Led and Button on Arduino and RaspberryPi

Antonio Natali

Alma Mater Studiorum – University of Bologna
viale Risorgimento 2, 40136 Bologna, Italy
antonio.natali@unibo.it

Table of Contents

0.1 Working with Arduino

In order to handle the serial line in Java, we use the java-simple-serial-connector by adding to our gradle file the new dependency:

```
// https://mvnrepository.com/artifact/org.scream3r/jssc
compile group: 'org.scream3r', name: 'jssc', version: '2.8.0'
```

0.2 Working with RaspberryPi

The Raspberry Pi is a series of small single-board computers developed in the United Kingdom by the Raspberry Pi Foundation to promote the teaching of basic computer science in schools and in developing countries.

Raspberry Pi 3 Model B was released in February 2016 with a 64 bit quad core processor, and has on-board WiFi, Bluetooth and USB boot capabilities. None of the current Raspberry Pi models have a built-in real-time clock, so they are unable to keep track of the time of day independently. The Raspberry Pi Foundation recommends the use of Raspbian, a Debian-based Linux operating system.

In the rest of this section we summarize basic notions for using the Unibo software built around the Jessie, the development code name for Debian 8.

0.2.1 Start

1. Read rpi_sd.pdf e guida_rpi.pdf
2. Download and install PuTTY
3. Download the unibo 2016-02-26-raspbian-jessie-iss.img.zip (zip 1GB) and generate your own Secure Digital card
4. Insert the SD card in a PC and look (in the unit boot) at the file mywifi.conf:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
ap_scan=1
eapol_version=1
fast_reauth=1

network={
        ssid="natspot"
        scan_ssid=1
        key_mgmt=WPA-PSK
        psk="123456789"
}
```

5. Insert the SD card into the RaspberryPi and extend the file system by running:

```
sudo raspi-config
```

0.2.2 Connect with cable

1. Connect the RaspberryPi to the PC via a Ethernet cable
2. Open PuTTY and connect to 192.168.137.2 port 22
3. Login with: pi pswd= raspberry
4. Look at the output:

0.2.3 Access to the RaspberryPi via Windows

1. connnetti unita di rete
2.

```
\\192.168.137.2\home_pi\
```

0.2.4 Connect with Wifi .

1. For Raspberry 1 and 2: insert a WIFI adapter (e.g. EDUP) in a USB slot.
2. Activate an hot spot (e.g by using a smart device) with name natspot and password 123456789
3. Turn on the RaspberryPi. It will acquire an address (192.168.43.xxx) as shown by the command ifconfig
4. Open PuTTY and connect to 192.168.43.xxx on port 22

1 A Led on Raspberry

The GPIO pins on a Raspberry Pi are a great way to interface physical devices like Buttons and Leds through some simple hardware connection. In this example the Led anode is connected to pin 25 in BCM code[1] while the Led cathode is connected to a GND pin.

However, a software design never interacts in direct way with the hardware level; at least some minimal support for the control of external devices must be provided by the operating system. In the case of Linux we can start from the shell or from a more advanced library.

1.1 Led control using files

The basic way provided by Linux to manage a device connected on a GPIO pin is reading/writing some (virtual) file associated with that pin.

```
# ---------------------------------------------------------------
# led25OnOff.sh
# Key-point: we can manage a device connected on a GPIO pin by
# reading/writing some (virtual) file associated with that pin.
#   sudo bash led25OnOff.sh
# ---------------------------------------------------------------

echo Unexporting.
echo 25 > /sys/class/gpio/unexport #
echo 25 > /sys/class/gpio/export #
cd /sys/class/gpio/gpio25 #

echo Setting direction to out.
echo out > direction #
echo Setting pin high.
echo 1 > value #
sleep 1 #
echo Setting pin low
echo 0 > value #
sleep 1 #
echo Setting pin high.
echo 1 > value #
sleep 1 #
echo Setting pin low
echo 0 > value #
#schmod +x ledOnOff.sh #
```

Listing 1.1. led25OnOff.sh

1.2 Led control using the GPIO shell library

WiringPi is a GPIO access library written in C[2] for the BCM2835 used in the Raspberry Pi. *WiringPi* includes a command-line utility gpio which can be used to program and setup the GPIO pins. We can use this to read and write the pins and even use it to control them from shell scripts.

```
# ---------------------------------------------------------------
# led25Gpio.sh
# Key-point: we can manage a device connected on a GPIO pin by
# using the GPIO shell library.
# The pin 25 is physical 22 and Wpi 6.
#   sudo bash led25Gpio.sh
# ---------------------------------------------------------------
gpio readall #
```

[1] BCM refers to the pin number of the BCM2835 chip, and this is the pin number used when addressing the GPIO using the /sys/class/gpio interface

[2] The *WiringPi* library was written by Gordon Henderson to allow GPIO communication from C,C++ in a style similar to the Arduino Wiring programming language

```
echo Setting direction to out
gpio mode 6 out #
echo Write 1
gpio write 6 1 #
sleep 1 #
echo Write 0
gpio write 6 0 #
```

Listing 1.2. led25Gpio.sh

1.3 Led control using python

The Raspbian Linux operating system has the RPi.GPIO library pre-installed. It is a Python library that handles interfacing with the GPIO pins.

```
# --------------------------------------------------------
# ledPython25.py
# Key-point: we can manage a Led connected on the GPIO pin 25
# using the python language.
#   sudo python ledPython25.py
# --------------------------------------------------------
import RPi.GPIO as GPIO
import time

'''
---------------------------------
CONFIGURATION
---------------------------------
'''
GPIO.setmode(GPIO.BCM)
GPIO.setup(25,GPIO.OUT)

'''
---------------------------------
main activity
---------------------------------
'''
while True:
    GPIO.output(25,GPIO.HIGH)
    time.sleep(1)
    GPIO.output(25,GPIO.LOW)
    time.sleep(1)
```

Listing 1.3. ledPython25.sh

1.4 Led control using Pi4j

Pi4J is an open source project intended to provide a bridge between the native hardware and Java for full access to the Raspberry Pi. In addition to the basic raw hardware access functionality, this project also attempts to provide a set of advanced features that make working with the Raspberry Pi an easy to implement and more convenient experience for Java developers.

Thus, the Pi4J library can be used as our basic software layer (our *technology assumption*) for the the control of a Led in Java.

2 A Led on Arduino

The basic way provided by Arduino to manage a device connected on a pin is to perform a read or a write operation on that pin. For example we can blink a Led connected on pin 13 as follows:

```
/*
=====================================================
 project it.unibo.buttonLedSystem.arduino
 Led13/Led13.ino ARDUINO UNO
 Pin 13 has a internal LED connected on ARDUINO UNO.
=====================================================
 */
int ledPin  = 13;
int count   = 1;
boolean on  = false;

void setup(){
  Serial.begin(9600);
  Serial.println( "--------------------------------------" );
  Serial.println( "project it.unibo.buttonLedSystem.arduino" );
  Serial.println( "Led13/Led13.ino" );
  Serial.println( "--------------------------------------" );
  configure();
}
void configure(){
  pinMode( ledPin, OUTPUT );
  turnOff();
}
/* ----------------------------
LED primitives
------------------------------- */
void turnOn(){
  digitalWrite(ledPin, HIGH);
  on = true;
}
void turnOff(){
  digitalWrite(ledPin, LOW);
  on = false;
}
/* ----------------------------
Blinck the led (non primtive)
------------------------------- */
void blinkTheLed(){
    turnOn();
    delay(500);
    turnOff();
    delay(500);
}

void loop(){
  if( count <= 10 ){
    Serial.println("Blinking the led on 13 count=" + String(count) );
    blinkTheLed();
    count++;
  }
}
```

Listing 1.4. `Led13.ino`

2.1 Led with input output

The next *sketch* does introduce a `cmdHandler` operation that looks at the serial line as an input device and turns the Led `on/off` if the input value is `1/0`.

```
/*
=====================================================
 project it.unibo.arduino.intro Led13Msg/Led13Msg.ino ARDUINO UNO
 Pin 13 has a internal LED connected on ARDUINO UNO.
=====================================================
```

```
 */
int ledPin  = 13;
int count   = 1;
boolean on  = false;

void setup(){
  Serial.begin(9600);
  Serial.println( "-------------------------------------" );
  Serial.println( "project it.unibo.arduino.intro" );
  Serial.println( "Led13Msg/Led13Msg.ino" );
  Serial.println( "-------------------------------------" );
  configure();
}
void configure(){
  pinMode( ledPin, OUTPUT );
  turnOff();
}
/* ------------------------------
LED primitives
------------------------------- */
void turnOn(){
  digitalWrite(ledPin, HIGH);
  on = true;
  forwardTheLedState();
}
void turnOff(){
  digitalWrite(ledPin, LOW);
  on = false;
  forwardTheLedState();
}
void forwardTheLedState(){
   //msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, SEQNUM )
  Serial.println("msg( info, dispatch, arduino, ANY, ledstate(" + String(on) + "), " + String(count++) + " )" );
}
/* ------------------------------
Blinck the led (non primtive)
------------------------------- */
void blinkTheLed(){
    turnOn();
    delay(500);
    turnOff();
    delay(500);
}
/* ------------------------------
Input handler
------------------------------- */
void cmdHandler(){
   int v = Serial.read(); //NO BLOCKING
   //if( v > 0 ) Serial.println("arduinio input=" + String(v) );
   if( v != -1 && v != 13 && v != 10 ){
     boolean isPressed = ( v == 49 ) ; //48 is '0' 49 is '1'
     if( isPressed ) turnOn();
     else turnOff();
   }
}
void loop(){
  cmdHandler();
}
```

Listing 1.5. `Led13Msg.ino`

3 A Button on Arduino

An input device such as the Button can be managed in Arduino in two ways: by polling or via interrupt.

3.1 Managing a Button by polling.

```
/*
=======================================
 buttonPolling.ino
 project it.unibo.arduino.intro
 Input from Pin 3 (Button)
 ARDUINO UNO
 =======================================
 Arduino has internal PULLUP resistors to tie the input high.
 Hook one end of the switch to the input (5V) and the other end to ground:
 when we PUSH the switch the input goes LOW.
 */

int pinButton = 3;
int pinLed   = 13; //internal led

void initLed(){
   pinMode(pinLed, OUTPUT);
}
void initButton(){
  pinMode(pinButton, INPUT);
  digitalWrite(pinButton, HIGH); //set PULLUP : PUSH=>0
}

void setup(){
  Serial.begin(9600);
  Serial.println( "--------------------------------------" );
  Serial.println( "project it.unibo.arduino.intro" );
  Serial.println( "buttonPolling/buttonPolling.ino" );
  Serial.println( "--------------------------------------" );
  initButton();
  initLed();
}

/*
 * When the button is PUSHED, the led is ON
 */
void loop(){
  int v = digitalRead( pinButton );
  Serial.println( "write " + v );
  digitalWrite( pinLed, v );
  delay(100);
}
```

Listing 1.6. `buttonPolling.ino`

3.2 Managing a Button via interrupt.

```
/*
=======================================
 buttonInterrupt.ino
 project it.unibo.arduino.intro
 Input from Pin 3 (Button)
 ARDUINO UNO
    pin 3 maps to interrupt 1,
    pin 2 is interrupt 0,
 =======================================
 Arduino has internal PULLUP resistors to tie the input high.
 Hook one end of the switch to the input (5V) and the other end to ground:
```

```
 when we PUSH the switch the input goes LOW.
 */

int pinButton = 3;
int pinLed   = 13;

void initLed(){
   pinMode(pinLed, OUTPUT);
}
void initButton(){
  pinMode(pinButton, INPUT);
  digitalWrite(pinButton, HIGH); //set PULLUP : PUSH=>0
  attachInterrupt(1, buttonInterruptHandler, CHANGE); // RISING CHANGE FOLLING LOW
}
/*
----------------------------
INTERRUPT HANDLER function
----------------------------
*/
void buttonInterruptHandler(){
  boolean ok = debouncing();
  if( ok ){
    int v = digitalRead(pinButton);
    Serial.println( "interrupt v=" + String(v));
    digitalWrite( pinLed, v );
   }
}
boolean debouncing(){
static unsigned long lastInterruptTime = 0;
static int lastVal = 0;
unsigned long interruptTime = millis();
int v = digitalRead(pinButton);
  if( (interruptTime - lastInterruptTime) > 100 ){
    if( (v != lastVal) ) lastVal = v;
    return true;
  }
  lastInterruptTime = interruptTime;
  return false;
}
void setup(){
  Serial.begin(9600);
  Serial.println( "--------------------------------------" );
  Serial.println( "project it.unibo.arduino.intro" );
  Serial.println( "buttonInterrupt/buttonInterrupt.ino" );
  Serial.println( "--------------------------------------" );
  initButton();
  initLed();
}
void loop(){
  //do nothing
}
```

Listing 1.7. `buttonInterrupt.ino`

4 A Button on Raspberry

In this example, one terminal of the Button is connected to the 5V pin (physical pin 2) while the other one (button-input pin) to pin 24 in BCM code. A resistor of 10K ohm[3]) is also inserted between the input pin and a GND pin (e.g. physical pin 9) to assure a connection to the ground when the button is not pressed.

4.1 Button control using files

The basic way provided by Linux to manage a device connected on a GPIO pin is reading/writing some (virtual) file associated with that pin.

```
# ------------------------------------------------------------
# buttonOn24Click.sh
# Key-point: we can manage a device connected on a GPIO pin by
# reading/writing some (virtual) file associated with that pin.
#   sudo bash buttonOn24Click.sh
# ------------------------------------------------------------

echo Unexporting.
echo 24 > /sys/class/gpio/unexport #
echo 24 > /sys/class/gpio/export #
cd /sys/class/gpio/gpio24 #
echo Setting direction to in.
echo in > direction #
echo Reading pin.
cat value
#schmod +x ledOnOff.sh #
```

Listing 1.8. buttonOn24Click.sh

4.2 Button control using the GPIO shell library

The gpio library allows us to write a program (bash file) that reads the button-input pin and writes a 0/1 value on the Led pin:

```
#!/bin/bash
# WARNING: document class text UNIX
# ------------------------------------------------------------------------
# button24Gpio.sh
# Key-point: we can manage a device connected on a GPIO pin by using the GPIO shell library.
# The pin 24 (button) is physical 18 and Wpi 5.
# The pin 25 (led)  is physical 22 and Wpi 6.
#   sudo bash button24Gpio.sh
# ------------------------------------------------------------------------
But=5
Led=6
LedBCM=25

gpio mode $But in
gpio -g mode $LedBCM out
gpio readall
echo "start"

while true
do
    gpio read $But > vgpio24.txt
    V1=`cat vgpio24.txt`
    if [ $V1 == "1" ] ; then echo "on" ; fi
    #echo $V1
    gpio write $Led $V1 #command the led
    sleep 0.1
done
```

Listing 1.9. button24Gpio.sh

[3] 10K colors: brown, black, orange and argent/gold

4.3 Button control using python

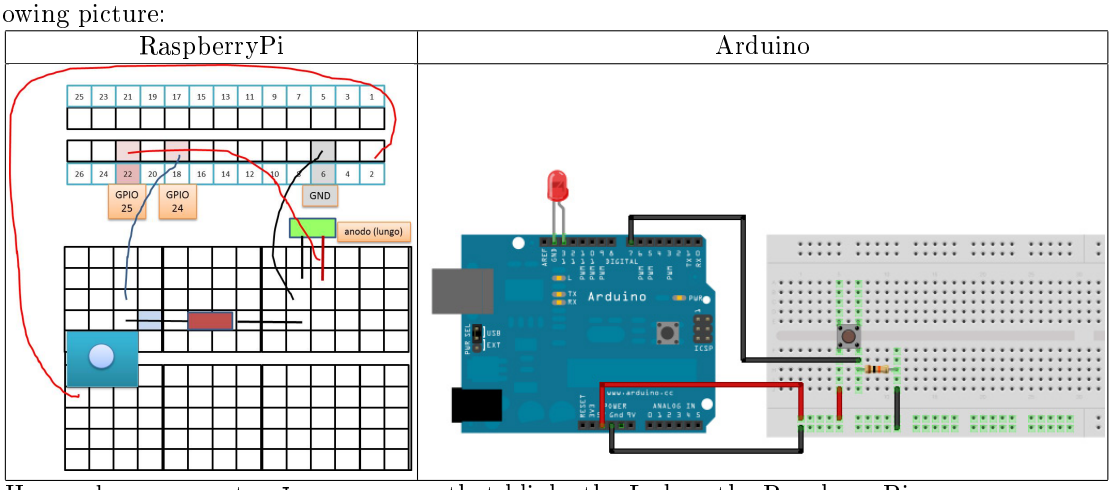
The Python library that handles interfacing with the GPIO pins allows us to write:

```
# ----------------------------------------------------------
# buttonPython24.py
# Key-point: manage in python a Button connected on the GPIO pin 24
#  using polling.
#   sudo python buttonPython24.py
# ----------------------------------------------------------
import RPi.GPIO as GPIO
import time

'''
----------------------------------
CONFIGURATION
----------------------------------
'''
GPIO.setmode(GPIO.BCM) #BCM or BOARD
GPIO.setup(24, GPIO.IN, pull_up_down=GPIO.PUD_DOWN )


'''
----------------------------------
main activity
----------------------------------
'''
while True:
    if GPIO.input(24):
        print('Button input HIGH')
    else:
        print('Button input LOW')
    time.sleep(1)  #wait 1sec

#while GPIO.input(24) == GPIO.LOW:
#    time.sleep(0.01) #wait 10msec
```

Listing 1.10. buttonPython24.py

5 From the components to a Button-Led system

The typical physical configuration of a Button-Led system on RaspberryPi and Arduino is shown in the following picture:



Hereunder we report a `Java` program that blinks the Led on the RaspberryPi:

```
1  package it.unibo.bls.gpio;
2  import it.unibo.gpio.base.GpioOnSys;
3  import it.unibo.gpio.base.IGpio;
4  import it.unibo.gpio.base.IGpioConfig;
5  import java.io.FileWriter;
6
7  public class BLSGpio {
8  protected IGpio gpio = new GpioOnSys();
9  protected FileWriter fwrLed;
10
11     public void doJob() throws Exception{
12         prepareGpioButton();
13         prepareGpioLed();
14         doCmdBlink();
15         System.out.println("BLSGpio bye bye");
16     }
17     protected void prepareGpioButton() throws Exception{
18         gpio.prepareGpio(IGpioConfig.gpioInButton);
19         gpio.openInputDirection(IGpioConfig.gpioInButton);
20     }
21     protected void prepareGpioLed() throws Exception{
22         gpio.prepareGpio(IGpioConfig.gpioOutLed);
23         fwrLed = gpio.openOutputDirection(IGpioConfig.gpioOutLed);
24     }
25     protected void doCmdBlink( ) throws Exception{
26         String inps = "";
27         for( int i = 1; i<=15; i++) {
28             //System.out.println("BLSGpio reading ...");
29             inps = gpio.readGPio(IGpioConfig.gpioInButton );
30             System.out.println("BLSGpio -> " + inps);
31             if( inps.equals("0") ){
32                 gpio.writeGPio( fwrLed, IGpio.GPIO_OFF);
33             }
34             if( inps.equals("1") ){
35                 gpio.writeGPio( fwrLed, IGpio.GPIO_ON);
36             }
37             // Wait for a while
38             java.lang.Thread.sleep(400);
39         }
40         gpio.writeGPio( fwrLed, IGpio.GPIO_OFF);
41     }
42     public static void main(String[] args) {
43         try {
44             BLSGpio sys = new BLSGpio();
45             sys.doJob();
```

```
46          } catch (Exception e) {
47              e.printStackTrace();
48          }
49      }
50  }
```

Listing 1.11. BLSGpio.java

> The reader should try to write the code (in the preferred language) in order to turn the Led on/off when the button is pressed/released.

5.1 Working at low-level

When we work with Arduino (and often also with Raspberry), we are in a *zooming phase*, with respect to our logical or project architecture expressed as a *QActor* model. In this phase, the software designers usually 'thing' by adopting the 'core-mentality' of the device they are using.

For example, the behavior model of Arduino (and in general of low-level micro-controllers) is based on an end-less loop (in the vocabulary of embedded systems, this is called a `Super Loop Architecture`) that contains all the tasks of the system. But, let us consider an autonomous robot that drives around via 2 motors given a series (could be very lengthy) of `digitalWrites` and delays to those motors. If we want to incorporate wall sensors, writing conditionals within a loop of considerable length would be useless and inaccurate.

5.2 Abstraction gap: the concept of coroutine

The problem of building systems that must actively operate while being able to look at the same time at the state of their environment is also known as the of *proactive-reactive* problem.

One way to overcome this problem without introducing pre-emptive threads, is to recur to the `Knuth` concept of `Coroutine`.

The basic idea is to be able to create blocks of code that execute sequentially, but can choose to stop temporarily and resume later. This is similar to threads, but in the case of coroutines, they never get pre-empted and will only give away focus if they explicitly yield.

Some support for coroutines in Arduino has been proposed . For example Coroutines in Arduino was inspired by Simon Tatham work and by Coroutines in Unity).

5.3 Remote control

Now, let us suppose to have the following problem:

> Build a distributed system composed of a Button and a Led, each controlled by a different computational node. The system must initially provide a very basic function: a Led is turned on and off each time a Button is pressed (by an human user).
> In the future, the functionalities of the systems could be extended, e.g. by allowing a Button to blink a Led, to turn on/off many Led etc.

We read the text file that describes the requirements and immediately plan some work to:

- define the model of the `Led` (question: *what* is a Led in the user domain space?);
- define the model of the `Button` (question: *what* is a Button in the user domain space?);
- define the (model of the) *Use Cases* (i.e. define the functions/features of the software system).

5.4 The key-role of problem analysis

After the analysis related to the system components, we ask ourselves whether the models of the Led and of the Button defined in cooperation with the user can be also suited to face the problem of building the required distributed software system. When it is not the case, we advert a conceptual distance between the basic components and the needs of the application (i.e. we find an abstraction gap) that can be reduced by describing the single components and the whole system by means of a custom meta-model, no more based on classical objects but on *actors*.

> A proper problem analysis that starts from the available components, formally expressed by (executable) models can provide a solid guide for agile (SCRUM-based) project and implementation phases, by reducing the risks and the costs for the software company.