MICROCONTROLLED SMART RELAY

Embedded system for environments refrigerated by air-conditioner unit(s)

Preliminary project by ***Brenda Fernandes Ribeiro***

v0.1 Prototype *version updated on April 22, 2021*
v0.2 Prototype *version updated on May 02, 2021*

Overview

This documentation describes the development process of an embedded system for application in environments refrigerated by air-conditioner unit(s). Its main purpose is energy saving, since, in these environments, open doors and/or windows can be a potential source of energy waste.

The system is composed by a microcontrolled relay for performing, by the click of a button, the energization of an AC unit, conditioned by the response of magnet sensors installed on doors and windows.

Requirements

Functional

- Main input: push button and sensors response.
- Other input: solicitation for usage information download to PC and/or smartphone. Usage information with event and timestamp must be stored in a linked list data structure. The list must be emptied to a desktop file and/or a smartphone app if demanded.
- Main output: relay activation (air-conditioner energization).
- Other output: Usage information file (to desktop and/or smartphone).

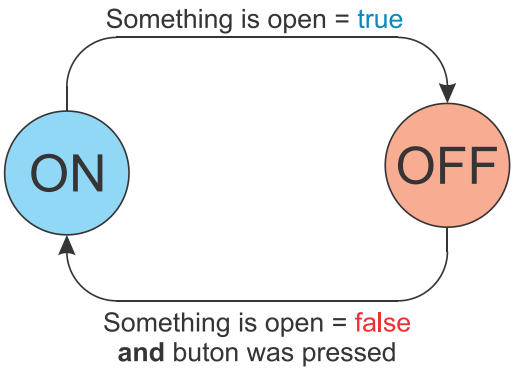
Non-functional

- The hardware and physical structure, after installed, must support room temperature range (22±10°C), and normal indoor weathering conditions (IP20, at least).
- Manufacturing costs must not exceed R$100/unit.
- Response time between main input and main output must not exceed 1s.
- Power autonomy is not necessary. The input must be connected to the 220Vac power grid. A battery can be optionally added to allow the system to support short-term power outages without losing track of time.
- The device must be portable (maximum dimensions being: 15 x 10 x 5cm).

Specifications

Firmware

The main function must be implemented based on the finite state machine (FSM) presented below. The code for the microcontroller must be object oriented (C++). Its design ought to provide easy comprehension and allow adaptation for many architectures, encapsulation is key.



Hardware

The hardware for the inputs should be:

- 1-position dip switch (for time enabling serial communication).
- 2 Push buttons (for starting the system and start the UART sending procedure).
- Magnet sensors.

As for the outputs:

- 3 LEDs (to indicate: if something is open, if the air-conditioner is energized, if the sending mode is enabled)
- Relay (to connect the air-conditioner to its power source).

Software

Desktop

The desktop software for downloading usage information into a file must be developed in C++ language. Its input should be the data sent from the microcontroller via USB serial communication. As for the output, a file containing the received data should be created. Other data processing functionalities may be added in future versions.

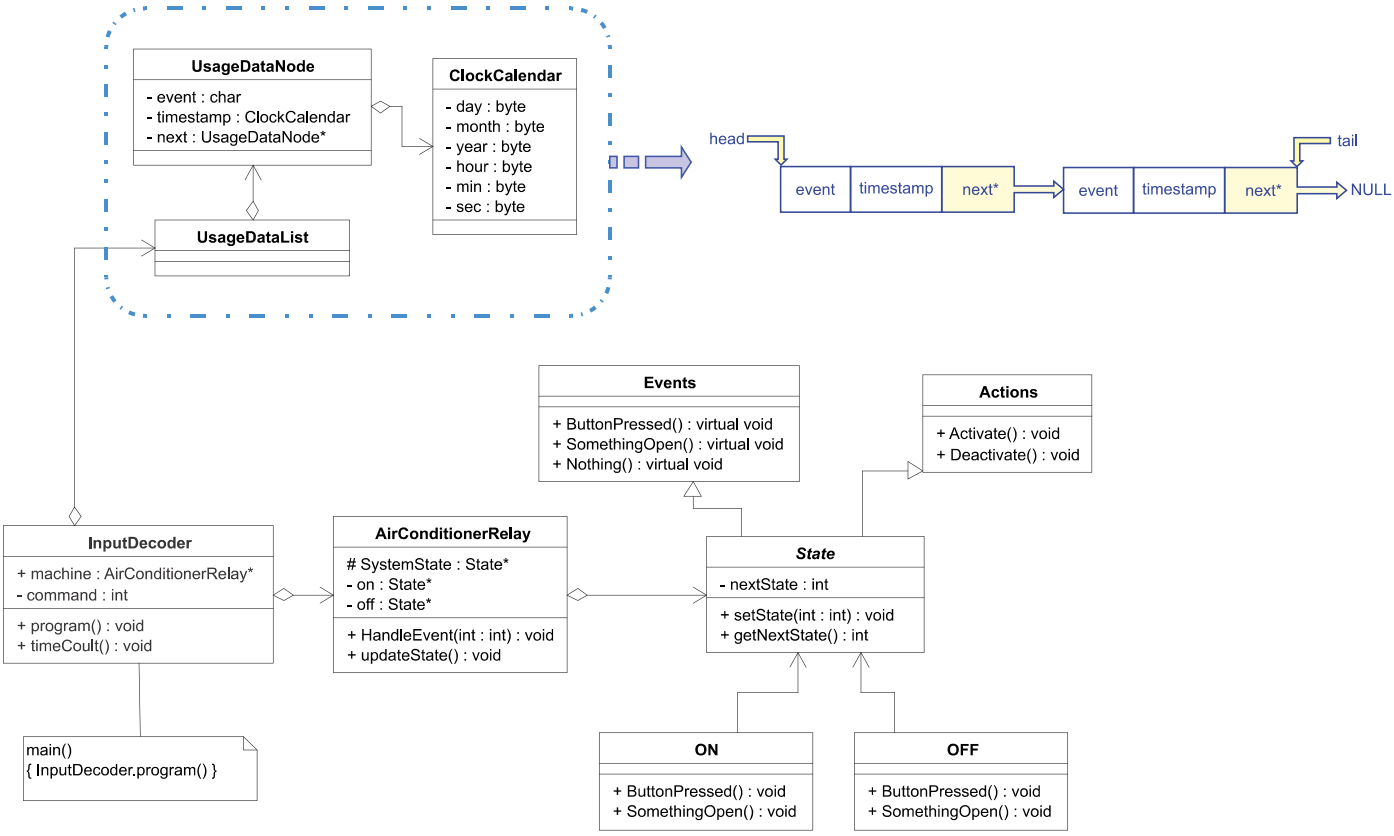
Smartphone

The smartphone app must also be developed in C++ language. It may allow the user to access, through wireless communication, the usage data information stored in the microcontrolled board.

Architecture

Firmware

A sketch of the class diagram is proposed below. The architecture specific commands should be included in the "Action" class (outputs) and in the "InputDecoder" class (inputs), in which the entries must be read and decoded. Not all attributes and methods are represented – these will be detailed in version update notes.



Hardware

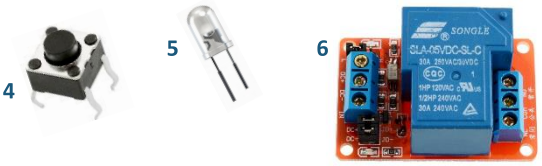
Microcontroller

The NodeMCU ESP8266 ESP-12E architecture should attend the descripted requirements, for it allows C++ programming, is open source, has low-cost (around R$10/unit, if imported directly from China), has portable dimensions (4.50 x 2.55 cm) and supports normal weathering conditions.

Peripherals

1. 1-position dip switch (generic).
2. Push button (plastic, robust).
3. Magnetic sensor (reed-switch).

4. Push button (generic, small).
5. LEDs (small, generic).
6. Relay module (30 A / 240 Vac).



Power

1st Option: No battery

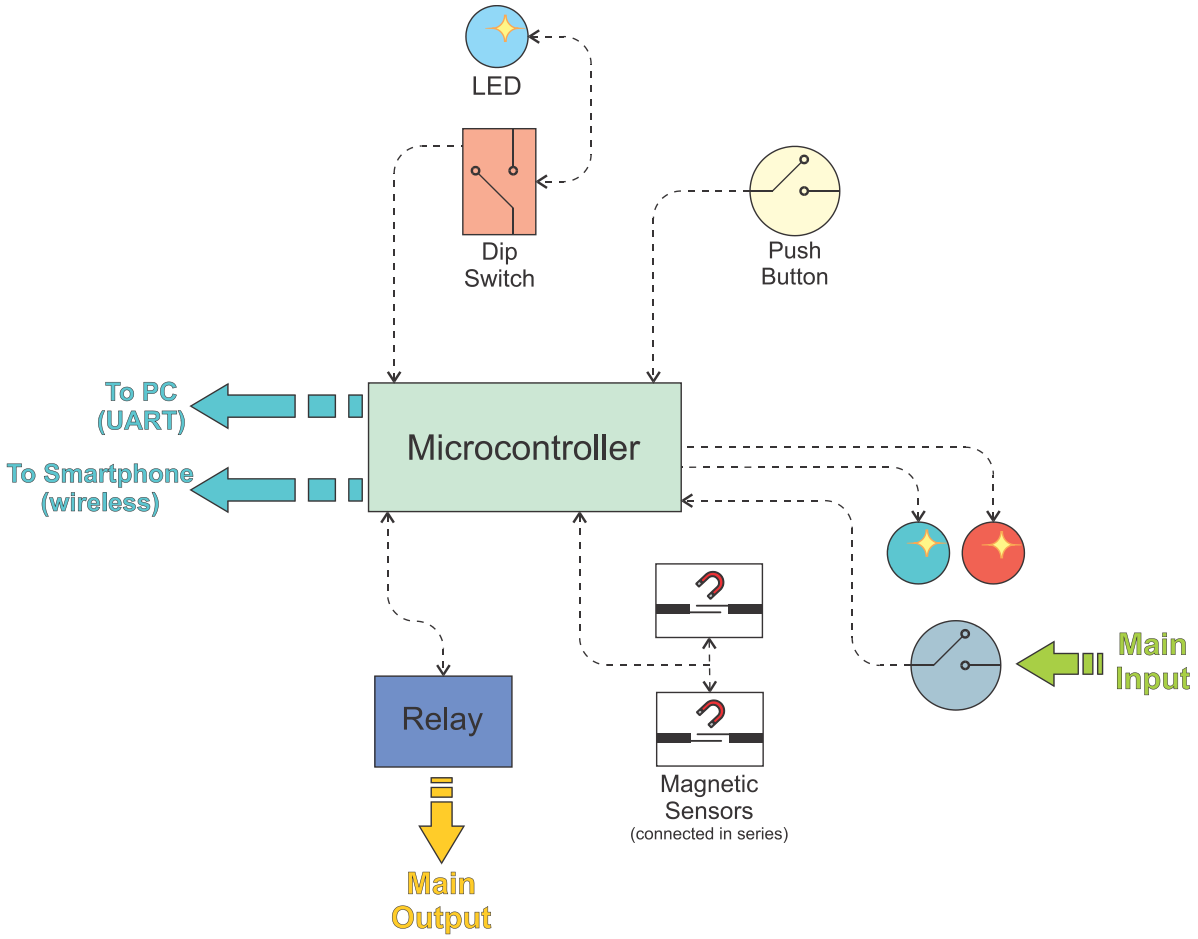
In this case, a 12Vcc/1A power supply is connected to the power grid and to the system through a Jack P4 connector. One step-down module, adjusted to 5Vcc, is needed (suggestion: LM2596-based module, or similar). The 5Vcc output will attend to the microcontroller and the peripherals.

2nd Option: With battery

For this option, the power circuit must also have a charge controller module to interchange between batteries and the power supply connected to the grid. Suggestion: LM317T-based circuit (ATTENTION: this component may require heat dissipator and cooling system).

Block diagram

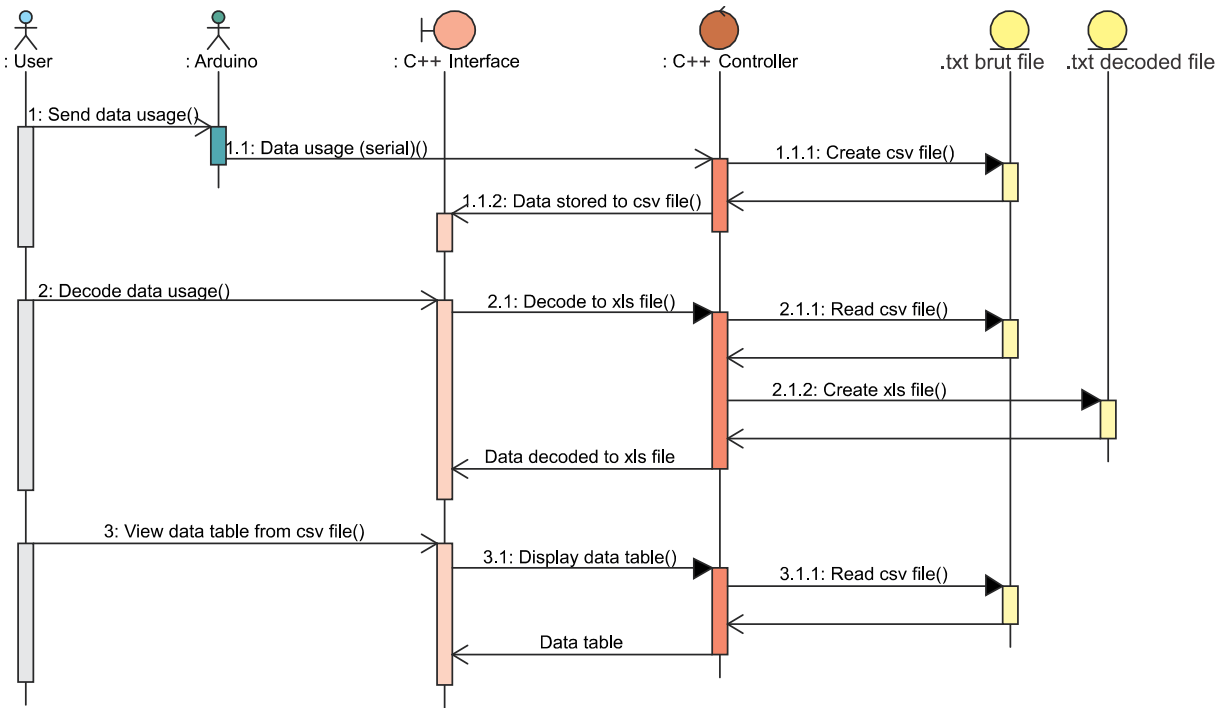
The block diagram bellow provides a preliminary view of the hardware schematics.



Software

Desktop

The desktop software must receive data from the microcontrolled board through serial communication via USB. The code and interface should interact with the user and board as represented in the preliminary sequence diagram displayed ahead. The program must also interact with a csv file generated by itself.



Smartphone

The smartphone software must receive data from multiple units through MQTT protocol. The software must have its controller developed in C++ language. Other languages can also be used, considering that the native Android language is not C++. For more details, see version updates.

Costs

Prototype v0.1

Description	***Price (R$)***
Microcontroller (NodeMCU ESP8266 ESP12-e)	44,90
4-position dip switch	2,40
Push-buttons (small) x2	0,98
LEDs x3	2,40
Resistors (diverse)	2,90
TOTAL	***53,58***

Product

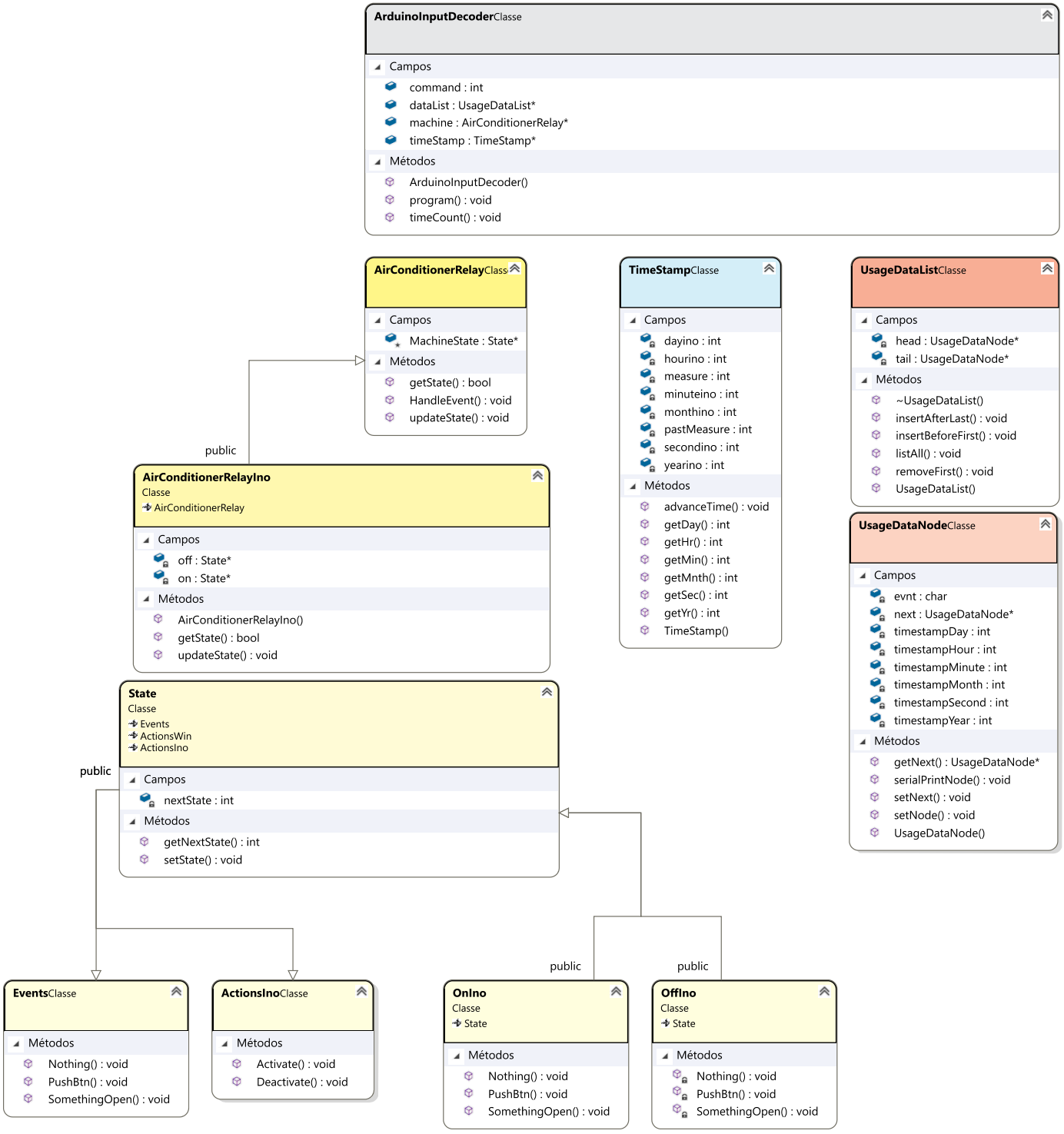
The estimated cost value for the product is detailed below (considering medium scale production).

Description	***Price (R$)***
Microcontroller (NodeMCU ESP8266 ESP12-e)	10,00
4-position dip switch	0,55
Push-button	2,00
Relay 30A	6,50
Resistors	0,40
Power supply	10,00
Plastic enclosure	5,00
Other costs (manufacture, fees, etc.)	5,50
TOTAL	***39,95***

v0.1: Prototype

Firmware

The code of the first prototype version (v0.1) is organized as shown in the class diagram below.



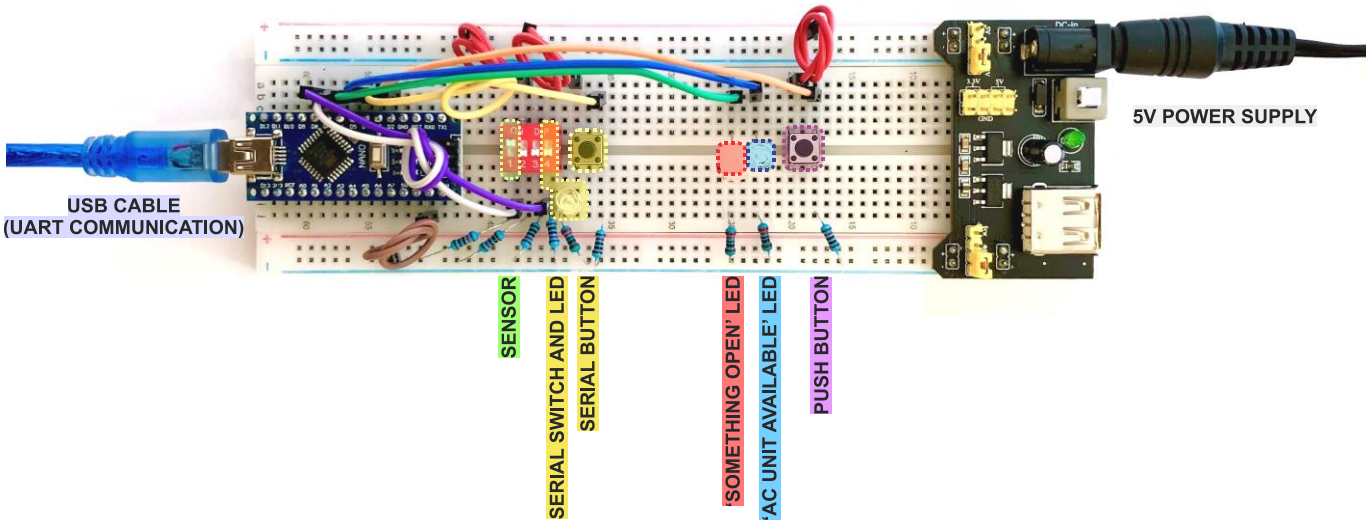
The main() program consists of system initialization routines and a loop containing ArduinoInputDecoder::program() method.

Hardware

The prototype was built using the following components:

- 5Vcc power supply
- Push button tactile micro switch (02 un)
- High bright led (02 un as outputs; 01 un in the circuit)
- 4-way dip switch
- Arduino nano microcontroller
- 10kΩ resistor (06 un)
- 220Ω resistor (03 un)
- Protoboard
- Jumpers (male-male)
- USB to USB mini cable (for UART communication with PC)

The circuit is shown below.



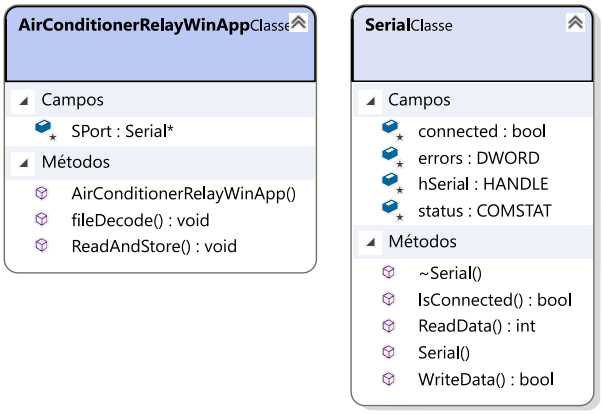
An USB cable is used to stablish serial communication (via UART protocol) to a computer. Also, a dip switch is used to allow data reading (serial switch). The data is sent by the hardware once this switch is ON and the serial button is pressed. This switch is connected to a LED, to visually inform the user about its state. This visual alert is particularly important, because, once the data is sent, it is deleted from the Arduino board.

In this prototype: a switch represents the set of magnetic sensors in the circuit; two LEDs indicate whether there is something open or the AC unit is available; a simple push button represents the robust button accessible to the final user.

Software

Desktop

The software was developed for running in Windows 10 environment. Its class diagram is shown below.



The “main.cpp” consists of a global “AirConditionerRelayWinApp” object and a void main() function containing an infinite loop for the console app menu. The menu has three options chosen accordingly to user’s input:

[1] Read and store: Reads data sent from the microcontrolled board via UART and stores to a *.txt* file.

[2] Decode file: Decodes data from a *.txt* file, generating a new file or appending the information to an existing one. This process transforms the brut data received from the board (*csv* alike) to a better comprehensible format.

[3] Exit: Leaves the console program.

Smartphone

Smartphone software is not contemplated in this version.

Tests

Methodology

No plans were developed for testing this version. A simple manual test was performed for evaluating the main functionalities.

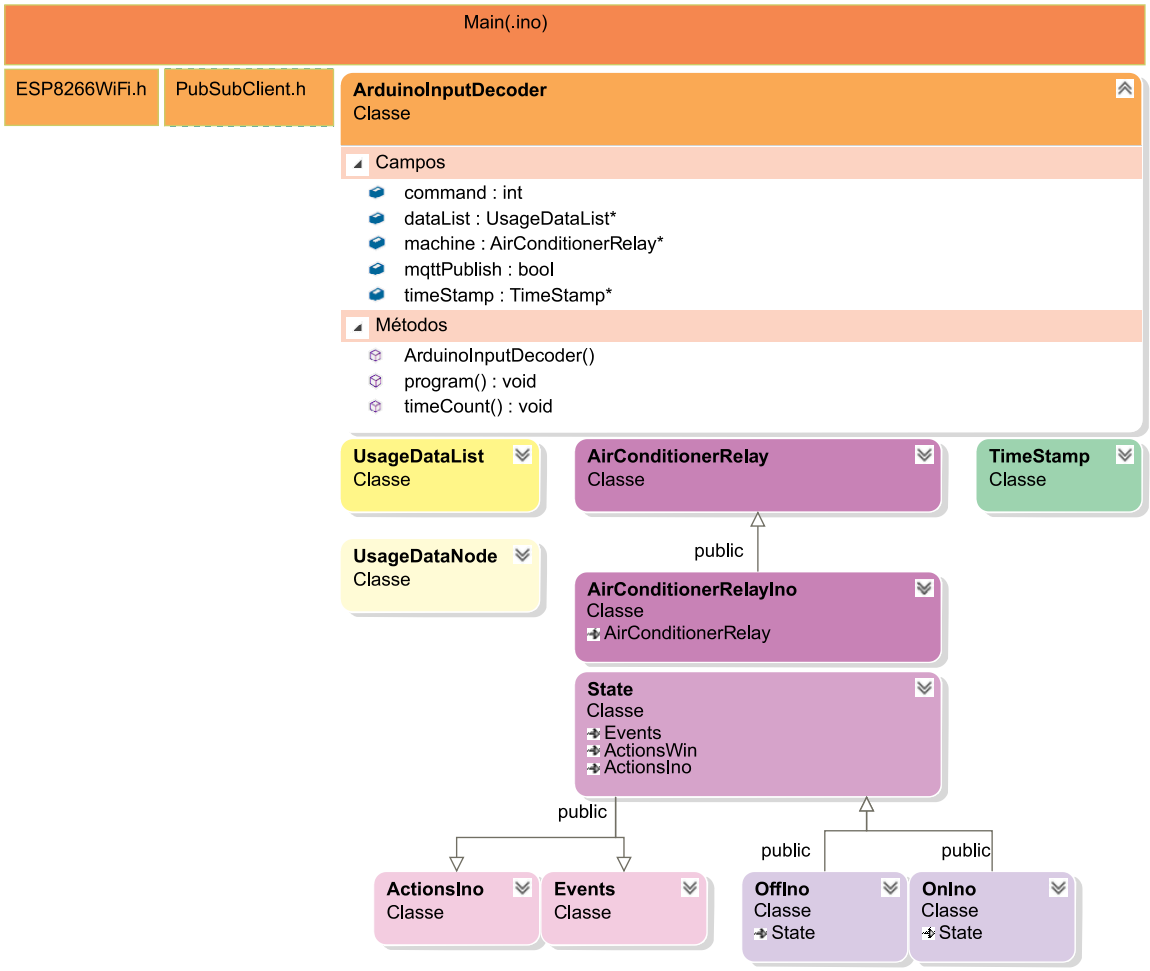
Results

The results were successful, thus, the overall structure passed for future versions (subject to improvements).

v0.2: Prototype

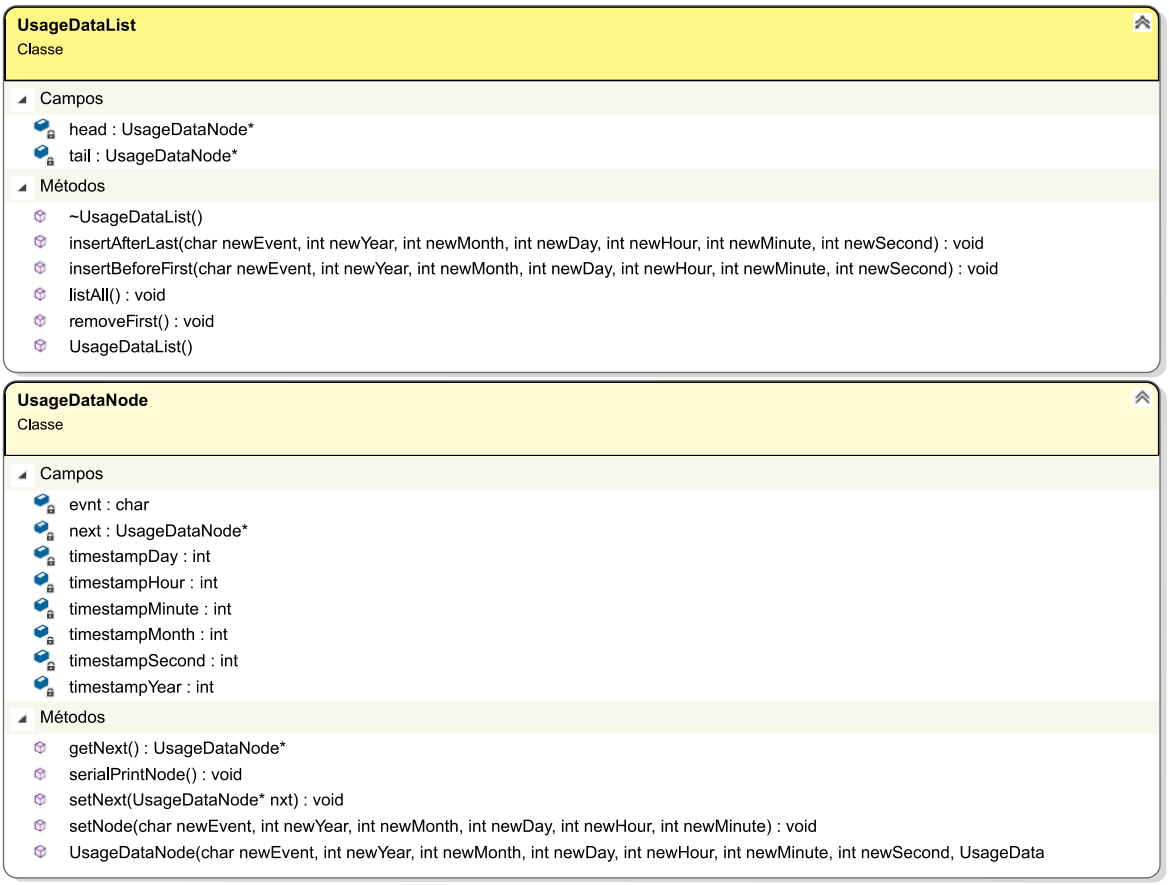
Firmware

The firmware of the second prototype version (v0.2) was developed in Arduino IDE with ESP8266's core (needed for deployment). The generated binary occupies 277900 bytes (26%) of the program memory and 27532 bytes (33%) of the dynamic memory. The code is organized as shown in the class diagram below.



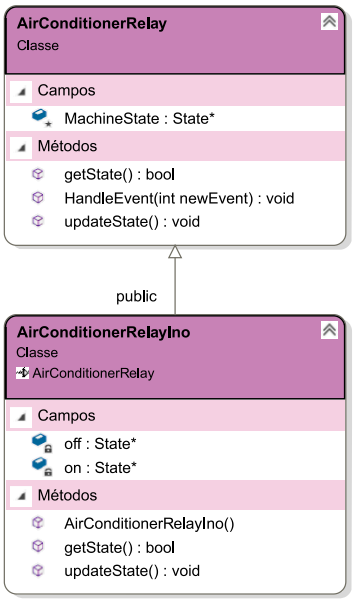
The main() program consists of system initialization routines (including Wi-Fi connection) and a loop containing: ArduinoInputDecoder::program() method and mqttPublish() function . ESP8266WiFi class and PubSubClient are used, respectively, for connecting to a Wi-Fi network and communicating to a MQTT broker.

The UsageDataList class is a singly linked list used as a queue by the input decoder. Its class diagram is detailed in the following picture.

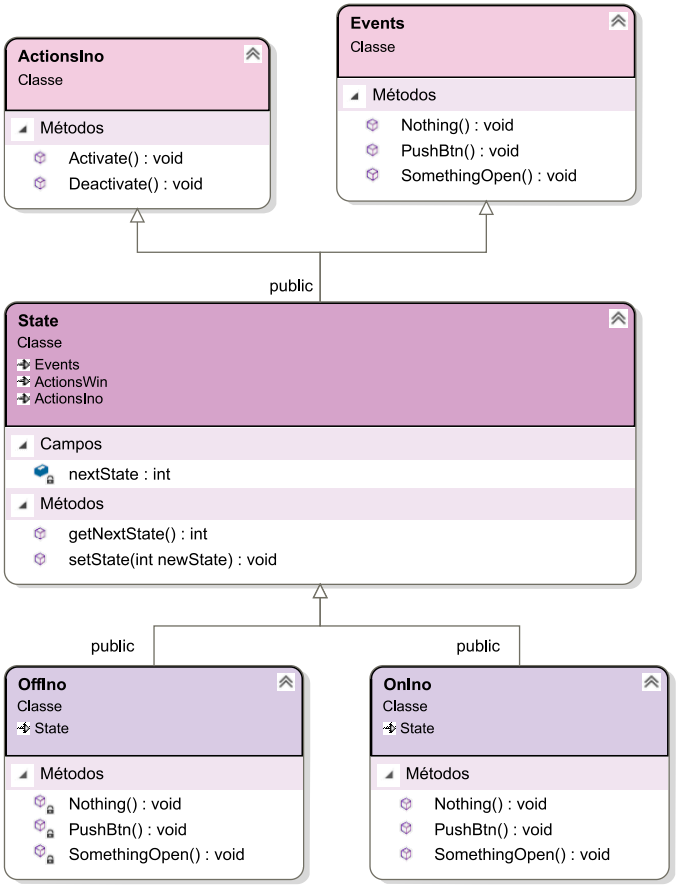


An UsageDataList contains none or multiple UsageDataNode objects. This data structure allows the usage data to be stored dynamically to memory. The list structure consists of two pointers, therefore, the total cost is 4 bytes (each pointer costs 2 bytes). As for the nodes, each node costs 27 bytes. Thus, the total cost of a list is given by: $cost_{(bytes)} = 4 + 27 \cdot n_{(nodes)}$. Since the source code itself occupies 33% of the dynamic memory, 54448 bytes are left, which gives 252 days of storage autonomy to the device (days without emptying the list), i.e., 8 months and 12 days, considering an average of 8 events being recorded per day.

The AirConditionerRelay is a base class used for implementing a finite state machine (FSM) for multiple architectures through polymorphism. In this application, its child is AirConditionerRelayIno, which is specific for Arduino and NodeMCU architectures. Both classes are detailed in the diagram hereafter.

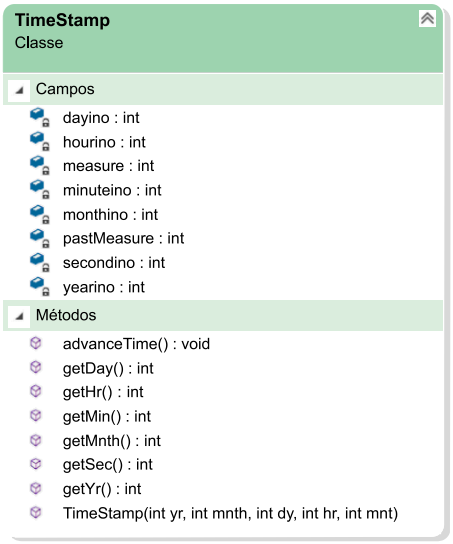


The polymorphism concept is also used for defining the states of the FSM, as shown in the following illustration.



In this scenario, the State class inherits Events (abstract, pure virtual), ActionsIno (implements Arduino/NodeMCU actions) and ActionsWin (implements actions for a Windows application, is omitted in the picture). The states OnIno and OffIno both inherit State class and each implements the actions to be performed accordingly to what event has happened.

Another important class is Timestamp. Its diagram is presented below.



This class is used for keeping track of time, it is based on the board's crystal and it considers leap years.

Wi-fi, time and mqtt settings can be defined by user in the .ino file (see picture below).

air_conditioner_v1 | ActionsIno.cpp | ActionsIno.h | ActionsWin.cpp | ActionsWin.

```
/*
  Smart Relay (for AC units) Firmware

  Developed in: ArduinoIDE (Windows 10 environment)
  Needs: Esp8266 NodeMCU core (open source)
  Developed by: Brenda Ribeiro
*/

// Type here the Wi-Fi information_____________________
#define wifiName "BrendaAP-2.4Ghz"                  //|
#define wifiPswd "redeBrendaAP"                     //|

// Type here the MQTT broker___________________________
#define mqttServer "broker.hivemq.com"              //|

// Type here the timestamp settings___________________
#define setYear   2021                              //|
#define setMonth  4                                 //|
#define setDay    30                                //|
#define setHour   10                                //|
#define setMin    10                                //|
//___________________________________________________/|
// Thecnical information, from this point forward, do not change
//(ESP8266 NodeMCU I/O ports)
```

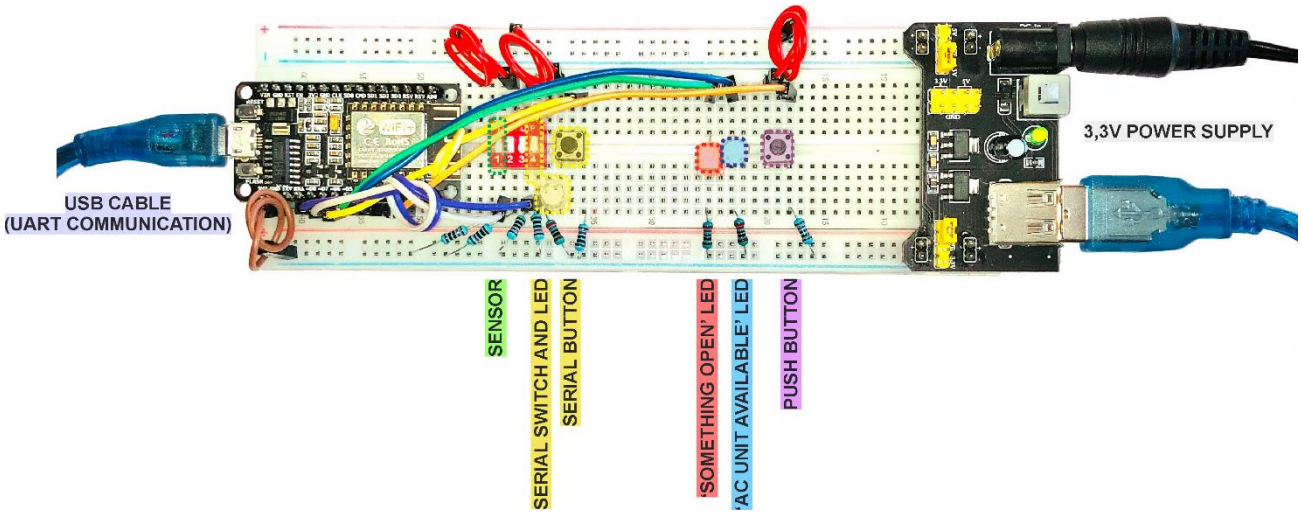
Hardware

The prototype was built using the following components:

- 5Vcc power supply
- Push button tactile micro switch (02 un)
- High bright led (02 un as outputs; 01 un in the circuit)
- 4-way dip switch
- ESP8266 NodeMCU ESP-12E microcontrolled board
- 10kΩ resistor (06 un)
- 220Ω resistor (03 un)
- Protoboard

- Jumpers (male-male)
- USB to USB mini cable (for UART communication with PC)

The circuit is show in the following image.

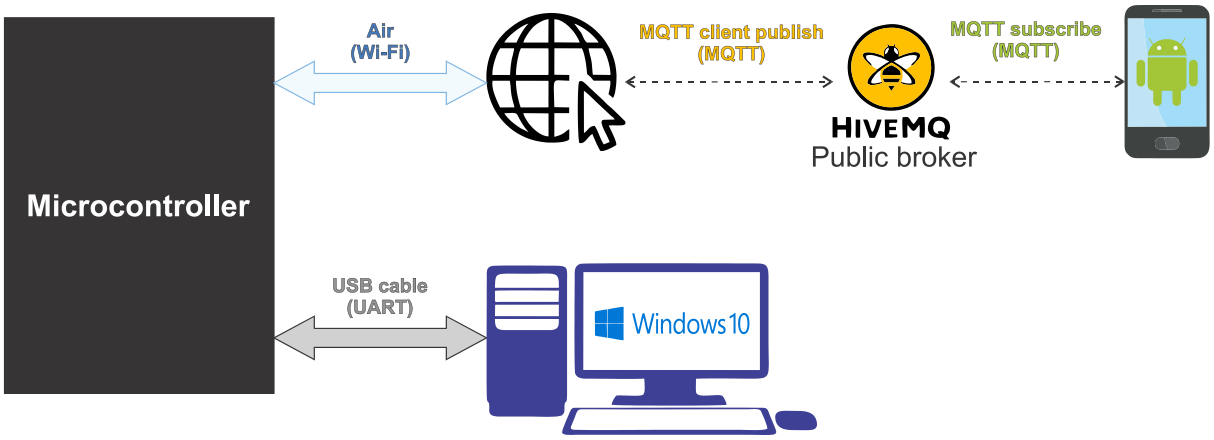


An USB cable is used to stablish serial communication (via UART protocol) to a computer. The ESP8266 NodeMCU board performs connection to the internet through Wi-Fi and sends data to a MQTT broker in real time.

A dip switch is used to allow data reading (serial switch). The data is sent by the hardware once this switch is ON and the serial button is pressed. This switch is connected to a LED, to visually inform the user about its state. This visual alert is particularly important, because, once the data is sent and the button is pressed again, it is deleted from the ESP8266 NodeMCU board.

In this prototype: a switch represents the set of magnetic sensors in the circuit; two LEDs indicate whether there is something open or the AC unit is available; a simple push button represents the robust button accessible to the final user.

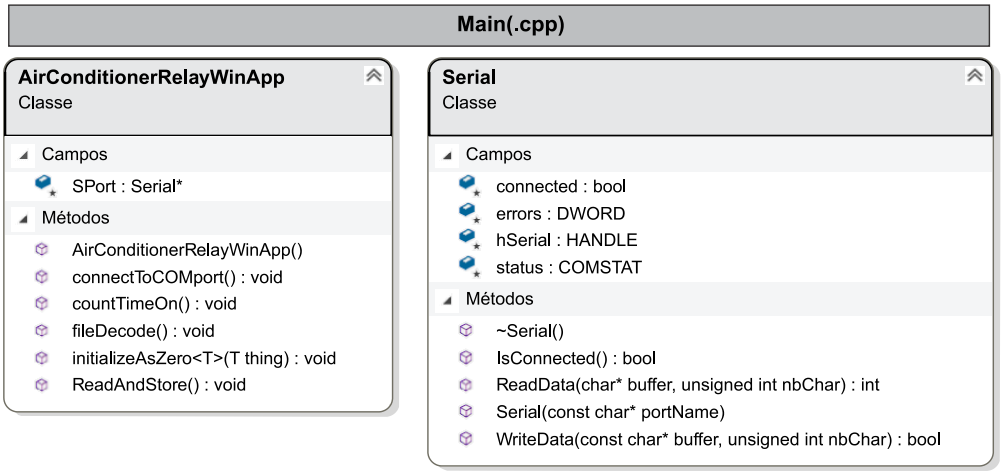
The communications diagram is displayed beneath.



Software

Desktop

The software was developed in Microsoft Visual Studio 2017 for running in Windows 10 environment. Its class diagram is shown below.



Main.cpp consists of a global AirConditionerRelayWinApp object, a bool isCOMconnected function (friends with AirConditionerRelayWinApp class) and a void main() function containing an infinite loop for the console app menu. The menu has five options chosen accordingly to user's input:

[0] Connect to COM port: connects to a serial port (its name is given by the user).

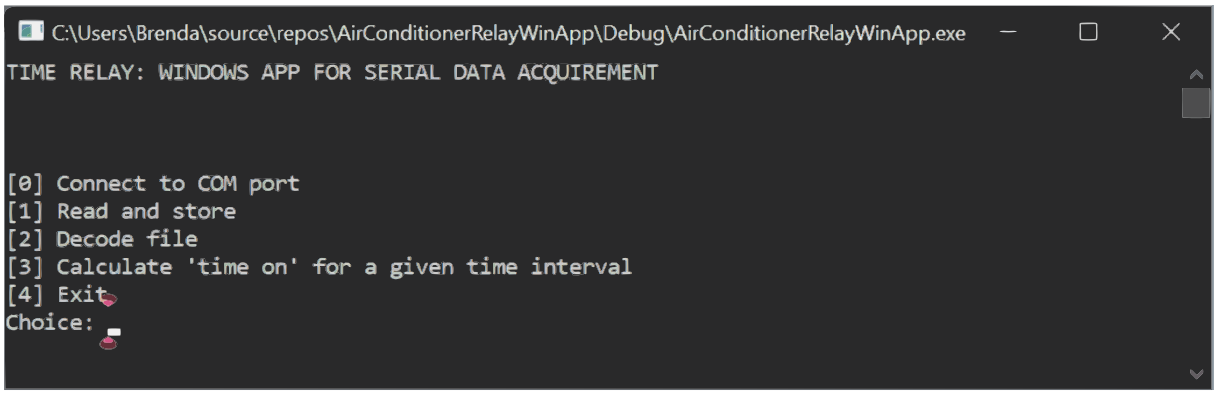
[1] Read and store: Reads data sent from the microcontrolled board via UART and stores to a *.txt* file.

[2] Decode file: Decodes data from a *.txt* file, generating a new file or appending the information to an existing one. This process transforms the brut data received from the board (*csv* alike) to a better comprehensible format.

[3] Calculate 'time on' for a given time interval: returns the total time the AC unit was powered, in minutes, for a given time interval.

[4] Exit: Leaves the console program.

A try-catch is used to prevent the user from entering invalid options. A view of the menu is presented below. After released, the total program size is 862 KB.



The resource usage is presented in the table below.

Option	**CPU* (%)**	**Memory (MB)**	**Disk (MB/s)**
None selected	0,0	6,9	0,0
*[0]**, [1]**	0,2	7,0	0,0
*[2]**	0,6	7,0	0,1
*[3]**	0,5	7,0	0,1

*Intel® Core™ i5-7200U CPU @ 2.50GHz
**Peak

Smartphone

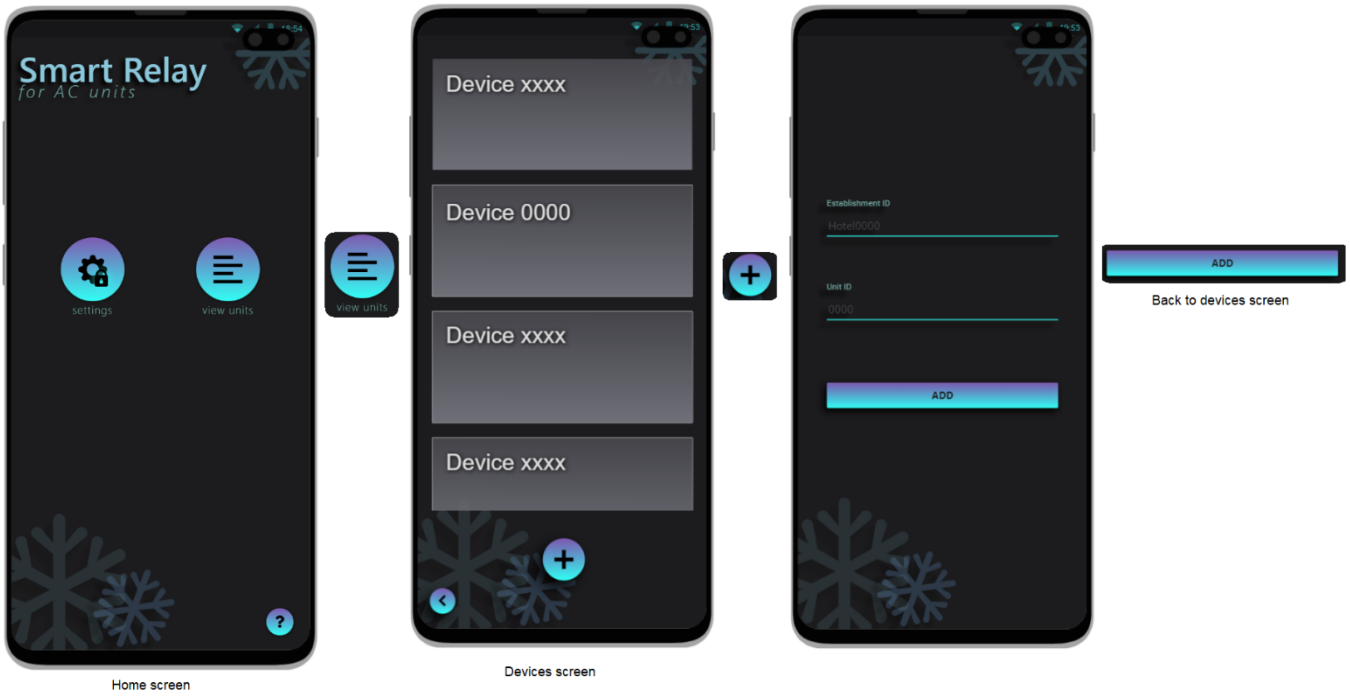
The smartphone app receives data from multiple indexed units through MPTT protocol, by subscribing to the unit's topic.

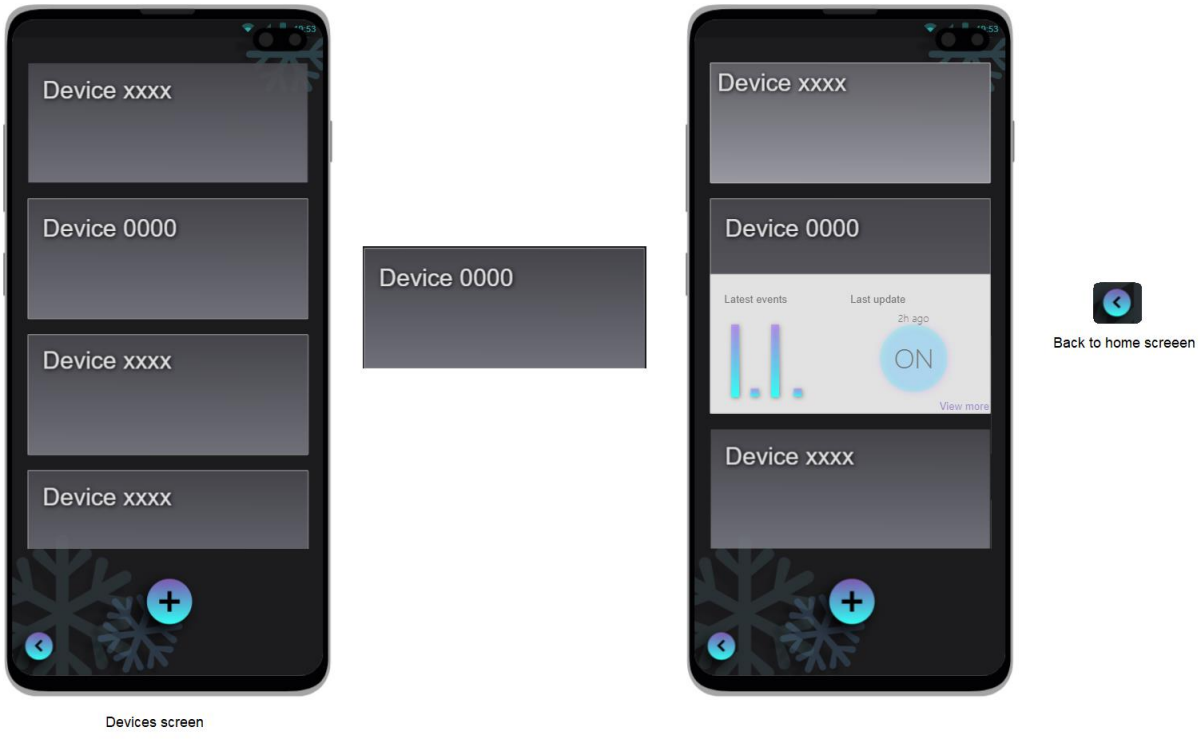
Interface plan

A few screenshots of the interface plan are presented ahead. The objects between the screens indicates the ones that need to be tapped for the transition to happen.

Home screen

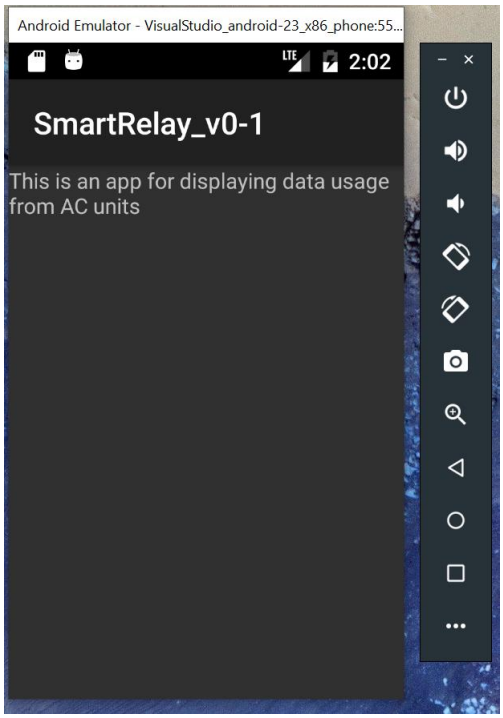
Back to home screen





Code

This version's code does not implement all required functionalities. It simply offers the foundation for C++ programming for Android, also known as *glue code*. It was generated in Microsoft Visual Studio 2017 (Windows 10 environment), through an extension for programming Android apps.



Suggested improvements

Firmware

Since the ESP8266 can drain up to 990 mW while transferring packets through Wi-Fi and 231 mW on its normal operation mode, improvements targeting power efficiency are important. For instance:

- Exploring *sleep* operational modes can save up to 65% more energy.
- Wi-Fi communication can be established only before sending data and can be ended immediately after.
- Data sending could be performed not in real time, but once per day, for example.

For optimizing the storage autonomy: the device could be capable to check, periodically, a MQTT topic to see if the usage data list can be deleted, making the presential maintenance almost unnecessary.

Software

Desktop

The software could be modified to support data transferring through MQTT, so it could be done remotely.

A user-friendly interface could be developed for allowing optimal usage.

Other environments could be supported, such as Linux and MacOS.

Smartphone

The smartphone app could implement all requirements with a user-friendly interface. It could also be available for other smartphone operating systems, such as Apple iOS.

Tests

Firmware

The hardware was exposed to 48 hours straight of functioning. During this time:

- Over 50 events were randomly induced through pressing the push button and activating/deactivating the switch that represents the sensors.
- The internet router was turned off, then, after a couple minutes, turned on.
- Data was read from the microcontrolled board through UART and MQTT subscription.

Results have shown that the firmware functioned correctly. Although the results were promising, it is suggested that other testing routines are performed:

- Exhaustion tests: with mechanical entries (I/O) or synthetic entries arriving until exhaustion (ex: dynamic memory overflow), for ensuring, for instance, the theoretical calculated storage autonomy.
- Long duration tests: leaving the device on for a long period of time (ex: couple months) for better evaluating the timestamp precision and defining an optimal frequency for updating the time counter through the internet.

Demonstration available in: https://youtu.be/6tfaT6IbHRA

Software (Desktop)

The desktop console app was executed multiple times in Windows 10 environment. Each time, random functionalities were explored in random orders and quantities.

All results have proved the correct functioning of the software. Even with the positive results, it is suggested that exhaustion tests with synthetic entries are performed in the future, for identifying the need of dealing with exceptions not identified until this moment.

Demonstration available in: https://youtu.be/sCXCv-7n8KA

Software (Smartphone)

This software was only tested in an emulator: VisualStudio_android-23_x86_phone.

Interface demonstration available in: https://youtu.be/7RxcRJHKdkU