A PharoThings Tutorial

Allex Oliveira

January 11, 2019

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

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.

1.1 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

1.2 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.

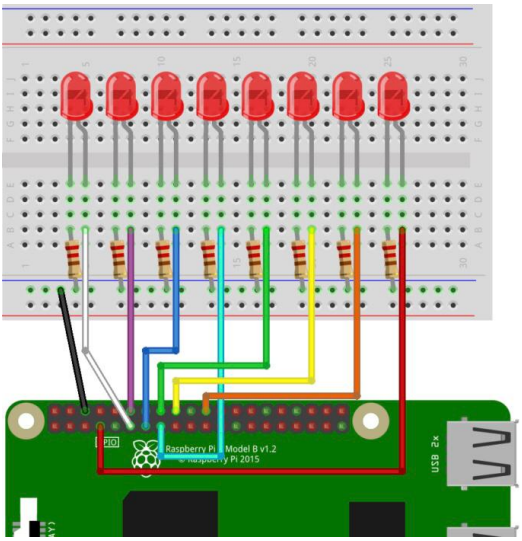

Figure 1-1 Schema connection 8 LEDs.

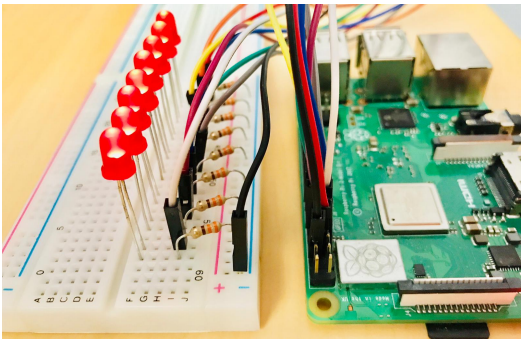

Figure 1-2 Physical connection 8 LEDs.

- 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 1-1.

The Figure 1-2 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.
```

1.3 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 1-3 and 1-4 shows the code and the LEDs turn On.

Inspector on a PotRemoteBoard (a RpiBoard3B in #[192 168 1 131]:40423)

a PotRemoteBoard (a RpiBoard3B in #[192 168 1 131]:40423)

P1 Devices Raw Meta

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	out	GPIO7	7	8	SerialPortTXD		14
		Ground (0v)	9	10	SerialPortRXD		15
17	out	GPIO0	11	12	GPIO1	out	18
27	out	GPIO2	13	14	Ground (0v)		
22	out	GPIO3	15	16	GPIO4	out	23
		3.3v	17	18	GPIO5	out	24
10		MOSI (SPI)	19	20	Ground (0v)		
9		MISO (SPI)	21	22	GPIO6	out	25
11		SCLK (SPI)	23	24	CE (SPI)		8
		Ground (0v)	25	26	CE (SPI)		7

```
gpioArray := { gpio0. gpio1. gpio2. gpio3. gpio4. gpio5. gpio6. gpio7 }  .
gpioArray do: [ :gpio | gpio beDigitalOutput ].
[
    gpioArray do: [ :item | item toggleDigitalValue. (Delay forSeconds: 0.3) wait ].
] forkNamed: 'FlowingProcess'.
```

Figure 1-3 Code on Inspector

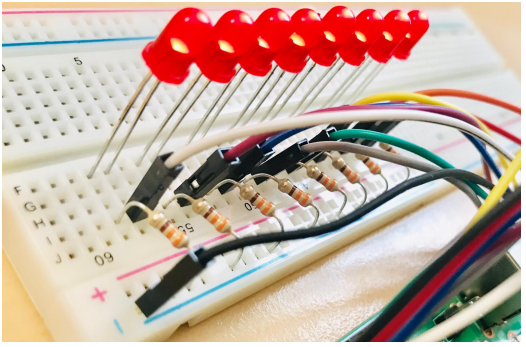

Figure 1-4 LEDs turn On.

1.4 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!

1.5 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!

1.6 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.
] ].
```

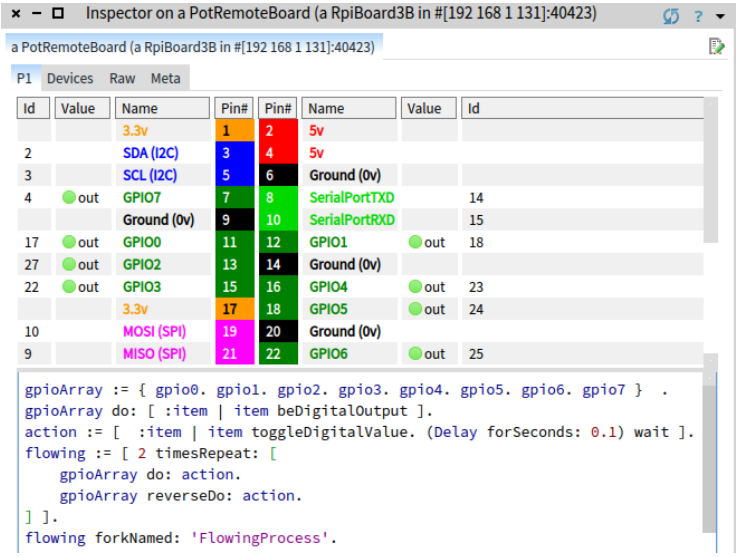

Figure 1-5 Process Browser terminate.

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 1-5. 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'.
```

1.7 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:

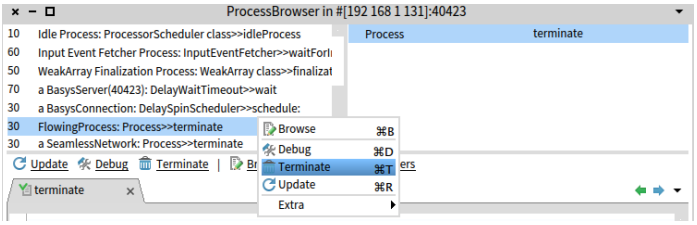

Figure 1-6 Process Browser terminate.

```
remotePharo openProcessBrowser.
```

Search the FlowingProcess and terminate it, like in Picture 1-6, 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.

1.8 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.