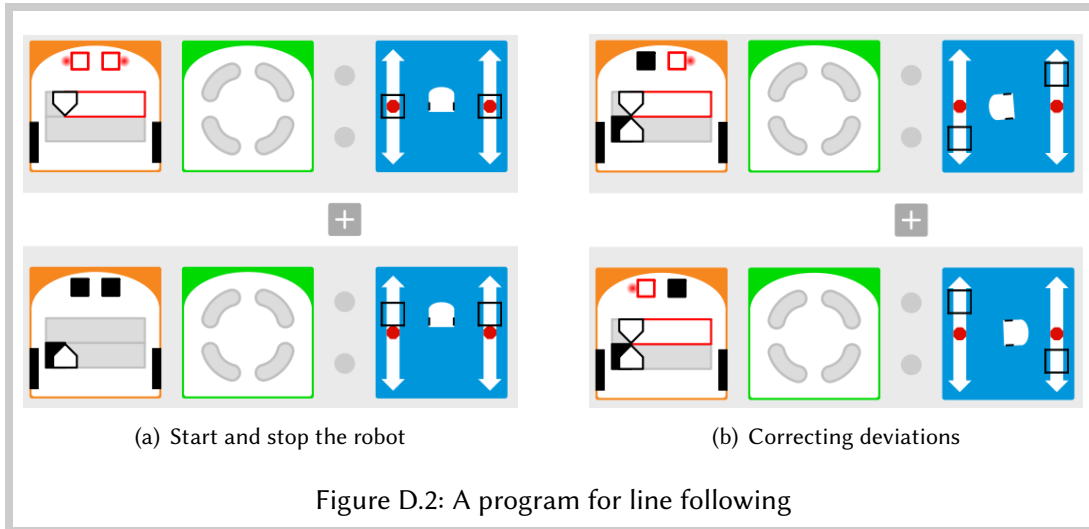


Figure D.2 shows the line-following program (Figure 5.2) in advanced mode. The sliders are set so that the threshold is very low: 100 for both the upper and lower thresholds.

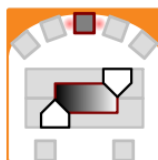
Multiple sensors

If several sensors are set, they share the same thresholds:



Events for values between the thresholds

There is an additional mode indicated by a dark gray square:



In this mode, an event occurs if the value read by the read is greater than the lower threshold (set by the lower slider) and less than the upper threshold (set by the upper slider).