LibsensorPy: A Library to Improve the Development of Ubiquitous Applications on Raspberry Pi

Edivaldo M. F. de Jesus Jr[*]
Instituto Federal da Bahia
Rua Emídio dos Santos, S/N
Barbalho, Salvador Bahia
juniorug@gmail.com

Manoel Carvalho Marques Neto[†]
Instituto Federal da Bahia
Rua Emídio dos Santos, S/N
Barbalho, Salvador Bahia
manoelnetom@gmail.com

ABSTRACT

Commonly, programmers who have no knowledge in electronics, and engineers, who have no programming experience, are involved in projects that aims to build ubiquitous systems. To create such systems, they often face the problem of not having the specific knowledge to deal with activities like: implement system's abstractions, configure logic connections between sensors and a computer device, capture, understand and process data from sensors in order to make them understandable, etc. To fill this gap, this paper presents a library named LibsensorPy, an extensible library which allows to abstract much of the complexity of these activities and more easily interact with an environment through sensors and actuators coupled to a Raspberry Pi computer.

Keywords

Ubiquitous Computing; Raspberry Pi; Sensors

Introduction

Researches on Ubiquitous Computing (UbiComp) include topics such as: basic access to any wireless device, transparent mobility support on the network, safety, context treatment, energy efficiency, multimedia content presentation, intelligent interactive environments, etc [1]. This work is focused on building intelligent interactive environments. In these environments, the fundamental idea is to create ways to allow the remotely and pervasive use of a computer device. For this purpose, the use of platforms to integrate devices that make up these environments is one of the key points to create such environments. Currently, there are some options to fill this gap like: Arduino, BeagleBoard, Intel Galileo and Raspberry Pi [6, 7, 10, 11]. In this paper we focus on Raspberry Pi (RPi) project.

The RPi is a widely used platform for professionals who are interested in the field of ubiquitous applications. It provides basic interfaces to prototype ubiquitous projects, especially for those who must be powered by battery. The platform allows the use of high-level programming languages that are quite widespread as C/C ++, Python and Java, in a range of projects. The use of platforms (such as RPi) by programmers and engineers for the development of ubiquitous projects that use context-aware computing often face a problem: commonly programmers have no knowledge in electronics and engineers have no programming experience. To create such systems, they need such specific knowledge to, for example, implement system's software abstractions; configure electrical and logic connections between sensors and a RPi; capture, understand and process data in order to make them understandable, etc. To fill this gap, this paper presents a library named LibsensorPy.

The LibsensorPy library allows users (programmers or engineers) to write software that use context-aware computing to interact with an environment through sensors and actuators coupled to the Raspberry Pi abstracting much of the complexity of these activities. We claim that this abstraction allows users with low experience in programming/electronics to develop their projects more easily. The library abstracts many sensors as software objects. These objects encapsulates all the specific functions to create units converters, create analog-digital converters, treating events, etc. Therefore, a user could focus only on using the software objects required in an application. Another consequence of library usage is reducing the number of lines of code needed to write a program.

This paper is structured in five sections. Section 1 presents an overview of the Raspberry Pi Plataform. Section 2 describes related work used as inspiration for this paper. Section 3 presents the LibsensorPy project and implementation. The Section 4 presents the conclusions of this paper and the last section lists the references used.

1. RASPBERRY PI OVERVIEW

The RPi is a credit card size computer that can be used as a common personal computer and/or in ubiquitous systems projects [10]. The purpose of the RPi is to be a low-

[*] Student from the course of Analysis and Systems Development (ADS).

[†] Ph.D. in Computer Science, Professor and Researcher.

WebMedia'15, October 27–30, 2015, Manaus, Brazil.
© 2015 ACM. ISBN 978-1-4503-3959-9/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2820426.2820452.

cost computer, with the ability to interact with the outside world through the sensors. It was first designed to be used in educational environments to assist and encourage the teaching of programming and help understand how a computer work. However, since the very beginning, the RPi has been used by people of different ages and interests in projects like automation, sensing and robotics, games, multimedia, etc. Currently there are three RPi models named A, B and B+. The main differences are the number of USB ports, the presence of Ethernet port and amount of available memory.

Another important aspect RPi hardware is the group of General Purpose I/O (GPIO) pins. These pins are programmable ports to input and output data used to provide an interface between the board and real world. The number and options of input and output from a RPi depend on the model used. The RPi does not have analog pins and all of then may have more than one function (for example, input or output).

Raspberry Pi uses Linux-kernel-based operating systems installed in a minimum 4 GB size SD card. As in traditional computers, these operating systems are the basis for the APIs and tools that enable software development. Most of these OS already includes many development tools like Java, Python, C, etc. In addition to the operating systems default API's, there are many others that enable the use of GPIO. The next section presents some related works that were used as inspiration for this paper. They all try to, somehow, improve the use of platforms such as RPi, Arduino, etc. However, they still require to their users some level of skill in programming and electronics.

2. RELATED WORK

The Pingo project [2] is an uniform python API to program devices like the Raspberry Pi, Arduino, pcDuino, Intel Galileo, etc. Its an object-oriented API where each board is a subclass instance of a Board class. Every board has a dictionary named "pins" which lists all GPIO pins on the board. Each pin is a subclass instance of a Pin class with attributes that users can inspect to learn about its capabilities.

The Pi4J project [7] is a wrapper based on WiringPi project [4] and aims to provide an API written in Java also for manipulating GPIO interface. It supports all the Raspberry Pi hardware models available, abstracts the low-level hardware native integration and the interruptions monitoring allowing Java developers to maintain focus on implementing the business logic of an application.

The Adafruit's RPi Python Code Library[1] (RPi.GPIO) can be used for controlling a variety of electronics with a Raspberry Pi. It is also a library written in Python used to control GPIO on RPi. This library provides a collection of Python scripts to work with sensors, providing low-level class/objects abstractions. Despite being a library with a considerable amount of functions, the library does not use a software pattern to abstract sensors.

The option to control GPIO pins in this paper was to use RPi.GPIO as the basis for the new library (see Section 3) proposed here. The main motivation was the use of the Python language. It considered a very high level language because of its simple syntax and its dynamic typing. Moreover, it is an interpreted language which makes it a great option for scripting and has a wide range of features for embedded systems.

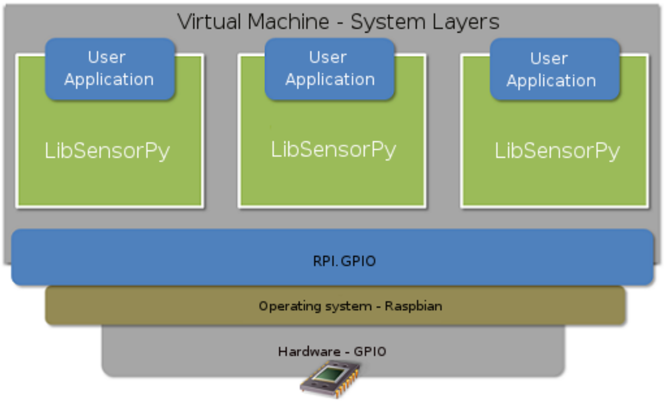

Figure 1: LibsensorPy's solution layers

One of the limitations of the libraries presented here is related to the use of sensors. None of the libraries compatible with the RPi provide, by default, constructs for representing sensors, actuators or events as objects. This implies that, to use a sensor, a user need to know all the complex details of its operation to design these objects. These details include: unit conversion, digital-analog processing, event handling and more. The problem is that these details require expertise, in both programming and electronics, which are not easy to understand and implement for both engineers and developers.

To fill this gap, the next section presents LibsensorPy, an extensible library based on RPi.GPIO, which allows a user to create ubiquitous systems even with a low-level of experience in electronics or programming. A user only need to focus on using software components that make up a solution since LibsensorPy provides the constructs for the main necessary resources in both programming and electronics expertises. It allows a user to easy write ubiquitous applications to interact with an environment through sensors and actuators coupled to the RPi. Moreover, it is extensible and supports adding new sensors / actuators.

3. LIBSENSORPY

A solution that uses LibsensorPy can be divided into five layers (see Figure 1): i) Hardware (GPIO, sensors, actuators, SDCard, network and power supply), ii) Operating System, iii) RPi.GPIO, iv) LibsensorPy and v) User Applications. LibsensorPy provides a simple abstraction layer between user's application and RPi.GPIO API. It makes it easier the use of hardware interface (especially GPIO), allowing, for example, to reduce the number o code lines necessary to write an application and also to include new hardware (sensors or actuators). The focus of this work is to abstract, through LibsensorPy, the complex details required to use sensors for data capture. Thus, we defined the Libsen-

[1] `https://pypi.python.org/pypi/RPi.GPIO`

sorPy main requirements in order to meet this focus. To implement these requirements we adopted the object-oriented analysis and design (OOAD) under the architectural pattern named Sense-Compute-Control (SCC) [3, 9].

3.1 Main Features

The LibsensorPy has some features that allow to create instances of components as well as a standardized and flexible methodology to add new components. The first import feature of LibsensorPy is the use of abstract factory pattern [5] to encapsulate a group of individual factories that have a common theme (sensors, actuators and events) without specifying their concrete classes. This ensures independence on how components are created, composed and represented. The main abstract factory classes of LibsensorPy are AbstractSensorFactory, AbstractSensor, AbstractActuator and AbstractEvent.

In addition to the abstract factory common classes, LibsensorPy also offers a number of concrete classes that represent sensors, actuators and events commonly used in ubiquitous applications. The library's structural view as well as a complete list of all components and the API documentation are on the library site [2].

Another feature of LibsensorPy is its ability to support sensors compositions. Some sensors may be used for more than one purpose. For example, DHT11 sensor can be used to measure temperature and humidity at the same time. In this case, LibsensorPy creates three sensors: i) to measure temperature - DHT11Temperature, ii) to measure humidity - DHT11Humidity and iii) a composite sensor that aggregates the two basic sensors - DHT11Composite. This solution is in line with the composite pattern [8]. It allows basic and composite sensors being viewed by a user in the same way. The idea is to allow the creation of lighter objects.

An important feature of LibsensorPy is the Extensibility Capacity. It allows the easy library update avoiding the impact of changes over an user application. However, note that a LibsensorPy update requires user programming skills. A user can add new components by implementing a new concrete factory (and its other components like concrete sensors, actuators, etc.) and overwriting the *createSensor()* method ensuring system compatibility. Thus, to add a new sensor family, a user have to create a new factory that implements the abstract methods of AbstractSensorFactory interface, observing thus a "contract", where attributes are specified and methods that a concrete sensor class in this family have to implement.

3.2 Example of Use

This section presents one example of use conducted with LibsensorPy. This example consisted in the development of an applications that aims to demonstrate the ease of use of LibsensorPy.

The example of use consisted in creating a simple application to measure the distance in centimeters between a HCSR-04 ultrasonic sensor and any real world object. To demonstrate the advantages of using LibsensorPy, first, we write the application code using only the RPI.GPIO API. After this, the example was completely rewritten with LibsensorPy. The following code shows the first version of this application:

```
import time
import RPi.GPIO as GPIO

GPIO.setwarnings(False)
GPIO.setmode(GPIO.BCM)

ECHO = 17
TRIG = 27

if sensor == 0:
    GPIO.setup(ECHO,GPIO.OUT)
    GPIO.setup(TRIG,GPIO.IN)
    GPIO.output(ECHO, GPIO.LOW)
    time.sleep(0.3)
    GPIO.output(ECHO, True)
    time.sleep(0.00001)
    GPIO.output(ECHO, False)

    while GPIO.input(TRIG) == 0:
      signaloff = time.time()

    while GPIO.input(TRIG) == 1:
      signalon = time.time()

    timepassed = signalon - signaloff
    distance = timepassed * 17000

    print ("distance in cm" + distance)

    GPIO.cleanup()
else:
    print "Incorrect usonic() function varible."
```

In the above code, we can realize that a user (programmer or engineer) must know the details of the HC-SR04 sensor to write an this application. These details include aspects of electronics, physics and programming languages. They will be summarized and explained below.

The HC-SR04 has four pins: ground (GND), Echo Pulse Output (ECHO pin), Trigger Pulse Input (TRIG pin), and 5V Supply (Vcc). We power the sensor using RPi Vcc pin, ground it using RPi GND pin, and use the RPi board to send an input signal to TRIG, which triggers the sensor to send an ultrasonic pulse. The pulse waves bounce off 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 returned pulse, and then sends a 5V signal on the ECHO pin. ECHO will be "low' (0V) until the sensor is triggered when it receives the echo pulse. Once a return pulse has been located ECHO is set "high" (5V) for the duration of that pulse. Pulse duration is the full time between the sensor outputting an ultrasonic pulse, and the return pulse being detected by the sensor receiver. The application must therefore measure the pulse duration and then calculate distance using time and distance formula: distance=time*soundspeed.

The HC-SR04 sensor requires a short 10uS pulse to trig-

[2] http://libsensorpy.com/

ger the module, which will cause the sensor to start the ranging program (8 ultrasound bursts at 40 kHz) in order to obtain an echo response. So, to create a trigger pulse, we set out trigger pin high for 10uS then set it low again. Another important detail is the speed of sound. We have used the speed of sound as 340m/s but we need to divide our time by two because what we've calculated using distance=time*soundspeed is actually the time it takes for the ultrasonic pulse to travel the distance to the object and back again.

An important question is: does a user need always to known all these complex details? To this work the best answer is: NO. The philosophy adopted in LibsensorPy is that for most of these applications the user simply wants the distance to an object. Following this philosophy, the code bellow shows the previous example now using LibsensorPy:

```
1 import CompositeSensorFactory
2
3 hcsr04=CompositeSensorFactory.createSensor("HCSR04")
4
5 print ("distance in cm" + hcsr04.distance_in_cm())
```

In the above code, we can realize that all the details of the previous example have been hidden in an HCSR04 object provided by LibsensorPy (line 3). This object encapsulates the implementation details, exposes the interface for use (through methods) and allows to calculate the distance in centimeters for real world objects. Another advantage realized in this example is the reduction of the amount of code required to write an application. Note that, in this example we use the default pins defined in LibsensorPy for HCSR04 object. However, the HCSR04 object allows to choose any of the available RPi pins.

4. CONCLUSION

The development of ubiquitous projects that use context-aware computing on platforms (such as RPi) often face a problem: commonly programmers have no knowledge in electronics and engineers have no programming experience. To fill this gap, this paper presented the LibsensorPy, an open source library designed to facilitate the creation of ubiquitous applications that use sensors and actuators to capture and process context data. It aims to simplify the creation of these systems using a Raspberry Pi board.

The LibsensorPy offers many concrete classes by default that represent the sensors, actuators and events commonly found in ubiquitous applications. Despite offering various components by default, there are still many others can be added to the library. This is the motivation why we created this library as an open-source initiative. This allows anyone to contribute improving the library.

The use of LibsensorPy in ubiquitous applications allows to reduce the amount of code lines, as well as increasing the abstraction of how to use the hardware components. The architecture and implementation of the LibsensorPy were explained in detail, facilitating the reproduction of this work as well as its extensibility. The LibsensorPy architecture is well established, but its definition is not trivial, since it allows the application to abstract details of an implementation.

The next steps of this research include the creation of mechanisms to make LibsensorPy a cross-platform library. This will be done through the creation of a new layer in LibsensorPy. This layer will be based on techniques of software prototyping to the design of interactive components. The main advantages of prototyping technique are: the closeness of the system with the needs of users, the reduction of misunderstandings between users and also the reduction in development effort. Prototypes are concrete models and represent both the description of information structure as information instance that must be present in an application.

5. REFERENCES

[1] Abowd, G. D., and Mynatt, E. D. Charting past, present, and future research in ubiquitous computing. *ACM Transactions on Computer-Human Interaction (TOCHI) 7*, 1 (2000), 29–58.

[2] Clube, G. H. Pingo, a uniform api to program devices like the raspberry pi, pcduino, intel galileo etc. http://www.pingo.io/docs/, October 2014. [Online; accessed 14-march-2015].

[3] Edwards, G., Garcia, J., Tajalli, H., Popescu, D., Medvidovic, N., Sukhatme, G., and Petrus, B. Architecture-driven self-adaptation and self-management in robotics systems. In *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on* (2009), IEEE, pp. 142–151.

[4] Henderson, G. Wiring pi gpio interface library for the raspberry pi. http://projects.drogon.net/raspberry-pi/, 2013.

[5] Hunt, J. Abstract factory pattern. In *Scala Design Patterns*. Springer, 2013, pp. 155–161.

[6] Kubitza, T., Pohl, N., Dingler, T., Schneegass, S., Weichel, C., and Schmidt, A. Ingredients for a new wave of ubicomp products. *IEEE Pervasive Computing*, 3 (2013), 5–8.

[7] Neto, M. C. M. Desenvolvimento de aplicações ubíquas com arduino e raspberry pi. In *Proceedings of the XX Brazilian Symposium on Multimedia and the Web* (João Pessoa, Paraíba, Brazil, 2014), WebMedia '14, ACM, pp. 30:1–30:30.

[8] Riehle, D. Composite design patterns. In *ACM SIGPLAN Notices* (1997), vol. 32, ACM, pp. 218–228.

[9] Taylor, R. N., Medvidovic, N., and Dashofy, E. M. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[10] Upton, E., and Halfacree, G. *Raspberry Pi user guide*. John Wiley & Sons, 2014.

[11] Wirth, M., and McCuaig, J. Making programs with the raspberry pi. In *Proceedings of the Western Canadian Conference on Computing Education* (2014), ACM, p. 17.