

Universitat Oberta de Catalunya

Degree's Project

**Analysis, Improvement, and Development of  
New Firmware for the Smart Citizen Kit  
Ambient Board**

Degree's Project on Computer Science  
Miguel Colom Barco

---

Supervised by:  
Pere Tuset Peiró

July 23, 2015

## **Analysis, Improvement, and Development of New Firmware for the Smart Citizen Kit Ambient Board.**

Miguel Colom Barco  
<http://mcolom.info>

This document is under the Attribution-ShareAlike 4.0 International (**CC BY-SA 4.0**) license<sup>1</sup>. You are free to:

- Share - copy and redistribute the material in any medium or format
- Adapt - remix, transform, and build upon the material for any purpose, even commercially.

Under these terms:

- Attribution - You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- ShareAlike - If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

---

<sup>1</sup><http://creativecommons.org/licenses/by-sa/4.0/legalcode>

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	Contextual bibliographic resources . . . . .	6
1.2	Keywords . . . . .	11
1.3	Main objectives . . . . .	11
1.4	Benefits . . . . .	12

## **I Improvement of the Firmware and Documentation of the current SCK**

**13**

<b>2</b>	<b>Current system overview</b>	<b>14</b>
2.1	System architecture . . . . .	14
2.2	Activity cycle . . . . .	16
2.3	Development environment . . . . .	17
<b>3</b>	<b>The sensors board</b>	<b>20</b>
3.1	I2C and SPI communication protocols . . . . .	20
3.2	Caching pattern . . . . .	22
3.3	The sensors . . . . .	23
3.3.1	Microphone . . . . .	23
3.3.2	Temperature and humidity (SHT21) . . . . .	24
3.3.3	$CO$ and $NO_2$ (MICS-4514) . . . . .	25
3.3.4	Lighting level (BH1730) . . . . .	30
3.3.5	Battery level . . . . .	31
3.3.6	Number of WiFi networks . . . . .	32
<b>4</b>	<b>The base SCK board</b>	<b>32</b>
<b>5</b>	<b>Engineering tasks in the official firmware</b>	<b>35</b>
5.1	Built-in Self Test (BIST) . . . . .	37
5.2	Contributions to the SCK firmware . . . . .	41
<b>6</b>	<b>Conclusions</b>	<b>46</b>

## **II Development of Firmware for the New RTX4100 Low-Power WiFi Module**

**48**

<b>7</b>	<b>Introduction</b>	<b>49</b>
----------	---------------------	-----------

<b>8 Objectives</b>	<b>49</b>
<b>9 RTX4100 hardware description</b>	<b>50</b>
<b>10 Software architecture</b>	<b>55</b>
10.1 Persistent data . . . . .	58
<b>11 The new SmartCitizen WiFi API</b>	<b>61</b>
11.1 Requirements . . . . .	61
11.2 Implementation details . . . . .	61
11.3 Hardware integration . . . . .	67
<b>12 Communication with the upper layer</b>	<b>68</b>
12.1 Introduction . . . . .	68
12.2 Hardware integration . . . . .	69
12.3 SPI commands reference . . . . .	71
<b>13 Conclusions</b>	<b>75</b>

## List of Figures

1	Current SCK with both layers mounted. Only the top layer with the sensors is visible. On the left, the battery that powers up the system. On the right, the SCK. . . . .	14
2	Detail of the sensors top board. . . . .	15
3	Detail on how both the sensors (up) and main (down) boards are connected together. As can be seen, on the right there is a connector that contains several input/output pins that are used to retrieve information from the sensors. For most of the sensors the I2C communication protocol is used, but it is possible to use SPI too, and even to read analog values. . . . .	16
4	Activity cycle of the SCK, from the time it is powered up. . .	18
5	Frontal detail of the Pro Signal ABM-705-RC microphone used in the SCK. Image obtained from the UK Farnell website. . .	23
6	Detail of the SHT21 temperature and humidity sensor by Sensirion. . . . .	24
7	MICS-4514 sensor without the protective cap, showing both $CO$ and $NO_2$ captors. . . . .	26
8	Measurement schematic for the MICS-4514, to detect $CO$ and $NO_2$ gas concentration. . . . .	27
9	MICS-4514 $CO$ concentration depending on the ratio $R_S/R_0$ . .	27
10	MICS-4514 $NO_2$ concentration depending on the ratio $R_S/R_0$ . .	28
11	Detail of the base board. . . . .	33
12	Commits and merges of all contributors in the forked SmartCitizen project. The two arrows that go from the official project (blue line) to the forked project (black line) are the merges with the upstream. Image from the GitHub website. . . . .	37
13	UML class diagram of the classes involved in the new SCK BIST. . . . .	40
14	The Wireless Sensor Application Board docking station used to develop and test the firmware. It has an USB connector to communicate the board with the PC, an FTDI chip that implements an USB to serial converter (UART), and two connectors to plug the RTX WiFi module with its batteries. The image shows the RTX WiFi module with the batteries already plugging into the WSAB. . . . .	51
15	The low-power RTX4100 WiFi Sensor Application Board, also known as <i>WSAB</i> . . . . .	52

16	The RTX4100 module (behind, not visible) powered up with two AAA-type batteries. See Figure 17 to see the circuitry behind. . . . .	53
17	The RTX4100 module showing the circuitry. The AAA-type battery connector is attached. See Figure 16 to see the battery connector with the two AAA-type batteries plugged in. . . . .	54
18	CoLa controller tool provided by the Amelie framework to upload CoLa applications to the RTX4100. . . . .	57
19	Software architecture in the RTX4100. Source: RTX41xx Wi-Fi Modules, User Guide UG3 Application Development [20]. . . . .	57
20	Mail mechanism used by the ROS protothreads. The main CoLa task protothread (PtMain) creates all protothreads needed in the firmware application. All threads send their mail messages to a mail scheduler, which puts each mail message into a queue. The main thread dispatches the mail messages in the queue, that is, it takes the message which arrived last and the scheduler sends it to the appropriate protothread recipient. This mechanism to put messages in a queue to be dispatched later and sent to the appropriate recipient is known as the <i>Publish-Subscribe</i> design pattern [15]. . . . .	59
21	Terminal session with the Putty program, executing the <code>test</code> command (unit test) using COM4 in a Windows system. Private data (WiFi SSID and key password) have been hidden. . . . .	63
22	GPIO pins distribution at the Raspberry Pi B. For SPI it uses pins #19 (MOSI), #21 (MISO), and #23 (SCLK) (red square). . . . .	68
23	SPI communication between the base board (upper layer) and the RTX4100 in production mode. The test the SPI commands, any board able to communicate through SPI is valid, as for example a Raspberry Pi or most of the Arduino boards. In red, the Arduino-like ICSP header (see Figure 24). . . . .	70
24	The ICSP header pins in Arduino boards used for SPI communication. . . . .	70

# 1 Introduction

This document describes my Project of the Computer Science degree at Universitat Oberta de Catalunya (UOC), in the *Arduino* area. Specifically, the objective was to develop firmware for both the Arduino-compatible boards of the Smart Citizen initiative, and for the RTX4100 low-power WiFi.

The objective of the Smart Citizen project [1] is to foster the people in cities to collect environmental data (level of noise, concentration of gases, or humidity, among others) using sensor boards owned by the citizen users (known as Smart Citizen Kit, SCK). These data are transmitted to a centralized platform that makes public all the collected information. This enables anyone to browse the data over a map in the web or a mobile application.

The specific goal of this project is to improve and optimize the firmware of the current versions of the sensor boards and also to write code for the next hardware version of the boards. Since the SCK, maintains compatibility with the Arduino IDE<sup>2</sup> [2] and both share the same Atmel microcontrollers found in the Arduino Leonardo [3]. Therefore it is possible to program it with the Arduino IDE, even if the SCK is not an Arduino board itself.

This project has two independent parts. In the first part the current state of the firmware code corresponding to the sensor board will be analyzed, optimized, and any found errors will be corrected. In the second part of the project it will be designed a brand new API<sup>3</sup> to interact with the new low-consumption WiFi module that will be used at the new SmartCitizen boards. These new boards are not yet in production and, consequently, there is not any firmware available. Therefore, it will be designed and written from scratch.

## 1.1 Contextual bibliographic resources

This section recommends five articles that will help the reader to better understand the context of the project, especially the Smart Cities concept. For each bibliographical resources it is listed: (1) Title, (2) Authors, (3) Date, (4) Source, (5) Summary, and (6) Explanation why the resource relevant for the project.

- 1) **Understanding Smart Cities: An Integrative Framework** [4]
- 2) Chourabi, H. and Taewoo Nam and Walker, S. and Gil-Garcia, J.R. and Mellouli, S. and Nahon, Karine and Pardo, T.A and Scholl, Hans Jochen.
- 3) January 2012.

---

<sup>2</sup>Integrated Development Environment.

<sup>3</sup>Application Programming Interface.

4) International Conference on System Science (HICSS), 2012 45th Hawaii.

5) The concept of “smart cities” is emerging strongly during the last years with the objective of solving problems of urban population in different kind of cities, caused by the fast grow in the number its habitants. However, just a few articles in the literature take care of the problem and this paper tries to explore the problem and identify what are the characteristics that should have any smart city. The authors identify up to eight significant factors: management and organization, technology, governance, policy context, people and communities, economy, built infrastructure, and natural environment.

The ability of the habitants of the city to acquire and share information is vital, since the kind of information that can be collected has to do with pollution, quality of air, quality of water, light pollution during night, or levels of noise, among others. These indicators have a clear impact on the economic and environmental health of the city and its citizens. If the data is made public governments can take improvement actions.

6) This article without any doubt is relevant for the project, since it defines objectively what is a smart city, with eight indicators, and the authors explain why having a smart city is a great benefit for the health and well-being of their habitants.

Although it is not a technical paper (as the selected other four), it is important because it gives a sense to the question of why we should have plenty of boards able to acquire data along the cities. They are cheap indeed, but this is not a reason by itself to have them around.

However, there are deeper reasons than the technical possibility to create smart cities, and these are the great benefits that come to the habitants: independent public data for everybody (independent from the official data that governments may collect), and the possibility to use this information to press governments to improve the quality of the cities.

1) **Sensor Networks for Ambient Intelligence** [5]

2) Pauwels, E.J. and Salah, Albert A. and Tavenard, R.

3) October 13-16.

4) IEEE 9th Workshop on Multimedia Signal Processing, 2007. MMSP 2007.

5) This article has two well differentiated parts. In the first, the authors do a review of the state of the art in multimodal sensor networks. Multimodal networks are made of sensors which acquire data of different nature, as sound level, light intensity, or gas levels, for example. Nowadays technology has evolved enough to allow the sensor boards to be small, cheap, and with low power consumption. Therefore, now it is possible to perform pervasive



computing techniques in a highly distributed and global system.

In the second part of the article it is discussed how to obtain temporal patterns using the data coming from several sensors. This cross-signal correlations allow to study the evolution of some magnitude along the space and time. For example, gas movement throughout the city along different times can be tracked.

Therefore, this article does not only brings up knowledge on the technical details of the state of art devices able to create large distributed sensor networks, but also proposes a method to obtain higher level information combining the data coming from sensors in different time series.

6) This article is relevant for this project since it reviews on one hand the state of the art on multimodal sensor boards and, on the other hand, how to integrate information from several sensors to obtain additional information.

The SCK is a multimodal board, since it contains sensors for noise level, light intensity, CO and NO<sub>2</sub> gases, temperature, and others, and therefore the review that can be found in this article is absolutely related to the kind of board the SCK board is. Knowing what are the best design pattern for both the hardware and software of sensor boards will allow to avoid design errors. In the case of this project, the hardware is already designed, but there is still room for software refactoring.

About the integration of data coming from several sources, it might be useful in order to detect defective boards that give outlier values compared to nearby different boards.

1) **An introduction to I2C and SPI protocols** [6]

2) Leens, F.

3) February 2009.

4) Instrumentation Measurement Magazine, IEEE

5) This article discusses about two protocols that have standardized the communication between electronic components: the inter-integrated circuit (I2C) and the serial peripheral interface (SPI) protocols. Both two are suitable and wide used to communicate integrated circuits among them with on-board devices.

The I2C protocol (“Inter IC<sup>4</sup>”) is a simple protocol designed to communicate integrated circuits inside a single board. Although initially it was designed for slow communication (up to 100 kbps), nowadays the protocol allows to a high speed mode up to 3.4 Mbps. It is widely used in part because of its simplicity, since only two bus lines are required, with flexible strict baud rates, simple master/slave relationships between all components,

---

<sup>4</sup>Integrated Circuit.

and software-addressable devices.

The other communication protocol covered by the article is SPI, for which it is possible to obtain higher data transfer rates than I2C. It has duplex capacity (communication in both senses at the same time). It follows a master/slave relationship as I2C, but it lacks any device addressing.

6) This article presents how the I2C and SPI protocols work in order to achieve that different integrated circuits over the same board are able to communicate. Of course, these two protocols have a major interest for this project, since the architecture of the SCK matches exactly the aim of the protocols: to intercommunicate with a common standard different kinds of sensors. In the case of the SCK, most of the sensors are able to communicate as slaves within the I2C protocol, while the ATmega microcontroller takes the role of the master. Even if the Arduino IDE offer an standard library to I2C communication (called “wire”), the details on how to initiate a communication, or the addressing are still needed to be known by the programmer. However, so far no SPI sensors are found in SCK, but they might be used in the future.

1) **Built-In Self-Test Techniques** [7]

2) McCluskey, E.J.

3) April 1985.

4) Design Test of Computers, IEEE.

5) This classic article discusses several techniques for the BIST<sup>5</sup>, that is, the possibility of configuring a device in a test mode in order that it generates patterns of input data to check if the device is working properly.

Most of the techniques described are based on the use of linear-feedback shift registers which are fed up with specific test patterns. The output of many components of the integrated circuits are analyzed while the test pattern is entering to look for any unexpected output. The improper outputs imply a malfunction of the integrated circuit. When the integrated circuit detects itself that some of its components is not working as it should, it signals the situation so the system can avoid using the failing component. This is especially important for fault-tolerant systems.

6) The hardware of the SCK board that will be used in this project is already build, but anyway any of its components may fail at any time. Therefore, doing a BIST for the complete board would improve the reliability of the whole system. For example, if a sensor fails and the firmware ignores the problem or does not attempt to detect the failure, then wrong data could be sent and shared. Although the article is now old, it is still valid for

---

<sup>5</sup>Built-In Self Test

nowadays designs.

A failure may not only affect the sensors, but also the ATmega microcontroller. Thus, performing a memory read/write test with several test patterns and checking that basic operations give the expected value (for example, sums, bitwise operations, or floating point computations, among others) may detect most of the hardware errors.

With the current version of the firmware only 4Kb of memory are available after programming the microcontroller. However, new versions of the board hardware may contain microcontrollers with more EEPROM, thus allowing BIST in production units.

**1) An Incremental Approach to Unit Testing During Maintenance [8]**

2) Harrold, M.J. and Souffa, M.L.

3) October 1988.

4) Proceedings of the Conference on Software Maintenance, 1988. IEEE.

5) This article describes a methodology to be used in parallel of the unit testing during the maintenance of the project. The authors define a framework where units tests are integrated in the programming environment. That is, unit testing is not performed not only at the end or after specific milestones have been reached, but continuously.

Each time a module is modified, the test case may be modified accordingly; if a new feature is added, then a new test case is created and added to the programming framework. When the modification is a major update (for example, a new version of the program with significant changes), then a human tester needs to decide which of the test cases must be modified, which removed, and how to modify the existing tests for the new version in order to adapt the tests set to the new version.

6) A significant part of the work during this project will be testing, because it is a requirement of the program that any change made to the firmware is tested for validity. In fact, this is a minimal requirement that any software component should meet.

The development methodology that will be followed in this project implies creating test cases for any functionality of the system, implementing the code for the functionality and committing the change to the code repository only if all tests are passed.

This approach (and what the article proposes) is very close to the Continuous Integration (CI) methodology. However, for this project it will not be possible to use the CI approach strictly, since it requires automating the build. However, this is not possible with the current version of the Arduino IDE.

Nevertheless, other steps of the CI can be met during the development of the code of this project: single source repository, many commits into the mainline, public visibility of the repository.

## 1.2 Keywords

These are the keywords which define this project:

- Ambient sensors
- Data sharing
- Embedded system
- Arduino
- Unit testing

## 1.3 Main objectives

The project has two clearly independent parts.

The first part is related to the current version of the SCK sensor board and implies:

- Correct any detected bugs in the current code;
- Improve the current code (speed and memory used);
- Perform continuous refactoring of the code in development;

The objectives of the project changed after they were first defined given that the tasks in the first part did not seem to be enough for a complete Final Degree Work. Therefore, a second part was added to the project.

The goal of the second part of the project is to write from scratch the firmware from the new low-power WiFi module for the next versions of the future SCKs. Right now, it does not exist such a firmware and, therefore, the code written as part of this project will be the first version. The new firmware will provide an API that will make possible the communication between the other modules of the SCK and the WiFi module.

Therefore, the objectives of the second part of the project can be summarized as follows:

- Analyze and document the low-power WiFi module.

- Write firmware to make the chip set of the WiFi module work and provide an API.
- Document how the API works.

The following tasks are common to both parts:

- Make explicit the design of the system using class and activity UML diagrams;
- Perform continuous refactoring of the code in development;
- Define a unit testing plan and make the system pass it completely each time a new feature is added. Create a new unit test case for each new feature. For this project, the testing is really significant: no component of the system will be considered valid if a code change does not pass all unit testing cases;
- Follow a methodology close to Continuous Integration;
- Finish the final report of the TFG before its deadline.

## 1.4 Benefits

The Smart Citizen project has clear benefits for their users. Perhaps the most evident is the possibility of acquiring data by themselves using cheap hardware and to be able to share it worldwide, without having to rely on official data (which sometimes simply does not exist).

This data allows to evaluate an important indicator of what are called as *Smart cities*: the quality of the environment. That is, the level of noise or the concentration of unhealthy gases, among others.

For the users who install the sensor board indoor there is still an important benefit, since they will be able to detect concentration of gases at home that can put human life in risk, as *CO* and *NO<sub>2</sub>* gases.

Part I

# Improvement of the Firmware and Documentation of the current SCK

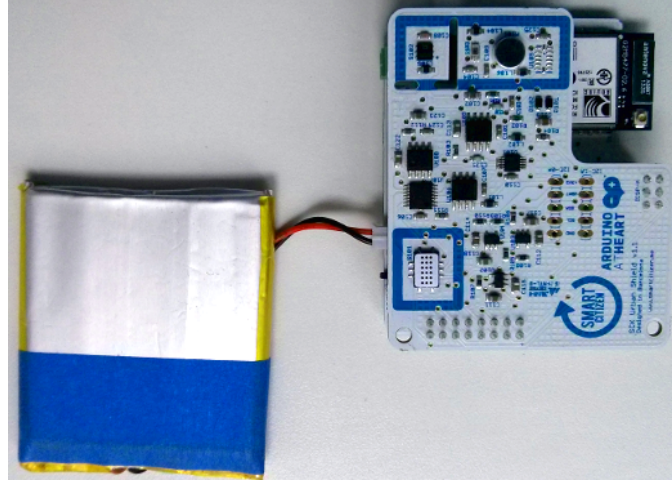


Figure 1: Current SCK with both layers mounted. Only the top layer with the sensors is visible. On the left, the battery that powers up the system. On the right, the SCK.

## 2 Current system overview

### 2.1 System architecture

The current version of the SCK consists on two boards mount one over the other. The top layer contains all the sensors and the bottom layer contains the ATmega CPU, the WiFi module, the circuit to charge the battery, and all other auxiliary circuitry. Both layers can work independently and it is possible to remove the current sensor layer and to change it by a different sensor board. The communication between the sensors and the bottom layer is done using standard inter-chip communication protocols (I2C mainly and SPI), making easy the interconnection of the two different layers.

Figure 1 shows on the right the current SCK with both layers mounted. It can only be seen the top layer, which contains the sensors. On the left, the battery which powers up the system when it works standalone. It is not strictly necessary to use a battery, since the SCK can also be powered by a micro USB<sup>6</sup> connector. Of course, using the battery allows to locate the SCK far away from the computer or any power source. Normally, the SCK is battery-powered.

Figure 2 shows a detailed view of the sensors board. The kind and number of sensors on the top board depends on the version of the SCK. In this version (SCK Urban Shield v1.1) the following sensors are available:

---

<sup>6</sup>Universal Serial Bus.

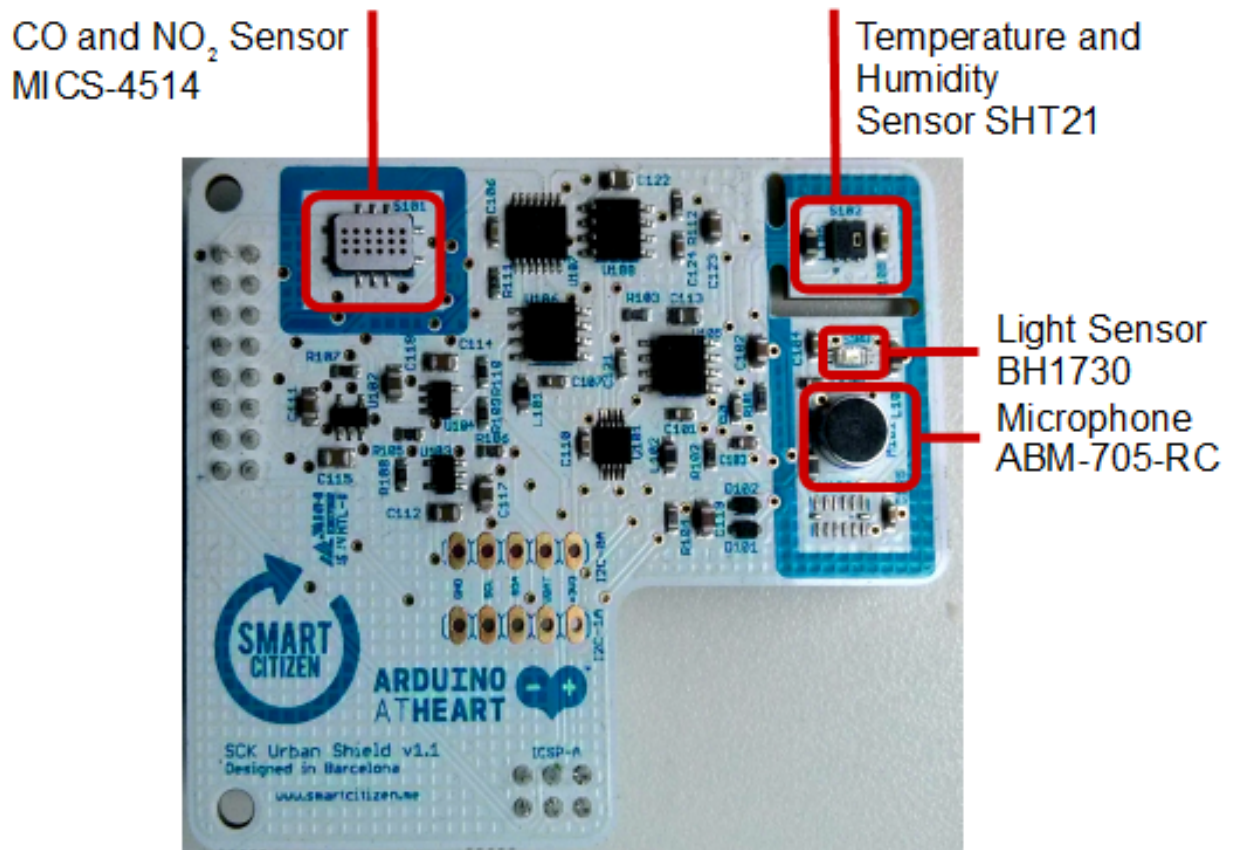


Figure 2: Detail of the sensors top board.

- Temperature and humidity (SHT21)
- $CO$  and  $NO_2$  gas concentration (MICS-4514)
- Lighting level (BH1730)
- Noise level (Pro Signal ABM-705-RC)
- Battery level (obtained by reading and averaging an analog pin)
- Number of WiFi networks (obtained from the RN131 “WiFly” module)

The sensors board contains all the ambient sensors and must be connected to the base board to work. Figure 3 shows how both boards are connected together: there is a bridge on the right (red circle) that contains several input/output pins that are used to retrieve information from the sensors.



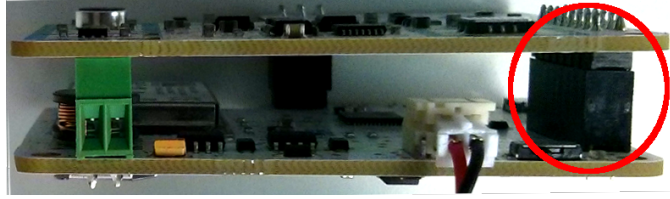


Figure 3: Detail on how both the sensors (up) and main (down) boards are connected together. As can be seen, on the right there is a connector that contains several input/output pins that are used to retrieve information from the sensors. For most of the sensors the I2C communication protocol is used, but it is possible to use SPI too, and even to read analog values.

For most of the sensors the I2C communication protocol is used, but it is possible to use SPI too, and even to read analog values (see Section 3.1).

The SmartCitizen firmware needs to minimize the energy consumption of the SCK. Since the WiFi module needs only to be active when it has to send information to the central platform, the rest of the time it is in *standby* mode. Note that it is only the WiFi module that is put in standby mode, not the microcontroller unit. It could be possible to use a timer to wake up the microcontroller and to put it in standby the rest of the time. In that mode the microcontroller does not execute any instructions and its energy consumption is minimal. It can be waken up by a hardware interruption and, in the case of the SCK, it could be TIMER2 or any other. However, this modification of the firmware is out of the scope of this part of the project focused on the sensors board and is left as a future improvement.

The details on the activity cycle can be found in Section 2.2.

## 2.2 Activity cycle

From the moment the SCK is powered up the firmware is continuously running, until it runs out of battery or it is unplugged from the USB host.

As any standard Arduino program, the SCK code contains two main parts.

The first part is the setup of the system, that is performed as soon as the board is powered up or reset. Function `setup()` is automatically executed after power up. In the case of the SCK, it contains a call to `ambient_.ini()`; which configures the SCK timer and initializes the sensors board.

After `setup()` has been called, the `loop()` function is invoked. This function is executed in a endless loop. Each time the function has finished, it is called again. The `loop()` function calls `execute()` of class `SCKAmbient`,

which checks the last time the sensors were read. If the elapsed time is large enough, it starts the transmission of the data to the centralized platform using the WiFi connection.

The microcontroller unit only needs to be active to read the sensors and to send the data to the platform. The rest of the time it can be in standby mode in order to save energy (this is specially important if the SCK is powered by a battery). It can enter the standby mode by setting the AWAKE pin to HIGH. In this mode, it does not execute any more instructions, the timers are active and it is responsive to hardware interrupts. In fact, the only way to make the microcontroller get out of the standby mode is by a hardware interrupt. In the case of the SCK, TIMER #1 is configured to callback `ambient_.serialRequests()` each elapsed second. Each time the timer fires, the microcontroller unit gets out of the standby mode and resumes normal operation.

Inside `ambient_.serialRequests()` the timer is stopped, the sensors update, the ambient information is sent to the SmartCitizen platform, the AWAKE pin is set to LOW (it enters standby mode again), and finally the timer is configured to wake up the SCK in the next iteration. This way, the microcontroller unit is only full active each time it needs to send data to the platform and remains in standby mode the rest of the time, saving energy.

## 2.3 Development environment

This section discusses about the decision of having made the SCK Arduino-compatible and which IDEs are available to develop the SCK.

The SCK hardware is compatible with the Arduino system (it even shares the same microcontroller unit as the Arduino Leonardo, see Section 4). Some of the advantages of Arduino are the fact that it is open source hardware and software. There is also a wide community of Arduino users that develop applications and libraries worldwide, making easy to find pieces of hardware/software already available for many possible applications of Arduino.

Of course, it would have been possible to choose other alternatives instead of an Arduino-like device, but the advantages are many:

- It is cheaper than other more complex alternatives, as Raspberry Pi, HummingBoard, BeagleBone, or MinnowBoard, among others.
- The ATmega microcontroller, that share both the SCK and the Arduino Leonardo, have many analog and digital pins, with integrated ADCs<sup>7</sup>. This makes the design of the hardware of the SCK cheaper

---

<sup>7</sup>Analog to Digital Converter.

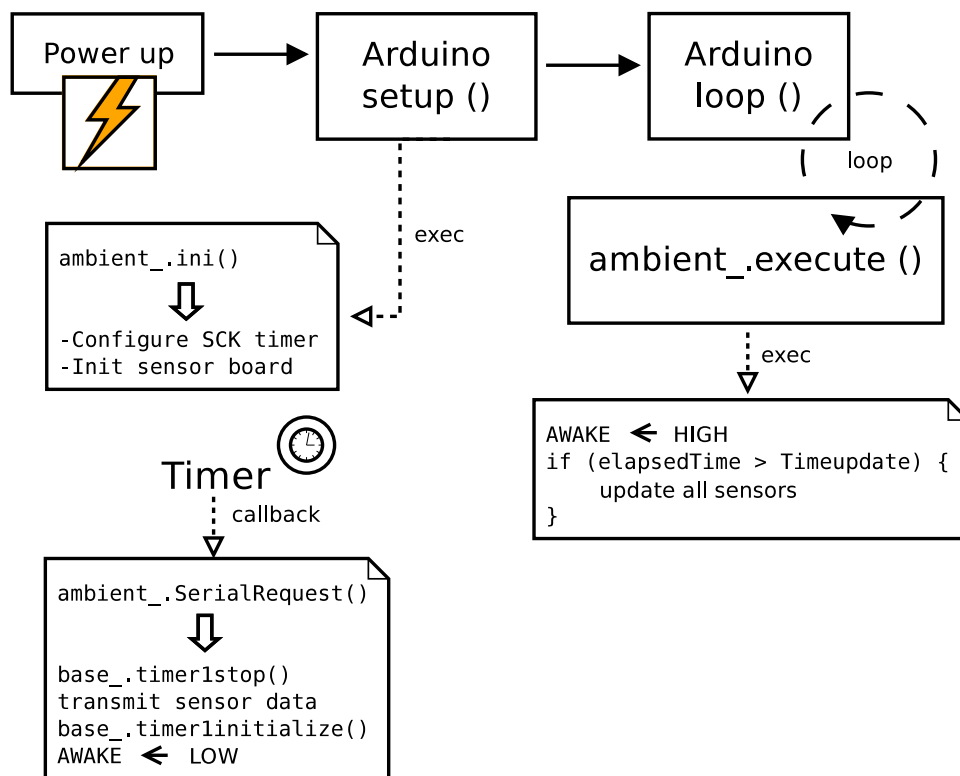


Figure 4: Activity cycle of the SCK, from the time it is powered up.

and simpler.

- Also, the ATmega microcontroller has built-in USB communication support and, again, it simplifies the design of the SCK hardware and also the software.
- There are plenty of well-known libraries for the Arduino that can be used directly on the SCK. For example, the `wire` library for I2C and SPI communication protocols.
- It is possible to directly use the C/C++ languages with Arduino. This is important for the SCK, since C/C++ allows for the low-level access required to communicate with the SCK hardware at the sensors board.

About the IDE environment, since the SCK is Arduino-compatible, it is possible to directly use the Arduino IDE (Integrated Development Environment) for coding. It simply suffices to configure the SCK as it was an Arduino Leonardo Board.

However, the Arduino IDE has serious limitations. For example, it is not multi-threaded and some operations needs to be finished before a new one can be started. For example, it is not possible to edit the source code if the IDE is compiling a large program. Another important limitation of the Arduino IDE is the impossibility to debug the code (could be solved by using a commercial debugging applications of an alternative IDE, Visual Micro, or the new 6.2 version of Atmel Studio). However, Atmel Studio is not free software (but cost-free).

The Arduino IDE lacks of code auto-completion and refactoring tools (rename, extract superclass, extract interface, use supertype where possible, encapsulate field, etc). Atmel studio supports code exploring, fast compile, firmware upload, debugging, serial terminal emulation, and its project format is compatible with the Arduino IDE. It supports the product development process with integrated tools and software extensions through the Atmel Gallery plugins.

As pointed out before, the Atmel Studio IDE alternative seems better from the point of view of the offered features, but it is not free software and could turn into proprietary software without notice, creating an undesired dependence.

There exist free software alternatives, as the Eclipse Arduino plugin, a professional and mature tool. It is also possible to use Arduino plugins with the NetBeans IDE, which are similar to the Eclipse Arduino plugins.

In conclusion, the Arduino IDE is the official IDE for the Arduino development, but it seems to be an immature tool yet. Better free software alternatives exist, as the plugins for Eclipse and Netbeans IDEs.

### 3 The sensors board

The SCK is made of two boards. On the top, the sensors board, which contains all the ambient sensors and communicates with the base board with the I2C or SPI protocols. As shown in Figure 3, there is a physical connector that joins both two boards and communicates the sensors and the base board with I2C or SPI. Section 3.1 gives a brief introduction of the I2C and SPI protocols.

The rest of the subsections here list the sensors used in the sensors board and discuss their characteristics and how the information is retrieved from them.

#### 3.1 I2C and SPI communication protocols

Both the I2C (Inter-Integrated Circuit) [9] and SPI (Serial Peripheral Interface) [10] protocols were designed to make it possible for different integrated circuits to communicate between them, using a common communication protocol and common electrical specifications.

The I2C protocol was created by Philips in 1982 in order to interconnect audio and other kinds of chips, while SPI was initiated by Motorola in 1985.

In the I2C protocol uses four wires, two for data transmission (SDA<sup>8</sup> and SCL<sup>9</sup>) and other two for power (Vcc and ground). It is a master-slave protocol, where the master initiates the communication and the slaves accept the query. Each device (master or slave) has an unique address (typically 7 bits, but up to 10 bits can be used for the address depending on the device).

The protocol is simple: the master puts the address of the slave device in the bus, then 1 bit to specify if it wants to read or write data, and then sends one or more commands. Then the slave will answer to the master and eventually there will be more data exchange between master and slave, depending on the concrete operation being executed.

The transmission is serial, bit by bit. The SDA line carries the actual information bits whereas SCL is a clock signal generated by the master. The maximum speed is 5 Mbps with Ultra Fast-mode (UFm).

In the case of the SCK, in the current version all sensors use I2C to communicate:

- SHT21 (temperature and humidity)
- MICS-4514 ( $CO$  and  $NO_2$ )

---

<sup>8</sup>Serial Data Line.

<sup>9</sup>Serial Clock Line.

- BH1730 (lighting level)

There are several advantages that justify the use of I2C:

- The hardware is easy to implement and debug, since it only uses two signals (data and clock).
- It is possible to select each device with an unique address. In the SCK case, it means that is easy to add more sensors without almost modifying the architecture of the system, just adding them over a new sensors board.
- Simple master/slave mechanism.
- In the case of the SCK, the I2C protocol is supported natively by the Arduino `wire` library.

On the other hand, there are some disadvantages:

- Not all devices may support high speed transmission rates. This is not a problem for the SCK, since most of the sensors do not need high speed rates.
- If low-speed devices are mixed with high-speed devices the final speed for all devices will be the slowest speed.
- Electrically I2C uses open-drain connections, which imply more current drain. For the SCK this is a clear disadvantage.
- Is not full duplex (but high-speed modes are available). For the SCK this is not an issue, since most of the time the SCK is in standby mode.

Since the new SCK boards will contain a new low-power WiFly module that communicates using SPI, this protocol will be discussed briefly here. Other sensor boards in the future may contain sensors that communicate with SPI too.

In the case of the SPI protocol four wires are used too:

- Clock (SCLK)
- Master Output Slave Input (MOSI)
- Master Input Slave Output (MISO)
- Chip Select (CS)

SPI also uses a master/slave schema, but it does not use numeric addresses to choose a device. Instead, it uses the CS signal to select which device the master needs to communicate with. When the CS signal connected to a device is active (with a LOW state), the slave device is selected and the communication begins. The MOSI and MISO signals determine if the master is writing and the slave reading (MOSI) or if the master is reading and the slave writing (MISO).

There are some advantages of SPI over I2C:

- It is full-duplex. However, for the SCK this is not important, since most of the information goes from the sensor to the microcontroller, and not the other way around.
- Less power consumption than I2C. This is important for the SCK, since it is battery-powered.
- Easy to select a slave chip using the CS signal. No need to use numeric addresses.

However, there are some disadvantages on using SPI:

- No hardware flow-control. In I2C it is possible for the slaves to pause the transmission if the rate is too high and their buffers are about to overflow. For the SCK it is not an issue, since most of the sensors are low-speed.
- There is not a formal standard for SPI and therefore it is not possible to say if a device is fully complaint with SPI or not. This is an issue for the SCK, since new sensors may be added to a sensors board and there is no warranty they work.

## 3.2 Caching pattern

The SCK implements the *catching* design pattern [11], in which the API is not forced to perform an actual and physical read of the sensor each time the layer that uses the API asks for sensor information.

Instead, the SCK timer fires once per second and updates the information of all sensors in a cache array. The sensors API simply returns the cached values each time the getter functions are invoked.

Using a cache for the sensors is specially important for the gas sensors, because it is not possible to read them continuously. Instead, a minimal amount of time (typically ten minutes) must elapse between reads. Therefore, a second layer of cache is used for the gas sensors. Even if the SCK timer



Figure 5: Frontal detail of the Pro Signal ABM-705-RC microphone used in the SCK. Image obtained from the UK Farnell website.

fires each second, the gas sensor getter simply checks if at least ten minutes have elapsed. If not, it returns the last cached value and returns it to the sensors API.

Note that the SCK does not pre-heat the sensors just before each read, but instead it keeps their corresponding heaters always activated. Of course, the disadvantage is the current consumption (needed to heat the sensors) but the reads are much more stable and accurate this way.

### 3.3 The sensors

In this section the captors over the sensors board are documented, giving details about the detection mechanisms, the communication protocols, and how the SCK accesses to them.

#### 3.3.1 Microphone

The SCK uses the Pro Signal ABM-705-RC microphone [12] (Figure 5 shows a detail of its frontal view).

It is a classic analog omni-directional microphone which can operate with a voltage that goes from 2v to 10v. This voltage range makes it adequate for its use within the SCK, since the typical 5v voltage used by low-power CMOS devices is within this range. The SCK can provide its required power voltage directly.

Also, its maximum current consumption is 0.5mA, which is really small and thus makes it a perfect candidate for its use in the SCK, given that it is battery-powered and it requires all devices to be low-power.

Its audio frequency range goes from 50Hz to 16KHz. This frequency range is sufficient for most applications, but it is possible to find noises whose frequency goes over 16KHz. We can think, for example, in the typical noise produced by old CRT televisions when they are working. This tone is 16744Hz and therefore it would not be detected by this microphone.



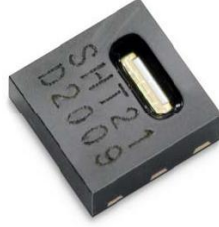


Figure 6: Detail of the SHT21 temperature and humidity sensor by Sensirion.

Also, it is possible for humans to hear (and to be bothered) by low-frequency sounds below the 50Hz threshold of this microphone. For example, the lowest note of a tuba or grand pipe organs is centered at 16.35Hz, and therefore not detectable by this microphone (but yet audible by humans!).

In conclusion, this microphone implemented in the SCK is valid for most applications but in next versions of the hardware of the SCK it should be considered to change it by some other microphone that at least covers the complete human frequency spectrum (20Hz-21KHz).

About its sensitivity, its datasheet is not clear about how many decibels corresponds to the low threshold, but the tests I've performed seem to indicate that the minimum noise level it is able to detect is about 15dB, which is sufficient for the SCK purposes.

In the SCK firmware, the noise level is obtained with `SCKAmbient::getNoise()`. The microphone is connected to an amplifier (the gain is configurable, but is fixed to 10000 in the code for this microphone type) and a filter. The output of the filter is connected to an ADC that turns the analog voltage level into a digital value between 0 and 1023. The output of the ADC is connected to the SCK digital pin  $S_4$ , which can be read using the Arduino `analogRead()` function.

Function `SCKAmbient::getNoise()` gives the results with respect to the  $V_{cc}$  level, that is,

$$L = \frac{\text{average}(S_4)}{1023} \times V_{cc} \quad (mV).$$

### 3.3.2 Temperature and humidity (SHT21)

The SCK uses the SHT21 sensor by Sensirion to obtain information about the environmental temperature and relative humidity. Figure 6 shows a detail of the sensor.

The interaction with the device is fully digital using the I2C protocol. It is able to give values of relative humidity with 12-bits and temperature

values with 14-bits resolutions.

There are several advantages on using this sensor in the SCK:

- Low power consumption (1mW when reading).
- It does not need any calibration and it is long-term stable. This is specially important for the SCK, since average users are not qualified or do not have the tools to calibrate the sensors. Compare this sensor with the SCK gas sensors described in Section 3.3.3, which need calibration and are not stable along time.
- Good resolution (12 bits for relative humidity and 14 bits for temperature reads).
- Wide temperature range, with accurate responses within the range -10 to 80 Celsius.
- Wide relative humidity range, with accurate responses within the range 8% to 90%.

The relative humidity (RH) is obtained with the following equation:

$$RH = -6 + 125 \times \frac{S_{RH}}{2^{RES}},$$

and the temperature (T) with this equation:

$$T = -46.85 + 175.72 \times \frac{S_T}{2^{RES}},$$

where RES is 12 bits when measuring relative humidity and 14 bits for temperature measurements.

In the SCK firmware, function `SCKAmbient::getSHT21()` updates two member variables when called: `lastTemperature` and `lastHumidity`. This function call another function `SCKAmbient::readSHT21(uint8_t type)` that receives as a parameter the corresponding command to the sensor.

`SCKAmbient::getSHT21()` calls `SCKAmbient::readSHT21(0xE3)` (read temperature) and `SCKAmbient::readSHT21(0xE5)` (read relative humidity) and updates `lastTemperature` and `lastHumidity`.

### 3.3.3 CO and NO<sub>2</sub> (MICS-4514)

The SCK can measure the level of CO and NO<sub>2</sub> gases with the MICS-4514 sensor (both captors are integrated inside the same package, as show in Figure 7). The sensor is completely analog and is based on the fact that the presence

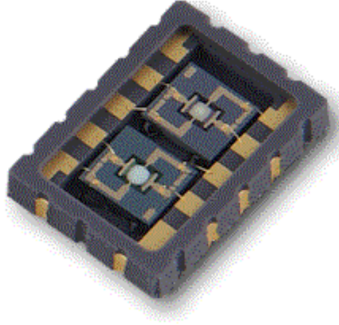


Figure 7: MICS-4514 sensor without the protective cap, showing both  $CO$  and  $NO_2$  captors.

of these gases produce free electrons in the surface of the detectors when they are absorbed. These free electrons are nothing else but a current that, after being amplified, can be detected. It is somehow similar to the process used to detect photons in CCD or CMOS devices with the photoelectric effect, but triggered by gas molecules instead of photons. Of course, both are completely different physic processes, but the idea behind the sensors is similar.

From the point of view of the detection circuits, the sensors can be seen as a resistance  $R_S$  whose value in ohms decreases as the concentration of gas increases.

Figure 8 show the typical measurement circuit. For both gas detectors, a load resistor  $R_0$  (typically  $820\Omega$ ) is used and the concentration of gas is obtained from two known curves. Figure 9 shows the  $CO$  concentration depending on the  $R_S/R_0$  ratio and Figure 10 the same for the  $NO_2$  concentration.

Since the detected gas concentration depends on the temperature of the sensor, it needs to be pre-heated. The objective is to bring the sensor to a high enough temperature that the ambient temperature is not significant with respect to the sensor temperature (more than  $200^\circ C$ ). This is a disadvantage for the SCK, since this makes impossible to perform consecutive reads without a time less than ten minutes between them. As explained in Section 3.2, the SCK implements a *caching pattern* strategy to be able to give interpolatable data to the platform, even if the less than ten minutes have elapsed since the last read. If the system reads the value before ten minutes have elapsed, the API simply returns the last value read. This way the software that uses the API can perform safely as many reads as needed without taking into account the elapsed time, since the firmware API will only execute an actual read in the sensor if the timing makes it possible.

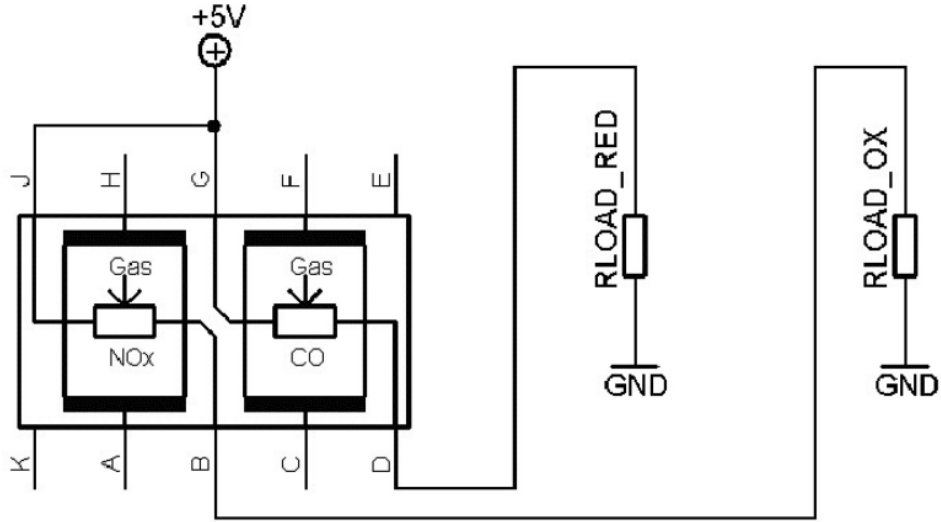


Figure 8: Measurement schematic for the MICS-4514, to detect  $CO$  and  $NO_2$  gas concentration.

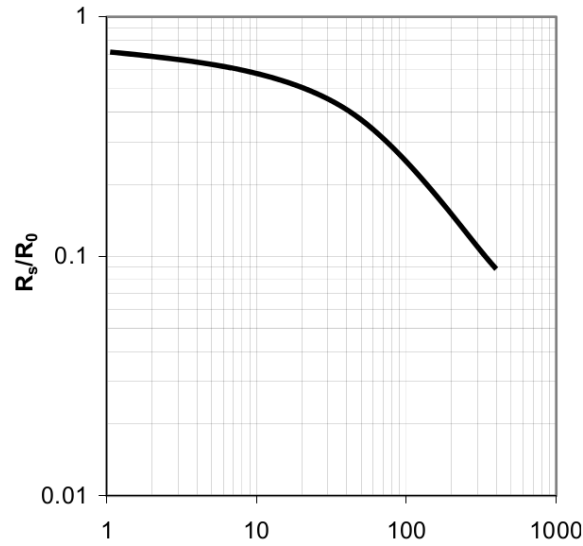


Figure 9: MICS-4514  $CO$  concentration depending on the ratio  $R_s/R_0$ .

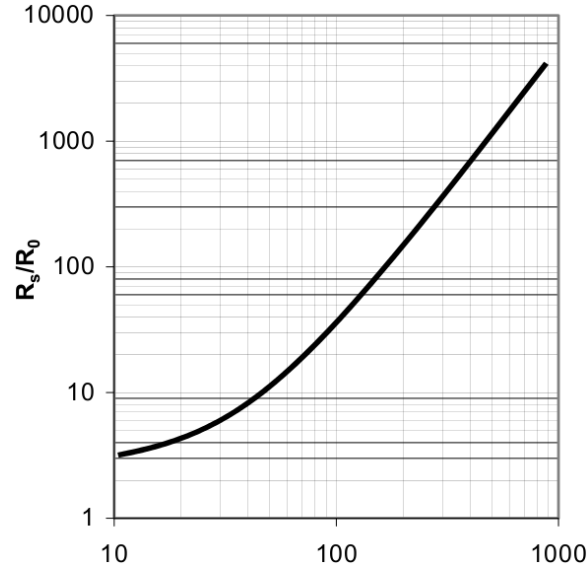


Figure 10: MICS-4514  $NO_2$  concentration depending on the ratio  $R_s/R_0$ .

There are several characteristics that make this sensor adequate for the SCK:

- The heating current is low. This is specially important since the SCK is battery-powered.
- The gas concentration detection range is wide.
- It can operate within a large temperature range.
- It combines in the same package both the  $CO$  and  $NO_2$  sensors.
- It is robust to vibrations and shocks. This is important for the SCK, since it is likely that some users place it outdoors, where the wind can shake it violently.

However, this sensor has a very important disadvantage (like many of the same technology), that is the necessity of re-calibrating the device after a few months, because the pairs of metal-oxide in the surface of the captor change their physical properties when exposed to the detectable gases. Therefore, the curves shown in Figures 9 and 10 are not valid after some months of continuous operation and the device needs to be re-calibrated in order to know exactly the actual concentration curves.

Of course, average users do not have access to the laboratory instruments needed to re-calibrate the MICS-4514 and they are likely to obtain wrong

values of gas concentration after some months of operation. This is a serious issue, since one of the objectives of the SCK project is to give citizens the possibility of obtaining reliable environmental data that could be used to convince governments of the need of improving the air conditions. If counterpart might are that the data is reliable because of the lack of calibration, the whole SCK could be considered non-reliable. For next versions of the SCK the use of better sensors that are not so sensitive to de-calibration should be strongly considered.

In the SCK firmware the actual read of the  $CO$  and  $NO_2$  (assuming that the ten minutes minimum time has already elapsed) is implemented with the following function:

```
void SCKAmbient::getMICS(){
    // Charging tension heaters
    heat(MICS_5525, 32); // mA
    heat(MICS_2710, 26); // mA

    RsCO = readMICS(MICS_5525);
    RsNO2 = readMICS(MICS_2710);
}
```

The `heat` function simply adjusts the current that goes through the heater in order to maintain a constant temperature.

Note that the firmware code assumes that in the circuit there are two different sensors (MICS-5525 and MICS-2710) instead of the MICS-4514. The reason is simply that MICS-4514 combines both the  $CO$  and  $NO_2$  sensors inside the same package, but from the point of view of the software there is not any difference between having separated MICS-5525 and MICS-2710 chips or a single MICS-4514, since the two captor in MICS-4514 operate independently (for example, they have they own pre-heater).

Function `void SCKAmbient::getMICS()` first pre-heat each captor and then read the value of concentration using function `readMICS`. Then, function `readMICS` reads the digital value at the output of the ADC connected before the output of the sensor and returns a value of concentration with respect to  $V_{cc}$ .

Note that function `void SCKAmbient::getMICS()` DO NOT give the value of the gas concentration using Figures 9 and 10, but just the value of  $R_S$  given by the sensor. This value without the calibration data is not useful but itself. The idea is that is value is transmitted to the SmartCitizen platform to be converted into concentration values. But still the problem is the sensor calibration is not solved.

In my opinion, the problem of the de-calibration of the gas sensors in the SCK is not solved at all and it should be absolutely addressed and prioritized in new versions of the SCK hardware.

### 3.3.4 Lighting level (BH1730)

To measure the quantity of ambient light the SCK uses a BH1730 sensor, which uses a LDR<sup>10</sup> combined with a ADC and the corresponding circuit that allows to communicate with the device with the I2C protocol.

There are some characteristics that make this device suitable for the SCK:

- No need of external ADC or linearization circuits (compare this with the gas sensors in Section 3.3.3. Uses the well-known I2C protocol (with slave address 0101001), which is supported natively by the Arduino library with the `wire` library.
- Its resolution is 16-bits for the range of 0.008 to 65535 lux, which is excellent.
- It is possible to adjust by a I2C command the kind of light that it should measure (visible or infrared). The infrared channel is not used in the current version of the SCK, but it could be considered in future versions.
- Low power consumption, which is really needed by the SCK.
- Able to reject light at 50Hz/60Hz (electric network frequency), which is excellent for its use within the SCK, since it filters the interference is artificial light.

In conclusion, this device is excellent for its use within the SCK.

In the SCK firmware the function that is used to obtain the light level is `uint16_t SCKAmbient::getLight()`. It does the following:

1. Starts a I2C communication to the device using this slave address 0101001 and sends the command to read visible light.
2. Waits for 100ms.
3. Reads two bytes (DATA0 and DATA1) from the I2C bus.
4. It obtains the intensity in lux units using expression

$$\alpha \times \text{DATA0} - \beta \times \text{DATA1},$$

where parameters  $\alpha$  and  $\beta$  depend on the ratio DATA1/DATA0.

---

<sup>10</sup>Light Dependent Resistance.

### 3.3.5 Battery level

Most of the data transmitted to the Smart Citizen platform comes from ambient sensors. However, the battery level it is an important parameter about the SCK state. Therefore, the battery charge level is sent along with the rest of ambient data coming from regular sensors.

The battery level is obtained by averaging an analog pin connected to the anode of the LiPo<sup>11</sup> battery. Note that the battery will return a value of  $3.7v$  when fully charged and values that go from  $0v$  to  $0.7v$  when discharged, but the SCK needs  $3.3v$  of  $V_{cc}$  to operate correctly. Then, a simple low-dropoff voltage regulator is used to set the voltage at the output to the desired value ( $3.3v$ ). If the ATmega microcontroller is powered with  $3.3v$  instead of  $5v$ , its performance is worse (it works at  $8MHz$  instead of  $16MHz$ ), but its current consumption is also lower. For the SCK it is clearly preferable to sacrifice the microcontroller speed for the sake of energy consumption.

In the SCK the battery level is obtained by averaging the voltage of the analog pin labeled as BAT (0xA7) with function `uint16_t SCKBase::getBattery(float Vref)`. It simply executes `average(BAT)`, rescales the obtained digital value between the range  $[0, 1023]$  and returns the rescaled value.

Note that with this procedure what is really read is the voltage at the output of the battery (before the voltage regulator). It is indeed related to the charge in the battery, but the relationship between the voltage at the anode and the actual charge is not linear. Therefore, a problem with this approach is that even if the power consumption remains stable, the users may perceive that the level of charge decreases faster when the charge is low, compared with the rate when the charge is high. This could be a problem, since most users are used to the fact that the battery level of their domestic devices (tablets, smartphones, GPS, etc) is related directly to the level of charge.

A possible solution to this problem is to encode in the SCK firmware the charge/voltage curves, but this improvement is out of the scope of this project.

Note also that in the original firmware the function `float SCKBase::readCharge()` appears, but it is not used elsewhere in the code. It seems to be dead code and therefore it was removed in the improved version of the firmware presented with this project.

---

<sup>11</sup>This is a commonly used nomenclature for the **Lithium-Polymer** batteries.



### 3.3.6 Number of WiFi networks

Similar to the case of the battery level (Section 3.3.5), the number of WiFi networks makes part of the data send by the SCK to the SmartCitizen platform.

This data is obtained from the RN131 “WiFly” module. In the firmware, it is obtained by calling function `base_.scan()`; , which calls `uint32_t SCKBase::scan()` to obtain the number of WiFi networks available at that particular moment. Note that the number of networks simply counts how many WiFi networks have been detected, regardless if the user has or has not the encryption key to connect to them.

The communication with the “WiFly” module is done by serial communication (supported directly by the Arduino libraries) and with very verbose commands. *Very verbose* means that the communication is not with the I2C or SPI protocols, but interchanging ASCII messages. For example, command `scan` is used to obtain the list of networks. The answer is a list of ASCII strings that the firmware splits and counts accordingly, each of them containing the name of each detected network. Function `uint32_t SCKBase::scan()` simply counts how many strings are in the verbose answer.

## 4 The base SCK board

In order to complete the documentation of the SCK, this section describes briefly the base SCK board. However, note that the objective of this first part of the project is to document an improve the part of the firmware related to the sensor board and not working on the base board of the SCK. However, since obviously the sensor board can not work without the base board, it is documented here for completeness.

As explained in the introduction, the SCK is made by two boards:

- The sensors board
- The base board

The sensors board (documented in Section 3) contains all the ambient sensors of the SCK and the bridge to communicate the sensors with the base board. Figure 3 shows how both boards are interconnected with the I/O bridge. Most of the communication is made through the I2C protocol. However, some of the sensors are analog. In that case, the data is first converted by an ADC before it goes from the sensors board to the base board.

The base board contains the following elements:

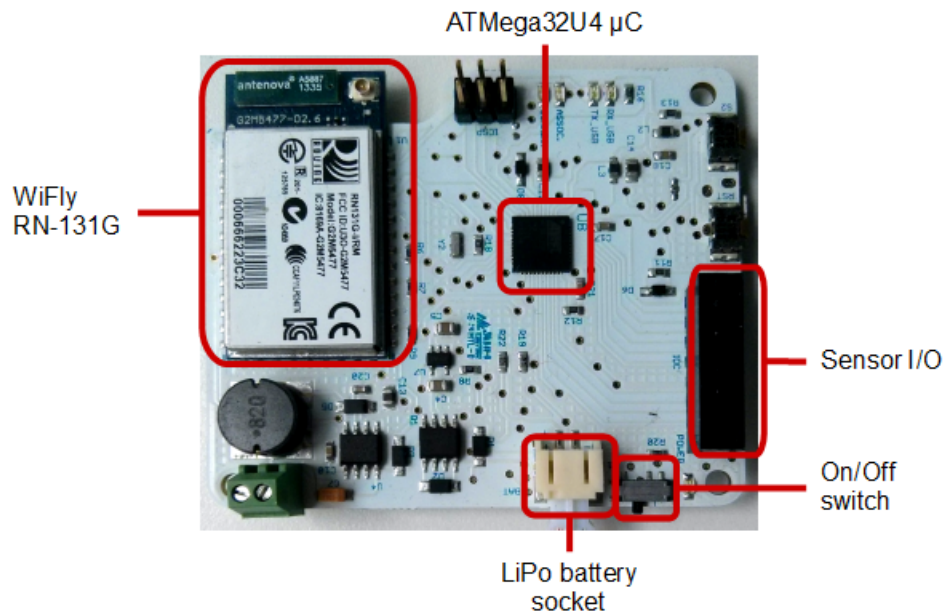


Figure 11: Detail of the base board.

- The ATMega32U4 microcontroller unit.
- The WiFi module. In the current version of the SCK is the RN131 “WiFly”.
- RTC<sup>12</sup> chips DS1339U and DS1307Z.
- Other components, like amplifiers, voltage regulators, power management chips, bus communication, etc.

Figure 11 shows a picture of the base board and identifies its main components. In this section, the ATMega32U4 microcontroller unit and the WiFly RN131 modules are discussed.

The ATMega32U4 unit is a low-power AVR 8-bit microcontroller, with RISC<sup>13</sup> architecture.

Some of its characteristics make it a good candidate for its use in the current SCK and future versions of the hardware of the SCK:

- Low power consumption with at maximum 200mA according to the official datasheet [3].

<sup>12</sup>Real Time Clock.

<sup>13</sup>Reduced Instruction Set Computing.

- It is possible to put the microcontroller in a low-power *sleep* mode and to resume normal operation when a timer interrupt fires (see Section 2.2), in order to save energy.
- It is exactly the same microcontroller unit used by Arduino Leonardo, and therefore most of the existing code for Arduino Leonardo can be reused or adapted in the firmware of the SCK. Also, this encourages users that may have written code for the Arduino Leonardo to adapt it for custom firmware in the SCK or even to try new custom sensors boards.
- It has built-in support for the USB 2.0 bus. This simplifies the SCK hardware, since it does not need to implement any extra circuits to transmit data through the USB bus to program the microcontroller with the firmware.
- It has 12-channel 10-bits ADCs built-in. Again, this simplifies the SCK hardware and avoid adding specialized hardware for analog to digital conversions.
- It is fast (it can run at 16MHz, 16 MIPS<sup>14</sup>).

However, there is an important limitation on the use of this microcontroller, since it only has 32Kb of internal EEPROM available for the firmware. With the Arduino libraries loaded, only 28Kb are actually available. The current version of the SCK is about 27Kb and therefore almost all the internal memory of the microcontroller is full after programming. This is a problem if in the future more sensors need to be added to the sensors board or new algorithms need to be implemented in the firmware.

The base board also contains the Microchip RN131 “WiFly” module [13] for wireless communication. Although the datasheet states that it has a *Ultra-low power design for battery powered applications*, its consumption is about 210mA (close to the microcontroller unit consumption). Nowadays it is no longer possible to consider such a current consumption as *ultra-low power* and there exist several alternatives that perform much better in terms of power consumption. The second part of this project is about writing the firmware of the new low-power WiFi module that will be used in the new versions of the hardware of the SCK.

There are other problems associated to this WiFi module. The range of temperatures within it can work is from 0 to 70 Celsius. The high limit is enough, but the lower limit at 0 Celsius is too high for most applications. If

---

<sup>14</sup>Millions of Instructions Per Second.

the SCK is used outdoors, it is likely that in some countries the temperature goes below 0 Celsius during winter. In this case, the SCK would not be able to transmit any data to the platform.

However, the RN131 module has some characteristics that make it interesting for its use within the SCK:

- Implements a full on board TCP/IP stack and therefore no external drivers are required. This simplifies the hardware of the SCK.
- It can use both verbose (ASCII) commands as well as UART and SPI communication.
- Supports both the softAP and infrastructure network modes.
- It has auto-sleep and auto-wake up modes. This is specially important for the SCK, in order to save energy.
- Supports WEP/WPA/WPA2 authentication schemes.

In conclusion, this WiFi module is versatile, but the range of working temperatures and the power consumption make it a bad candidate for its use within the SCK.

## 5 Engineering tasks in the official firmware

One of the main objectives of this first part of the project is to improve the current firmware of the SCK in order to correct bugs and add new functionalities. Note that the official firmware code from SmartCitizen uses about 27Kb out of the available 28Kb of the microcontroller and therefore there is not much room to add new features. However, it was possible to add a new BIST<sup>15</sup> mode (see Section 5.1) by removing some dead (unused) code and reorganizing in order to avoid duplicated code and creating functions for common operations. Also, the code was analyzed with an automated lint tool to detect and correct potential problems and improve the firmware code.

The development model used is close to Continuous Integration (CI) [14], with frequent commits in the `master` branch and continuous refactoring. However, note that one of the requirements of CI is that the source code is compiled automatically on the development server to check that it compiles correctly and passes all unit test. In this project the source code is not compiled and checked automatically on a server, but the new firmware implements a BIST unit test that checks that all sensors work, that the SCK

---

<sup>15</sup>Built-in Self Test

is able to read them correctly, and that the overall running cycle of the SCK works well. Also, several commits are performed into the master branch of the forked SmartCitizen project and pull requests are proposed to integrate the improvements into the upstream.

The official SmartCitizen project is stored in GitHub, at

<https://github.com/fablabbcn/Smart-Citizen-Kit>

This project was forked in my personal GitHub account, at

<https://github.com/mcolom/Smart-Citizen-Kit>

The workflow for the development is the following:

1. The official SmartCitizen project was forked from GitHub and a derived project created.
2. A local copy of the forked project was stored in my computer.
3. The forked source code is modified (new functionalities, bug correction, documentation, etc).
4. When needed, a **push** of the local **master** branch is performed into the remote **master** branch in GitHub.
5. It may happen that while I contribute to the forked project, the engineers of SmartCitizen and collaborators push their own changes in the official repository. In this case I resynchronize my forked project by merging my code with the upstream (the official project in GitHub).
6. I send **push requests** from my forked GitHub project to the official project, in order to integrate my contributions. The SmartCitizen team studies my changes and they are integrated into the **development** branch if accepted. Later, their **master** branch is merged with the **development** branch to integrate the changes definitively.

Figure 12 shows the commits and merges of all contributors to the SmartCitizen project from different forks. Note that my forked project (user **mcolom**, black line) is several commits ahead of the official project because my contributions are not integrated yet. See also that my forked project was merged twice with the official project (blue line).

Section 5.1 discusses the new BIST mode that was added to the SCK, and Section 5.2 explains in details all the contributions made to the SCK firmware, other improvements, and recommendations for further development.

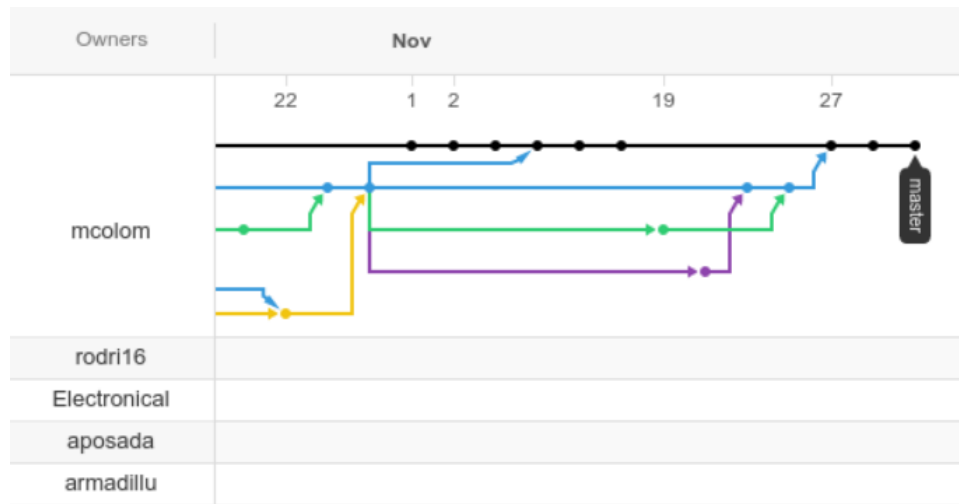


Figure 12: Commits and merges of all contributors in the forked SmartCitizen project. The two arrows that go from the official project (blue line) to the forked project (black line) are the merges with the upstream. Image from the GitHub website.

## 5.1 Built-in Self Test (BIST)

One of the contributions of this project to the official SCK firmware has been the addition of a new test mode that allows to check if the hardware of the SCK works correctly. It can be used also as a unit-test to check if the SCK still works correctly after modification of its firmware.

This kind of test is known as Built-in Self Test (BIST) since the idea is that the same device that implements the BIST is able to check itself when configured in test mode [7]. In the case of the SCK, the test checks all ambient sensors and the overall communication with the SCK.

Since only 1Kb of free EEPROM memory is available after programming the microcontroller with the official firmware, it was not possible in principle to add much more code. The solution that I found was to use a macro of the C++ preprocessor to control if the firmware should include the code of the BIST or not.

If `TESTMODE` is defined, the C++ preprocessor adds the corresponding code to execute the BIST check when the SCK is rebooted. This constant can be activated by passing it as a compiler option, or adding it into file `Constants.h`.

```
void setup() {
    ambient_.begin();
```

```

#ifndef TESTMODE
    ambient_.ini();
#endif

#ifdef TESTMODE
    if F_CPU == 8000000
        SCKTestSuite test_suite;

        while (true) {
            test_suite.run_all_tests();
            Serial.println("\n*** Repeating test suite in 30s...\n");
            delay(30*1000); // ms
        }
    #else
        Serial.println("***TEST MODE NOT AVAILABLE FOR THIS CPU");
    #endif
#endif
}

```

If the SCK is configured in test mode it skips some initializations (it does not call `ambient_.ini()`) that are not needed for the test (for example, activating the WiFi or configuring some timers). Also, it checks for the CPU type (if `F_CPU` is equal to 8000000 it means that it is ATmega32U4). The CPU check is performed because I only had a particular SCK board and it does not make sense to write code for other kind of SCK versions that I can not actually test without having the hardware physically. In case the SCK has a different hardware, it simply shows an error message and stops execution.

Class `SCKTestSuite` encapsulates the list of all possible tests. It has a public function `run_all_tests()` that runs all known tests. This function executes each test and prints the result (passed/failed) using the serial line. A final summary is also printed at the end. The test suite is repeated after 30 seconds. Note that this can be considered a unit-test of the sensors.

From the point of view of the class design, all tests are subclasses of a common `SCKTestBase` base class. The idea of this design is that `SCKTestSuite` contains a list of tests without knowing their particular type, so it can execute a known method `execute()` of each object and obtain the result, regardless the concrete type. In object oriented design, this way of creating objects and referring them by the base class type is known as the *abstract factory* pattern [15].

The test base class `SCKTestBase` is a pure abstract C++ class, which means that it contains abstract virtual methods and therefore it can not be instantiated directly. Each of the actual tests is a subclass of `SCKTestBase` which implements the virtual abstract method `read_sensor()`. This method reads the sensor (which sensor depends on each particular test) and return the obtained value as a float number.

The workflow is the following:

1. Object `test_suite` executes `run_all_tests()`.
2. Method `run_all_tests()` obtains a list of all possible test, each of generic type `SCKTestBase`.
3. Each test is instantiated, but referred with the base type `SCKTestBase` (abstract factory pattern).
4. For each test, function `execute` is invoked. This function returns two values. One with the float value obtained from the sensor and a boolean flag indicating if it was possible to read the sensor.
5. If the obtained value is within the valid range for that sensor, the test is passed. If `execute` returned `false` or the read of the sensor failed, the test is failed. Note that `execute` invokes function `read_sensor` which is pure abstract in the base class. Therefore, its behavior is defined by the concrete subclass.

Figure 13 shows the corresponding UML class diagram of the BIST design. Six different tests are performed for the ambient sensors:

- Light test
- Noise test
- *CO* level test
- *NO<sub>2</sub>* level test
- Temperature test
- Humidity test

I considered that it was important to add a BIST test to the SCK, since it is likely that some SCKs get damaged (specially if used outdoors) and their users had no way to check if the sensors or the hardware were working correctly. Also, this BIST test can be used as a test-unit to check that the SCK is able to read all sensors correctly after any firmware modification (i.e. continuous refactoring).



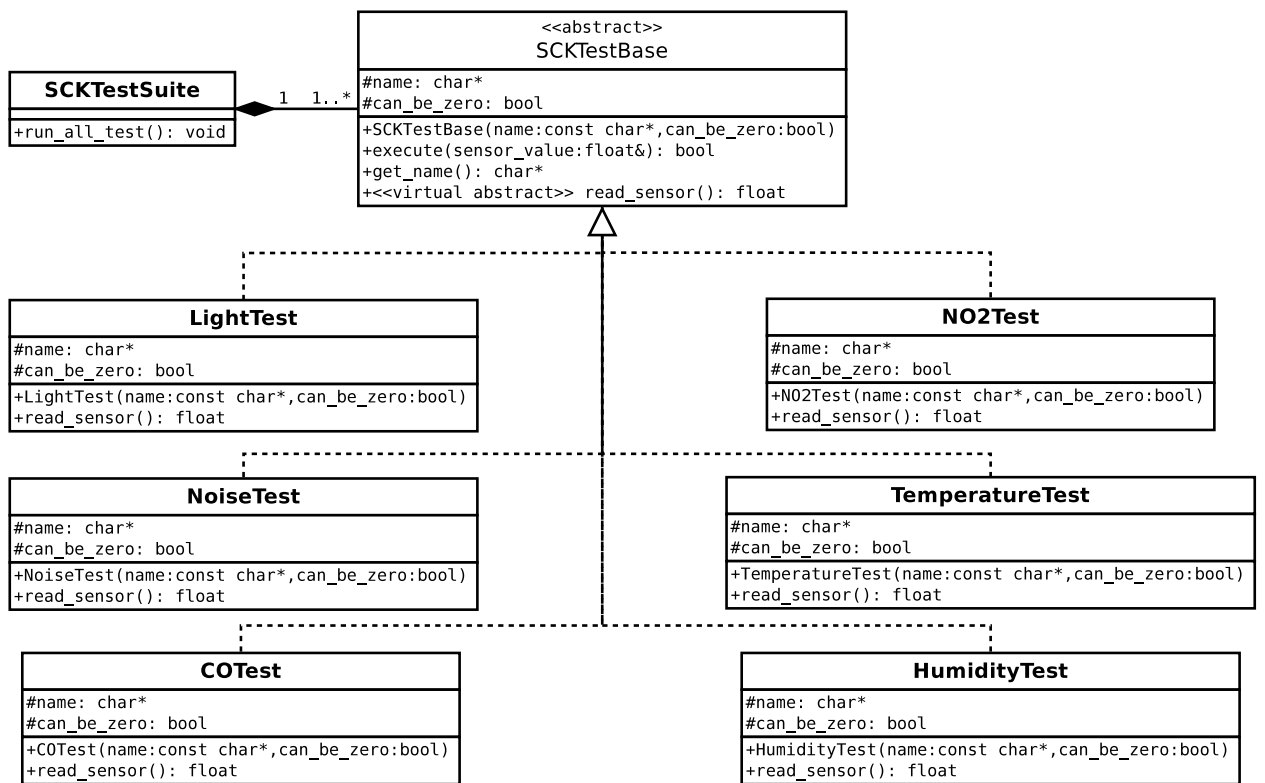


Figure 13: UML class diagram of the classes involved in the new SCK BIST.

## 5.2 Contributions to the SCK firmware

My contributions to the official SCK firmware are documented in this section.

**Commit:** 5c4b238f91c59479ece8242cb0e21f8c4665fad5

**Date:** Sat Aug 9 18:47:31 2014 +0200

- In this commit a first problem in the firmware code was identified: all files should start with the license, as requested by the GPL license<sup>16</sup>, but most of the sources lack this header. It was added.
- The firmware had a `set apikey` verbose command to set the platform API key, but it lacked any command to set it. Actually the system is able to work since the API key is set by a browser plugin that is able to communicate with the SCK (or any Arduino-compatible device) via USB. However, it seems that it is better to provide a command to set the API key and let the firmware deal with the details on how to do it, instead of giving the browser plugin the addresses that it should write into to set the API key. With this new function, it is possible to encapsulate this functionality and making private the implementation details.

**commit** 627db6ae598a4f1d1ea2afff0dd29c6400798275

**Date:** Sat Aug 9 18:57:49 2014 +0200

- Variables `val` and `upSpeed` were moved to private scope in class `AccumulatorFilter`. Before this change, it was possible to modify these variables from outside the class. But they should be read-only and accessible only from read-only getters.

**commit** 0176046eda8f3d2330f23a8304143041a45fbefd

**Date:** Sun Aug 10 00:26:49 2014 +0200

In file `SCKAmbient.cpp` the construction

```
if (report == 1) Serial.println(F("Wifly Updated.));  
else if (report == 2) Serial.println(F("Update Fail.));  
else if (report == 0) Serial.println(F("WiFly up to date.));  
else if (report == -1) Serial.println(F("Error reading the wifi version.));
```

was changed and now a `switch` statement is used instead.

Variable `byte retry = 0;` was moved to a more restricted scope with less visibility.

Construction

---

<sup>16</sup>The SCK is indeed under the GPL license: <https://github.com/fablabbcn/Smart-Citizen-Kit/blob/master/LICENSE.txt>

```

if (temp>RES) data = RES;
    else if (temp<0) data=0;
    else data = temp;

```

was changed to the shorter equivalent comparison:

```

if (data > RES) data = RES;
else
    if (data < 0) data = 0;

```

In function `void SCKAmbient::writeRL(byte device, long resistor)` variable `data` is initialized to zero, but just after that it is rewritten with another value. This dummy initialization was removed and the new code now is simply `int data (int)(resistor/kr1)`. The same problem happened in function `SCKAmbient::getMICS` with variable `DATA`.

The code

```

if (Rs < 2000) writeRL(device, 2000);
else writeRL(device, Rs);

```

was written in a compact form that avoids repeating code:

```

writeRL(device, Rs < 2000 ? 2000 : Rs);

```

In this part of the code the second comparison is useless if the first comparison is successful.

```

if (temp>1000) temp=1000;
if (temp<0) temp=0;

```

Therefore, an `else` statement was added in between.

Changed construction

```

if (nets_temp == 0) pos = 0;
else pos = nets_temp - 1;

```

by this much more compact form: `pos = (nets_temp == 0 ? 0 : nets_temp - 1)`.

commit 7c34b83c318d727e84cb9bcf879ca10a094d8d9e

Date: Sun Aug 10 16:19:36 2014 +0200

It was detected that the code that handles the I2C communication had many duplicated code, because the operations needed to start I2C transactions and to read/write data from the sensors are very similar for each sensor. Also, some procedures also produced duplicated code, as for example the code to wait until the I2C bus is available or a certain pin has toggled its logical value.

In order to prevent code duplicity, the following functions were created in class `SCKAmbient` and called when needed:

- `static void i2c_transaction(int device, int data, int num);` and `static void i2c_transaction(int device, int data)` to start an I2C transactions.
- `static void i2c_transaction_reg_val(int device, int address, int val);` to start an I2C transaction and read a single byte from the register of the specified device. This is a procedure that is executed many times and therefore it is convenient to isolate it in a common function.
- `static boolean wait_wire_available(int timeout_ms);` to wait until the I2C bus is available for reading/writing.
- `static boolean wait_pin_change(int pin, int current_value);` waits until a certain pin has changed its logical value given its current value.

`commit e1227e230388dbdf88037a45ae5cbbea971bbbea`

Date: Mon Aug 11 13:02:24 2014 +0200

In the original code, in `SCKAmbient.cpp` each time it was needed to access the base board, the server communication routines, or the ambient routines, the following objects were created:

```
SCKBase base_;
SCKServer server_;
SCKAmbient ambient_;
```

However, each of this declarations creates a whole new instance of the object, which in reality is not needed. In order to avoid creating new objects each time, in the revised firmware the objects are created just once at the beginning and they are accessed as static objects instead. This saves time (since the objects do not need to be constructed again) and memory (since the objects are created once and they are static).

`commit f25151d4681051de374d27cb37cb54859562cce1`

Date: Mon Aug 11 13:56:24 2014 +0200

This commit is simply a refactoring. The new code about I2C transactions was move to `SCKBase.cpp` from `SCKAmbient.cpp`, since it is actually common code (thus, it should belong to `SCKBase`), and not ambient related.

`commit f126b70d003a088fef2c7cbd999cf87375ca8b2b`

Date: Mon Aug 11 20:39:20 2014 +0200

In order to prevent code duplicity, three functions were created:

- `readByte` to read a single byte from the internal CPU EEPROM or the external EEPROM.

- `writeByte` to write a single byte from the internal CPU EEPROM or the external EEPROM.
- `static void i2c_write_many(int device, byte data[], int num);` to start an I2C transaction and write many bytes to the specified register of a device given its I2C address.

```
commit 66bfebb3daecdaf9f86cb9396edde1e81979acd8
Date: Sat Nov 1 20:11:07 2014 +0100
```

A `.gitignore` file was added to the root of the project tree in the git repository. The reason is that file `Constants.h` contains the WiFi password hardcoded. Therefore, it might happen that accidentally some users committed and pushed this files to the repository, exposing private data. In order to avoid this, a `.gitignore` file was added with instruction to avoid tracking this particular file. Of course, changes to this file can be done using the web interface of GitHub and committing it afterwards.

```
commit c11e8eb5e5faef4946f3ed8464e13d4347a7f90e
Date: Sun Nov 2 15:32:40 2014 +0100
```

In this commit, the BIST test was pushed to the forked repository. Section 5.1 describes in detail this new feature.

```
commit 22aaaff9bc49fdc1875ad2809f28dc9813bb12b7
Date: Sun Nov 2 18:51:24 2014 +0100
```

The official code of the firmware lacked of any special notation to generate automated documentation. It is very important to use tools to document the source code structure, because it allows:

- To generate automatically detailed documentation.
- The input and output of any function are clear.
- It is possible to generate UML diagrams of the current architecture of the code automatically, in any moment.
- The documentation is always updated.
- Some IDEs are able to interpret the automatic documentation notation and to improve accordingly their edition capabilities (i.e.: code autocompletion, grouping functions, give class metainformation, etc.)

In this commit, all functions were added Doxygen<sup>17</sup> comments in order to be able to generate automated documentation and class diagrams afterwards.

For example,

---

<sup>17</sup><http://www.doxygen.org>

```

/**
 * @brief Reads the resistance of a device (ohms)
 * @param device : address of the device
 * @return Resistance of the device (ohms)
 */
float readRs(byte device);

```

Note that the lack of documentation (even automated documentation) may make fail any project. For the SmartCitizen project, it is strongly encouraged to keep using Doxygen (or any other alternative) to document the code.

```
commit 53d11f17842045a4c100f5c0d1bbfc3f989f93c0
```

```
Date: Thu Nov 27 20:48:39 2014 +0100
```

It was detected that function `readCharge()` to obtain the remaining level of battery was not used in the code. It seems that this function was replaced by `getBattery` (see Section 3.3.5) and `readCharge()` remained as dead code.

Removing dead code is specially important for two reason:

- It makes the code unclear about what it is doing. Without knowing that a function is in reality dead code, it is difficult to understand why there are two different functions for the same feature.
- The unused function occupies EEPROM memory. This is an important problem in the SCK, since the official firmware only leaves 1Kb free in the microcontroller internal memory.

```
commit d86862db72edc36581963e77dde04b28042e011d
```

```
Date: Thu Nov 27 20:51:05 2014 +0100
```

Constant `MCP3` (the value of the adjustable resistance used to measure the remaining battery level) was added to file `Constants.h`. Before this change, this constant was declared in `SCKAmbient.cpp`. This did not make sense, since the other `MCP1` and `MCP2` definitions were already in `Constants.h`.

Apart of these changes, there were many other commits that corrected minor bugs in the code, improved some inefficient structures, and corrected some typos (for example, the word *voltage* written in Catalan as *voltatge*, or the word *reads* being written in Spanish as *lecturas*).

Some of the problems could have been detected by some automatic *lint* tool. These tools analyze the source code and print a report that indicates which parts of the code may be improved or have potential problems.

For the SCK code I used the `checkcpp` tools, and its report was the following (only the relevant parts are shown):

```
cppcheck --enable=all .
```

```
Checking SCKAmbient.cpp...
```

```
[SCKAmbient.cpp:555]: (warning) Comparison of a boolean with an integer.
```

```
[SCKAmbient.cpp:852] -> [SCKAmbient.cpp:850]: (style) Found duplicate branches for 'if' and 'else'.
```

```
[SCKAmbient.cpp:920]: (style) The scope of 'pos' can be reduced.
```

```
[SCKAmbient.cpp:736] -> [SCKAmbient.cpp:740]: (performance) Variable 'ok_read' is reassigned a value before the old one has been used.
```

```
[SCKAmbient.cpp:852] -> [SCKAmbient.cpp:845]: (style) Found duplicate branches for 'if' and 'else'.
```

```
[SCKAmbient.cpp:506]: (style) The scope of 'temp_x' can be reduced.
```

```
[SCKAmbient.cpp:507]: (style) The scope of 'temp_y' can be reduced.
```

```
[SCKAmbient.cpp:508]: (style) The scope of 'temp_z' can be reduced.
```

```
[SCKBase.cpp:443]: (style) The scope of 'byteRead' can be reduced.
```

```
[SCKBase.cpp:445]: (style) The scope of 'timeOutTarget' can be reduced.
```

```
[SCKBase.cpp:581]: (style) The scope of 'auth' can be reduced.
```

```
[SCKBase.cpp:582]: (style) The scope of 'ssid' can be reduced.
```

```
[SCKBase.cpp:583]: (style) The scope of 'pass' can be reduced.
```

```
[SCKBase.cpp:584]: (style) The scope of 'antenna' can be reduced.
```

```
[SCKBase.cpp:590]: (warning) String literal compared with variable 'auth'. Did you intend to use strcmp() instead?
```

```
[SCKServer.cpp:263]: (warning) Redundant assignment of 'cycles' to itself.
```

```
[SCKServer.cpp:207]: (warning) Assignment of function parameter has no effect outside the function.
```

All these problems has been solved in the forked project. It is strongly recommended that the SmartCitizen team executes `checkcpp` or any other lint tool to check regularly the code. It avoids introducing bugs that are difficult to detect afterwards.

## 6 Conclusions

The SCK is using low-power consumption sensors, which is needed since it is powered up with batteries. However, the WiFly module can drain about 200mA according to the official datasheet. It is the only component of the SCK that can be considered non low-power, and it will be changed by a device with better energy performance in the next versions of the hardware

of the SCK. In fact, the second part of this project is about writing the firmware of the new WiFi module, from scratch.

As discussed in Section 3.3.3, the gas sensors have a very important problem that need to be addressed as soon as possible in next versions of the SCK: after a few months the calibration of the sensors is no longer valid and they need to be re-calibrated using laboratory instruments. Of course, this kind of equipment is out of reach for most of the users. If the sensors are not calibrated they give totally incorrect results, which is absolutely unacceptable for the reliability of the SCK. If the calibration problem of the gas sensor can not be solved, in my opinion they should not be used in the sensor board.

It was added a BIST mode to the test. The BIST is a classic test that many devices (specially electronic chips) perform to find any hardware defects. The BIST that has been implemented in the SCK is not as sophisticated as the kind of tests performed inside some microchips, but is able to detect malfunctions of the sensors and buses. The BIST also implements unit-testing.

It was possible to improve the official firmware, as detailed in Section 5.2, as bug corrections, memory saving with static objects, moving variables to private or lower scope, creating functions to avoid duplicated code, etc. The `cppcheck` lint tool was used to obtain an automatic report of potential problems, which were solved. The refactoring of the firmware will go on during the second part of the project, since it is a continuous process.



Part II

# **Development of Firmware for the New RTX4100 Low-Power WiFi Module**

## 7 Introduction

The SCK is battery-powered and therefore it needs that all its components (sensors, CPU, and other chips) have a low current consumption. As explained in Sections 3 and 4, all sensors in the production SCK are low-power, with the exception of the RN131 WiFi module.

It is therefore necessary to change the RN131 WiFi module for a different chip that exhibits a better power performance in the new versions of the SCK.

## 8 Objectives

The objectives of this second part of the project are:

- Analyze and document the hardware of the RTX4100 WiFi module (Section 9) and the CoLa application model (Section 10).
- Do a requirement analysis of the needs of the new WiFi layer (Section 11.1) and define a new API. This new API does not need to maintain compatibility with the previous versions of the WiFi API or previous versions of the SCK, since the hardware of the new SCK boards is completely different.
- Define the software interface of the new WiFi API.
- Write the firmware corresponding to the new API, inside the RTX CoLa application framework.
- Define a protocol to communicate the RTX4100 with the main SCK board using SPI (see Section 11.3). Previous versions of the SCK used the RN131 WiFi module, which communicates with the main board of the SCK with verbose ASCII commands. The SmartCitizen team want to change verbose ASCII communication with binary commands send via the SPI protocol.
- Connect the RTX4100 WiFi module with an external board (an Arduino Leonardo or a Raspberry Pi B board) in order that the external board sends SPI commands to the RTX4100 to test the new firmware. This implies writing a test program in the external board. See Section 11.3 for more details.

- Communicate with the SmartCitizen team (in particular, with Guillem Camprodon) and the RTX helpdesk in case additional technical support is needed during firmware development.

## 9 RTX4100 hardware description

In order to develop firmware for the RTX4100, it is convenient to use a development board to be able to transfer the compiled firmware to the RTX4100 and to debug the uploaded applications. Figure 14 shows the development board that is used in this project, known as the Wireless Sensor Application Board docking station. It has an USB connector to communicate the board with the PC, an FTDI 2232D [16] chip that implements an USB to serial converter (UART<sup>18</sup>), and two connectors to plug the RTX<sup>19</sup> WiFi module with its two AAA-type batteries. The image shows the RTX WiFi module with the batteries already plugging into the WSAB.

The WSAB docking station is a  $86 \times 86$ mm development board provides an easy way to code the RTX4100. Once it is connected to the PC using the USB cable (mini-USB connector), it provides two virtual serial ports<sup>20</sup>. RTX provides some command line and GUI based applications which communicate with the WSAB docking station and allow to update the firmware of the RTX4100 easily.

It contains the following elements:

- A mini-USB connector to communicate the WSAB docking station with the PC, providing two virtual serial ports.
- Two status LEDs that blink when each of the serial ports are accessed.
- A connector for the WiFi Sensor Application Board (WSAB), the actual Wi-Fi module that will be installed in the SmartCitizen boards.
- A 40 pin expansion connector, that can be used to for additional debugging purposes. They are connected to the I/O ports of the WSAB microcontroller.
- Boot button: used for firmware updates. However, for the kind of project used in this project (a CoLa application, see Section 10 for details) this button is not needed.

---

<sup>18</sup>Universal Asynchronous Receiver-Transmitter.

<sup>19</sup><http://www.rtx.dk/>

<sup>20</sup>Two COM ports in Windows system, and devices `/dev/ttyUSB0` and `/dev/ttyUSB1` in GNU/Linux systems.

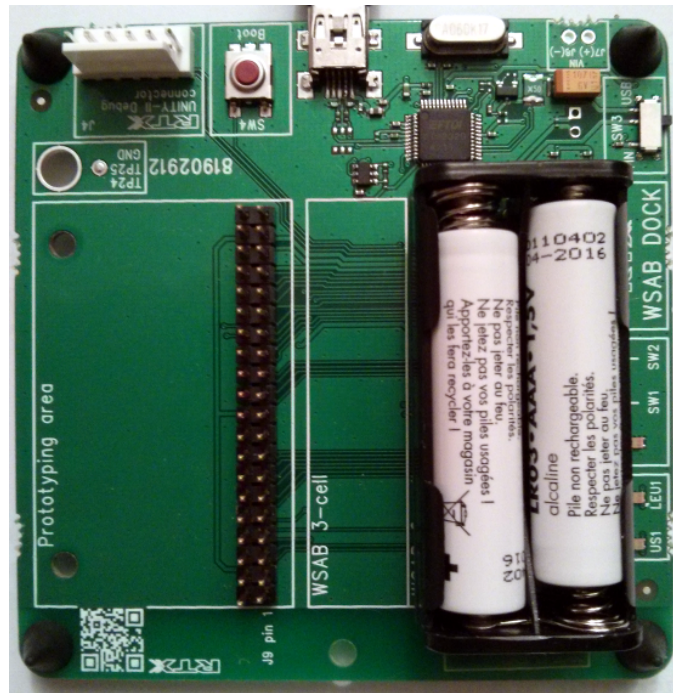


Figure 14: The Wireless Sensor Application Board docking station used to develop and test the firmware. It has an USB connector to communicate the board with the PC, an FTDI chip that implements an USB to serial converter (UART), and two connectors to plug the RTX WiFi module with its batteries. The image shows the RTX WiFi module with the batteries already plugging into the WSAB.

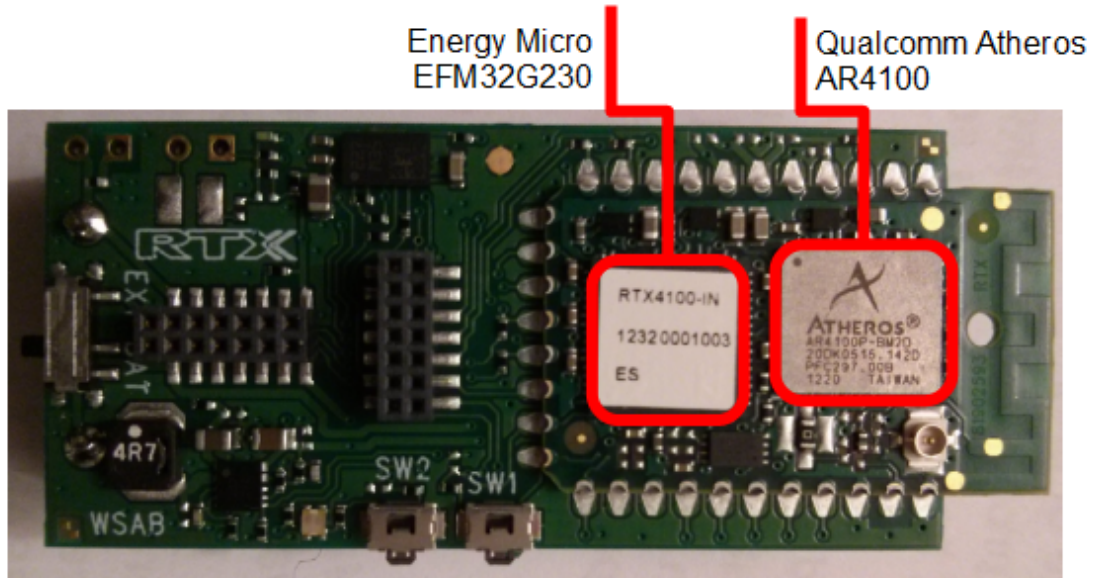


Figure 15: The low-power RTX4100 WiFi Sensor Application Board, also known as *WSAB*.

- A RTX Unity-II debug connector, a debugging board. It is possible to extend the WSAB docking station with a RTX2040 Unity-II debugger, but this piece of hardware is not used in this project.

The RTX4100 unit (also known as the *WSAB*) is the WiFi module that will be used in the SmartCitizen boards. It combines in a single board a Qualcomm Atheros AR4100 WiFi SIP chip (with built-in antenna) [17], an Energy Micro EFM32G230 microcontroller [18], 24Kb of EEPROM memory, and 3Kb of RAM memory available for the applications. Figure 15 shows the RTX4100.

Since the WiFi RTX4100 module has its own microcontroller unit, it can work standalone without needing to be embedded inside another system. In the case of SmartCitizen boards this has following advantages:

- The EFM32 microcontroller in the SmartCitizen board does not need to perform any operations which are naturally related to WiFi control. This isolates well the responsibility of each part of the hardware in the system architecture.
- It improves the concurrency of the system, since the EFM32 microcontroller can do other kind of operations at the same time the main



Figure 16: The RTX4100 module (behind, not visible) powered up with two AAA-type batteries. See Figure 17 to see the circuitry behind.

microcontroller in the base board performs different tasks. For example, the EFM32 microcontroller can take care of maintain a TCP connection while the main microcontroller in the base board builds a JSON request.

- It is more adequate to use the EFM32 microcontroller, since the WiFi operations are less CPU-cycle demanding than the operations needed in the SmartCitizen base board. Using the dedicated low-power EFM32 microcontroller to manage the WiFi hardware optimizes the energy consumption of the complete system.

Figure 16 shows the RTX4100 working with two AAA-type batteries as power source. It is possible to completely detach RTX4100 from the docking station and to make it work standalone. An example is the "Nabto" application, which reads the environmental temperature and sends this data to a centralized server <sup>21</sup>. Figure 17 shows the same, but seen from a different point of view.

The key features of the Atheros AR4100 chip are:

- Little electric consumption (85mA typical)
- Very fast sleep to wake-up. This is important for the SmartCitizen boards, since it makes it possible to activate the WiFi module only when the system needs to send data to the platform, and leave in standby mode the rest of the time.
- The libraries include full support for IPv4 and IPv6 stacks

---

<sup>21</sup><http://nabto.com/>

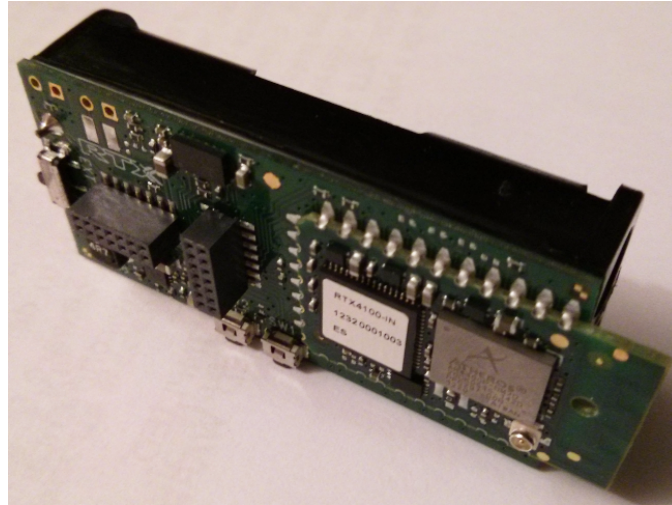


Figure 17: The RTX4100 module showing the circuitry. The AAA-type battery connector is attached. See Figure 16 to see the battery connector with the two AAA-type batteries plugged in.

- It supports WEP, WPA, and WPA2 encryption standards.
- It can be communicated to other modules with UART, SPI, and I2S interfaces
- It is small: 18mm  $\times$  30mm. For the SmartCitizen boards this is an advantage, since it makes it possible to keep the complete SmartCitizen boards small and therefore they can be installed almost everywhere.

On the other hand, the main advantages of the Energy Micro EFM32G230 microcontroller used in the RTX4100 are the following:

- It is fast. Up to 32Mhz ARM Cortex-M3 CPU.
- Up to 128 kB flash and 16kB RAM memory. This is important for the SmartCitizen boards, since it is possible to save the WiFi configuration directly on the RTX4100 module instead of using dedicated memory in the main SmartCitizen board. Also, its 16Kb of RAM enable the microcontroller to execute simple programs without requiring interaction with the main SmartCitizen board.
- It has five different energy modes. Since low-power consumption is an important requirement of the SmartCitizen boards, this is needed. It consumes 180 $\mu$ A/MHz in normal mode and 45 $\mu$ A/MHz in sleep mode.

- It implements 16 external interrupts. This means that it is possible for SmartCitizen boards to get the EFM32 leave the standby mode and put it back to normal mode when needed. For example, it can wake up when a timer in the main SmartCitizen fires, wake up, wake up the Atheros AR4100 WiFi chip, and send the data.
- It implements UART and SPI communication. This is required for the new SmartCitizen boards, the need to leave the verbose commands used in previous versions and use SPI.
- Wide temperature range: from -40 to +85 Celsius. This allows SmartCitizen boards to operate in extreme environmental conditions.

Therefore, the RTX4100 is a great candidate to be used in SmartCitizen boards. The main advantage is the low-power electric consumption, and the possibility of putting the Atheros AR4100 WiFi module in standby mode and being able to resume operation quickly. Also, it has fully support for the IP/TCP protocols and the vastly used WEP, WPA, and WPA2 encryption protocols which are needed by the SmartCitizen boards. On the other hand, the EFM32 microcontroller is also low-power, implements SPI communication, and operates in a wide temperature range.

However, the main advantage of the RTX4100 is the low-power consumption. Its maximum consumption is 85mA, much less than the 210mA used by the RN131, the WiFi module used in the previous version of the SmartCitizen board (see Section 4).

## 10 Software architecture

The applications for the RTX4100 do not generally access the hardware directly, but use an abstraction layer provided by its own multithreaded<sup>22</sup> RTX Operating System (ROS). These applications are known as Co-Located Applications (CoLa) in the terminology used by RTX.

The vendor (RTX) provides a complete development environment (AmelieSDK) that allows to

- Compile CoLa applications.
- Upload and install CoLa applications at the RTX4100.

---

<sup>22</sup>Note that the ARM Cortex-M3 CPU in the RTX4100 has only one core. Therefore concurrency is obtained by OS context switching, since it is impossible to obtain real parallel execution with a single core.



- Debug CoLa applications using the RTX2040 debugging unit. This piece of hardware was not provided to develop this project and therefore is neither documented nor used here. However, its use is recommended for future developments.
- Use a large set of libraries, including TCP/IP support or SPI communication, among others.
- Using multithreading.

The CoLa architecture uses external tools, such the ARM development tools<sup>23</sup> to compile the source code, and the protothreads developed by A. Dunkels et al [19]. The *protothreads* are like usual threads in a concurrent system, but optimized to work in systems with very limited memory, as in the case of ARM microcontrollers.

Of course, it is possible for the developer to remove the CoLa layer from the RTX4100 and to write code that directly accesses the hardware by re-programming the platform firmware. However, this would imply rewriting from scratch much of the code already available in the provided libraries, managing the system memory, and context switching. Since in general these features are needed, clearly it is better to simply build the firmware within the CoLa layer. In this project, the developed firmware is a CoLa application.

The Amelie framework gives an specific tools called *CoLa Controller* (see Figure 18) that allows to choose a compiled CoLa application and update the RTX4100 firmware with it.

Figure 19 shows a diagram of the software architecture of the RTX4100 ROS. The block on the bottom shows the hardware abstraction layer and ROS API. It provides

- Drivers and hardware abstraction. This way, the CoLa applications do not need to know the details of the hardware (I2C addresses, or protocol timing), since the access is done through an API which abstracts the hardware details.
- NVS (Non-Volatile Storage) management, to access the EEPROM memory. With the provided API it suffices to give the memory position to read/write the RTX4100 EEPROM.
- Several APIs for application-level access to WiFi management, DNS, and HTTP operations.

The block on top shows the components available to a CoLa application:

---

<sup>23</sup><http://www.arm.com/products/tools/index.php>

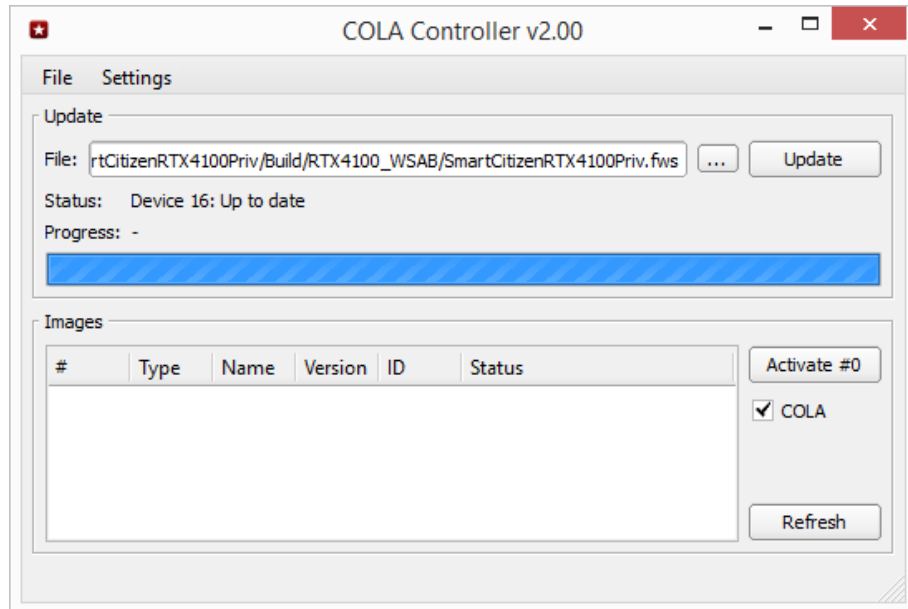


Figure 18: CoLa controller tool provided by the Amelie framework to upload CoLa applications to the RTX4100.

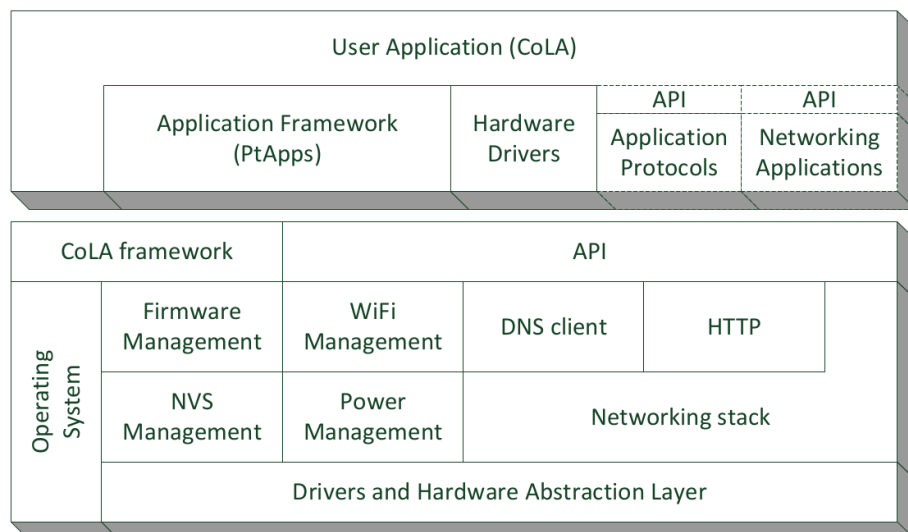


Figure 19: Software architecture in the RTX4100. Source: RTX41xx Wi-Fi Modules, User Guide UG3 Application Development [20].

- Application Framework (PtApps): it consists on several applications that run as protothreads and provide useful common functionality. For example: WiFi power management, status LED control, or thread management, among other.
- Hardware drivers: software components to access the hardware drivers in the hardware abstraction layer (see bottom layer).
- APIs for application protocols (for example, serial communication using the UART, among others) and networking (DHCP, IPv4, IPv6, among others).

The main thread is referred as the *CoLa task*, which is in charge of creating and managing the rest of the threads in the application. Once the CoLa task has created a new thread, there are two ways to communicate the threads among them:

- Using global variables. This is a valid option, but abusing of global variables reduces the readability of the source code of the application and might be the cause of race condition problems which are difficult to detect and correct.
- Using ROS *mails*. The mails are simply messages sent by a thread to an specific recipient. Since they are put in a queue and dispatched by the CoLa task (with function `PtDispatchMail()`), this is the preferred way. For example, a device driver may put a mail message when it has received data from the network or when a timer has fired. The main CoLa task protothread (PtMain) creates all protothreads needed in the firmware application. All threads send their mail messages to a mail scheduler, which puts each mail message into a queue. The main thread dispatches the mail messages in the queue, that is, it takes the message which arrived last and the scheduler sends it to the appropriate protothread recipient (See the diagram in Figure 20). This mechanism to put messages in a queue to be dispatched later and sent to the appropriate recipient is known as the *Publish-Subscribe* design pattern [15].

## 10.1 Persistent data

Most applications need to load and save configuration data in a persistent storage that allows the data to be recovered after a power-down. The WiFi firmware which is developed in this project is not an exception, since it

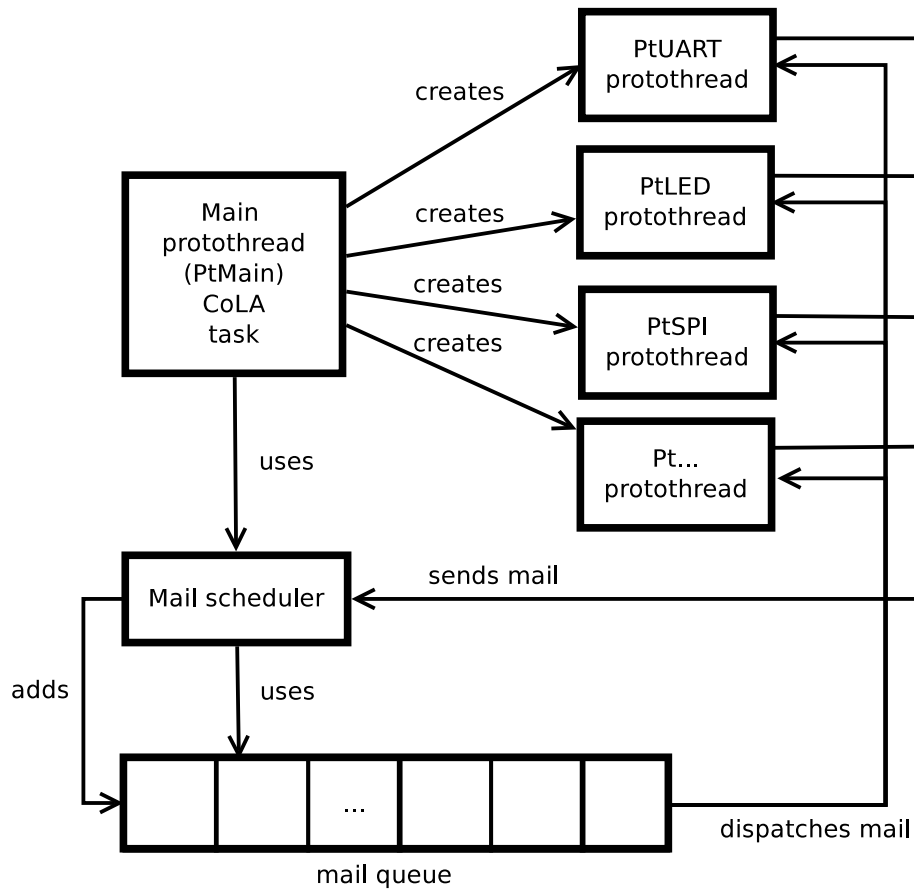


Figure 20: Mail mechanism used by the ROS protothreads. The main CoLa task protothread (PtMain) creates all protothreads needed in the firmware application. All threads send their mail messages to a mail scheduler, which puts each mail message into a queue. The main thread dispatches the mail messages in the queue, that is, it takes the message which arrived last and the scheduler sends it to the appropriate protothread recipient. This mechanism to put messages in a queue to be dispatched later and sent to the appropriate recipient is known as the *Publish-Subscribe* design pattern [15].

will need to store the password of known WiFi networks, the ESSID of the preferred network, the number of channel, or if DHCP or static IP is used, among others.

The RTX4100 ROS offers support to access Non-Volatile Storage (NVS) to load and save persistent data. In practice, the provided API accesses certain positions of the 1Kb EEPROM inside the EFM32 microcontroller. The RTX4100 platform firmware reserves the first 512 bytes for its own use, but the following bytes are available to CoLa applications. Some of the PtApps (see Figure 19) store its own data in the last 256 bytes of the EEPROM. Therefore, it is only safe to assume that 256 bytes of NVS are available for the firmware being developed in this project, just after the 256 bytes reserved for the RTX4100 platform firmware.

The header file `NvsDef.h` defines two constants which declare the address of the beginning of the PtApps data (`NVS_OFFSET(RtxAppData)`) and the initial address of the 256 bytes available for new CoLa application `NVS_OFFSET(Free)`. The API provides two functions, `NvsRead()` to read and `NvsWrite()` to write values to the NVS area.

For example, this code might be used to write a structure `PreferredNetworkSettings` which holds the parameters of the preferred WiFi network:

```
NvsWrite(NVS_OFFSET(Free),
        sizeof(PreferredNetworkSettings),
        (rsuint8*)&PreferredNetworkSettings);
```

The firmware developed in this project will use the “Free” NVS area to avoid conflicts with the PtApps. The persistent application data is stored in the `app_data` object of type `AppDataType`:

```
typedef struct {
    ApInfoType ap_info;
    rsuint8 use_dhcp;
    ApiSocketAddrType static_address, static_subnet, static_gateway;
} AppDataType;
```

Where `ap_info` stores the configuration of the WiFi AP<sup>24</sup> (wireless network SSID, and encryption details), if DHCP or a static IP address is used, and the static IP address (if DHCP is not used), the subnet, and gateway.

---

<sup>24</sup>Access Point.

## 11 The new SmartCitizen WiFi API

### 11.1 Requirements

Since the firmware being developed in this project will be used in SCK boards whose hardware is completely different from previous versions of the SmartCitizen boards, it is not mandatory to maintain API compatibility with them.

However, there are some minimal requirements that new WiFi API needs to meet:

- It should provide at least functions to send and receive data over a TCP/IP stream.
- It should include power-control functions in order to ensure that the electrical consumption is minimal, since low-power is one of the reasons to have changed the WiFi module.
- Its protothreads must be managed such a way that the RTX4100 EFM32 microcontroller stays in standby mode most of the time, in order to save energy.
- It must be easy to test and debug. To achieve it, the API will provide communication through the UART so the developer can execute commands to test the firmware, without having to necessarily programming an external unit which communicates with SPI.
- It should be able to receive SPI commands and return status information using SPI. The UART communication is only for the developer convenience and to make debugging easy. See Section 11.3 for more details on hardware integration.
- It should use a binary communication protocol through SPI, avoiding verbose commands.
- The source code must be properly documented. It will be used also the `Doxygen` tool in order to obtain automated documentation.

### 11.2 Implementation details

This section describes the API which is exposed by the firmware of the new WiFi low-power module used in the new version of the SmartCitizen boards. Only those parts of the code which are significant will be shown here, to avoid having large listing in the document.

The source code of the firmware is developed locally using the Amelie SDK provided by RTX using a development procedure close to Continuous Integration. For this reason, many changes are commits to a local `git` repository and also to a remote repository in `github`. Although only relevant part of the code are show in this section, the complete source code being developed can be browsed in `github` at <https://github.com/mcolom/SmartCitizenRTX4100>. The source code is licensed under the GPL<sup>25</sup> and therefore it is publicly available.

As explained in Section 11.1, in order to ease the process of developing and testing the firmware, it has been implemented a simple terminal to communicate the RTX4100 with the PC. If the terminal is available, the RTX4100 creates a virtual port (a COM port in Windows or a `/dev/ttyUSB` serial device in Linux) which can be opened by any terminal emulator program. For example, Figure 21 shows an a session in the terminal, where the `test` command (unit test) has been executed.

The main CoLa task is the following:

```
/**
 * @brief Main CoLa task event handler
 * @param Mail : protothread mail
 */
void ColaTask(const RosMailType *Mail) {
    // Pre-dispatch mail handling
    switch (Mail->Primitive) {
        case INITTASK:
            // Init GPIO PIN used for timing of POWER measurements
            POWER_TEST_PIN_INIT;

            // Init the Buttons driver
            DrvButtonsInit();

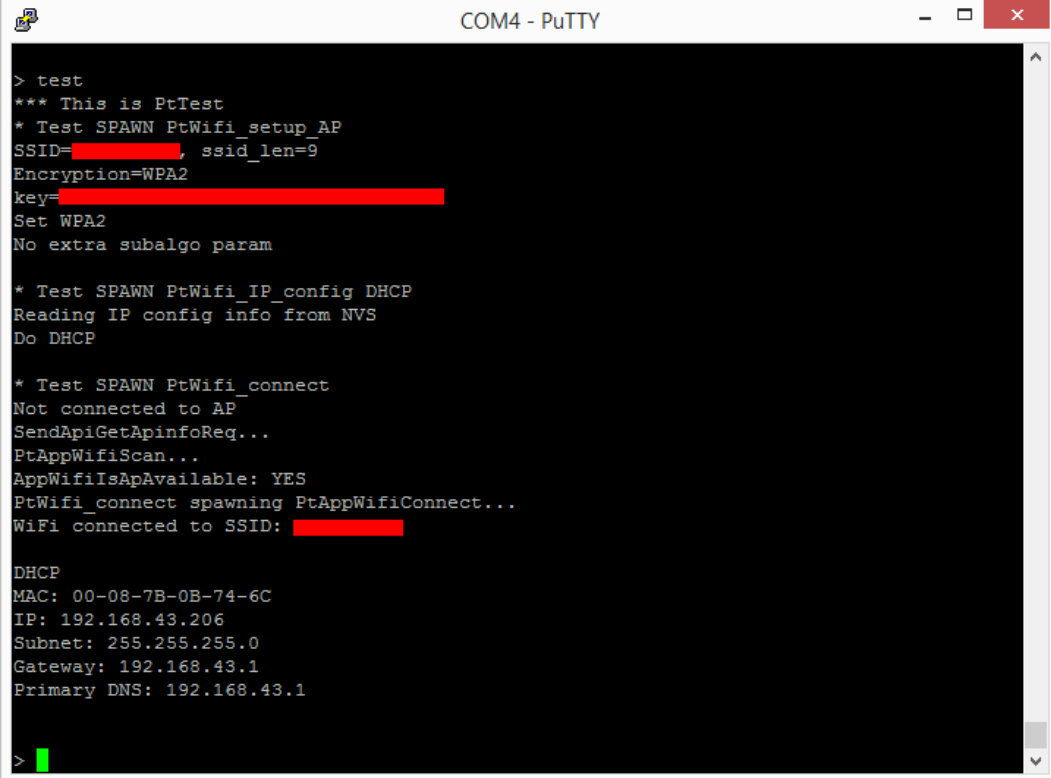
            // Init the Protothreads lib
            PtInit(&PtList);

            // Init the LED application
            AppLedInit(&PtList);

            // Init the WiFi management application
            AppWifiInit(&PtList);
```

---

<sup>25</sup>GNU Public License.



```
> test
*** This is PtTest
* Test SPAWN PtWifi_setup_AP
SSID=[REDACTED], ssid_len=9
Encryption=WPA2
key=[REDACTED]
Set WPA2
No extra subalgo param

* Test SPAWN PtWifi_IP_config DHCP
Reading IP config info from NVS
Do DHCP

* Test SPAWN PtWifi_connect
Not connected to AP
SendApiGetApinfoReq...
PtAppWifiScan...
AppWifiIsApAvailable: YES
PtWifi_connect spawning PtAppWifiConnect...
WiFi connected to SSID: [REDACTED]

DHCP
MAC: 00-08-7B-0B-74-6C
IP: 192.168.43.206
Subnet: 255.255.255.0
Gateway: 192.168.43.1
Primary DNS: 192.168.43.1

> 
```

Figure 21: Terminal session with the Putty program, executing the `test` command (unit test) using COM4 in a Windows system. Private data (WiFi SSID and key password) have been hidden.



```

    // Start the Main protothread
    PtStart(&PtList, PtMain, NULL, NULL);
    break;

case TERMINATETASK:
    RosTaskTerminated(ColaIf->ColaTaskId);
    break;

case API_SOCKET_SEND_CFM:
    #ifdef USE_LUART_TERMINAL
    PRINTLN("API_SOCKET_SEND_CFM (send confirmation)");

    if (((ApiSocketSendCfmType *)Mail)->Status == RSS_SUCCESS)
        PRINTLN("Send OK");
    else
        PRINTLN("Send ERROR");
    #endif
    break;

case APP_EVENT_SOCKET_CLOSED:
    #ifdef USE_LUART_TERMINAL
    PRINTLN("APP_EVENT_SOCKET_CLOSED");
    #endif
    TCP_is_connected = false;
    break;

case API_SOCKET_CLOSE_IND:
    #ifdef USE_LUART_TERMINAL
    PRINTLN("API_SOCKET_CLOSE_IND");
    #endif
    TCP_is_connected = false;
    break;

case API_SOCKET_RECEIVE_IND: {
    #ifdef USE_LUART_TERMINAL
    PRINTLN("API_SOCKET_RECEIVE_IND");
    #endif

    // Save pointer to TCP allocated buffer.
    // The buffer will be freed in Wifi_TCP_receive().
    ApiSocketReceiveIndType *socket = (ApiSocketReceiveIndType *)Mail;

```

```

    TCP_receive_buffer_ptr = socket->BufferPtr;
    TCP_Rx_bufferLength = socket->BufferLength;

    // Move data to rx_buffer
    TCP_Rx_bufferLength = socket->BufferLength;
    if (TCP_Rx_bufferLength >= TX_BUFFER_LENGTH)
        TCP_Rx_bufferLength = TX_BUFFER_LENGTH;
    memcpy(rx_buffer, socket->BufferPtr, TCP_Rx_bufferLength);

    // Activate the flag that indicates that TCP data has been received.
    // The buffer is not freed. The data must be read with Wifi_TCP_receive,
    // which will read the data and clear the buffer.
    TCP_received = true;
    break;
}
}

// Dispatch mail to all protothreads started
PtDispatchMail(&PtList, Mail);
}

```

It does the following:

1. When the **INITTASK** mail is received (it is sent at the application initialization), it initializes the protothreads library, and starts the main protothread. The main protothread listens for commands (from the LUART or the SPI bus) and executes them.
2. When the **TERMINATETASK** mail is received (when a CoLa task is terminated), it notifies ROS in order to finish cleanly. In practice, the threads created to use the SmartCitizen WiFi will be never finished, but processing this mail message makes it possible to detect threads terminating when they should never do. That is, detecting abnormal thread terminations due to design problems or coding errors.
3. When the **APP\_EVENT\_SOCKET\_CLOSED** or **API\_SOCKET\_CLOSE\_IND** mails are received it means that the currently open TCP connection has been closed by the server. It updates the **TCP\_is\_connected** flag to **false**.
4. When the **API\_SOCKET\_RECEIVE\_IND** mails is received it means that new data has arrived to the TCP stream. It copies the arrived

data to the TCP buffer (`rx_buffer`) and sets the flag `TCP_received` to `true`.

The main thread (`PtMain`) manages the communication with the outside. It can be configured in two different ways:

- Debug mode
- Production mode

In the debug mode it accepts verbose ASCII commands using the UART. The RTX4100 is connected using the USB cable with the PC and a serial terminal is emulated<sup>26</sup>. The terminal make it easy to query the status of the system and execute single commands in order to debug the firmware. It can be activated by leaving the `#define USE_UART_TERMINAL` statement in the source code.

The terminal does the following:

1. It declares a buffer to the UART terminal commands (`buffer`) and a pointer to the last received character (`buf_ptr`).
2. It starts the UART application which allows to send and receive characters from a virtual serial port: `PT_SPAWN(Pt, &childPt1, PtDrvLeuartInit(&childPt1, Mail));`
3. It resets the WiFi module: `PT_SPAWN(Pt, &childPt2, PtAppWifiReset(&childPt2, Mail));`
4. It reads, character by character, a complete command from the UART. If the command is recognized, it is processed. If not, it simply prints the *Unknown command* message and waits for a valid command.
5. It processes special characters, as for example backspaces.
6. It echoes back the characters received, in order to simulate a terminal.

However, when the RTX4100 is used in production mode it only communicates through SPI (see Section 3.1) and the terminal is not available. It uses a simple binary protocol with 15 different commands (see Section 12 for details).

---

<sup>26</sup>The FTDI chip is able to emulate a serial port using the USB connection.

### 11.3 Hardware integration

One of the requirements of the new WiFi firmware using the RTX4100 (see Section 11.1) is to avoid using verbose commands (as in previous versions of the SCK) and to directly use a binary protocol to communicate the main board of the SCK with the RTX4100.

During the development, it is easier to simply communicate with the RTX4100 with the UART and the terminal, but in the final version of the product it is needed that the communication is performed using SPI. The RTX4100 docking station has several I/O GPIO<sup>27</sup> pins which can be connected to external boards (see Figure 14). Some of these pins can be used for SPI communication.

Communication with the RTX4100 via SPI brings up a new problem: how to send the SPI commands to the RTX4100. To do it, an external board must be programmed with some test code which communicates with the RTX4100 using SPI. There are plenty of boards which can use SPI, as for example Arduino Leonardo or any Raspberry Pi board.

For this project, the hardware testing and integration will be done using a Raspberry Pi B board, instead of directly with an Arduino board, for several reasons:

- It is possible to create a Python program to communicate using SPI. In the case of the Arduino, changing the program implies reprogramming the microcontroller EEPROM, which takes much more time.
- Since the Raspberry Pi board runs a complete Linux system it is possible to open a terminal using SSH and simply modifying the program which uses RPI using a GNU/Linux console or even a complete graphical environment.
- It is possible to test SPI interactively by using the `ipython` tools and the appropriate libraries. In the case of Arduino boards it is still possible to use the serial terminal, but the interaction is much more limited compared to a full GNU/Linux Bash terminal.

The Raspberry Pi B has the pins needed for SPI located at pins #19 (MOSI), #21 (MISO), and #23 (SCLK). Of course, the GND pins of both the RTX4100 and the Raspberry Pi boards must be connected together to have a unique ground.

The integration procedure consists on connecting the SPI pins of both the RTX4100 and Raspberry Pi and then executing the Python test program in the Raspberry Pi.

---

<sup>27</sup>General Purpose Input/Output.

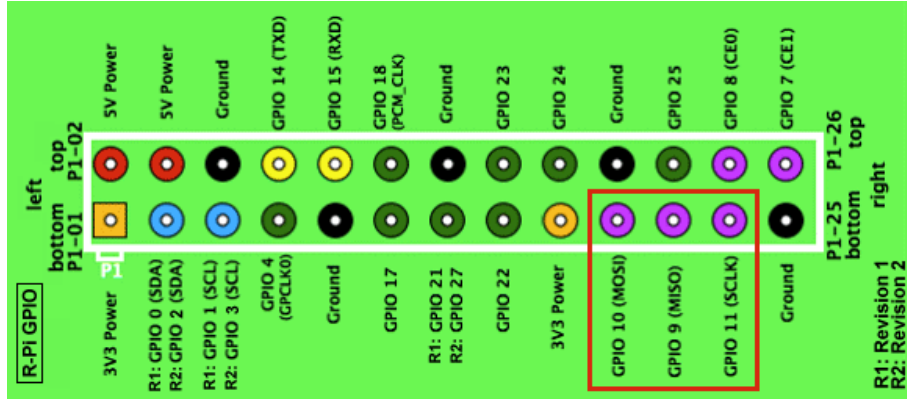


Figure 22: GPIO pins distribution at the Raspberry Pi B. For SPI it uses pins #19 (MOSI), #21 (MISO), and #23 (SCLK) (red square).

The Python test program will initialize SPI communication and then, for each possible operation, it will send the appropriate command and analyze the response. For example, a possible command is to power down the Atheros AR4100 WiFi chip. Therefore, after sending the power off command, the next step is to read the power status of the WiFi chip. If the WiFi chip is in power down mode, then the test has succeeded. Otherwise, it has failed. Note that what this program does is really close to *unit testing*, which is one of the objectives which were established at the beginning of the project.

## 12 Communication with the upper layer

### 12.1 Introduction

The firmware which has been developed in this UOC project is fully autonomous in the sense that it does not need the support or any other hardware to perform its WiFi communication tasks. However, to be useful to an application in an upper layer, it needs to listen for commands and execute them using some known API. Therefore, it follows a classic *master-slave* architecture.

The provided API is application-agnostic, in the sense that is generic enough to be useful to plenty of different applications, and the SmartCitizen boards are just one of them. Also, it offers several alternatives for energy saving, as regulating the power of the transmitted radio signal, the use of several power-save modes, to suspend and resume the whole RTX4100, and even the possibility of powering on and off completely the WiFi chip. The best energy configuration (to suspend or to power off, for example) depends

on the application and it should be the user upper layer which decides.

For example, if the interval between the transmission of the data to the SmartCitizen platform is large enough (say, several minutes) the SmartCitizen firmware should use the command to suspend the WiFi chip. If the interval is extremely low (say, every few seconds) clearly it should not suspend and use a low-power profile using command #12. And if the system is running out of battery, then it should power off the RTX4100 with command #11 and store the data in the system EEPROM instead, without transmitting any data with the WiFi connection.

This section provides an explanation of all the SPI commands available and what is the binary protocol used. For more details, please look at the implementation in <https://github.com/mcolom/SmartCitizenRTX4100>.

## 12.2 Hardware integration

As explained in Section 3.1, the SPI uses four signals to control the communication: clock (SCLK), Master Output Slave Input (MOSI), Master Input Slave Output (MISO), and Chip Select (CS)<sup>28</sup>.

In production mode the communication between the base board (upper layer) and the RTX4100 is performed using only SPI communication with the binary protocol given in Section 12.3. The base board always takes the initiative to start the communication with the RTX4100. Thus, it follows exactly a master/slave architecture, as shown in Figure 23.

To test the slave RTX4100 with the SPI commands, any board able to communicate through SPI is valid, as for example a Raspberry Pi or most of the Arduino boards. However, even if the logical connection is trivial (it suffices to connect both SCLK, MOSI, MISO, and CS pins together in both the master and slave devices), the electric characteristics of each device must be taken into account. In the case of the RTX4100, the EFM32G230 microcontroller needs  $V_{cc}=3.0v$ <sup>29</sup>. The RTX4100 boards is powered with 5v, so it uses a LDO<sup>30</sup> to adapt the 5v to the 3.0v needed by the microcontroller. However, the GPIO pins of the EFM32 microcontroller are connected directly to the SPI interface. It means that if a logical signal has a level over 3.0v it can damage the microcontroller. For example, an Arduino Leonardo boards uses 5v to signal a logical “1”, which would case damage to the RTX4100.

Nevertheless, an Arduino Leonard board can be used to test the RTX4100 communication using a voltage divider in order to adapt the voltage levels,

---

<sup>28</sup>Therefore, it needs 6 wires if we take into account the Vcc and ground pins.

<sup>29</sup>See section 3.8 in its datasheet [http://downloads.energymicro.com/devices/pdf/d0005\\_efm32g230\\_datasheet.pdf](http://downloads.energymicro.com/devices/pdf/d0005_efm32g230_datasheet.pdf)

<sup>30</sup>Low-dropout regulator.

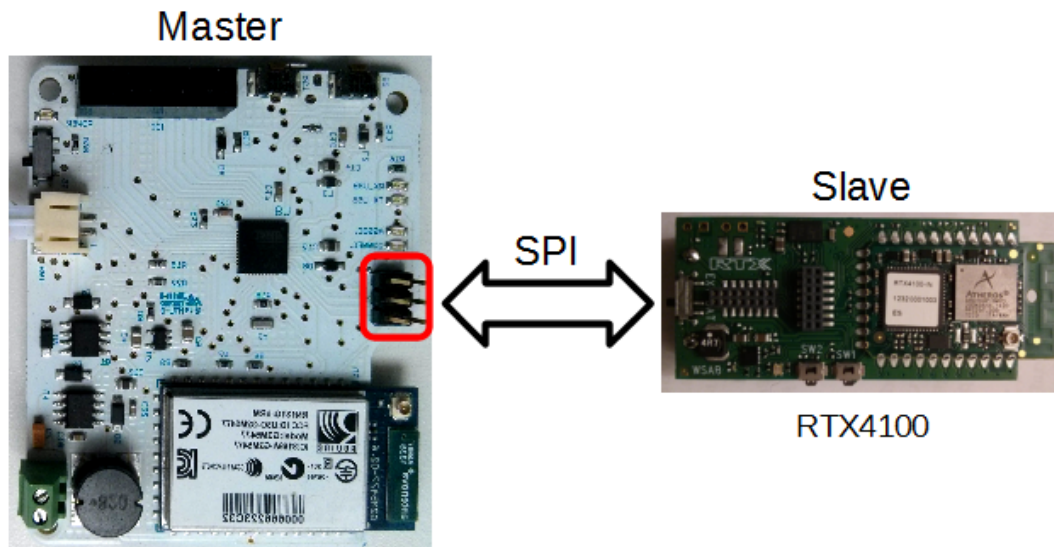


Figure 23: SPI communication between the base board (upper layer) and the RTX4100 in production mode. The test the SPI commands, any board able to communicate through SPI is valid, as for example a Raspberry Pi or most of the Arduino boards. In red, the Arduino-like ICSP header (see Figure 24).

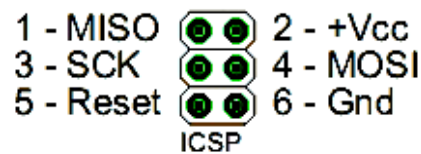


Figure 24: The ICSP header pins in Arduino boards used for SPI communication.

or a dedicated integrated circuit (as the LM3940 chip<sup>31</sup>, for example).

Some Arduino boards (Leonardo, for example) do not have GPIO pins which can be used for SPI communication, but it is possible to use the ICSP header for that purpose. Figure 24 shows the ICSP pinout when used for SPI (except pin 5, which is used to reset the Atmel microcontroller).

The RTX4100 has a driver<sup>32</sup> which implements SPI communication in the following way:

- Slave mode.

<sup>31</sup><http://www.farnell.com/datasheets/78785.pdf>

<sup>32</sup>In Drivers/DrvSpiSlave.c.

- 8 bit data frames. Thus, it can not transmit single bits, and the minimum transfer unit is an octet.
- Uses SPI mode 0 (CLKPOL = 0, CLKPHA = 0).

This driver uses the generic GPIO pins of the RTX4100 (port C) to implement SPI, and sets pin 2 as MOSI, pin 3 as MISO, pin 4 SCLK and pin 5 as CS.

### 12.3 SPI commands reference

In production mode, the firmware waits for commands in the SPI channel, executes them, and return back information using also SPI communication. The details of the SPI communication protocol used are given in Section 12.3.

The SPI interface allows to communicate the RTX4100 with the outside using 15 different commands. These commands are documented in this section. The command must be always initiated by the upper layer by sending a byte which identifies the command which must be executed.

The list of commands and the binary protocol is as follows.

#### **Command #1 (get status)**

This command is used to poll the status of the RTX4100. It returns a byte, where each bit has the following meaning:

1. Bit #0 (LSB): 1 if the WiFi chip is associated and connected to the AP, and 0 otherwise.
2. Bit #1: 1 if the TCP connection has been established, and 0 otherwise.
3. Bit #2: 1 if new data has been received at the TCP stream, and 0 otherwise.
4. Bit #3: 1 if the WiFi chip is suspended, and 0 otherwise.

#### **Command #2 (DNS<sup>33</sup> resolve)**

It allows to resolve a DNS name. The protocol is as follows:

1. Read the size in bytes of the name to resolve.
2. Read the name to resolve.
3. Resolve the name using the DNS system and return it as a 32 bits unsigned word.

---

<sup>33</sup>Domain Name System.



### **Command #3 (IP config)**

It configures how the RTX4100 should obtain the IP address (static IP or by DHCP). If the IP address is static, it configures also the static IP address, subnet, and gateway. After executing this command, the upper layer can associate and connect to the AP with command #5.

The configuration is given as a byte stream. If the byte stream is empty (its size is zero), it assumes that the configuration is stored at the NVS and it is read from there. If the byte stream is not empty, it is parsed and processed.

### **Command #4 (TCP start)**

It receives the server IP address and TCP port and starts a TCP connection. This function is asynchronous exists immediately and the upper layer can look at the global status (SPI command #1) in order to check when the connection has been established.

The protocol is:

1. Read 4 bytes (rsuint32) from the SPI channel. This rsuint32 type represents the IPv4 address of the server.
2. Read 2 bytes (rsuint16) from the SPI channel. This rsuint16 type represents the TCP port of the server.
3. Start the TCP connection asynchronously.

### **Command #5 (associate and connect to the AP)**

It associates and connects to the AP which was already configured with command #3. No SPI data transfer is needed in this command. After executing the command, the upper layer can poll the status of the system in order to know when the WiFi chip could associate with the AP.

### **Command #6 (disassociate and disconnect from the AP)**

It disassociates and disconnects the WiFi chip from the AP. The WiFi chip is assumed to have been already associated to an AP by using first command #5.

### **Command #7 (setup AP)**

It configures the AP which the RTX4100 must associate and connect with. The configuration is given as a stream of bytes which contain all the information needed. If the configuration stream is empty (its size is zero), it means that the configuration data has been already stored at the NVS and therefore it should be used. This frees the upper layer to store the configuration and pass it as an argument to the RTX4100 each time it needs to connect to the AP. The configuration is only specified once, and the rest of the times it is simply read from the NVS.

### **Command #8 (close TCP connection)**

It closes the already established TCP connection.

### **Command #9 (TCP receive)**

When new TCP data arrives, an internal event fires and event handler copies the new data to the TCP receive buffer (`rx_buffer`). The number of bytes in the queue is stored at the global variable `TCP_Rx_bufferLength` by the event handler.

When the upper layer wants to read the arrived data it executes this function. It simply writes `TCP_Rx_bufferLength` bytes from `rx_buffer` to the SPI.

### **Command #10 (TCP send)**

This command is used to send data to the TCP stream. The procedure is as follows:

1. Read a word of two bytes (`rsuint16`) with the number of bytes which it should send to the TCP stream.
2. Read that amount of bytes from the SPI channel. This data is copied to the `tx_buffer` buffer.
3. Write the read data to the TCP stream.

### **Command #11 (Wifi chip power on/off)**

This is used to power on/off the WiFi chip. Note that if the WiFi chip is powered off and the powered on, the WiFi chip must be associated and connected to the AP again, and the IP configuration procedure must be performed again as well. Normally this should be avoided, since it takes several seconds.

It is provided only in the case where the SCK is powered by batteries and the charge is very low. In that case, the SCK can power off the WiFi chip and store the data in the SD card instead of transmitting it.

This command reads a byte from the upper layer using SPI. If the byte is equal to 0, it means poweroff. If it is 1, it means poweron.

Note that for a quick suspend and resume the commands which must be executed are commands #14 and #15, namely.

### **Command #12 (Wifi set powersave profile)**

It configures the WiFi chip powersave profile. This commands read a byte from the upper layer using SPI, and sets the powersave profile accordingly. The code is:

- 0: low power

- 1: medium power
- 2: high power
- 3: maximum power

#### **Command #13 (Wifi set transmit power)**

It reads a byte from the SPI and sets the wireless transmission power accordingly. Its power level goes from 0 (minimum) to 18 (maximum). Of course, if the power level is low, the battery current drain is also low, but it shortens the minimum allowed distance to the AP.

#### **Command #14 (Wifi chip suspend)**

This command puts the Atheros AR4100 WiFi chip in suspend mode and the EFM32 microcontroller in EM2 (low-power mode) in order to save energy.

When the WiFi is suspended it can neither send nor receive information using the radio channel, but the energy consumption is minimal. In the case of SmartCitizen boards the suspend mode is strongly recommended, since the SCK only needs to transmit information at large time intervals (once per minute, or even more). It does not make sense to keep the WiFi activated, since it would be idle most of the time. It must be considered that even if the WiFi is idle, it needs to use the radio to receive WiFi beacon frames for synchronization.

If the WiFi chip is suspended for more than approximately 60 seconds it needs about one second to wake up and be ready for transmission again. The reason is that the synchrony is lost and therefore it needs to look for at least one complete beacon frame. Typically, these beacon frames are sent by the AP once per 102.4ms. The tests performed during these projects show that, in practice, it takes from approximately 0.5 to 1.0 seconds to be ready for transmission after the WiFi chip has been suspended, because it needs to wake up the WiFi chip, to receive one or more beacon frames. Also, the RTOS needs to receive the WiFi wakeup event from the queue.

The firmware presented in this project does not only put the Atheros AR4100 WiFi chip in suspend mode, but also the EFM32 microcontroller in the RTX4100, in order to save even more energy. There are four energy modes: EM1, EM2, EM3, and EM4, where EM1 is the full-power mode and EM4 means that the microcontroller is totally stopped and it can be only woken up with an external reset signal. It was checked that mode EM3 can not be used directly with RTX4100, since the board resets when trying to wake up. Mode EM2 works well with the proposed firmware and it allows to wake up the microcontroller by an external interrupt (when SPI data is received or a timer fires, among other events).

In any case, the decision to suspend the WiFi chip with this command, or to simply leave it in a low energy operation mode (with command #12) depends on the application. For the SmartCitizen boards, the recommended procedure is the following:

1. Suspend the whole SCK (base and RTX4100, with this command) for a given time period (for example, one minute or more), to save energy.
2. Resume the RTX4100 operation with command #15.
3. Read the sensors and send the data to the SmartCitizen platform.
4. Go to step 1.

#### **Command #15 (Wifi chip resume)**

This command is used to put the suspended (with command #14) Atheros AR4100 WiFi in normal operation mode. It might take up to a second to resume, depending on the AP beacon interval, the Atheros chip reactivation, and the RTOS message handling. This function does not need to make the EFM32 microcontroller get out of the suspend mode, since this is done automatically when an external interrupt is detected. When this command is executed, the EFM32 will automatically get out of the suspend mode because of activity in the SPI channel.

## **13 Conclusions**

This section presents the conclusions for the second part of the project. The conclusions of the first part are given in Section 6.

The objective of this second part of the project was to develop firmware for the RTX RTX4100 low-power WiFi module from scratch in order to use a WiFi module with an energetic efficiency better than RN131's, and offer a complete API to control the WiFi operations up to the transport OSI level (TCP connections).

The requirements of this API were the following:

1. It should be able to associate and connect to an AP. The configuration of the AP (SSID, encryption key and others) should be stored at the RTX4100 NVS.
2. It should allow to configure the low-power powersave modes of the Atheros AR4100 chip and the power of the signal.
3. It should provide functions for DNS resolution.

4. It should allow connecting and transmitting data with a TCP socket asynchronously. This way the upper layer can initiate an operation and do some other tasks while the RTX4100 is working. Functions to query the state of the system are provided (for example, to check if the TCP connection has been established or new TCP data has arrived, among others).
5. It allows full control of the energy profile. This is specially important for battery-powered applications, as SmartCitizen SCKs. Functions to suspend and resume the WiFi chip are provided, as well as functions to choose different powersave modes, functions to choose the wireless signal power, and even functions to completely power off the WiFi chip if needed. When an suspend command is executed, the EFM32 microcontroller is also suspended (energy mode EM2), and not only the AR4100 WiFi chip. This ensures that the energy consumption is minimal when the RTX4100 is suspended.
6. The whole firmware should be designed such a way that it saves as much as energy as possible. To achieve it, it should be event-driven, without polling loops which consume not required CPU cycles.
7. It should be easy to debug with verbose commands.
8. It should use only non-verbose SPI commands in production units.
9. It should be free-software.

All these requirements have been accomplished with the firmware developed during this project. The project was finished at the proposed dates and the cross requirements common to both parts of the project have also been met:

- Continuous refactoring;
- communication with the persons responsible for the Smart Citizen project at IAAC<sup>34</sup> (Guillem Camprodon and Alex Posada, mainly);
- periodic and frequent unit testing; Follow an approach close to Continuous Integration.
- writing documentation. In particular, automated documentation using Doxygen.

---

<sup>34</sup>Institute for Advanced Architecture of Catalonia.

About the development of the firmware, in this project the RTX2040 Unity-II debugger unit was not used. It is not absolutely needed, but it allows to watch the CPU registers, execute instructions step by step, or to look up and modify values stored in RAM, among others. Using such a hardware tool would have accelerated up the development of the firmware. There is not any reason to avoid using it, so next developers are encouraged to use it.

The RTX4100 board seems to be a very good candidate, given the results of the evaluation presented in this project. It is really low-power, their libraries implement the TCP/IP stack, and the ROS can provide multithreading. Also, two different tickets were open at the RTX help desk to ask for help during the development, and the response was useful and quick. Clearly, it is a very good candidate to be used in next versions of the SmartCitizen boards.

## Glossary

**ADC:** Analog to Digital Converter.  
**AP:** Access Point.  
**API:** Application Programming Interface.  
**ASCII:** American Standard Code for Information Interchange.  
**BIST:** Build-In Self Test.  
**CI:** Continuous Integration.  
**CO:** Carbon Monoxide.  
**CPU:** Central Processing Unit.  
**CS:** Chip Select (SPI signal).  
**CoLa:** Co-Located Application.  
**DHCP:** Dynamic Host Configuration Protocol.  
**DNS:** Domain Name System.  
**GNU:** GNU's not Unix.  
**GPIO:** General Purpose Input/Output.  
**GPL:** GNU Public License.  
**I/O:** Input/Output.  
**I2C:** Inter-Integrated Circuit.  
**IAAC:** Institute for Advanced Architecture of Catalonia.  
**IC:** Integrated Circuit.  
**ICSP:** In Circuit Serial Programming.  
**IDE:** Integrated Development Environment.  
**IP:** Internet Protocol.  
**LDO:** Low-dropout regulator.  
**LDR:** Light Dependent Resistance.  
**LSB:** Least Significant Bit.  
**LiPo:** Lithium-Polymer.  
**MIPS:** Millions of Instructions Per Second.  
**MISO:** Master Input Slave Output (SPI signal).  
**MOSI:** Master Output Slave Input (SPI signal).  
**NO<sub>2</sub>:** Nitrogen Dioxide.  
**NVS:** Non-Volatile Storage.  
**RISC:** Reduced Instruction Set Computing.  
**ROS:** RTX Operating System.  
**RTC:** Real Time Clock.  
**SCK:** Smart Citizen Kit.  
**SCL:** Serial Clock Line (I2C signal).  
**SCLK:** Clock (SPI signal).  
**SDA:** Serial Data Line (I2C signal).  
**SPI:** Serial Peripheral Interface.

**SSID:** Service Set IDentifier.  
**TCP:** Transmission Control Protocol.  
**UART:** Universal Asynchronous Receiver-Transmitter.  
**UOC:** Universitat Oberta de Catalunya.  
**USB:** Universal Serial Bus.  
**Vcc:** Continuous Current Voltage.  
**WSAB:** WiFi Sensor Application Board.  
**WiFi:** Wireless Fidelity.



## References

- [1] Smart Citizen Team, “The Smart Citizen platform”, <http://www.smartcitizen.me/>, Oct. 2014.
- [2] Arduino Team, “Arduino IDE”, <http://arduino.cc/en/Main/Software/>, Oct. 2014.
- [3] Atmel Corporation, “The ATmega32U4 microcontroller unit”, <http://www.atmel.com/devices/atmega32u4.aspx/>, Oct. 2014.
- [4] H. Chourabi, Taewoo Nam, S. Walker, J.R. Gil-Garcia, S. Mellouli, Karine Nahon, T.A Pardo, and Hans Jochen Scholl, “Understanding smart cities: An integrative framework”, in *System Science (HICSS), 2012 45th Hawaii International Conference on*, Jan 2012, pp. 2289–2297.
- [5] E.J. Pauwels, Albert A. Salah, and R. Tavenard, “Sensor networks for ambient intelligence”, in *Multimedia Signal Processing, 2007. MMSP 2007. IEEE 9th Workshop on*, Oct 2007, pp. 13–16.
- [6] F. Leens, “An introduction to I2C and SPI protocols”, *Instrumentation Measurement Magazine, IEEE*, vol. 12, no. 1, pp. 8–13, February 2009.
- [7] E.J. McCluskey, “Built-in self-test techniques”, *Design Test of Computers, IEEE*, vol. 2, no. 2, pp. 21–28, April 1985.
- [8] M.J. Harrold and M.L. Souffa, “An incremental approach to unit testing during maintenance”, in *Software Maintenance, 1988., Proceedings of the Conference on*, Oct 1988, pp. 362–367.
- [9] NXP, “Official I2C specification”, [http://www.nxp.com/documents/user\\_manual/UM10204.pdf](http://www.nxp.com/documents/user_manual/UM10204.pdf), Nov. 2014.
- [10] Freescale Semiconductor, “Official SPI block guide v03.06”, <http://www.ee.nmt.edu/~teare/ee3081/datasheets/S12SPIV3.pdf>, Nov. 2014.
- [11] Michael Kircher and Prashant Jain, *Pattern-oriented software architecture volume 3: patterns for resource management*, Wiley, 2004.
- [12] Pro-Signal, “ABM-705-RC microphone datasheet”, <http://www.farnell.com/datasheets/1671459.pdf>, Nov. 2014.
- [13] Microchip, “WiFly RN131 data sheet”, <http://ww1.microchip.com/downloads/en/DeviceDoc/rn-131-ds-v3.2r.pdf>, Dec. 2014.

- [14] Martin Fowler and Matthew Foemmel, “Continuous Integration”, <http://www.thoughtworks.com/ContinuousIntegration.pdf>, Dec. 2014 2006.
- [15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: elements of reusable object-oriented software*, Pearson Education, 1994.
- [16] FTDI Chip, “FT2232D dual USB to serial UART/FIFO IC datasheet”, [http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS\\_FT2232D.pdf](http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT2232D.pdf), Dec. 2014.
- [17] Qualcomm, “AR4100 single-stream 802.11n SIP for the Internet of Everything”, <http://www.qca.qualcomm.com/wp-content/uploads/2013/11/AR4100.pdf>, Dec. 2014.
- [18] Silicon Labs, “EFM32G200 datasheet”, <http://www.silabs.com/Support%20Documents/TechnicalDocs/EFM32G200.pdf>, Dec. 2014.
- [19] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali, “Protothreads: simplifying event-driven programming of memory-constrained embedded systems”, in *Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM, 2006, pp. 29–42.
- [20] RTX, “RTX41xx Wi-Fi modules, user guide UG3 application development”, Dec. 2014.