

# A PharoThings Tutorial

Alex Oliveira

June 5, 2019

Copyright 2019 by Alex Oliveira.

The contents of this book are protected under the Creative Commons Attribution-ShareAlike 3.0 Unported license.

You are **free**:

- to **Share**: to copy, distribute and transmit the work,
- to **Remix**: to adapt the work,

Under the following conditions:

**Attribution.** You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page:

<http://creativecommons.org/licenses/by-sa/3.0/>

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.



Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license):

<http://creativecommons.org/licenses/by-sa/3.0/legalcode>

# Contents

|  |           |
|--|-----------|
| <b>Illustrations</b>   | <b>iv</b> |
| <b>1 Installations</b>   | <b>3</b>  |
| 1.1 Running Pharo IoT in a Raspberry that has Raspbian . . . . .                 | 3         |
| 1.2 Installing Raspbian and Pharo IoT from scratch (Raspberry Pi Headless Setup) | 4         |
| 1.3 Run Pharo IoT IDE on your Linux, Windows or Mac OSX computer. . . . .        | 5         |
| 1.4 In the next chapter . . . . .  | 8         |
| <b>2 Lesson 1 – Turning LED on/off</b>   | <b>9</b>  |
| 2.1 What do we need? . . . . .   | 9         |
| 2.2 Experimental theory . . . . .  | 10        |
| 2.3 Experimental procedure . . . . .   | 11        |
| 2.4 Experimental code . . . . .  | 12        |
| 2.5 What did we learn? . . . . .   | 14        |
| 2.6 In the next lesson . . . . .   | 14        |
| <b>3 Lesson 2 – Blinking LED</b>   | <b>15</b> |
| 3.1 What do we need? . . . . .   | 15        |
| 3.2 Experimental code . . . . .  | 16        |
| 3.3 In the next lesson . . . . .   | 16        |
| <b>4 Lesson 3 – A brief introduction to Pharo object-oriented language</b>       | <b>17</b> |
| 4.1 Developing a simple LED blinker . . . . .                                    | 17        |
| 4.2 Our use case . . . . .   | 17        |
| 4.3 Create your own class remotely . . . . .                                     | 18        |
| 4.4 Create a package . . . . .   | 19        |
| 4.5 Create a class . . . . .   | 19        |
| 4.6 Create a protocol . . . . .  | 19        |
| 4.7 Creating an initialize method . . . . .                                      | 21        |
| 4.8 Using your new class . . . . .   | 22        |
| 4.9 Save your work . . . . .   | 23        |
| 4.10 Conclusion . . . . .  | 23        |

|          |   |           |
|----------|---|-----------|
| <b>5</b> | <b>Lesson 4 - LED Flowing Lights</b>  | <b>25</b> |
| 5.1      | What do we need? . . . . .  | 25        |
| 5.2      | Experimental procedure . . . . .  | 25        |
| 5.3      | Experimental code . . . . .   | 27        |
| 5.4      | Adding features . . . . .   | 28        |
| 5.5      | Reversing the flow . . . . .  | 29        |
| 5.6      | Going and backing the flow . . . . .  | 29        |
| 5.7      | Terminating the process . . . . .   | 30        |
| 5.8      | In the next lesson . . . . .  | 31        |
| <b>6</b> | <b>Lesson 5 - LED Flowing Lights using OOP</b>  | <b>33</b> |
| 6.1      | The goal . . . . .  | 33        |
| 6.2      | Creating the class . . . . .  | 34        |
| 6.3      | Creating the initialize method . . . . .  | 34        |
| 6.4      | Creating access methods . . . . .   | 34        |
| 6.5      | Creating process control methods . . . . .  | 35        |
| 6.6      | Creating action methods . . . . .   | 35        |
| <b>7</b> | <b>Lesson 6 - LED Breathing PWM</b>   | <b>37</b> |
| 7.1      | What do we need ? . . . . .   | 37        |
| 7.2      | Experimental procedure . . . . .  | 38        |
| 7.3      | Connecting remotely . . . . .   | 38        |
| 7.4      | Experimental code . . . . .   | 38        |
| 7.5      | In the next lesson . . . . .  | 39        |
| <b>8</b> | <b>Lesson 7 - Controlling LED by Button</b>   | <b>41</b> |
| 8.1      | What do we need? . . . . .  | 41        |
| 8.2      | Experimental theory . . . . .   | 42        |
| 8.3      | Experimental procedure . . . . .  | 42        |
| 8.4      | Connecting remotely . . . . .   | 42        |
| 8.5      | Experimental code . . . . .   | 43        |
| 8.6      | Terminating the process . . . . .   | 44        |
| 8.7      | In the next lesson . . . . .  | 44        |
| <b>9</b> | <b>Lesson 8 - I2C Sensors (Temperature, Humidity, Pressure and<br/>Accelerometer)</b> | <b>47</b> |
| 9.1      | What do we need? . . . . .  | 47        |
| 9.2      | Experimental theory . . . . .   | 48        |
| 9.3      | Experimental procedure . . . . .  | 49        |
| 9.4      | Configuring the Raspberry Pi I2C . . . . .  | 50        |
| 9.5      | Connecting remotely . . . . .   | 50        |
| 9.6      | Exploring the properties of a remote object with the remote inspector . . .           | 51        |
| 9.7      | Getting the temperature with BME280 . . . . .   | 52        |
| 9.8      | Getting the humidity and pressure with BME280 . . . . .                               | 52        |
| 9.9      | Getting the temperature with MCP9808 . . . . .  | 52        |
| 9.10     | Getting the axis X, Y, and Z with ADXL345 . . . . .                                   | 53        |
| 9.11     | Conclusion . . . . .  | 53        |

|   |               |
|---|---------------|
| <b>10 Lesson 9 - Ultrasonic Sensor (Distance)</b>           | <b>55</b>     |
| 10.1 What do we need? . . . . .                             | 55            |
| 10.2 Experimental theory . . . . .                          | 55            |
| 10.3 Experimental procedure . . . . .                       | 56            |
| 10.4 Connecting remotely . . . . .                          | 56            |
| 10.5 Experimental code . . . . .                            | 57            |
| <br><b>11 Lesson 10 - LCD Display</b>                       | <br><b>59</b> |
| 11.1 What do we need? . . . . .                             | 59            |
| 11.2 Experimental theory . . . . .                          | 59            |
| 11.3 Experimental procedure . . . . .                       | 60            |
| 11.4 Connecting remotely . . . . .                          | 60            |
| 11.5 Experimental code . . . . .                            | 61            |
| <br><b>12 Lesson 11 - Building a Mini-Weather Station</b>   | <br><b>63</b> |
| 12.1 What do we need? . . . . .                             | 63            |
| 12.2 Experimental theory . . . . .                          | 63            |
| 12.3 Experimental procedure . . . . .                       | 64            |
| 12.4 Creating the ThingSpeak account . . . . .              | 64            |
| 12.5 Creating the application . . . . .                     | 64            |
| 12.6 Creating the Superclass . . . . .                      | 66            |
| 12.7 Creating the subclass DisplayLCD . . . . .             | 66            |
| 12.8 Creating the subclass PostData . . . . .               | 67            |
| 12.9 Starting the application . . . . .                     | 67            |
| 12.10 Visualizing your data . . . . .                       | 67            |
| <br><b>13 Lesson 12 - Building a Webserver on Raspberry</b> | <br><b>69</b> |
| 13.1 What do we need? . . . . .                             | 69            |
| 13.2 Experimental theory . . . . .                          | 69            |
| 13.3 Experimental procedure . . . . .                       | 69            |
| 13.4 HTML page . . . . .                                    | 70            |

# Illustrations

|     |   |    |
|-----|---|----|
| 1-1 | User interface of PiBackery. . . . .                        | 5  |
| 1-2 | User interface of Pharo. . . . .                            | 6  |
| 1-3 | Remote GPIO inspector . . . . .                             | 8  |
| 2-1 | LED polarity and resistors. . . . .                         | 11 |
| 2-2 | Breadboard scheme. . . . .                                  | 11 |
| 2-3 | Physical connection LED. . . . .                            | 12 |
| 2-4 | Remote Board Inspector. . . . .                             | 14 |
| 3-1 | Remote playground. . . . .                                  | 16 |
| 4-1 | Remote System Browser. . . . .                              | 18 |
| 4-2 | Creating a package remotely. . . . .                        | 19 |
| 4-3 | Creating a class remotely. . . . .                          | 20 |
| 4-4 | Creating a package remotely. . . . .                        | 20 |
| 4-5 | Looking for ID and GPIO number on Remote Inspector. . . . . | 21 |
| 4-6 | Creating the initialize method. . . . .                     | 22 |
| 4-7 | Creating an operation method. . . . .                       | 23 |
| 4-8 | Remote playground. . . . .                                  | 23 |
| 5-1 | Schema connection 8 LEDs. . . . .                           | 26 |
| 5-2 | Physical connection 8 LEDs. . . . .                         | 26 |
| 5-3 | Code on Inspector . . . . .                                 | 28 |
| 5-4 | LEDs turn On. . . . .                                       | 28 |
| 5-5 | Process Browser terminate. . . . .                          | 30 |
| 5-6 | Process Browser terminate. . . . .                          | 31 |
| 7-1 | Fading a led with PWM . . . . .                             | 38 |
| 8-1 | Controlling LED by touch button. . . . .                    | 43 |
| 9-1 | Devices connected using I2C bus. . . . .                    | 48 |
| 9-2 | Bits sequence. . . . .                                      | 49 |
| 9-3 | Physical sensors connection. . . . .                        | 50 |
| 9-4 | Inspecting remote object. . . . .                           | 51 |

Illustrations

10-1 Physical sensors connection. . . . . 57

11-1 Physical sensors connection. . . . . 60

12-1 ThingSpeak Channel Configuration. . . . . 65

12-2 Mini Weather Station code. . . . . 65

12-3 ThingSpeak Channel. . . . . 68

13-1 Web Page. . . . . 71





In this booklet we will show you how to develop a little application that collect weather information. We will start to show how we can play with leds and others.





# Installations

In this chapter you will learn how to:

- Run Pharo IoT in a Raspberry Pi that has Raspbian already installed;
- Install Pharo IoT and Raspbian from scratch in headless mode (without keyboard/mouse/screen);
- Run and use Pharo IoT IDE on your Linux, Windows or Mac OSX computer.

When you buy a new Raspberry Pi, the Operating System is not factory installed. The first step you need to do to start with Pharo IoT is to install an Operating System on your Raspberry Pi.

To maintain compatibility between this tutorial and your Raspberry, we recommend installing the Raspbian operating system. Raspbian is the Foundation's official supported operating system, a Linux OS based on Debian Stretch to run on ARM processors.

## 1.1 Running Pharo IoT in a Raspberry that has Raspbian

If you already have Raspbian running on your Raspberry Pi, you can simply use the Pharo IoT zero-conf. Open a terminal window in your Raspberry Pi (local or remote SSH) and enter the following commands:

```
[wget -O - get.pharoiot.org/server | bash
```

That way you will download the server side and extract the files to the pharoiot-server folder. Go to the folder `cd pharoiot-server` and run the `./pharo-server`.

```
[ cd pharoiot-server
  ./pharo-server
```

If everything is alright, you will see this message: 'a TlpRemoteUIManager is registered on port 40423'. This means that you have TelePharo running on your Raspberry on TCP port 40423.

Now you can use Pharo IoT on your computer to connect to your Raspberry and create IoT applications remotely.

If you still do not have Raspbian installed on your Raspberry Pi or want to know how to install everything in headless mode, the next procedure will explain how to do this.

## 1.2 Installing Raspbian and Pharo IoT from scratch (Raspberry Pi Headless Setup)

There are many options to install Raspbian on your Raspberry. The most common way is to download the ISO image from the official Raspberry website and follow the steps to install it. Basically, they are: copy an ISO image to an SD card, insert it in Raspberry Pi, turn On the Rasp and use a keyboard/-mouse/screen to use it as a normal computer.

But we will not use keyboard/mouse/screen to install Raspbian and run Pharo IoT! We do not need them.

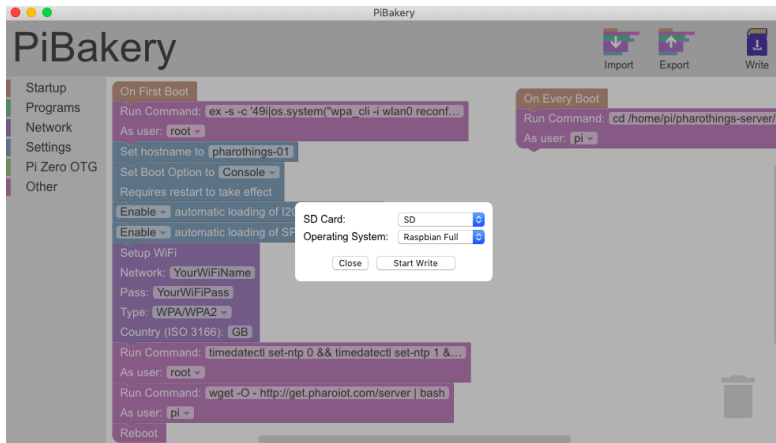
We will use a third-party program to perform these tasks automatically:

- Install Raspbian Full OS
- Setting the Raspberry Pi hostname
- Set boot to console
- Enable the I2C and SPI modules
- Connect it in your WiFi network
- Download Pharo IoT (requires Raspberry Pi connected on the internet)
- and start the Pharo IoT server at every boot

### PiBakery

- Download the PiBakery: <https://www.pibakery.org/download.html> and install it on your computer. It's a large file, around 2GB, because it has the Raspbian OS ready to install on your SD Card;
- Download the configuration file: <http://get.pharoiot.org/pibakeryPharoIoT.xml> and use the button Import in PiBakery to import these configurations;

### 1.3 Run Pharo IoT IDE on your Linux, Windows or Mac OSX computer.



**Figure 1-1** User interface of PiBakery.

- Change your hostname and WiFi configuration in PiBakery;
- Insert the SD card into your machine, click Write and select the Operation System Raspbian Full;
- After the process, insert the SD in the Raspberry and wait about 3 minutes to complete the automatic configuration. Time depends on the speed of your internet;
- You can now find your Raspberry by the Hostname you defined above.

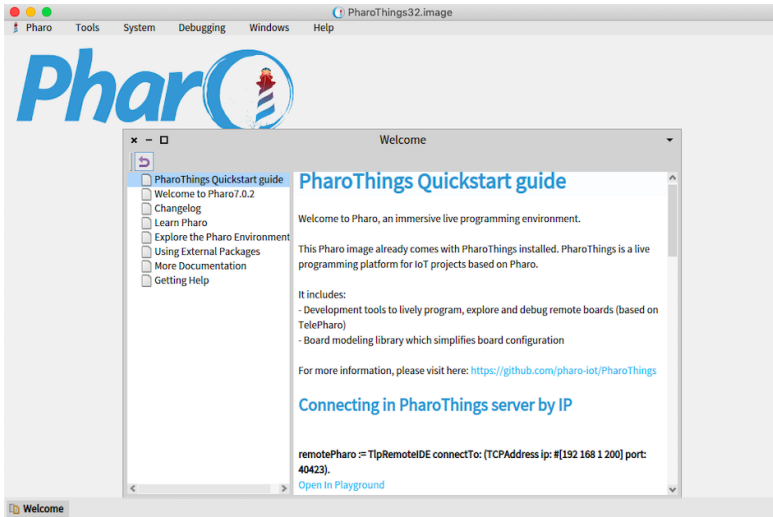
You do not need to do anything else on your Raspberry Pi. It is already loaded with Pharo IoT starting at every boot on the TCP port 40423. Now you can use the Pharo IoT client in your computer to connect in your Raspberry and create IoT applications remotely.

### 1.3 Run Pharo IoT IDE on your Linux, Windows or Mac OSX computer.

In this point, you should have the Pharo IoT running in your Raspberry Pi, as we saw in the previous tutorial. Now we will see how to run Pharo IoT IDE on Windows, Linux, and Mac. In this way, you can connect in you Raspberry using Pharo IoT IDE to write your applications on a remote environment.

#### Download

We think of everything to make your IoT journey so much easier. To use the Pharo IoT IDE, you just need to download one zip file, extract and run the



**Figure 1-2** User interface of Pharo.

application pharo-ui.

```
[get.pharo-iot.org/multi.zip
```

You can download this same file in your Windows, Linux or Mac OSX. After download, double click on pharo-ui (Linux, Mac) or pharo.bat (Windows).

## Pharo User Interface

When you run pharo-ui or pharo.bat, maybe some security warning will be shown. Just allow to run the program and Pharo will open as shown in the Picture 1-2;

You can see a quickstart guide window with some code to copy and past in the Playground. You can also click in Open in Playground below the code to open a new playground with that code.

## Connecting from your computer to Raspberry Pi

If you followed the tutorial to run Pharo IoT in you Raspberry, you should have now your Raspberry connected in your WiFi network running TelePharo in the TCP 40423 port.

Now with the server running in your Raspberry, you can connect your local Pharo in it. In your local machine, open the Playground and connect using the Raspberry IP address or the Hostname that you set before.

1.3 Run Pharo IoT IDE on your Linux, Windows or Mac OSX computer.

## Connecting in Pharo IoT server by Hostname

```
ip := NetNameResolver addressForName: 'pharoiot-01'.
remotePharo := TlpRemoteIDE connectTo: (TCPAddress ip: ip port:
    40423).
```

## Connecting in Pharo IoT server by IP

```
remotePharo := TlpRemoteIDE connectTo: (TCPAddress ip: #[192 168 1
    200] port: 40423).
```

## Working remotely

If you don't receive any error, this means that you are connected. Now you can call the Remote Playground, Remote System Browser, and Remote Process Browser:)

```
remotePharo openPlayground.
remotePharo openBrowser.
remotePharo openProcessBrowser.
```

## Inspect the remote Raspberry Pi GPIO board

You can inspect the physical board of your Raspberry Pi. So for your board model you need to choose an appropriate board class. For Raspberry, it will be one of the RpiBoard subclasses. Currently, you can use the following classes according to the models:

- RpiBoardBRev1: Raspberry Pi Model B Revision 1
- RpiBoardBRev2: Raspberry Pi Model B Revision 2
- RpiBoard3B: Raspberry Pi Model B+, Pi2 Model B, Pi3 Model B, Pi3 Model B+

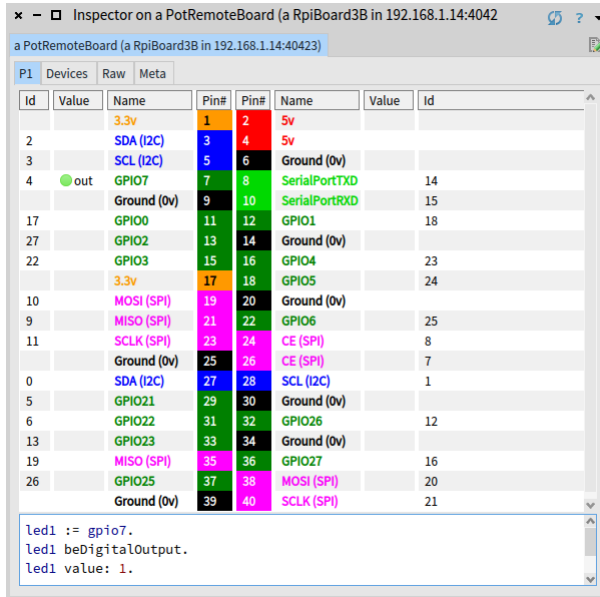
With the chosen class evaluate the following code to open an inspector:

```
remoteBoard := remotePharo evaluate: [ RpiBoard3B current].
remoteBoard inspect.
```

And will be open the inspector showing the PIN scheme, as shown in Figure 1-3

## GPIOs

The board inspector shows a layout of pins similar to Raspberry Pi docs. But here it is a live tool which represents the current pins state. The evaluation pane in the bottom of the inspector provides bindings to gpio pins which you can script by #doIt/printIt commands, as shown in Figure 1-3.



**Figure 1-3** Remote GPIO inspector

Digital pins are shown with green/red icons which represent high/low (1/0) values. In case of output pins you are able to click on the icon to toggle the value. Icons are updated according to pin value changes. If you click on physical button on your board the inspector will show the updated pin state by changing its icon color.

## Saving the remote image

```
[ remotePharo saveImage.
```

## Disconnect all remote sessions

```
[ TlpRemoteIDE disconnectAll.
```

Now that you are connected remotely in your Raspberry Pi, you can use sensors, buttons, LEDs, send data to the Cloud and create powerful applications.

## 1.4 In the next chapter

Now that we have installed the Operation System and Pharo IoT in the Raspberry Pi, we can play with LEDs, sensors, LCD displays and more.

In the next chapter we will see how turning on/off a LED using PharoThings.





# Lesson 1 – Turning LED on/off

One of the classic analogies in electronics to “Hello World” is turn on a LED (Light-Emitting Diode) or lamp. In this first lesson, we will learn how to connect correctly an LED to your Raspberry Pi and how to use PharoThings to control this LED by turning it on and off.

Coding in Pharo is very simple, but it is very powerful and you can control all the GPIOs of your Raspberry Pi remotely.

If you did not yet see how to install the PharoThings on your Raspberry Pi and how to control it remotely, you can find the instructions in Chapter 1.

## 2.1 What do we need?

In this lesson we will use a very simple setup.

### Components

- 1 Raspberry Pi connected to your network (wired or wireless)
- 1 Breadboard
- 1 LED
- 1 Resistor (330 ohms)
- Jumper wires

## 2.2 Experimental theory

Before constructing any circuit, you must know the parameters of the components in the circuit, such as their operating voltage, operating current, etc.

### The LED

LEDs (abbreviation of Light-Emitting Diode) is a semiconductor that produce light when current flows through it. Light is produced by an effect called electroluminescence.

To turn on the LED, we need to send the correct voltage and current to it. The voltage and current can't be too high, otherwise, the LED will burn, or in some cases, damage the Raspberry.

Typically, the forward voltage of an LED is between 1.8 and 3.3 volts. It varies by the color of the LED. A red LED typically drops 1.8 volts, but voltage drop normally rises as the light frequency increases, so a blue LED may drop from 3 to 3.3 volts[1].

Most 3mm and 5mm LEDs will operate close to their peak brightness at a drive current of 20 mA. This is a conservative current: it doesn't exceed most ratings (your specs may vary, or you may not have any specs—in this case, 20 mA is a good default guess [2]).

In this experiment, the operating voltage of the LED is between 1.5V and 2.0V and the operating current is between 10mA and 20mA.

### The Resistor

We must always use resistors to connect LEDs up to the GPIO pins of the Raspberry Pi to limit the voltage and current between the LED and the Raspberry to a safe value.

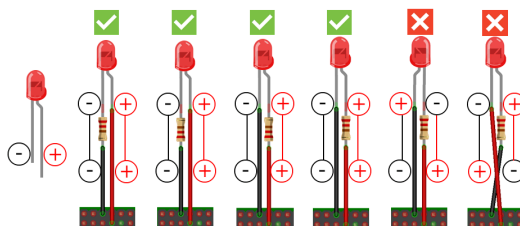
A small change in voltage can produce a huge change in current (see more: LED Current vs. Voltage [2])

In this experiment, we will use a 330ohm resistor. To identify the correct resistor, follow one of the following color sequences, depending on the number of bands [3]:

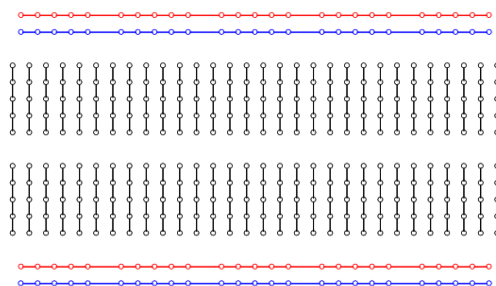
- If there are four colour bands, they will be Orange, Orange, Brown, and then Gold;
- If there are five bands, then the colours will be Orange, Orange, Black, Black, Brown.

It does not matter which way round you connect the resistors (in this experiment). Current flows in both ways through them. This means that you can connect the resistor at the positive pole or the negative pole of the LED, as well as starting with the first or last color, as shown in the Picture 2-1.

## 2.3 Experimental procedure



**Figure 2-1** LED polarity and resistors.



**Figure 2-2** Breadboard scheme.

But the LEDs will only work if the power is supplied correctly (if the polarity is correct). You will not burn the LEDs if they connect the wrong way – they just will not turn on.

### The Breadboard

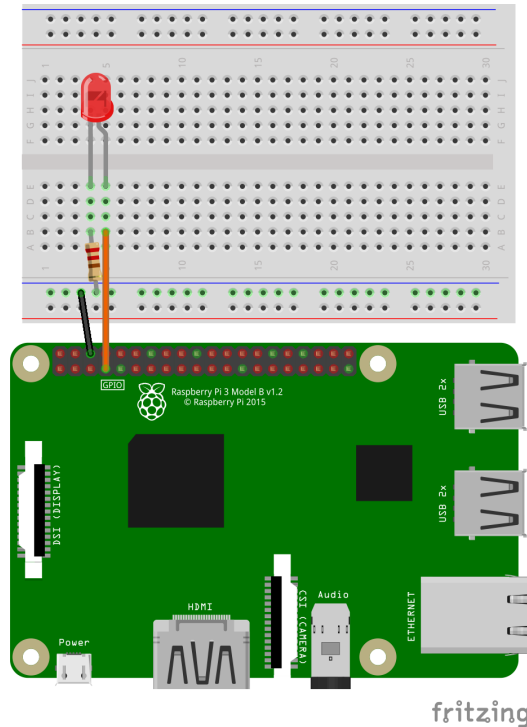
A breadboard is used to build prototyping of electronics. With a breadboard, it is not necessary to use solder, this way you can reuse the board. This makes it easy to use to create temporary prototypes and experiment with circuit design.

The holes in the breadboard are connected following a pattern, as shown in the Picture 2-2.

- The red (+) and blue (-) rail are connected horizontally;
- The holes in the middle are connected vertically.

## 2.3 Experimental procedure

Now we will build the circuit. This circuit consists of an LED that lights up when power is applied, a resistor to limit current and a power supply (the Rasp).



**Figure 2-3** Physical connection LED.

- Connect the Ground PIN from Raspberry in the breadboard blue rail (-). Raspeberry Pi models with 40 pins has 8 GPIO ground pins. You can connect with anyone. In this experiment we will use the PIN6 (Ground);
- Then connect the resistor from the blue rail on the breadboard (-) to a column on the breadboard, as shown in the Picture 2-3;
- Now push the LED legs into the breadboard, with the long leg (with the kink) on the right;
- And insert a jumper wire connecting the rigth column and the PIN7 (GPIO7).

The Figure 2-3 shows how the electric connection is made:

## 2.4 Experimental code

Now, we can write some code in Pharo to control the GPIOs using PharoThings and turn the LED on. We have 2 options to do this:

- Write the code locally on the Raspberry;
- Use TelePharo to connect from your computer into the Raspberry and do all the work remotely.

In this experiment, we will use the second option: connect and do all the work remotely.

If you didn't see how to install the PharoThings on your Raspberry Pi and how to control it remotely, take a look in the Chapter 1: Installations.

## Connecting remotely

Through your local Pharo image, let's connect in the Pharo image running on Raspberry, enable the auto-refresh feature of the inspector and open the inspector. Run this code in your local playground:

```
remotePharo := TlpRemoteIDE connectTo: (TCPAddress ip: #[193 51 236
    167] port: 40423)
GTInspector enableStepRefresh
remoteBoard := remotePharo evaluate: [ RpiBoard3B current].
remoteBoard inspect.
```

In your inspect window (Inspector on a PotRemoteBoard) you can see a scheme of pins similar to the Raspberry Pi docs. But here it is a live tool which represents the current pins state.

Digital pins are shown with green/red icons which represent high/low (1/0) values. In case of output pins you are able to click on the icon to toggle the value.

To control the LED we first introduce the named variable #led which we assigned to GPIO7 pin instance:

```
[ led := gpio7.
```

Then we configure the pin to be in digital output mode and set the value:

```
[ led beDigitalOutput.
  led value: 1.
```

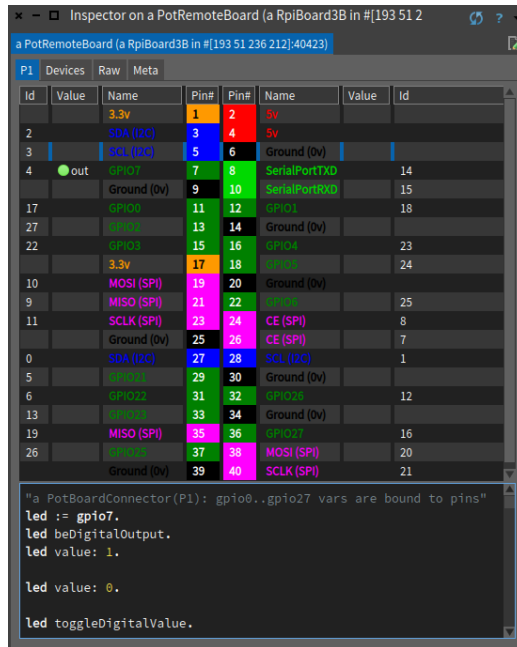
It turns the LED on.

You can notice that gpio variables are not just numbers/ids. PharoThings models boards with first class pins. They are real objects with behaviour. For example you can ask pin to toggle a value:

```
[ led toggleDigitalValue.
```

Or ask a pin for current value if you want to check it:

```
[ led value.
```



**Figure 2-4** Remote Board Inspector.

## 2.5 What did we learn?

With PharoThings you can remotely control all the GPIOs in your running board!

You can:

- Interact remotely with pins and boards;
- See the current pins state in real time;
- Run the code dynamically;
- Easy, powerful.

## 2.6 In the next lesson

Let's use what we learned in this lesson and write a simple code to blink the LED.



## Lesson 2 – Blinking LED

Now we can play with the LEDs, turn them on and off. Let's use this basic setup to write some code on the inspector playground to blink the LED. Next, we will learn how to remotely create a very simple application using classes, methods, and instances to control the LED.

### 3.1 What do we need?

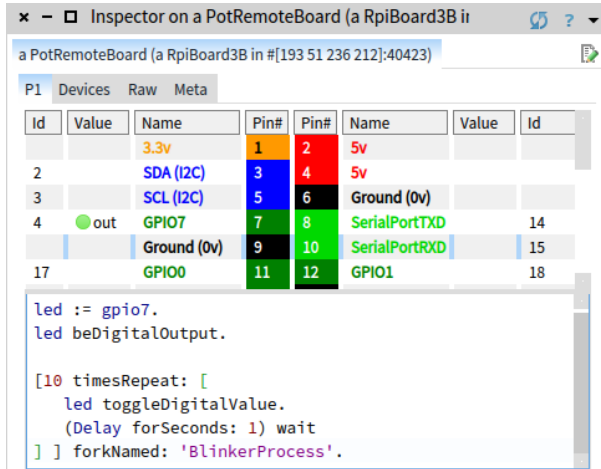
We are using the same setup as the last section: 1 Raspberry Pi, 1 Breadboard, 1 LED, 1 Resistor 330ohms. If you didn't do the last lesson to understand how to do the connections, go back to Chapter 2 and do it.

#### Connecting remotely

Through your local Pharo image, let's connect to the Pharo image by running on Raspberry, enable the auto-refresh feature of the inspector, and open the inspector.

Run this code in your local playground:

```
remotePharo := TlpRemoteIDE connectTo: (TCPAddress ip: #[193 51 236  
    212] port: 40423)  
GTInspector enableStepRefresh.  
remoteBoard := remotePharo evaluate: [ RpiBoard3B current].  
remoteBoard inspect.
```



**Figure 3-1** Remote playground.

### 3.2 Experimental code

In your inspect window (Inspector on a PotRemoteBoard), let's initialize the led and set the pin 7 to be in digital output mode as we did in the last lesson:

```
led := gpio7.
led beDigitalOutput.
```

To blink the LED let's create a simple loop to change the value of the LED every 1 second by 10 times. To change the value of the object (led value), let's call the method `toggleDigitalValue`, as we saw previously:

```
[ 10 timesRepeat: [
  led toggleDigitalValue.
  (Delay forSeconds: 1) wait
] ] forkNamed: 'BlinkerProcess'.
```

Run this code, as shown in Figure 4-5 and... cool! Now your LED is blinking!

Change the values to repeat more times and to wait less time between toggling. This will cause the LED to blink faster.

### 3.3 In the next lesson

In this tutorial, you learned how to blink a LED by typing some code in the remote inspector. But with Pharo we can do more! And in the next lesson, let's start to use OOP (object-oriented programming). Let's create a simple application, write classes and methods, all remotely.





## Lesson 3 – A brief introduction to Pharo object-oriented language

Pharo is a new generation reflective language and programming environment. The last code was executed inside the remote inspector. To get started using OOP (Object-Oriented Programming) with classes, methods, and instances, I invite you to implement a simple application to blink the LEDs.

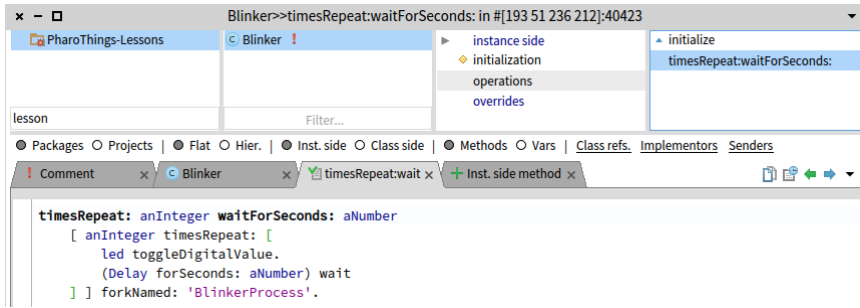
### 4.1 Developing a simple LED blinker

The following part of this chapter and application was based on the exercise Developing a Simple Counter, of Week 1 of Pharo MOOC (<https://mooc.pharo.org/>). I strongly recommend that you read and do the "counter exercise" to better understand the concepts explained here. And, of course, do the MOOC to learn how to develop using Pharo and the OOP concept.

### 4.2 Our use case

We want to create a blinker LED using a few parameters such as time to repeat the blinking LED and how many seconds to wait between blinks. The following code should run in the playground when we finish this lesson:

```
[|blinker|  
  blinker := Blinker new.  
  blinker timesRepeat: 10 waitForSeconds: 1.
```



**Figure 4-1** Remote System Browser.

Here is a short explanation of this code:

- In the first line, we declare the variable `blinker`. We can use any name. We will use this variable to create an object using the `Blinker` class;
- In the second line, we instantiate the `Blinker` class (with uppercase B) in the `blinker` variable, creating an object. In this lesson, we will create this class and methods to control the LED;
- In the third line, we send some messages to the `blinker` object, for how long and how many times per second. This will make the GPIO behave according to the parameters sent.

Now we will develop all the mandatory class and methods to support this scenario.

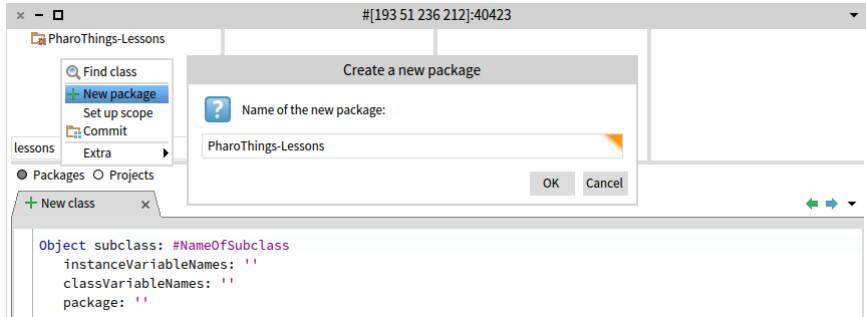
### 4.3 Create your own class remotely

Let's create our first class. To create a class in Pharo, we need first to create a package. Inside the package, you can create many classes and inside the classes, you can create many methods. The methods are organized in protocols, to become more easily navigate between them. Take a look in Figure 4-1 to better understanding. \*edit image, put name in windows

In your local playground, call the Remote System Browser of your Raspberry Pi. If you are already connected to your Raspberry Pharo, you do not need to run the first line below again. This will open a window as shown in Figure 4-1

```
remotePharo := TlpRemoteIDE connectTo: (TCPAddress ip: #[193 51 236
    212] port: 40423).
remotePharo openBrowser.
```

## 4.4 Create a package



**Figure 4-2** Creating a package remotely.

## 4.4 Create a package

Let's create a package using the Remote Browser. Right-click the package area and enter the package name, as shown in Figure 4-2. In this example, we will create a package named `PharoThings-Lessons`.

## 4.5 Create a class

To create a new class, edit the default class template by changing the `#NameOfSubclass` to the name of the new class. In this example let's create the class `#Blinker`. Take care that the class name begins with a capital letter and that you do not remove the hash symbol (`#`) in front of `NameOfSubclass`.

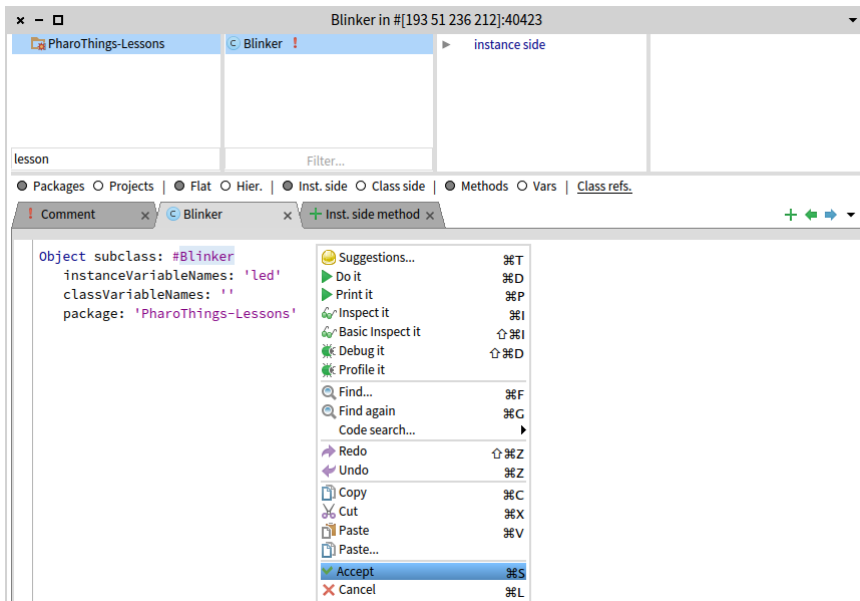
You must then fill in the names of the instance variables for this class. We need an instance variable called `led`. Be careful to leave the string quotes!

```
Object subclass: #Blinker
  instanceVariableNames: 'led'
  classVariableNames: ''
  package: 'PharoThings-Lessons'
```

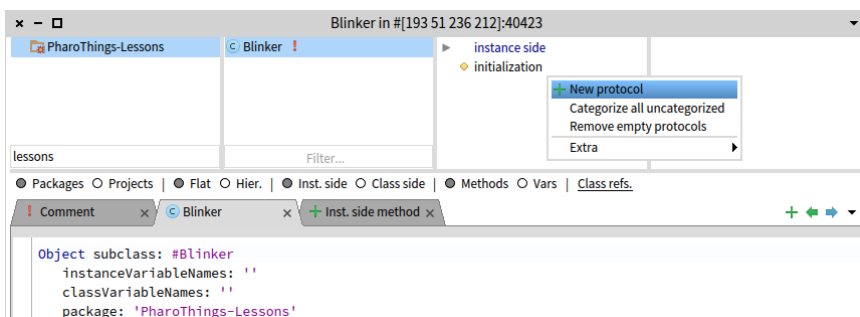
Now we need to compile it. Right click on the code area and select `Accept` option. The `Blinker` class is now compiled and added to the system, as shown in Figure 4-3.

## 4.6 Create a protocol

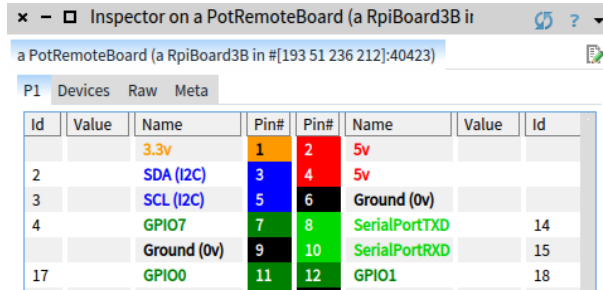
Let's create a new protocol to organize the methods. The first protocol we are going to create is `initialization`, as shown in Figure 4-4.



**Figure 4-3** Creating a class remotely.



**Figure 4-4** Creating a package remotely.



The screenshot shows a web-based inspector for a PotRemoteBoard (a RpiBoard3B in #[193 51 236 212]:40423). It displays a table of GPIO pins with their IDs, values, names, and pin numbers.

| Id | Value | Name        | Pin# | Pin# | Name          | Value | Id |
|----|-------|-------------|------|------|---------------|-------|----|
|    |       | 3.3v        | 1    | 2    | 5v            |       |    |
| 2  |       | SDA (I2C)   | 3    | 4    | 5v            |       |    |
| 3  |       | SCL (I2C)   | 5    | 6    | Ground (0v)   |       |    |
| 4  |       | GPIO7       | 7    | 8    | SerialPortTXD |       | 14 |
|    |       | Ground (0v) | 9    | 10   | SerialPortRXD |       | 15 |
| 17 |       | GPIO0       | 11   | 12   | GPIO1         |       | 18 |

**Figure 4-5** Looking for ID and GPIO number on Remote Inspector.

## 4.7 Creating an initialize method

Inside this protocol, we will create an `initialize` method. This means that every time we create a new object using this class, in this case, the `Blinker` class, this method will be executed to define some variable in the new object.

Let's use the instance variable `led`, which we defined when we created the class. The instance variable is private to the object and accessible by any methods inside this class. These methods can access this variable to get or set any value to it.

```
initialize
  led := PotClockGPIOPin id: 4 number: 7.
  led board: RpiBoard3B current; beDigitalOutput
```

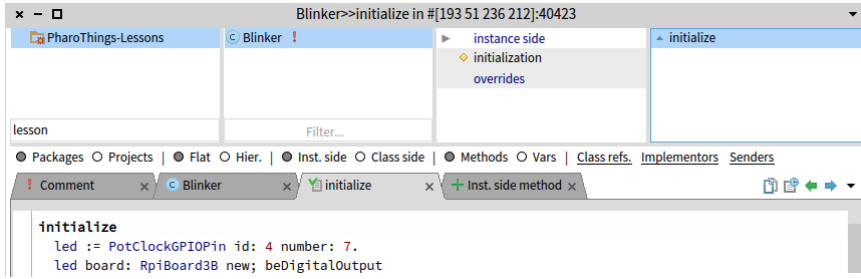
Here is a short explanation of this code:

- The first line defines the name of the method;
- In the second line, we configure the GPIO that we wanna use. Note that we need the GPIO number and ID. The ID is required to communicate with Wiring Pi Library. You can see the ID and GPIO number in PotRemoteBoard inspector, as shown in Figure 4-5;
- In the third line, we define the model of the Raspberry board and configure this GPIO as `beDigitalOutput`. This means that when the GPIO change to value:1, the power will go out of the GPIO to power the LED.

Compile your code (cmd + S) and the method will be shown in the remote browser, as shown in Figure 4-6:

### Creating a method to do actions

Now let's create a method to control the object `led` inside the class `Blinker`. Let's take the code that we used in PotRemoteBoard inspector to do the LED



**Figure 4-6** Creating the initialize method.

to blink and replace the numbers on code for two arguments. Create the protocol operations and inside this protocol, create the following method:

```
[ timesRepeat: anInteger waitforSeconds: aNumber
  [ anInteger timesRepeat: [
    led toggleDigitalValue.
    (Delay forSeconds: aNumber) wait
  ] ] forkNamed: 'BlinkerProcess'.
```

Here is a short explanation of this code:

- In the first line, we define the message with timesRepeat: and waitforSeconds:. We inform the kind of value will be received, creating 2 variables: aNumber and anInteger;
- We replace these variables in the code and now we have the control to say how many times repeat and for how many seconds;
- We finished the code by putting everything inside a fork to create a process in Pharo. While the process is running, you can open the Remote Process Browser (remotePharo openProcessBrowser) and see the process. This is useful when you wanna kill the remote process.

Compile your code (cmd + S) and the method will be shown in the remote browser, as shown in Figure 4-7:

## 4.8 Using your new class

Now we can use the class that we created, the Blinker class. To do this, let's open the Remote Playground:

```
[ remotePharo openPlayground.
```

and run the code that we saw in the begin of this lesson:

## 4.9 Save your work

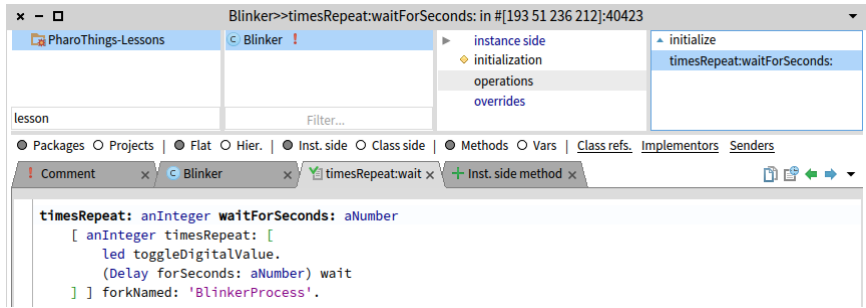


Figure 4-7 Creating an operation method.

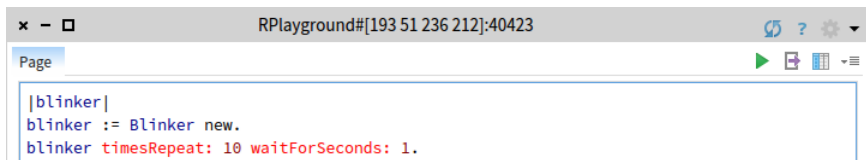


Figure 4-8 Remote playground.

```
[|blinker|
blinker := Blinker new.
blinker timesRepeat: 10 waitForSeconds: 1.
```

Run this code, as shown in Figure 4-8 and... cool! Now your LED is blinking! And the better, you did this using object-oriented programming!

You do not need to change your code every time you wanna change these parameters. Just change the messages you send to the object and it will behave as you want.

## 4.9 Save your work

Don't forget to save your work remotely. To do this, run this command on your local playground:

```
[remotePharo saveImage.
```

## 4.10 Conclusion

In this tutorial, you learned how to define packages, classes, and methods. The flow of programming that we chose for this first tutorial is similar to most of the programming languages.

With PharoThings you can remotely develop and manage your Raspberry GPIOs. Very easy and powerful.

In the next lesson, let's use what we learned in this lesson and write a simple code to flow lights using 8 LEDs.





## Lesson 4 - LED Flowing Lights

Now we can play with the LEDs, turn them on, off, and blink. Let's put 8 LEDs on the breadboard and create a code to turn on/off one at a time. Let's use some methods to change the flow direction and control the flow time. As we did in the last lesson, let's write the first code in playground and then create a class with methods to better control the flow of LED lights.

### 5.1 What do we need?

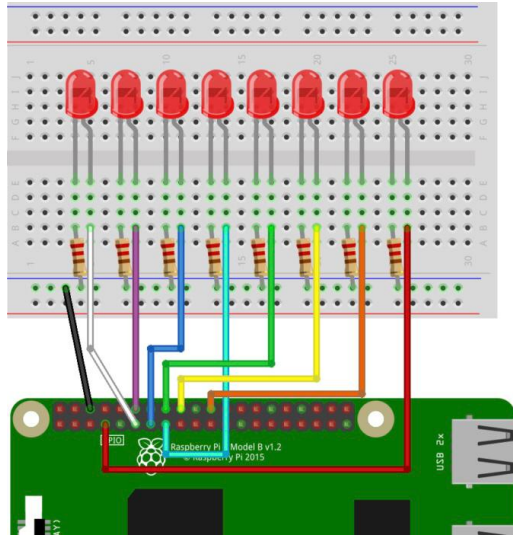
We are using the set of the first lesson, but let's use 8 LEDs and 8 resistors and some more jumper wires.

#### Components

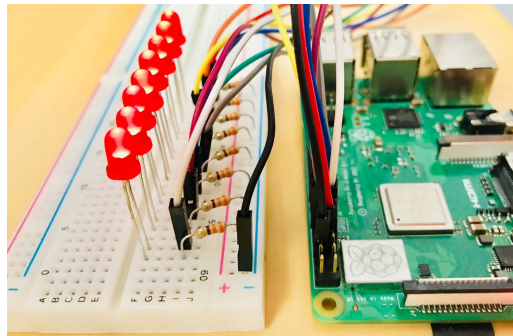
- 1 Raspberry Pi connected to your network (wired or wireless)
- 1 Breadboard
- 8 LEDs
- 8 Resistors 330ohms
- Jumper wires

### 5.2 Experimental procedure

We saw in lesson 1 how to connect the LED and resistors on the breadboard. Now let's do the same, but putting more 7 LEDs and resistors on the breadboard.



**Figure 5-1** Schema connection 8 LEDs.



**Figure 5-2** Physical connection 8 LEDs.

- Connect the Ground PIN from Raspberry in the breadboard blue rail (-).
- Then connect the 8 resistors from the blue rail (-) to a column on the breadboard, as shown below;
- Now push the LED legs into the breadboard, with the long leg (with the kink) on the right;
- And insert the jumper wires connecting the right column of each LED to GPIO from 0 to 7, as shown in the Picture 5-1.

The Figure 5-2 shows how the electric connection is made.

## Connecting remotely

Through your local Pharo image, let's connect in the Pharo image by running on Raspberry, enable the auto-refresh feature of the inspector, and open the inspector.

Run this code in your local playground:

```
remotePharo := TlpRemoteIDE connectTo: (TCPAddress ip: #[193 51 236
    212] port: 40423)
GTInspector enableStepRefresh.
remoteBoard := remotePharo evaluate: [ RpiBoard3B current].
remoteBoard inspect.
```

## 5.3 Experimental code

In your inspect window (Inspector on a PotRemoteBoard), let's create an array and initialize the 8 LEDs, putting each one in a position of the array. This way we can send messages more easily to all objects. Look at the second line, we set the GPIOs to beDigitalOutput only using the method do: to move through the entire array:

```
gpioArray := { gpio0. gpio1. gpio2. gpio3. gpio4. gpio5. gpio6.
    gpio7 }.
gpioArray do: [ :item | item beDigitalOutput ].
```

To change the value of the object (led value), let's call the method toggleDigitalValue, as we saw previously. You can also use the method value: and send 1 or 0, instead toggleDigitalValue, but let's use this last. To do this fast and simple, let's use again the method do: to send the parameters to all objects on the array. In this example, we turn On all the LEDs at the same time:

```
gpioArray do: [ :item | item toggleDigitalValue ].
```

Let's put a Delay after changing the led value, to wait a bit time before to change the next LED value. Let's also put this inside a process using the method forkNamed:

```
[
    gpioArray do: [ :item | item toggleDigitalValue. (Delay
        forSeconds: 0.3) wait ].
] forkNamed: 'FlowingProcess'.
```

Execute this code and... cool! Now your LEDs are on by flowing an ordering!

Change the value of the method forSeconds: to wait less time between toggling it. This will cause the line LEDs to turn on faster. The Figure 5-3 and 5-4 shows the code and the LEDs turn On.

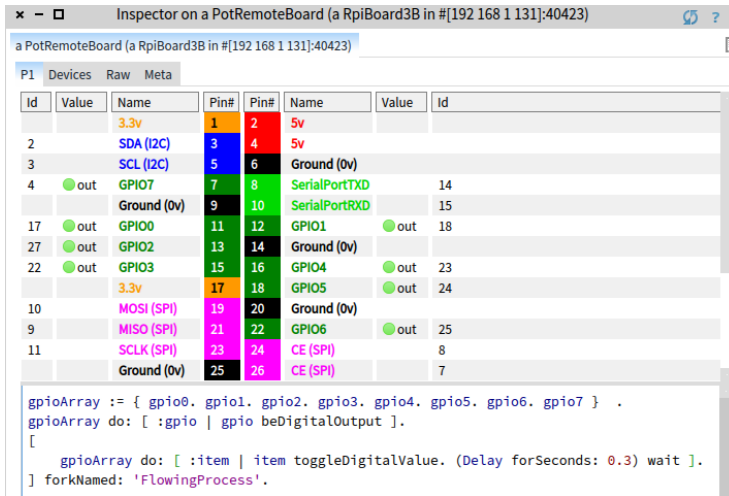


Figure 5-3 Code on Inspector

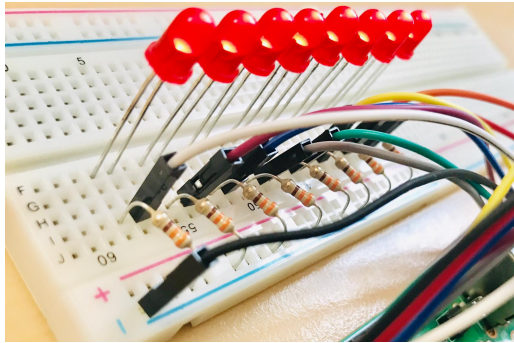


Figure 5-4 LEDs turn On.

## 5.4 Adding features

Every time you run this code, the LEDs toggles the state, from Off to On or vice versa. Let's reduce the delay time and add the `timesRepeat:` method, as we did in the last lesson, to repeat the alternation as many times as we want:

```

[ 2 timesRepeat: [
    gpioArray do: [ :item | item toggleDigitalValue. (Delay
forSeconds: 0.1) wait ].
] ] forkNamed: 'FlowingProcess'.

```

Execute this code and... cool! Now your LEDs are flowing On and Off!

## 5.5 Reversing the flow

We can have more fun with this experiment by changing the order of where to start changing the value of LEDs. To do this is very easy, just call the method `reverseDo:` and it will solve all for you:

```
[ 2 timesRepeat: [
  gpioArray reverseDo: [ :item | item toggleDigitalValue. (Delay
    forSeconds: 0.1) wait ].
] ] forkNamed: 'FlowingProcess'.
```

Execute this code and... cool! Now your LEDs are flowing on reverse order!

## 5.6 Going and backing the flow

To finish this experiment, let's combine the flowing On and Off with the Reverse!

```
[ 2 timesRepeat: [
  gpioArray do: [ :item | item toggleDigitalValue. (Delay
    forSeconds: 0.1) wait ].
  gpioArray reverseDo: [ :item | item toggleDigitalValue. (Delay
    forSeconds: 0.1) wait ].
] ] forkNamed: 'FlowingProcess'.
```

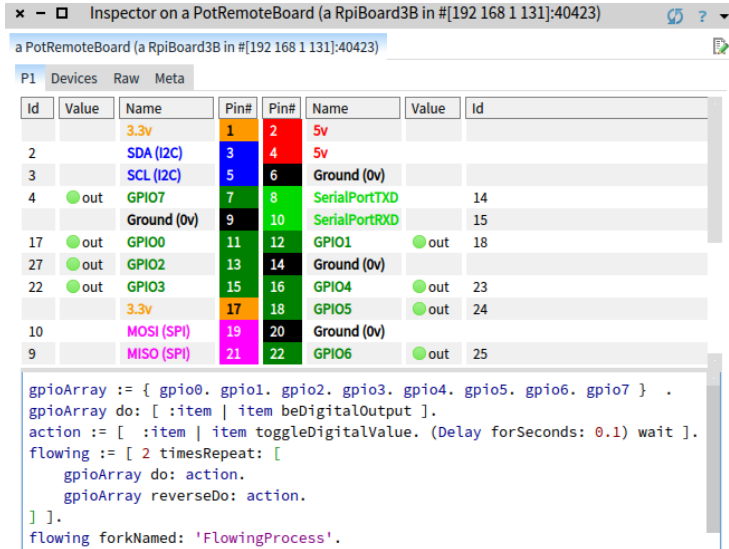
Execute this code and... cool! Now your LEDs are flowing On and Off and on normal and reverse order!

We can improve this code. Do you see this part where the code is repeating `"[ :item | item toggleDigitalValue. (Delay forSeconds: 0.1) wait ]"`? Let's put this inside a variable named `action`, so we can call it when we want:

```
action := [ :item | item toggleDigitalValue. (Delay forSeconds: 0.1)
  wait ].
[ 2 timesRepeat: [
  gpioArray do: action.
  gpioArray reverseDo: action.
] ] forkNamed: 'FlowingProcess'.
```

We can put the code inside the block closure `"[]"` in a variable also and call it in just one line. Let's put it inside the variable `flowing`:

```
action := [ :item | item toggleDigitalValue. (Delay forSeconds: 0.1)
  wait ].
flowing := [ 2 timesRepeat: [
  gpioArray do: action.
  gpioArray reverseDo: action.
] ].
```



**Figure 5-5** Process Browser terminate.

Now we can start the process just send the method `forkNamed:` to the object `flowing`, like in the following line:

```
[flowing forkNamed: 'FlowingProcess'.
```

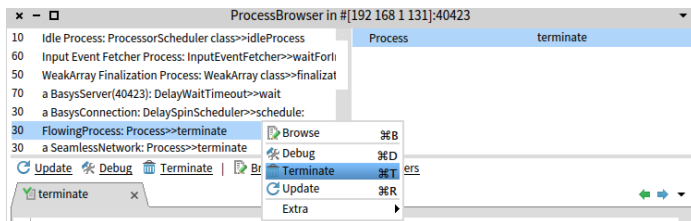
Your final code will seem like the Picture 8-1. Run this code once and when you want to flow the LEDs again, just run the last line. But remember, to each change in the code, you need to run the part that you changed.

```
gpioArray := { gpio0. gpio1. gpio2. gpio3. gpio4. gpio5. gpio6.
    gpio7 }.
gpioArray do: [ :item | item beDigitalOutput ].
action := [ :item | item toggleDigitalValue. (Delay forSeconds: 0.1)
    wait ].
flowing := [ 2 timesRepeat: [
    gpioArray do: action.
    gpioArray reverseDo: action.
] ].
flowing forkNamed: 'FlowingProcess'.
```

## 5.7 Terminating the process

As we saw in the Blinking LED lesson, you can finish this process remotely, case you don't want to wait it finish. To do this, call the Remote Process Browser:

## 5.8 In the next lesson



**Figure 5-6** Process Browser terminate.

```
[ remotePharo openProcessBrowser.
```

Search the `FlowingProcess` and terminate it, like in Picture 5-6, using one of these options:

- selecting the process and using the shortcut “Cmd + T”;
- selecting the process and using the button `Terminate`;
- or right-click and select `Terminate`.

## 5.8 In the next lesson

In this tutorial, you learned how to use an Array and control 8 objects at the same time by typing some code in the remote inspector. But with Pharo we can do more!

You can create your own program in classes and methods using the codes you learned in this lesson. Go ahead and try to do this yourself to test your knowledge.

And in the next lesson, let’s use object-oriented programming, OOP to create a simple program, using these codes, to control the flow like as we want.







## Lesson 5 - LED Flowing Lights using OOP

Now we can play with the LEDs, turn them on, off, blink it and manipulate many at the same time. Let's use object-oriented programming, OOP to create methods and classes, to build a simple program, to control the LEDs flow like as we want.

### 6.1 The goal

In this chapter we will create a program to control the LEDs with some features:

- We want to set how many times the LED line will flow;
- We want to set the speed of this flow;
- And we want to set the direction if the flow will start from left or right and which order will be executed. For example, if we want to set to flow starting from the right and after the start from left again, we will use the message 'lrl'. The program needs to be able to flow in any combination of directions, like llr, rllrr, rlrllr etc.

The final code executed in the playground will be a code like this:

```
 leds := Flowing new.  
 leds times: 2 delay: 0.1 direction: 'lrl'.  
 leds flowStart.  
 leds flowStop.  
 leds turnOn.  
 leds turnOff.
```

## 6.2 Creating the class

```
Object subclass: #Flowing
  instanceVariableNames: 'ledArray flowProcess flowDirection
    toggleDelay timesRepeat'
  classVariableNames: ''
  package: 'PharoThings-Lessons'
```

## 6.3 Creating the initialize method

```
initialize
  ledArray := {
    (PotGPIOPin id: 17 number: 0).
    (PotGPIOPin id: 18 number: 1).
    (PotGPIOPin id: 27 number: 2).
    (PotGPIOPin id: 22 number: 3).
    (PotGPIOPin id: 23 number: 4).
    (PotGPIOPin id: 24 number: 5).
    (PotGPIOPin id: 25 number: 6).
    (PotGPIOPin id: 4 number: 7)
  }.
  ledArray do: [ :item | item board: RpiBoard3B current;
    beDigitalOutput; value:0 ].
  timesRepeat := 2.
  toggleDelay := 0.5.
  flowDirection := 'lr'.
```

## 6.4 Creating access methods

```
times: anInteger delay: aNumber direction: aString
  timesRepeat := anInteger.
  toggleDelay := aNumber.
  flowDirection := aString

flowDirection
  ^flowDirection

flowTimesRepeat
  ^timesRepeat

toggleDelay
  ^toggleDelay
```

## 6.5 Creating process control methods

```

[ flowStart
  flowProcess := [ (self flowTimesRepeat) timesRepeat: [
    self action
  ] ] forkNamed: 'FlowingProcess'.

[ flowStop
  flowProcess terminate

```

## 6.6 Creating action methods

```

[ action
  flowDirection do: [ :character | character == $l ifTrue: [ ledArray
    do: self toggleLedArray ].
    character == $r ifTrue: [ ledArray
    reverseDdo: self toggleLedArray ] ]

[ toggleLedArray
  ^[ :item | item toggleDigitalValue. (Delay forSeconds: self
    toggleDelay) wait ]

[ turnOn
  ledArray do: [ :item | item value:1 ].

[ turnOff
  ledArray do: [ :item | item value:0 ].

```





## Lesson 6 - LED Breathing PWM

**PWM support is currently broken in PharoThings.**

In a previous lesson, we learned how to switch on and switch off a LED. How can we fade a LED now ? In order to do that, we can do a simulation of an analog pin with a digital pin. This technique is known as PWM (Pulse-Width modulation). See more details on Wikipedia page here: [https://en.wikipedia.org/wiki/Pulse\\_width\\_modulation](https://en.wikipedia.org/wiki/Pulse_width_modulation)

### 7.1 What do we need ?

We are using the same set as the first lesson:

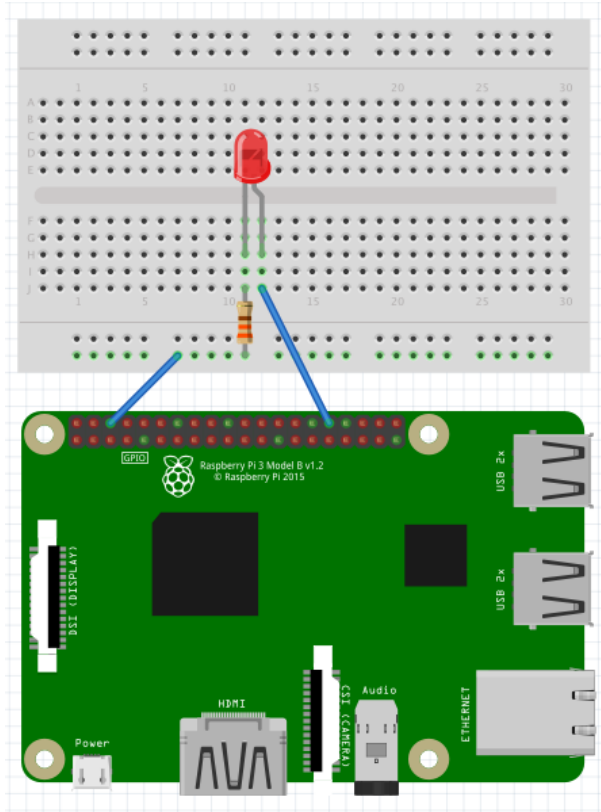
#### Components

- 1 Raspberry Pi connected to your network (wired or wireless)
- 1 Breadboard
- 1 LED
- 1 Resistor 330ohms
- Jumper wires

#### Experimental theory

The Raspberry Pi GPIO pin can be set in PWM mode. The GPIO1 pin is already setup as a PWM output. We will connect the LED to this pin instead of GPIO7.

**Figure need to be change, to use GPIO1 instead of GPIO26.**



**Figure 7-1** Fading a led with PWM

## 7.2 Experimental procedure

## 7.3 Connecting remotely

## 7.4 Experimental code

In your inspect window (Inspector on a PotRemoteBoard), let's initialize the led and set the pin GPIO26 to be in PWM output mode as we did in the last lesson:

```
[led := gpio26.  
led bePWMOutput.
```

Try to change the fading of the LED like that:

```
[led writePWMValue: aValue
```

## 7.5 In the next lesson

aValue could be between 0 and 1023. Try to wrote a program that will fade the LED progressively from on to off.

## 7.5 In the next lesson







## Lesson 7 - Controlling LED by Button

In the previous lessons, we learned how to control the GPIOs putting them in mode OUT. This means send power to wire connected in on specific GPIO. Now we will put the GPIO in mode IN, to read the pin state. This means that our application can know when a button is pressed. Let's create code on the remote inspector to turn on one LED each time we press the button.

### 8.1 What do we need?

We are using the set of the first lesson, and we add one pushbutton.

#### Components

- 1 Raspberry Pi connected to your network (wired or wireless)
- 1 Breadboard
- 1 LED
- 1 Resistor 330ohms
- 1 Pushbutton
- Jumper wires

## 8.2 Experimental theory

The Raspberry Pi GPIOs can be set in OUT or IN mode. When we set them to IN, the selected pin can have change the state from 0 to 1 or vice-versa when we connect it in the 3.3V or ground PIN.

In this lesson, we will use the pull-up mode with an internal resistor. This means that we will use the 3.3V pin to change the GPIO value. If we used the pull-down mode, we would use the Ground Pin to change the GPIO value. To know more, you can access the Wiring Pi website, <http://wiringpi.com/reference/core-functions>.

In this scenario, we will put a push button between the 3.3V pin and GPIO pin, to control when to send energy to GPIO.

## 8.3 Experimental procedure

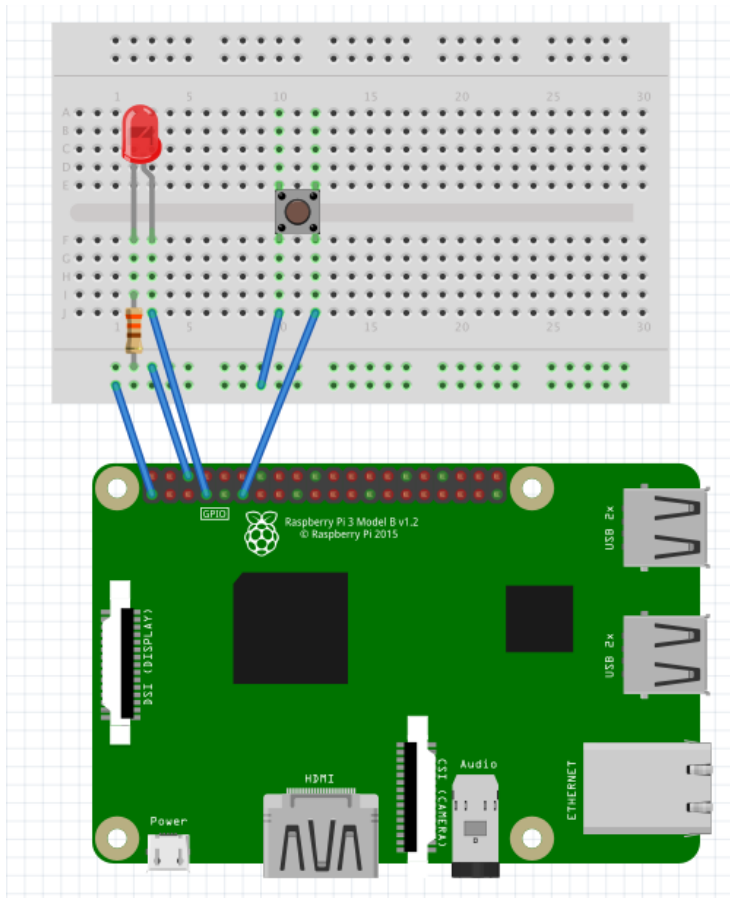
Now we will build the circuit. This circuit consists of an LED, a resistor to limit current and a push button, to control when to send power to GPIO IN.

- Connect the Ground PIN from Raspberry in the breadboard blue rail (-). In this experiment we will use the PIN6 (Ground);
- Then connect the resistor from the same row on the breadboard to a column on the breadboard, as shown below;
- Now push the LED legs into the breadboard, with the long leg (with the kink) on the right;
- And insert a jumper wire connecting the right column and the PIN7 (GPIO7);
- Connect the 3.3V PIN in the red rail (+);
- Then insert the push button on the breadboard and connect one leg on PIN11 (GPIO0);
- And another button leg on the red rail (+). Pay attention to the position of the button!

The figure below shows how the electric connection is made:

## 8.4 Connecting remotely

Through your local Pharo image, let's connect in the Pharo image by running on Raspberry, enable the auto-refresh feature of the inspector, and open the inspector. Run this code in your local playground:



**Figure 8-1** Controlling LED by touch button.

```

remotePharo := TlpRemoteIDE connectTo: (TCPAddress ip: #[193 51 236
    212] port: 40423)
GTInspector enableStepRefresh.
remoteBoard := remotePharo evaluate: [ RpiBoard3B current].
remoteBoard inspect.

```

## 8.5 Experimental code

In your inspect window (Inspector on a PotRemoteBoard), let's create the instances of the LED and button.

To instantiate the LED, let's do how we did previous, putting it in beDigitalOutput mode:

```
[ led := gpio7
  led beDigitalOutput.
```

To instantiate the button let's do the same, but putting it in beDigitalInput and setting it to enablePullUpResister mode:

```
[ button := gpio0.
  button beDigitalInput.
  button enablePullUpResister.
```

This means each time that we connect the 3.3V on this GPIO (pushing the button), the state will be changed to 1, and back to 0 after release the button.

You ask the value of the button using the value method. Do this test running this line with the button pressed and after with the button released. This test is good for you to check if your button is working correctly:

```
[ button value.
```

Now let's create a process to check when the button is pressed and send this value to led object:

```
[ [ 100 milliSeconds wait.
    led value: (button value=1) asBit
  ] repeat
  ] forkNamed: 'button process'.
```

Your LED will turn on when the button is pressed!

## 8.6 Terminating the process

As we saw in the Blinking LED lesson, you can finish this process remotely, case you don't want to wait it finish. To do this, call the Remote Process Browser:

```
[ remotePharo openProcessBrowser.
```

Search the FlowingProcess and terminate it:

- selecting the process and using the shortcut “Cmd + T”;
- selecting the process and using the button Terminate;
- or right-click and select Terminate.

## 8.7 In the next lesson

In this lesson, you learned how to configure a push button, take the value of the button and send it to LED value, doing the LED turn On or Off using the button.

## 8.7 In the next lesson

Next lesson we will start to use the sensors using the I2C protocol. We will see the BME280 (temperature, humidity, and air pressure), ADXL (accelerometer X, Y, Z) and MCP9808 (temperature).





## Lesson 8 - I2C Sensors (Temperature, Humidity, Pressure and Accelerometer)

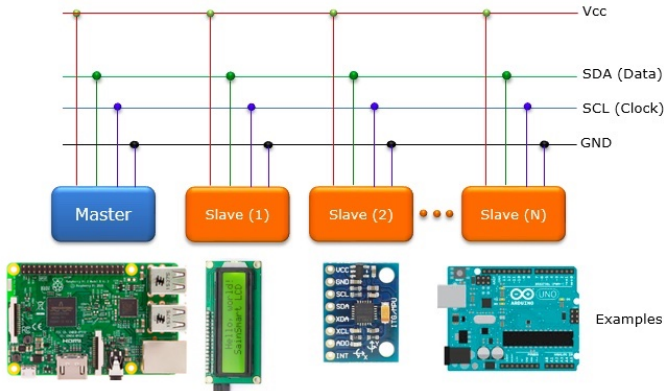
In the previous lessons, we learned how to control LEDs and to use a button to interact with LEDs. Now let's start using some sensors to interact automatically with the real world, taking the temperature, air pressure and humidity. This kind of sensor uses the I2C protocol to communicate.

### 9.1 What do we need?

In this lesson, we will use a setup with 3 different I2C sensors.

#### Components

- 1 Raspberry Pi connected to your network (wired or wireless)
- 1 Breadboard
- 1 BME280 temperature, humidity and pressure sensor
- 1 MCP9808 temperature sensor
- 1 ADXL345 accelerometer sensor
- Jumper wires



**Figure 9-1** Devices connected using I2C bus.

## 9.2 Experimental theory

Before constructing any circuit, you must know the parameters of the components in the circuit, such as their operating voltage, operating circuit, etc.

### The I2C protocol

The I2C communication protocol can be easily implemented in many electronic projects, being a very popular and widely used protocol. It is possible to perform communication between one or more master devices and several slave devices. It is an easy-to-implement protocol because it uses only 2 wires to communicate between up to 112 devices using 7-bit addressing and up to 1008 devices using 10-bit addressing.

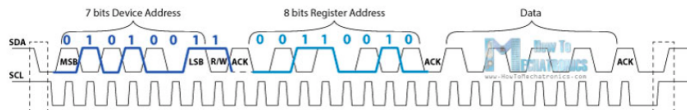
The Figure 9-1 shows how you can connect the devices using the same I2C bus.

### How I2C works?

How can we communicate with multiple devices using only two wires? For this to happen, each device has set an ID or a unique address. Then the master device can choose which device to communicate with.

The two wires are called Serial Clock (or SCL) and Serial Data (or SDA). The SCL wire is the clock signal that synchronizes the data transfer between devices on the I2C bus and is generated by the master device. The other wire is the SDA line that carries the date.





**Figure 9-2** Bits sequence.

## Protocol

The data is transferred in 8-bit sequences like you can see in Figure 9-2. After a special starting condition occurs, comes the first 8-bit sequence that indicates the address of the slave to which the data is being sent. For example, for the ADXL345 accelerometer device, the default address is 16r53 (0X53) or 0101 0011 (the last bit activated means the device is on reading mode).

After each 8-bit sequence follows a bit called Acknowledge. After the first Acknowledge bit, in most cases another addressing sequence comes, but this time to the internal registers of the slave device.

The internal registers are locations in the slave's memory containing various information or data. For example, the ADX345 accelerometer has a unique device address (16r53) and addition internal record addresses for the X, Y, and Z axes (16r32, 16r33, 16r34, etc.). Therefore, if we want to read the X-axis data, we first need to send the address of the device and then the internal register address specific to the X-axis.

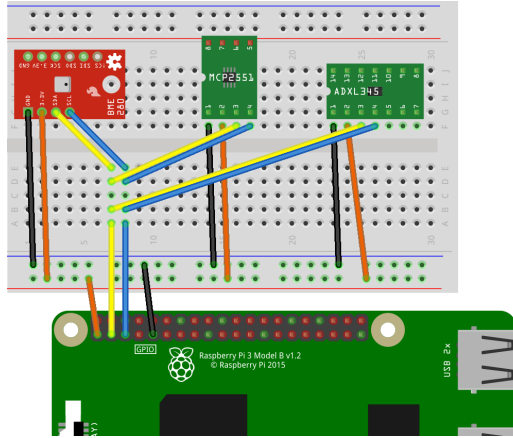
After the addressing sequences, the data streams are as many as they are sent until the data is completely sent and ends with a special stop condition.

## 9.3 Experimental procedure

Now we will build the circuit. This circuit consists of three sensors (BME280, MCP9808, ADXL345) and a power supply (the Rasp).

- Connect the Ground PIN from Raspberry in the breadboard blue rail (-). In this experiment we will use the PIN6 (Ground);
- Then connect the 3.3V (PIN1) pin in the red rail (+).
- Now let's connect the SCL (PIN5) and SDA (PIN3) wires. Connect them like as shown in the Figure 9-3 ;
- Now push the sensors in the breadboard;
- And insert the jumper wires connecting the sensor in the bus, like as shown in the Figure 9-3;
- The last step is to connect the power 3.3V (+) and ground (-) wires in each sensor.

The Figure 9-3 shows how the electric connection is made.



**Figure 9-3** Physical sensors connection.

## 9.4 Configuring the Raspberry Pi I2C

We need to configure the Raspberry Pi to use the I2C protocol. To do this, access the Raspberry using SSH and go to file `/boot/config.txt` and add the line `dtoverlay=i2c1=on`.

You can run the follow command to do this:

```
[sudo echo dtoverlay=i2c1=on >> /boot/config.txt
```

Add the 'pi' user to I2C group and restart the Raspberry

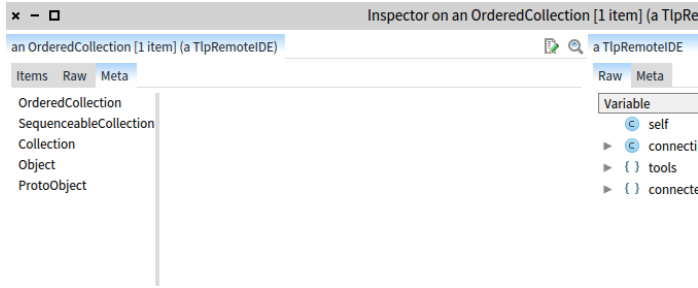
```
[sudo adduser pi i2c
reboot
```

Now your Raspberry is configured to communicate with the sensor using the I2C protocol.

## 9.5 Connecting remotely

Through your local Pharo image, let's connect in the Pharo image by running on Raspberry, enable the auto-refresh feature of the inspector, and open the inspector. Run this code in your local playground:

```
[remotePharo := TlpRemoteIDE connectTo: (TCPAddress ip: #[193 51 236
212] port: 40423)
GTInspector enableStepRefresh.
remoteBoard := remotePharo evaluate: [ RpiBoard3B current].
remoteBoard inspect.
```



**Figure 9-4** Inspecting remote object.

## 9.6 Exploring the properties of a remote object with the remote inspector

In your inspect window (Inspector on a PotRemoteBoard), let's create the instances of the first sensor.

```
[ a := board installDevice: PotBME280Device new.
```

With this proceeding, we are creating an object and we can inspect it to see the properties and values. To see the details of this object, like for example what you can do and what you can ask to it (methods), press `cmd + I` to inspect it and you will see the inspect window.

In the Meta tab, you can see the Class and Subclass of this object. To see what you can do with this object (methods), you can start selecting the top Class and will be shown in the right window the methods that you can use.

Let's see what the method `readParameters` can do. When you select it, you can see the code of method on the bottom of the window. When you call this method, it returns an array with 3 values: temperature, pressure, and humidity. The carrot symbol (little hat `^`) is to return something when the method is called.

So let's use the method `readParameters` to ask to the object what are the values of temperature, pressure, and humidity. To get the return of this object, select the following code and press `cmd + P`. You will see a big number. This number is an array with 3 indexes like the Figure 9-4.

```
[ a readParameters.
```

To see the details of this object, you can press `Cmd + I` to inspect it and you will see the 3 indexes separately.

## 9.7 Getting the temperature with BME280

You can also ask to only one value, selecting the array index using the method `at: 1`. In this case, we are selecting only the index 1 of an array and will return only the temperature:

```
[a readParameters at: 1.
```

If you want to format the number to be more legible, you can use:

```
[(a readParameters at: 1) printShowingDecimalPlaces: 2.
```

## 9.8 Getting the humidity and pressure with BME280

Now you know how to ask to the object the specific value that you want. To get the humidity and pressure is very simple, just choose the index of each one and format them as you want. To get the pressure:

```
[(a readParameters at: 2) printShowingDecimalPlaces: 2.
```

and to get the humidity:

```
[(a readParameters at: 3) printShowingDecimalPlaces: 2.
```

This is very simple and you can get these values and stored in a variable, to use to different proposes, like send a message to an LCD display, send the values to a cloud server and simply do some action, like for example turn on a LED or turn on an Air Conditional (using relays). In the following example, after you put the temperature value in a temperature variable, you can select the variable and press `Cmd + P` to see just the temperature.

```
[temperature := (a readParameters at: 1) printShowingDecimalPlaces: 2.  
temperature.
```

## 9.9 Getting the temperature with MCP9808

How we see before, is very easy to read the values of the sensors. To read the values of the MCP9808 sensor, just create an object using the MCP9808 sensor with the follow code and read the temperature with the method `readTemperature`:

```
[b := board installDevice: PotMCP9808Device new.  
b readTemperature.
```

You can format the answer as you want also:

```
[(b readTemperature) printShowingDecimalPlaces: 2.
```

## 9.10 Getting the axis X, Y, and Z with ADXL345

The process is the same before. Let's create an object with the sensor and ask to it what is the value of the X, Y, Z axis:

```
[ c := board installDevice: PotADXL345Device new.
  c readCoordinates.
```

As like the BME280 sensor, the ADXL345 is returning an array with 3 indexes, the 3 axes. To ask to a specific axis, you can select the position that you want. In the following case we are getting the X-axis:

```
[ c readCoordinates at: 1.
```

## 9.11 Conclusion

In this tutorial, you learned how to inspect remote objects to understand what this object can do. You learned also how to get the value of temperature, humidity and pressure to store in a variable, as like the X, Y, Z axis.

In the next lesson, let's see a different kind of sensor, the ultrasonic sensor. It uses only 2 wires to work, but don't use the I2C protocol. See you there.





## Lesson 9 - Ultrasonic Sensor (Distance)

In the previous lessons, we learned how to control LEDs and to use a button to interact with LEDs. We learned also how to use I2C sensors to read the temperature, humidity, pressure, and x, y, z-axis. Now let's use a different kind of sensor, that doesn't use I2C protocol.

### 10.1 What do we need?

In this lesson, we will use a setup with an ultrasonic sensor.

#### Components

- 1 Raspberry Pi connected to your network (wired or wireless)
- 1 Breadboard
- 1 HC-SR04 sensor
- 1 Resistor (1K ohms)
- 1 Resistor (2K ohms)
- Jumper wires

### 10.2 Experimental theory

Before constructing any circuit, you must know the parameters of the components in the circuit, such as their operating voltage, operating circuit, etc.

## The ultrasonic measure

The ultrasonic sensor that we will use in this tutorial has 4 PINs: GND Ground, VCC 5V, TRIG to send a pulse and ECHO to show how long time the pulse spend to go and back.

## How the HC-SR04 works?

It uses our Raspberry Pi to send a signal using the TRIG, which triggers the sensor to send an ultrasonic pulse. Pulse waves reflect any nearby objects and some are reflected back to the sensor. The sensor detects these return waves and measures the time between the trigger and the returned pulse and sends a 5V signal on the ECHO pin. So is just to see how long time the ECHO pin will stay ON and calculate the distance because we already know the sound speed.

## Limiting the return voltage with resistors

The ECHO signal in sensor HC-SR04 uses 5V. But the Raspberry Pi PINS uses 3.3V. If we send 5V to Raspberry GPIO, it can damage the Rasp. To avoid this let's put 2 resistors to limit the voltage.

## 10.3 Experimental procedure

Now we will build the circuit. This circuit consists of 1 ultrasonic sensor HC-SR04, 1 resistor 1K ohms, 1 resistor 2K ohms and a power supply (the Rasp).

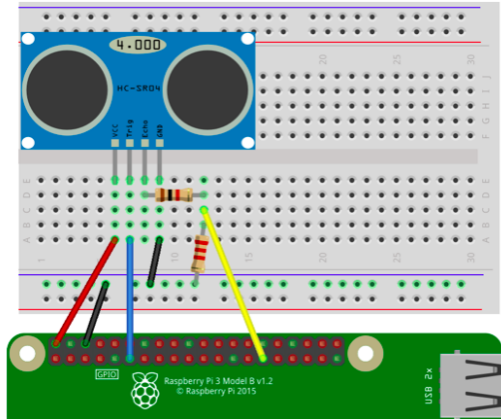
- Connect the Ground PIN from Raspberry in the breadboard blue rail (-). In this experiment we will use the PIN9 (Ground);
- Then connect the 5V (PIN2) pin in the red rail (+).
- Now push the HC-SR04 sensor in the breadboard;
- And insert the jumper wires connecting the sensor leg TRIG in the GPIO0 (PIN11) and the sensor leg ECHO in the breadboard, like the scheme shown in the Figure 10-1;
- The last step is to connect the power 5V and ground (-) wires from the breadboard rails in the sensor VCC + and GND (-) legs.

The Figure 10-1 shows how the electric connection is made.

## 10.4 Connecting remotely

Through your local Pharo image, let's connect in the Pharo image by running on Raspberry, enable the auto-refresh feature of the inspector, and open the inspector. Run this code in your local playground:





**Figure 10-1** Physical sensors connection.

```

remotePharo := TlpRemoteIDE connectTo: (TCPAddress ip: #[193 51 236
    212] port: 40423)
GTInspector enableStepRefresh.
remoteBoard := remotePharo evaluate: [ RpiBoard3B current].
remoteBoard inspect.

```

## 10.5 Experimental code

In your inspect window (Inspector on a PotRemoteBoard), let's create an instance of the ultrasonic sensor.

```

[d := board installDevice: PotHCSR04Device new.

```

As we saw before, we can inspect the remote object to see some properties and methods. Let's use the method `readDistance` to read the distance:

```

[d readDistance.

```





## Lesson 10 - LCD Display

In the previous lessons, we learned how to control LEDs and to use a button to interact with LEDs. We learned also how to use the I2C sensors to read the temperature, humidity, pressure, and x, y, z-axis. Also, we saw how to use a no I2C sensor, an ultrasonic sensor. Now we will learn how to use an LCD Display without I2C.

### 11.1 What do we need?

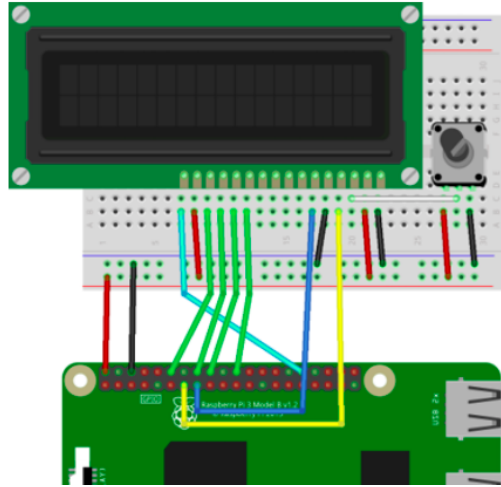
In this lesson, we will use a setup with an LCD Display 1602.

#### Components

- 1 Raspberry Pi connected to your network (wired or wireless)
- 1 Breadboard
- 1 LCD Display 1602
- 1 Potentiometer (10K ohms)
- Jumper wires

### 11.2 Experimental theory

Before constructing any circuit, you must know the parameters of the components in the circuit, such as their operating voltage, operating circuit, etc.



**Figure 11-1** Physical sensors connection.

## The LCD Display 1602

### How the LCD 1602 works?

## 11.3 Experimental procedure

Now we will build the circuit. This circuit consists of three sensors and a power supply (the Rasp).

- Connect the Ground PIN from Raspberry in the breadboard blue rail (-). In this experiment we will use the PIN6 (Ground);
- Then connect the 5V (PIN2) pin in the red rail (+).
- Now push the LCD 1602 in the breadboard;
- Push the potentiometer in the breadboard;
- And insert the jumper wires connecting the LCD Display in the Potentiometer and breadboard, as the scheme shown in the Figure 11-1.

The Figure 11-1 shows how the electric connection is made.

## 11.4 Connecting remotely

Through your local Pharo image, let's connect in the Pharo image by running on Raspberry, enable the auto-refresh feature of the inspector, and open the inspector. Run this code in your local playground:

```

remotePharo := TlpRemoteIDE connectTo: (TCPAddress ip: #[193 51 236
    212] port: 40423)
GTInspector enableStepRefresh.
remoteBoard := remotePharo evaluate: [ RpiBoard3B current].
remoteBoard inspect.

```

## 11.5 Experimental code

In your inspect window (Inspector on a PotRemoteBoard), let's create the instances of the LCD Display.

```

lcd := board installDevice: PotLCD1602Device new.

```

As we saw before, we can inspect the remote object to see some properties and methods. Let's use the method `message:` to send some message to LCD Display. To break line you can use `"/n"`. And to clear the LCD you can use the method `clear`:

```

lcd message: 'Hello everybody!\Pharo is cool!'.
lcd clear.

```





## Lesson 11 - Building a Mini-Weather Station

In the previous lessons, we learned how to control LEDs, sensors, LCD displays and how to use OOP to create applications to control them. Now we will use everything that we learned to build a Mini-Weather Station.

### 12.1 What do we need?

In this lesson, we will use a setup with an LCD Display 1602 and BME280 sensor.

#### Components

- 1 Raspberry Pi connected to your network (wired or wireless)
- 1 Breadboard
- 1 BME280 temperature, humidity and pressure sensor
- 1 LCD Display 1602
- 1 Potentiometer (10K ohms)
- Jumper wires

### 12.2 Experimental theory

Before constructing any circuit, you must know the parameters of the components in the circuit, such as their operating voltage, operating circuit, etc.

In this lesson, we will get the temperature, pressure, and humidity using the BME280 sensor, show this information on the LCD Display each 1 second and send this data to a Cloud Server every 60 seconds.

## 12.3 Experimental procedure

Connect the sensor and LCD Display on your breadboard as we did in the previous lessons.

## 12.4 Creating the ThingSpeak account

In order to store the measure collected by the weather station, we will use a web service called ThingSpeak. ThingSpeak is a cloud application that is dedicated to IoT (Internet of Things).

Before we start to create an application, let's create our account on the website Thingspeak. We will go to use this website to receive and store the data.

Follow this tutorial to create your account and your channel:

<https://roboindia.com/tutorials/thingspeak-setup>

Once you've created your channel, you need to enable three channels on it. Let's use Field1 to Temperature, Field2 to Humidity and Field3 to Pressure. You can see in Picture 12-1 how your channel will look like.

You can test your channel by simulating sending some data to it. Let's for example send the values:

- 25 to Temperature field (field1);
- 56 to Humidity field (field2);
- and 1012 to Pressure field (field3).

To test your channel, just copy the following URI and passed in your web browser, changing YOUR-WRITE-API-KEY to your ThingSpeak write API code. You can find it in the API Keys tab.

[https://api.thingspeak.com/update?api\\_key=YOUR-WRITE-API-KEY&field1=25&field2=56&field3=1012](https://api.thingspeak.com/update?api_key=YOUR-WRITE-API-KEY&field1=25&field2=56&field3=1012)

So check you channel to see if it was updated with this values:

<https://thingspeak.com/channels/YOUR-CHANNEL-ID>

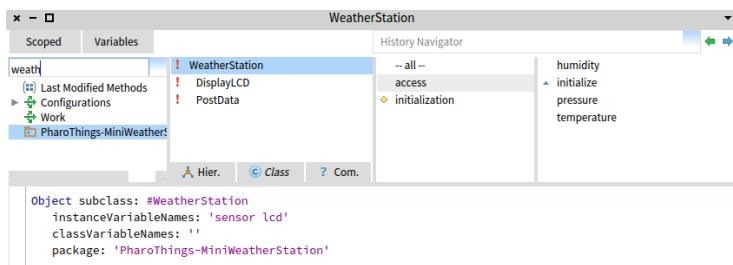
## 12.5 Creating the application

The first step lets create a Superclass to initialize and install all devices that we will use. So let's create 2 subclasses to do the actions. The first will dis-



The screenshot shows the ThingSpeak interface for a channel named 'PharoThings Monitor - INRIA Allex's Office'. The channel ID is 562010, the author is 'allexoliveira', and the access is 'Public'. The channel description is 'Getting temperature, humidity, and pressure from a BME280 sensor using PharoThings running in Raspberry Pi 3 B+'. The 'Channel Settings' tab is selected, showing the 'Percentage complete' at 50%. Below this, the 'Channel ID' is 562010, the 'Name' is 'PharoThings Monitor - INRIA Allex's Office', and the 'Description' is 'Getting temperature, humidity, and pressure from a BME280 sensor using PharoThings running in'. There are three fields: 'Field 1' is 'Temperature', 'Field 2' is 'Humidity', and 'Field 3' is 'Pressure'. Each field has a checkbox that is checked.

**Figure 12-1** ThingSpeak Channel Configuration.



**Figure 12-2** Mini Weather Station code.

play this information on the LCD and the second will send the data to the cloud. Your final code will seem like the Picture 12-2.

## 12.6 Creating the Superclass

```
Object subclass: #WeatherStation
  instanceVariableNames: 'sensor lcd'
  classVariableNames: ''
  package: 'PharoThings-MiniWeatherStation'
```

### Creating the initialize method

```
initialize
  lcd := (RpiBoard3B current) installDevice: PotLCD1602Device new.
  sensor := (RpiBoard3B current) installDevice: PotBME280Device new.
```

### Creating access methods

```
humidity
  ^((sensor readParameters at: 3) printShowingDecimalPlaces: 2)
  asString.

pressure
  ^((sensor readParameters at: 2) printShowingDecimalPlaces: 2)
  asString.

temperature
  ^((sensor readParameters at: 1) printShowingDecimalPlaces: 2)
  asString.
```

## 12.7 Creating the subclass DisplayLCD

```
WeatherStation subclass: #DisplayLCD
  instanceVariableNames: 'lcdprocess'
  classVariableNames: ''
  package: 'PharoThings-MiniWeatherStation'

lcdStart
  |text|
  lcdprocess := [ [
    text := 'Temp: ',self temperature,'\H:',self humidity,' P:',self
    pressure.
    lcd home.
    lcd message: text.
    (Delay forSeconds: 1) wait.
  ] repeat ] forkNamed: 'lcdprocess'.

lcdStop
  lcdprocess terminate.
  lcd clear.
```

## 12.8 Creating the subclass PostData

```

WeatherStation subclass: #PostData
  instanceVariableNames: 'apiKey postProcess'
  classVariableNames: ''
  package: 'PharoThings-MiniWeatherStation'

  apiKey
    ^apiKey

  apiKey: anString
    apiKey := anString .

  dataStart
    |url uri |
    url := 'https://api.thingspeak.com/update'.
    postProcess := [ [
      uri := url,'?api_key=',self apiKey,'&field1=',self
      temperature,'&field2=',self humidity,'&field3=',self pressure.
      ZnClient new get: uri.
      (Delay forSeconds: 60) wait.
    ] repeat ] forkNamed: 'postprocess'.

  dataStop
    postProcess terminate.

```

## 12.9 Starting the application

To start the application, we need to start the two subclasses. To start the LCD application run this in the remote playground:

```
(DisplayLCD new) lcdStart.
```

and to start to send the data to the Cloud, use this, replacing to your apiKey of Thingspeak:

```
(PostData new) apiKey:'F1MKEG7PJ44930L8'; dataStart.
```

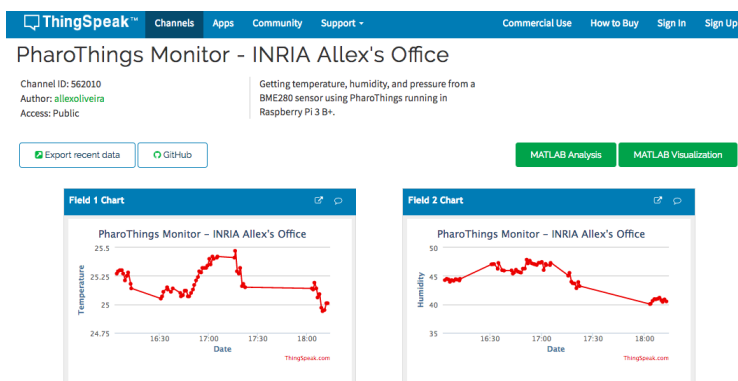
Remember, when you run these commands, you are creating a process running in the background in the Raspberry. So if you run this commands many times, will be created many processes. Take care to don't do this.

## 12.10 Visualizing your data

To see if your channel is receiving the data of your PharoThings, just access your channel and check if the data is being updated:

<https://thingspeak.com/channels/YOUR-CHANNEL-ID>

You will see some graphics like the Picture 12-3.



**Figure 12-3** ThingSpeak Channel.



## Lesson 12 - Building a Webserver on Raspberry

In the previous lessons, we learned how to control LEDs, sensors, LCD displays and how to use OOP to create applications to control them and how to build a Mini-Weather Station. Now we going to build a Webserver to interact with GPIOs.

### 13.1 What do we need?

#### Components

- 1 Raspberry Pi connected to your network (wired or wireless)
- Jumper wires

### 13.2 Experimental theory

Before constructing any circuit, you must know the parameters of the components in the circuit, such as their operating voltage, operating circuit, etc.

### 13.3 Experimental procedure

First of all, you have to create a class to implement the web app for the weather station :

```
Object subclass: #WeatherStationWebApp
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'PharoThings-Lessons'
```

and implement the following methods:

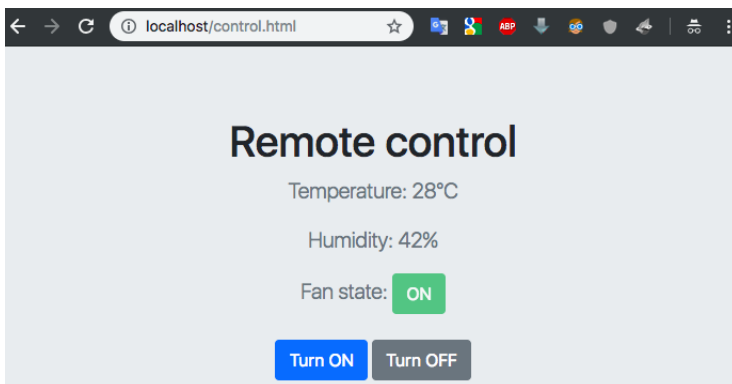
```
handleRequest: request
  request uri path = #sensors
  ifFalse: [ ^ ZnResponse notFound: request uri ].
  ^ ZnResponse ok: (ZnEntity html: self html)

temperature
  ^ 0

value: request
  ^ self handleRequest: request
```

## 13.4 HTML page

```
html
  ^ '<html>
    <head>
      <title>Remote control</title>
      <!-- Bootstrap core CSS -->
      <link
        href="https://getbootstrap.com/docs/4.0/dist/css/bootstrap.min.css"
        rel="stylesheet">
    </head>
    <body >
      <main role="main">
        <section class="jumbotron text-center">
          <div class="container">
            <h1 class="jumbotron-heading">Remote control</h1>
            <p class="lead text-muted">Temperature: 28°C</p>
            <p class="lead text-muted">Humidity: 42%C</p>
            <p class="lead text-muted">Pressure: 1017 hPa</p>
            <p class="lead text-muted">Fan state:
              <button type="button" class="btn btn-success"
disabled="disabled">ON</button>
            </p>
            <p>
              <a href="#" class="btn btn-primary my-2">Turn
ON</a>
              <a href="#" class="btn btn-secondary my-2">Turn
OFF</a>
            </p>
          </div>
        </section>
      </main>
```



**Figure 13-1** Web Page.

```
<!-- Bootstrap core JavaScript -->
<script
  src="https://code.jquery.com/jquery-3.2.1.slim.min.js"></script>
</body>
</html>
```

Then you can start the Zn webserver like that:

```
ZnServer startDefaultOn: 8080.
ZnServer default delegate map: #image to: MyFirstWebApp new.
```

This page looks like the Picture 13-1.

