UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

SMARTPHONE INTERACTION WITH THE IOT. SMARTPHONE DEVELOPMENT

A Degree Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Borja Rosas Ruiz

In partial fulfilment

of the requirements for the degree in

Sistemes Audiovisuals **ENGINEERING**

Advisor: Anna Mª Calveras Auge

Barcelona, September 2017

Abstract

The purpose of this project is to design and develop a system which enables interaction between a smartphone and an external device. By means of a smartphone application, the final user is able to detect the states of the device upon the user is interacting with, and also is able to send orders that will change states and modify parameters of the system, through a communication system based on binary chains of flashes.

The motivation for the completion of this project is born from the need of a communication system with a low manufacturing cost, that will result in a cheap and **affordable** product. This system could be used for a wide range of applications such as: air condition controlling, audio system controlling, smart home controlling, **garden irrigation controlling**, etc.… Being the last example the one which will be deployed in this project.

Acknowledgements

I want to acknowledge Anna Maria Calveras for her guidance, support and advice given to me in order to be able to accomplish and finish this project. I want to thank her specially the opportunity she has given me to be my tutor and resume with this the degree.

Table of contents

1. Introduction

The motivation and basis of this project is to design and develop a communication system between a smartphone and an external device/system which will allow the user to act on it. The requirement that will primarily lead the direction of the following project is that this system needs to be as much **affordable** as possible, making it an excellent choice that does not add extra costs to integrated applications, such as home controlling systems. Also, we want this system to be **robust** and **safe**, since a miscommunication among the actors could lead to undesired situations. The chosen means of communication for the system to send information from the smartphone to the external device, is via **flash signals**. The reason behind this decision is that nowadays almost everybody owns a smartphone with a camera integrated on it, and this camera also comes with a flash torch. This means that a Morse/binary kind-of communication system using the flash torch with chains of flashes would be the basis of our communication system, and the hardware for this communication protocol is already integrated on almost the majority (if not the whole) of the smartphones out there nowadays. Hence, **no extra costs** for additional pieces of communication hardware would be needed on the external device. Having a light-pulse-driven communication system means that a **luminosity sensor** is required in the controlled device's side. For this purpose, a first approach has been carried out on this project using a luminosity sensor. A future work on this behalf would be to change the sensor to a Light-Emitting-Diode (LED) acting as a receiver, since it is much cheaper than the sensor used on this project. We also want our system to be **robust,** and for this reason we need a way to ensure and verify that the state in which actions are performed is the correct one and also that the orders and commands given to the external device are correctly understood and applied. In order to achieve it, **image caption** and further **analysis** is performed in the mobile application using analysis algorithms on the image captured by the camera of the smartphone. We will need to design a common **communication protocol** between both actors. Finally, the results of our project will be translated in a real system prototype formed by the mobile application and the external controlled device.

2. State of the art of the technology used or applied in this thesis

A previous research on the subject matter of this project revealed some interesting results. There already exist in the market irrigation systems which work with a mobile application and an external system where both are communicated via Wi-Fi.

Nevertheless, none of them communicate with a protocol based on light signals, which would make our system a unique kind among the direct competitors. These systems offer a wider range of features including adaption to local weather, terrain type, seasons change or different configurations for different zones of irrigation besides the basic features such as the watering schedules that will be implemented in ours. On one hand these commercialized professional systems offer clearly more features than ours but in the other hand they are an expensive choice with prices varying from 120$ up to almost 200$. In this sense, our intention is basically to try to make a difference between these systems and our system by offering a competent irrigation system for a much more affordable price.

Fig. 1. Shows some irrigation systems in the market

Another aspect we researched for was the existence of applications that use the torch light of the smartphone to communicate and receive information also in form of light. We found that there is an application called *Morse Code Reader Flashlight* which is able to send and receive Morse signals by means of the light torch of the smartphone. Although the main purpose of that app is to be able to communicate using Morse language, the idea to capture the receiving signals coming from a light source on the smartphone application could be applied in a future development in order to extend the scope of our system to a bidirectional focused communication, since for the scope of this project a unidirectional kind-of drift has been applied.

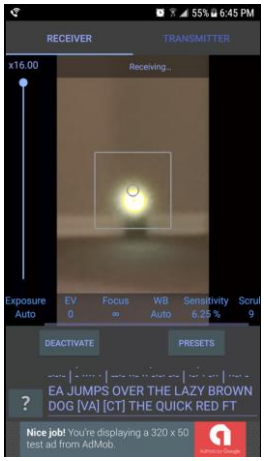


Fig. 2. Morse receiver/transmitter application

3. Functional design and System architecture

3.1. Functional Design Specification Overview

Our system is composed by two different elements, a smartphone application and a microcontroller embedded in an external device. The application is manipulated by the user and through the different possibilities it gives, the user will be able to interact and communicate actions to the microcontroller, which will understand and process every action received according to a shared communication protocol. Next, we will break down the functional point of view for these two actors from a user perspective.

3.1.1. Smartphone application

The **inputs** of our application are mainly **two**, image input and user interaction input. The first one, is retrieved by the camera of the smartphone which will capture image in real time. The second one is the user touch input when interacting with the application's different options.

The **outputs** of the application are also two. The first output is each one of the different Graphical User Interfaces, from now on GUI, that will be rendered in the screen and presented to the user. The second output is the torch light that will be the vehicle of our communication protocol system. The application will detect the state of the external device, giving the user different options according to what the system's state is, in every moment. This check will be done in the start of the application through an **action button** taped by the user, in the launch screen.

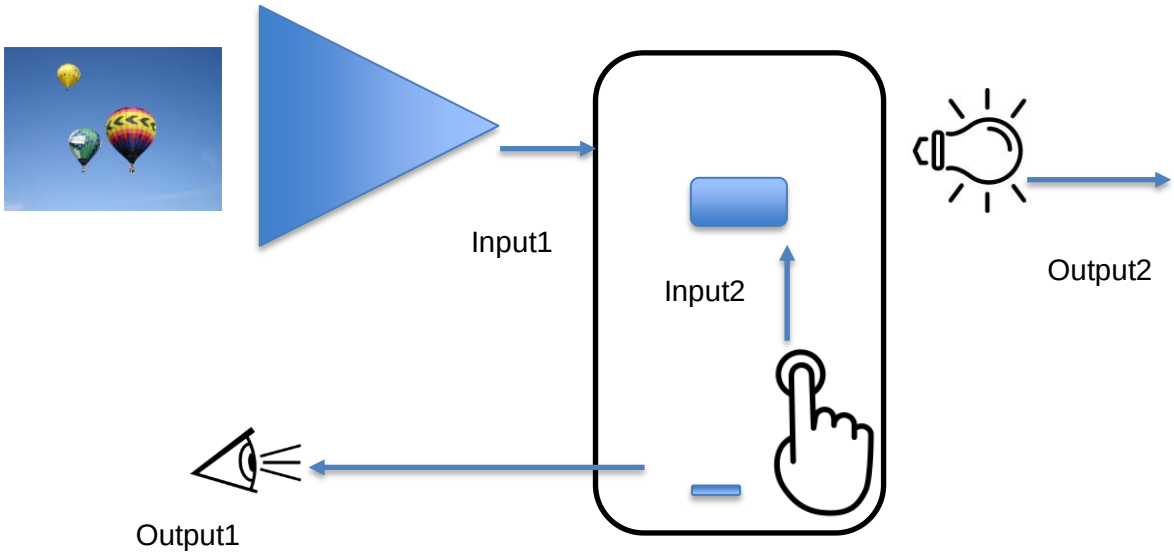


Fig. 3. Inputs & Outputs of the app

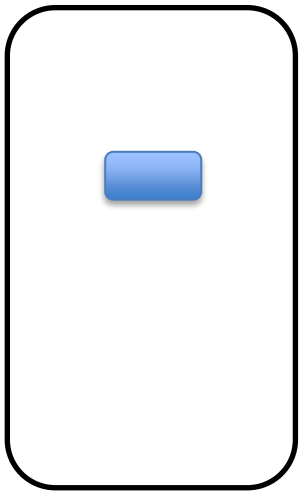


Fig. 4. Design of the Launch Screen

GUI 1.1 Figure 3: Launch Screen – General Requirements	
GUI 1.1.1	Screen name is Launch Screen
GUI 1.1.2	Contains 1 action button
GUI 1.1.2.1	Action button text: *Start*

Table 1. Shows the general requirements for Launch Screen

When the detection is performed, a system message will give the user the possibility to change the current state of the external device, or in case the system is already in a state to start receiving data and be programed, then the user will be prompted to a different GUI. The rest of the GUI tables, like *Table 1*, are appended in the *Annex*.

3.1.2. External device

As per the external device which contains the microcontroller here we are going to break down the requirements and specifications. We only need an **input** which is the light that comes from the torch light of the smartphone and is captured by a sensor and processed by the microcontroller. We find that the external device generates **2 different outputs**. The first one is the programmed data that is received, translated and stored in the memory of the microcontroller, and the other are the different LEDs that are lit depending on each situation. The first output, the data that contains the information that we want to transmit from the application to the microcontroller is used by the system itself and is the result of our process, being this result successfully carried out or not, depends on the communication protocol and the correct implementation of our system. The other outputs, the LEDs, serve to a visual reference and assurance for the user, so that at every moment the current state of the device can be tracked and monitored, and also to the state detection of the application.

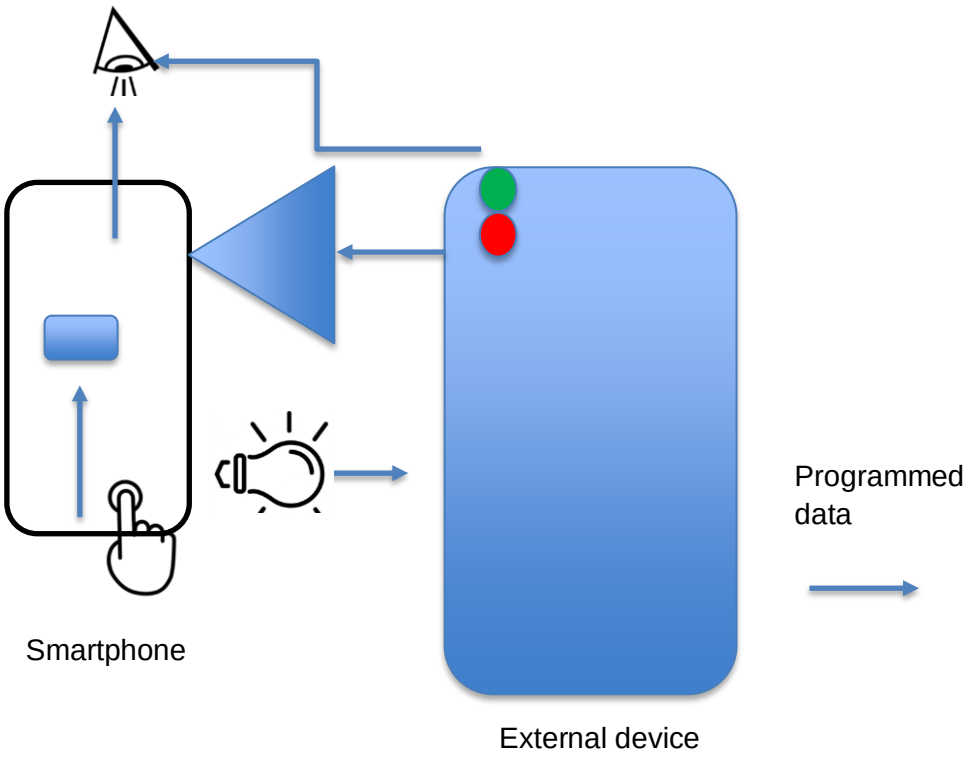


Fig. 5. System architecture

3.2. System architecture

Our system application is a simple client based mobile application, also known as fat client, designed to be self-organised and autonomous without any connexion to a server, although in a future work it could be implemented so that data from users can be retrieved and monitored. The data gathering would be needed when creating business operations such as CRM and customer operations, but for now this is not on the scope for this project. Systems should be designed with considerations for the user, the system, and the business goals. For each one of these areas, key scenarios, important quality attributes and satisfaction areas should be outlined. And here it will be exposed the overall architecture of our system application, some use cases, and some functional and quality requirements will be handled.

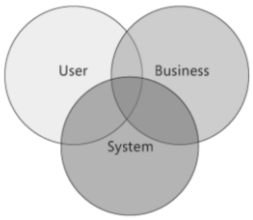


Fig. 6. Three pillars of our system design

The approach we are going to address this section is by following **two** of the many mechanisms that are used for software description, which is the **architecture functional viewpoint** and the **CRC cards**. By breaking down the desired system in single viewpoints we can specify in a precise way each of the important areas that our system encapsulates. With the CRC cards, we will have an easy and visual way to define the architecture of our system

3.2.1. Functional viewpoint

This englobes the functionalities that our system will require to fulfil the user needs.

Hence the main functionalities we will offer to our consumers will be:

- **Programming the days of the week**
- **Programming the start hour/end hour**
- **Synchronization with the mobile's phone calendar**
- **Checking of the programmed data**

The main idea is to give a simple but robust system which, with few actions the user can program easily the irrigation system of his/her house. The next UML diagram shows the use cases that are covered by the functionalities that our system offers to the users.

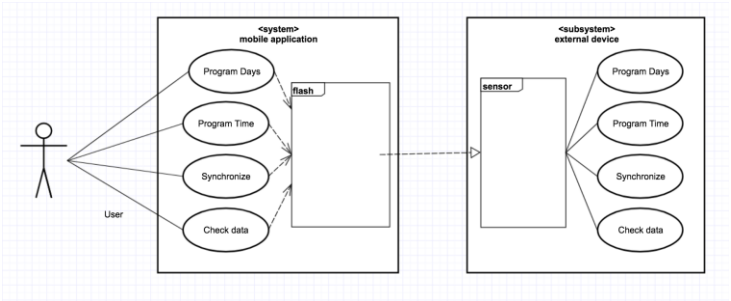


Fig. 7. Use cases of the system

3.2.2. Overall architecture viewpoint

The system architecture of our future prototype **mobile application** is presented next.

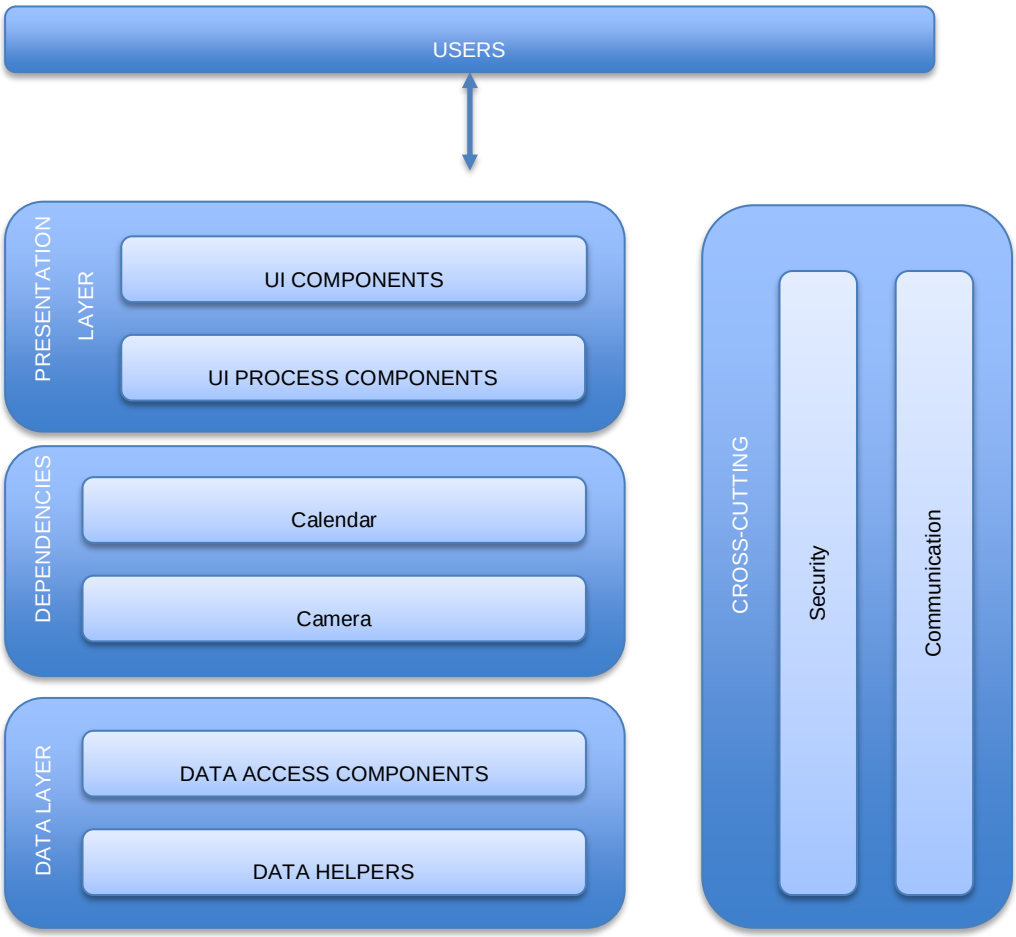


Fig. 8. Shows the application architecture

3.2.3. CRC Cards

A **Class Responsibility Collaborator** is a collection of standard index cards that have been divided into three sections. A Class represents a collection of similar objects, a Responsibility is something that a class knows or does, and Collaborator is another class that a class interacts with to fulfil its responsibilities. The top row contains the name of the class, the left column the responsibilities of that class and in the right column its interactions and collaborations with other modules or classes.

ViewController	
capture Output from Camera detect States: On/Off present Alerts send flash sequences to Switch On/Off	ProgramViewController

Table 2. Shows the CRC for the ViewController class

The rest of the **CRC** cards for the rest of classes that will be implemented on our application can be found in the *Annex*.

4. Methodology / project development

The proposed name for the project is **uWater**, which is the name for the smartphone application and for the project as a whole. From now on when referring to uWater, we are meaning the project, app and system. The uWater project was born from the basic idea of giving the user the possibility to control the irrigation system of his/her house via a smartphone application using a communication system based on light signals, meaning that the user would need to lay the smartphone device down onto the surface of the **home irrigation system**, which has been referred previously as an external device. The system is composed then by a **smartphone application** and the external device which we will call **Home System,** at which we will refer from now on as **HS**. We will dissect in this section the whole project into these two main and clearly different systems. Methodologies used for both hardware and software designs will be described here. An important remark needs to be highlighted, which is that all the code used in this project has been written from scratch and it is not any continuation of a previous existing project. Nevertheless, some parts of the code have been taken from examples from online courses or helping blogs.

4.1. uWater App – Smartphone App:

As presented before, uWater is the name we have given to our project and also to our application. This application has been designed and built only for iOS (Apple) devices for the moment, and the reason for programing on iOS instead of doing so also for Android, is that as a first approach, we want to present a prove of concept which works in one platform and then if needed, it could be ported to another platform. When it comes to Android App development and iOS App development, there are a few clear demarcations which make for obvious preferences. Although Android development is easy due to a deeper access to their OS and high levels of customization are possible, it is slightly more vulnerable to hacks and cyber threats. Fragmentation is also a problem in Android due to the huge variety of models and varieties available, and this becomes a problem to maintain when it comes to bug fixing. The uWater application has been developed on Xcode 8.3.3 using Swift language. It required to take a previous course on Swift and Xcode, since it was a language and a software we never used before, nevertheless we found lot of information and lessons on this matter.

4.1.1. Launch screen

When the user starts the application, a launch screen is displayed with one single “Start” button in the middle of the screen. The launch screen view is ruled by the *ViewController* class file. View controllers are the foundation of a Switch app’s internal structure. Every app has at least one view controller, and most apps have several. Each view controller manages a portion of the app’s user interface as well as the interactions between that interface and the underlying data. View controllers also facilitate transitions between different parts of the user interface.

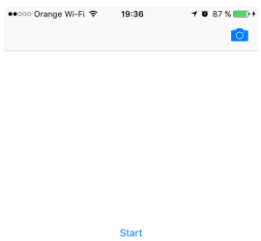


Fig. 9. Shows the application’s launch screen

Everything starts with the *viewDidLoad()* method. This method is called after the view controller has loaded its view hierarchy into memory. This is usually overridden to perform additional initialization on views. In this case our *viewDidLoad()* method includes the *prepareCamera()* method which initializes the AVCapture methods to access the camera and retrieve images from it through another method called *beginSession(.)*. These methods prepare the scenario in order to be able to access an AV resource and retrieve an AV output. By tapping the Start button displayed in the launch View, a *detectState*() action method is called. This changes the value of a Boolean variable called *detect* from the pre-set false value that it initially has, into true. Having this flag enabled lets the process start capturing and analysing a row of 22 frames. For every one of these 22 frames an analysis of the image is performed to verify which is the current state of the HS. The state can be **OFF** or **ON,** and according in what is the result of this detection, a system pop-up will prompt informing the user what is the current status detected by the app, and what is the following action to perform. The HS indicates its current state via LEDs. If the HS is **OFF**, then a red LED will be shining, but if it is **ON**, a green LED will be shining instead. The current main state of the HS is detected from the current shining LED which is contained both on the inner core of the HS device, which is not seen by the user, and another one emerging from the surface of the front side of the devices structure, which the user sees. This architecture has been designed in order to let the user visualize the current state of the system by the front side LEDs, and for the application to detect in the core of the system through the mobile camera, which are only accessible through a thin groove on the top side of the device covered with a translucent piece of glass.

a) Main state detection

Once the user taps the Start button, the flag is raised and the camera captures an image of the inner core of the system which is accessible through this glass. This image is a captured picture of the current shining LED of the inner part of the device. For this image, a group of pixels located in the centre position of the captured frame are analysed extracting for each of one the RGB components and determining which is the predominant one. This group of pixels is a matrix of **21x21 pixels**, meaning that for every frame, **441 pixels** are analysed. For each given pixel of the matrix of this image, a calculation of its RGB components is carried out, and if the red or green value from each component is superior to the other ones, then the pixel is considered to be a green/red pixel. Two methods are the ones responsible to do these calculus, *detectStateOn()* and *detectStateOff()*. They are mainly the same but they only differ in the component that is

taken into account when it comes to compare its value to the other ones, and verify if it complies with the condition to be considered a green/red pixel. The method *detectStateOn()* checks if the pixels inside the matrix of the current frame are mainly green, meaning that the light shining inside the HS is the green one, which indicates that the system is already ON. If the green component of the RGB for a given pixel is superior from the R and B components, then a summation variable increments by one, and the loop keeps evaluating each pixel until the whole matrix is done. Finally, if the value of the summation value exceeds a given threshold, it is considered that for that frame the light detected at that moment is a green light and hence the HS is already switched on. In this case the output of the method will be a Boolean flag with value true. If the summation value is under the given threshold for it to be considered as a green light detection, then the output value will be false. Same procedure is applied for the method *detectStateOff(),* returning true if the light detected is red and false if it is not the case. The output of these functions is used to consider the total of frames detected as StateOn/StateOff of the whole 22 frames. After having the number of greens and the number of reds detected, the method to decide whether the state detected is OFF or ON is another comparative threshold that is set after experimenting with different values. The condition for considering the detection of a state to be safe is that the detected green/red number of frames is equal or higher than **12** out of 22 frames in total for the ON case, and **15** out of 22 for the OFF case.

b) Alerts

Once the detection of the main state has finalized we can have three possible scenarios. In each of the possible paths, an alert informing the user what is the outcome of the detection previously done is prompted in the foreground of the application, remaining in the background the launch screen. Depending on the situation different alerts and decisions will be presented to the user.

i. **Detected green**:

The outcome of the detection algorithm is that the current state of the HS is ON, meaning that the method *detectStateOn()* has returned true for a majority of frames. At this point, after the *detect* flag is set back to false, an alert is prompted through the method *presentAlert_isON()*, by which the user is informed that the HS is already ON, and asks whether the desired action to perform next is to switch it off, go on to the **Program Mode** or Cancel. If the user decides to switch the HS off, a private method will be called and it will trigger the encoded signal to switch it off. This method is called

flashButtonSwitchOff(), and we will explain it in the next section. In the other hand if the user decides to go to the Program Mode, the current view controller will be dismissed and a new view controller called *programViewC* will be instantiated. Finally, if the user taps the Cancel option, the alert will be dismissed and the application will recover the view of the launch screen which was previously on the background.

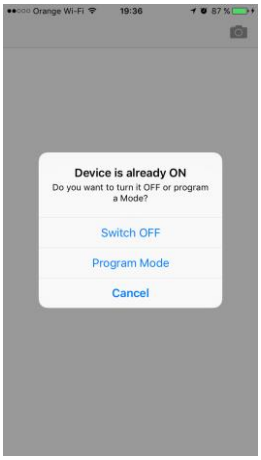


Fig. 10. Shows the alert *presentAlert_isON()*

ii. Detected red:

The outcome of the detection algorithm is that the current state of the system is OFF, meaning that the method *detectStateOn()* returned false for most of the frames, and that the method *detectStateOff()* was true instead. At this point, after the *detect* flag is set back to false, an alert is prompted through the method *presentAlert_isOFF()*, by which the user is informed that the HS is switched down, and asks whether the desired action to perform next is to switch it on or Cancel. If the user decides to switch the HS on, a private method will be called and it will trigger the encoded signal to switch it on. This method is called *flashButtonSwitchOn()*, and will be explained it in the next section. Finally, if the user taps the Cancel, the alert will be dismissed.

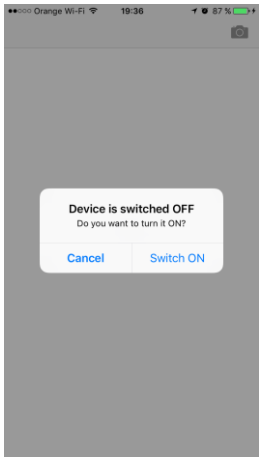


Fig. 11. Shows the alert *presentAlert_isOFF()*

iii. **Detected unknown:**

The output of both methods *detectStateOn()* and *detectStateOff()* is a false value, which means that neither a positive green or a red light have been detected. At this point, after the *detect* flag is set back to false, an alert is prompted through the method *presentAlert_isUnknown()*, by which the user is informed that the state of the HS could not be detected, and asks whether the user wants to try again the detection process or not. If the user chooses to try again, the *detect* variable will be set to true, and the whole process of capturing frames and analysing them will be carried out again.

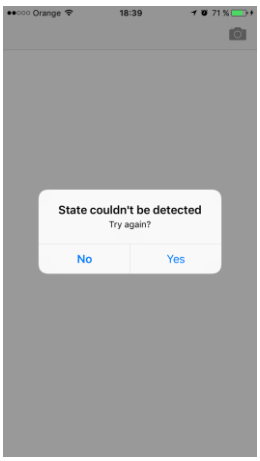


Fig. 12. Shows the alert *presentAlert_isUnknown()*

c) Flash methods

These methods are responsible of accessing the torch, if available, and send the corresponding signal to the HS in order to perform a concrete action.

I. **Switch On method:**

This method is responsible to communicate to the HS the order to turn itself on. The communication is based on an encoded light signal in a form of a chain of light/no-light events pulses of a limited duration. The duration of each pulse is 0.250 seconds and it will be the decisive factor of the velocity of the communication system in which the data is transmitted from the mobile to the HS. It is at the same time, determined by the maximum cadence in which the receiver, contained in the HS, can detect and interpret the data is received. The chosen encoded signal to communicate this to the receiver is the following: *On-Off-Off-On* → ***1 0 0 1***

After the torch finishes sending the signal, a Boolean flag called *verifyOn* is raised. This flag will be important once the flow gets back to the method *captureOutput()*, since it will be determinant to ensure that the change of state has resolved correctly before going forward to the Program mode view.

II. **Switch Off method:**

This method is responsible to communicate to the HS the order to shut down. The procedure is the same as the previously explained method and the chosen encoded signal to communicate it to the receiver is the following: *On-Off-Off-Off* →***1 0 0 0***

After the torch has finished sending the signal, a Boolean flag called *verifyOff* is raised. This flag, as well as the one explained for the previous method will be the key to reinforce and ensure that the HS has been turned off.

d) Validation methods

As described back in the first lines of this section, the system is thought and designed to be robust against the actions taken by the user to ensure a well-functioning system, free of inconsistencies and undesired situations. An inconsistency, for the sake of giving the reader a practical example, could be that when detecting the state of the system, being that the system is switched Off, the user gives the order to the application to turn the device on, and after the command has been sent, the application changes its view to the Program mode window but the reality is that the communication failed and the HS remains off.

I. Validation On:

This validation is performed after having detected that the HS is off, and the user gives the order to switch it on. The flag *verifyOn* is raised, and the main process steps into the *captureOutput()* method again, getting this time inside the condition expressed by the flag where the code validates the correct change of state from Off to On. The procedure is the same as the one explained in the *Main state detection* section, but the difference is that this time the alerts that will be prompted are different and so are the actions after confirming or refute the change of state. If the frames analysed detect that more than 12 frames of them are containing green pixels, meaning that the state has changed to On, an instance of the Program View controller will be called and presented, after dismissing the current view of the launch screen, and the application will change the view to the Program Mode one directly. If on the contrary, the detection shows that the state of the HS remains off, then the method *presentAlert_failedOn()* will be called, and a system message informing the user that the system could not be switched on will be prompted, giving the user the option to retry or Cancel the process, in which case the system pop up will be dismissed and the application will resume on the launch screen on the initial state. If the state cannot be identified as On/Off the alert *presentAlert_failedOn()* is shown as well. In all the three possible scenarios, the Boolean flag *verifyOn* is set back to its original false value.

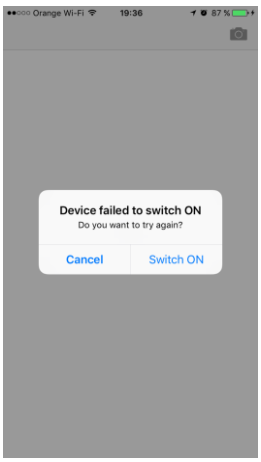


Fig. 13. Shows the alert *presentAlert_failedOn()*

II. Validation Off:

This validation is performed after having detected that the HS is on, and the user gives the order to switch it off. The flag *verifyOff* is raised, and the main process steps into the *captureOutput()* method again, getting this time inside the condition expressed by the flag

where the code validates the correct change of state from On to Off. The procedure is the same as the one explained in the *Main state detection* section, but the difference is that this time the alerts that will be prompted are different and so are the actions after confirming or refute the change of state. If the frames analysed detect that more than 15 frames of them are containing red pixels, meaning that the state has changed to Off, the Boolean flag will be reset back to its original value, false. If the command fails and the state detected is still On, an alert message will be prompted by the method *presentAlert_failedOff()* which will give the user the possibility to retry to switch the HS off or Cancel the process and leave the system as it is, in which case the alert will be dismissed and the application will resume and recover the launch screen that was in the background.

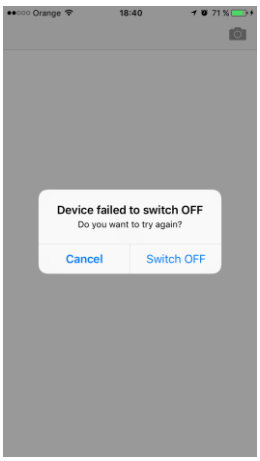


Fig. 14. Shows the alert *presentAlert_failedOff()*

4.1.2. Main Menu screen

The Main Menu or the Program View, is the window from where the different functionalities that the application offers, can be performed. This controller called *ProgramViewController,* is reached once the HS has been successfully switched on. The instance is called from the system alert that is prompted under the circumstances described before on the launch screen view.

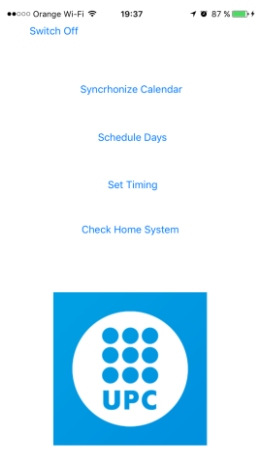


Fig. 15. Shows the application's Main Menu screen

The appearance of this window is a simple view with a column of buttons in the central axis, and a separated button located in the bottom left corner of the screen which switches off the HS. Each button contains a text which briefly describes its function. When the user taps one of them, an encoded light-pulse signal is triggered. This unique encoded signal is defined in each method associated for each button. Once the light-pulse signal finishes, the application changes automatically the context and another view controller is instantiated. The definition of the different controllers for each screen view that belongs to the associated action button is set in separated view controller Swift files, the same as the launch screen, and the Program View ones since all of them have different appearances for different purposes. The definition of these controllers will be added in this very section in order to synthetize and facilitate this project overview. So far, the application gives the user 4 different options which will be explained in the following pages, but a first glance will help us understand better the whole picture. For the first draft of this application prototype we give the user four different functionalities/options. The irrigation system can be programed by choosing which days of the week and which time the user desires the system to do it. But before being able to schedule the timings and the days we want our irrigation system to work, it is required to firstly synchronize the date of the device's calendar with the timestamp on the HS's micro controller. Here we need to specify that the Arduino controller we are using in this project does not have a timestamp library, so when the synchronization is done, it will be checked on the Arduino output's data in form of a digital date. The last functionality we included is to know what is the scheduled data so far in the HS.

4.1.3. Synchronize Date with calendar

This functionality is essential to correctly program the system and it is a requirement prior to the weekday and time scheduling. Such thing is required since the HS has no other way to set the current time by any other means than the retrieval of data from the smartphone. When the user taps on the button in the top with the name *Synchronize Calendar*, the method *synchronizeCalendarFlash()* is called which contains the code to access the torch and activate the flash chain signal. This light-pulse signal indicates to the HS that the action we want to perform is to synchronize the smartphone's date with it. The line below shows the shape of the command signal: **1 0 1 0**

Right after the order has been sent, the delegate dismisses the current *ProgramViewController*, and instantiates a new *DateViewController.* For the time being there is no validation that the HS understands this order, but in a further work, this needs to be added to reinforce even more the robustness and stability of the whole system by enabling a bidirectional communication. The synchronization date screen is very basic, containing a dynamic label text with the current date with the following format: *"dd.MM.yyyy"*

This label contains the formatted date retrieved from the OS of the device through a variable of type *Date(),* which is formatted as shown above. The format of the date affects the way the date is displayed in the current screen, so that the user can visualize and identify in a fast way that the date shown is correct, and also affects the processing of its value afterwards.

Fig. 16. Shows the application's Synchronize Date screen

Under the label, there is a button with the text *Synchronize Calendar* and on the top left corner of the view, there is a button with the text *Main Menu*. This last one dismisses the

current view going back to the Program View, regardless of whether the user synchronized the date previously or not. The method called *synchronizeDate(),* retrieves the current date and converts every digit into a binary string of 4 bits to finally transform these bits into flash signals. The first thing this method does when called is to retrieve the value of the text label and assign this value to a String type variable. Then, an Array of the characters contained in the date String is created, and for each character containing a number of the date (not taking into account the dots), a variable is created with this value. These variables convert the String value of the Array into an Integer type. Now that we have located and isolated all the digits of the date that we want to transmit, a conversion is done from integers into bit Strings of 4 digits by a private method contained in the *DateViewController* called *pad().* Since we want to transmit digits from the range between 0 and 9, and we want to transmit it in a binary form, we will need at least four bits to be able to transmit the range of numbers desired, being *0000* the binary sequence for 0 and being *1001* the one for 9. After this conversion, a String is formed with the concatenation of all the binary strings for the eight digits (two for the day, two for the month and four for the year), which conform the total of 32 bits of the array that will be sent. The last thing that will perform the *synchronizeDate()* method is to call another private function called *sendDateCodeWithFlash(),* with the concatenated String as a parameter. This last step accesses the torch and emits the light-pulse encoded signal the same way as it has been described previously. In order to start sending the signal, we need to synchronize the application with the HS luminosity sensor so that the sensor gets ready to detect the signal about to be sent. This concept will be explained in detail in the section dedicated to the HS development. For the ignition signal the torch is lighted for 0.250 seconds, and after this pulse, the actual binary signal is transmitted. The algorithm goes over the String, analysing one by one each of the characters that belong to the Array. If the current position contains a ‘1’, the torch light is turned on and remains as it is for 0.250 seconds, until the following digit is analysed. If the following digit is again another ‘1’, the torch remains on for another 0.250 seconds, but if it is a ‘0’, then the torch is turned off and remains off for 0.250 seconds. As this process goes on, the application will be busy and the user is not able to perform any action until the signal is sent completely. Once the action has resolved and the light signal sent, the user gets back the control of the application and can do the synchronization again or go back to the Program View by tapping the button in the top with the name Main Menu.

4.1.4. Set Timing

This functionality gives the user the ability to program before hand at what time we want the irrigation system to start working, and also at what time we want it to stop.

This functionality is attached to the action committed when the third button, the one with the label text *Set Timing*, is taped. When the user taps on it, a sequence of flashing lights is ignited with the information that will say to the HS to navigate to the controller where times can be set. The sequence used here is the following: → **1 0 1 1**

Once this signal is sent, the Program View controller is dismissed, and a new controller of type *TimePickerViewController*() is instantiated. The new view contains the back button we already explained in the last bullet point, that redirects back to the Main Menu, two separated displays with a digital-clock-like time pickers with its corresponding text label above them, and an action button that triggers the sending of the signal. Both time pickers are defined by *pickerView()* methods, which build the structure of the picker with variables for start/end hour and start/end minute *UIPickerView* variables. Linked to the picker views we have arrays to set the values that are going to be displayed for every picker view. So, in one hand we have an *hourArray* which contains the values from *00* to *23*, and in the other hand another array called *minuteArray* which contains the values *00, 15, 30* and *45*. The values from the minute array are designed to be quarters of an hour so that the user can choose fractions of hour for the start/end times. Above the top time picker, the text label reads *Start time*, and the bottom one has a text label with the title *End time*. The action button in the lowest part of the view is the one in charge of sending the encoded signal with the information of the start and end times to the HS, and its title is *Program selected time*.

Fig. 17. Shows the application's Set Timing screen

The method *programSelectedTime()* is called when the user taps on the bottom action button. At the moment of the interaction, the current values of the time pickers are retrieved, and through the method *pad(),* it is proceeded to transform the values into String arrays, which ultimately will be concatenated in a larger String. As we have two main data objects which are the start date and the end date, and each of these dates are composed by four digits, we will need to convert and transmit 8 different integer values in a 4-bit binary format. This means that a total of 32 bits will conform the message that will be transmitted. This final String, which is called *timeString*, is then sent as a parameter in the method *sendTimeCodeWithFlash()*. This method is similar to the method *sendDateCodeWithFlash()* described in the previous section. It transforms the binary chain of characters into lights/no-lights which will be received by the HS and decoded to reconstruct the binary chain again which will finally be converted to integers. Before the start of the message transmission, an ignition signal is sent to the HS in order to let it know that the message is going to start being transmitted. The ignition signal has the same format as it is explained in the previous point Synchronization with the calendar.

4.1.5. Schedule Days

One of the other main functionalities that this irrigation system gives to the user, is the option to program which day or days of the week the system is desired to function. A button with the text *Schedule Days* in the Main Menu screen is the one by which tapping it, triggers the encoded light-pulse signal that tells the HS that the user is about to start scheduling the days of the week. Such button is located below the *Synchronize Calendar* one. The following sequence is the one that gives the HS the order to switch to this functionality: →**1 1 0 0**

After this sequence, the context of the application is changed, the Program View controller is dismissed and the control is delegated to a new instance of a controller called *DayViewController*. The appearance of this new screen follows the simple and plane tone of the whole application implementation. A column in the main axe of the view composed by 7 equidistant buttons with the seven names of the days of the week. Also, a text label is placed on the top of the view with the text *Select irrigation days*, and finally, two different action buttons complete the whole scene. In the top left side of the view, the first button called *Main Menu* has the same functionality as the same so-called buttons of the other scenes. The second action button, with the label *Program selected days*, is placed in the bottom part of the scene, below the day buttons and is the one by which interacting with, the preferred selected set of days is transmitted to the HS.

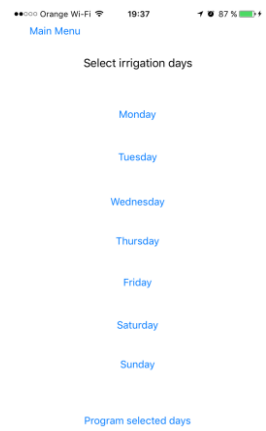


Fig. 18. Shows the application's Schedule Days screen

The user selects the preferred day or days to program the irrigation system to work and then taps the program button in order to send the set of selected days to the HS. When taping on one of the day buttons, it will be highlighted, indicating that the selected day will be programed, not the others. The process behind tapping a day button is that, besides highlighting it, which serves as a visual indicator for the user to know which days are being selected and will be programed, it also raises a flag with the name of the day which will be used afterwards by the *Program selected days* action button, to select which days have been highlighted, and hence selected, and to transmit the signal containing them to the HS. The method with the name *programDaysFlash()* is called by tapping the action button at the bottom, and the process to send the selected days starts. The function is nothing but a set of '*if*-else' condition validations, starting with the usual ignition signal. After the first ignition signal, the Boolean variables correspondent to each of the days, starting with Monday and ending with Sunday, are checked. If the value of the Boolean variable linked to a day is true, then the torch is turned on and the application keeps waiting with the torch lighted for 0.250 seconds. Then the next Boolean variable for the subsequent day is checked and the same logic is applied. If the torch was already turned on, and the next day check turns it on again, no conflict will happen, since it will remain turned on. The same applies for the case that the torch is off and the following check turns the torch off again. The resulting signal's length is 7 pulse periods, each one for each day of the week. Once the encoded signal transmission finishes the application delegates the control again to the ProgramViewController().

4.1.6. Check Scheduled Data

This is the last functionality that the application delivers to the user and by using it, the user is able to check what is the programed parameters so far that have been

programmed in the HS, but not yet in the mobile application. By selecting the button with the text label *Check HS,* the flashing sequence is transmitted to the HS. The sequence is the one below these lines: **→1 1 0 1**

No new view controller has been created for this functionality since it is relying entirely on the HS side.

4.2. uWater Home System

The so-called HS is the other key piece of the uWater project, which together with the uWater application gives sense to the whole system. The HS is the responsible to receive and interpret the orders from the smartphone, and ultimately perform the actions that the user commanded. It is formed by a single-board microcontroller, which processes and carries out the different functions of our system, a luminosity sensor that is the responsible to detect the light signals coming from the smartphone and some LEDs that will be accountable for indicating the HS state at every moment. A detailed electrical scheme of the designed circuit of the HS is attached in *Annex*. First, we are going to break down the HS in its **different components**, and deepen into their specifications and the role and dependencies among them. Then, we will explain in detail the code designed and developed to bring the HS to life.

4.2.1. Microcontroller board

The microcontroller used in the prototype of this project is an **Arduino Adafruit Flora** board. In order to supply the power to the board, an USB cable is plugged into it and it is connected to the laptop, which besides serving as a battery, it also monitors the output logs of the microcontroller that will help us visualize the result of the HS operations. A further work in this regard would be to incorporate a small LED screen into the HS device which would give visual information to the user about what is the programmed data on the HS. For the moment, we are going to rely on the logs generated by the board and displayed on the console. The language used for programming the microcontroller is C++, and the software used to program it is the **Arduino IDE**, which is an open-source Arduino Software.

The board has a set of outputs/inputs that let us design our circuit according to the specifications we want for it. These are the pins used and its function:

1. **GND:** Ground of our electrical circuit.
2. **3.3V:** The power supply pin for the luminosity sensor

3. **SCL/SDA:** Part of the I²C bus, with the Serial Data Line (SDA) and the Serial Clock Line (SCL), which will be used to connect the board with the luminosity sensor and share data information.
4. **D12:** Output connected to the red LED.
5. **D6:** Output connected to the green LED.
6. **D9:** Output connected to one blue LED.
7. **D10:** Output connected another blue LED.

4.2.2. Luminosity sensor

This component of the HS has the role of capturing the luminosity in every cycle of the clock that we, as programmers decide, and communicate it to the microcontroller which will make use of it. The model of luminosity sensor used in this prototype is the **TSL2561**. A further work in this regard will be to substitute this sensor by a LED functioning as a receiver. This change would decrease the total consumer cost of the system and with this make it available to a broader audience. The pins that are used are:

1. **GND:** Ground of our electrical circuit.
2. **VCC:** Power supply in.
3. **SCL/SDA:** Part of the I²C bus, with the Serial Data Line (SDA) and the Serial Clock Line (SCL), which will be used to connect the board with the luminosity sensor and share data information.

The sensor is a key piece and it captures the luminosity of the light that falls upon its transductor, and converts it into lux values. In order to be able to use this component we need to include an Arduino library into our IDE project where we have the code to process the data coming from the sensor. In addition, some configuration regarding the gain to be applied to the sensor or the integration time, which will affect the resolution and velocity of the data conversion, can be chosen when programing with this component. Since the purpose of the sensor will be to detect a flash light, we do not care about the resolution of the data, but we do care about the velocity of this process, since we need the fast system possible.

4.2.3. LEDs

These components play an important role on the system, since they serve as indicators of the system's state to the user, and also to the smartphone application. We will use LEDs of three different colours. We will make use of **three green LEDs** connected in parallel. Two of these will be placed inside the box, just below a little groove located on

the top surface, and this LED will be the indicator of the system's state that the application will detect through its camera. The other one will be mounted on the front side of the HS surface, and will be the visual indicator for the user. This LED will be turned ON when the system detects a signal coming from the app commanding it to turn on. We also use **two red LEDs** connected in parallel following the same scheme and logic used for the two green LEDs explained before. The inner LED will be the one used for the state detection and the one facing the exterior of the device, placed just next to the green one will be the visual indicator for the user. To indicate to the user the mode in which the HS is currently working, **two blue LEDs** will be placed in the front side face of the device just below the red and green LEDs. Since the Arduino microcontroller board has only 4 outputs, and two of them are used as the ON/OFF indicators, only two remain free. This limitation is worked around with a combination of these LEDs. For the different modes our system has, which in fact for now are four, a specific LED or combination of them is triggered.

1. Synchronization of the Calendar: For this mode, the first blue LED is turned on. The LED will be turned ON once the detected light-pulse signal is verified to belong to the command to change the working mode into the synchronisation of the calendar. Once the synchronisation has ended, the LED is turned off.
2. Setting of the Time or: For this mode, the second blue LED is turned on. The LED will be turned ON once the detected light-pulse signal is verified to belong to the command to change the working mode into the days programming. Once the scheduling has ended, the LED is turned off.
3. Scheduling of Days: For this mode, both blue LEDs are turned on and will be kept shining until the schedule has finished and the HS gets back to the Main Menu state. The LEDs will be turned ON once the detected light-pulse signal is verified to belong to the command to change the working mode into the time scheduling. Once the schedule has ended, the two LEDs are turned off.
4. Mode 4 or Check Programmed Data: For this mode, both blue LEDS are turned on, then the information of the HS will be displayed in the console window and finally the LEDs will flicker two more times before turning off. Once this happens the microcontroller will get back to the Main Menu state.

4.2.4. Home System software implementation

The HS is programed in a loop contained inside the main function *loop().* Before the *loop()* function kicks in, a set of variables that will have a role inside the program are initialized and given a value. The more important variable among these, is the so called

currentStateMode. This parameter will guide and dictate the current state in which the microcontroller is receiving and processing the data, and will be dynamically changing its value according to what commands are received from the application. Another important value is one called *currentStateLed*, which will take by default the value of a constant called *redLed.* This value is '12' which is at the same time the number of the output pin that is connected with the red LED. When the system is switched on for the first time, meaning that it is plugged into the mains, all the variables are initialized with pre-set or memory values. These values, among the others, are meant to drive the system into the OFF state. Also, the variable *currentStateMode* is given a default value of 0. In such state, the microcontroller can receive orders to switch itself off/on or navigate through the different functionalities.

The main action of our program will be receiving and processing the data coming from the luminosity sensor, and such thing will happen almost all of the time, and according to if this data resembles to some patterns matching the same encoded patterns coming from the application, other processes will be called. As soon as the HS is booted, luminosity data from the sensor will be retrieved through the method *tsl.getEvent(&event)* where the parameter *&event* is from type *sensors_event_t.* Right after the data from the sensor is taken, the red led will be lit by means of calling the method *digitalWrite(currentStateLed,HIGH).* The approach we decided to follow for the project is a **state machine**, in which through a series of condition validations we will navigate from states. The main state, as it has been previously commented, is the one defined by the value '0' of the *currentStateMode,* which will be the one in charge of receiving the signals to change between the different states. We jump into this state after successfully passing this condition, then the system is ready to receive and process the light-pulse encoded signals from the application. This waiting status, which we could call as 'stand-by' state, will be retrieving data from the light sensor every **10ms** and in every cycle, it will be checked if the value received from the sensor is above a predefined value. This value is assigned to a variable called *threshold*, which has a default value of 175 lux. In order to consider that the sensor is capturing what it could be a potential message coming from the phone, we need to stablish a threshold where any captured data from the sensor above this level will be considered part of the light signal. Once the microcontroller has verified that the sensor has detected a sample, we consider that a message coming from the device is being sent. At this point we store the detected value in an array called *stateArray*, which has a length of 4 positions corresponding to the 4 bit-length modality sequences. These samples are stored with a delay between them of **240ms**. After the array is filled with the value of the samples taken from the sensor, it proceeds to evaluate

if it matches any of the known patterns for the possible states or actions that the sent message intents to transmit. It is carried out by a sequence of '*if-else if*' statements that will validate if the sequence is recognizable. Depending on which is the validation that succeeds, different processes will take place.

1. The detected signal corresponds to the **Switch ON** action: The most important thing here will be to raise a Boolean variable flag with the name *modeOn.* Its function in our program is to act as a filter for the incoming signals, in order to discern when the HS is ON and ready to execute the different functionalities of the application, from when it is OFF and there is no need to validate those incoming messages since the system is off. This serves as a safe measure to avoid doing extra work when there is no need from a functional point of view. Another important thing is to activate the green LED and turn off the red one. To do so, the value of the variable *currentStateLed* is changed into the value of constant called *greenLed*, which has as value '6'. However, before this change, the red LED is turned off by calling the method *digitalWrite(currentStateLed, LOW)*. Before the green LED is lit, a couple of flickers of the blue LEDs happens, giving the user the chance to validate that the blue LEDs are working fine.

2. The detected signal corresponds to the **Switch OFF** action: The most important thing here, as it happens with the opposite action, will be to set the variable *modeOn* to 'false'. The variable *currentStateLed* is given the value of the constant *redLed,* but only after having switch off the old state LED, no matter if it was already the red led.

3. The detected signal corresponds to the **Synchronize Calendar** action: The *currentStateMode* is changed to the value of the constant ***mode1*** the value of which is '9', which at the same time matches the output number for the pin connected to the according blue LED. Such LED is lit by mean of the already known method *digitalWrite(currentStateMode, HIGH).*

4. The detected signal corresponds to the **Set Timing** action: The *currentStateMode* is changed to the value of the constant ***mode2*** the value of which is '10', which at the same time matches the output number for the pin connected to the according blue LED. Such LED is lit by means of the already known method *digitalWrite(currentStateMode, HIGH).*

5. The detected signal corresponds to the **Schedule Days** action: The *currentStateMode* is changed to the value of the constant ***mode3*** the value of which is '11', which does not match any pin output number, but will be used when validating the state mode in which the microcontroller is currently working. As we mentioned before, since we do not have in

our hands the capacity to lit four different LEDs, we will turn on both blue LEDs and let them light until the data coming from the application is successfully received.

6. The detected signal corresponds to the **Check Scheduled Data** action: The *currentStateMode* is changed to the value of the constant ***mode4*** the value of which is '12'. For this last mode, what will happen is that the data programed until the time being will be prompted through the PC console meanwhile the two blue LEDs flicker for 3 times and then remain off.

So far, we have seen what happens when an incoming message is detected, stored and its reference to a known pattern is acknowledged. Now we are going to get into detail of what happens when the microcontroller founds itself working inside a different state than the mode '0' we explained before.

- **Mode 1 or Synchronize of the data**

This state is entered when the variable *currentStateMode* is assigned the value of the constant *mode1* in the detection of the command in the main state '0'. After validating that the *currentStateMode* does not equal to the first condition, and is not equal to '0', then inside the '*else'* expression, again another validation is performed with a series of '*if-else if*' condition statements validating whether the current state mode is *mode1, mode2, mode3* or *mode4.* Once it is verified that the current mode is *mode1*, the microcontroller will keep retrieving and processing the luminosity data coming from the sensor. Unlike the previous case, where the signals expected to be received where the ones for changing between states (or modes), now the signal to be received is different, as it has been explained in the sections of the uWater application. The binary signal containing the information of the current date on the calendar is formed by 32 bits. When the first sample has been detected to surpass the threshold level, then the microcontroller waits for **376ms**. This delay is necessary in order to start capturing samples from the pulses in a position on each pulse, approximately in the middle of the duration of this pulse. After waiting this time by means of the *delay(x)* method, where *x* is the number in milliseconds the microcontroller will wait, the 32 samples are captured with a delay in between samples of **239ms**. After the caption of the whole 32 samples is done and stored in the array called *synchDateArray*, then the algorithm proceeds to convert the captured samples from lux values into binary and then to decimal by a method included in the microcontroller program called *binary2decimal*. After the decimal value is calculated, the output value is stored in an 8-element long array called *dateArrayDec.* This array will contain the 8 digits decoded from the incoming signal containing the information of the

smartphone's date in the same format they were encoded, being it the following: *"dd.MM.yyyy"*

After having retrieved the date, we will pass it to the console by using the *Serial.print* method, in order to visually verify if the synchronized date on the HS is the correct one. As mentioned before, a further work would be to show this information through a small LED screen for example. For now, on this prototype we will rely on the logs written by the microcontroller on the debug console.

Before we step out from the mode1 piece of code, few things still remain to do. First, we need to raise the Boolean flag called *synchronizedDate*, for it will be used in the 'Check the programmed data' mode. Another thing is to get back to the *currentStateMode* = '0'. Like this, we get back to the main state, since we have already synchronized the calendar. Finally, the last thing remaining is to turn down the blue LED.

- **Mode 2 or Set the Timing**

This state is entered when the variable *currentStateMode* is assigned the value of the constant *mode2* in the detection of the command in the main state '0'. Once it is verified that the current mode is *mode2*, the microcontroller will keep retrieving and processing the luminosity data coming from the sensor. The same process that has been explained previously is followed. This time, we also need to retrieve 32 samples, since we will be receiving 8 different decimal values, encoded with 4 bits each. The format of the digits transmitted will start with the first number of the start time, and finish with the last one of the end time. Start: '**h**h:mm' / End: 'hh:m**m**'

Being the following string the one been transformed to binary: **h**hmmhhm**m**.

Basically, the process will be the same applied in the mode1, since we want these 32 bits of data captured to be transformed into decimal values and then send the result to the console. This time we also modify at the end the value of *currentStateMode* to '0', and the flag we will raise is the one called *programmedTime*, which will be used in the *mode4* or Check Programmed Data. The delays are also the same ones applied in the *mode1* for capturing the samples in the correct time, which are **376ms** after the ignition signal is detected, and **239ms** between each sample. In order to verify what is the result of the detected signal, the converted decimal 8 digits are prompted on the console debug window, displaying what are the programmed start and end time for the irrigation system. Before exiting the *mode2* state we will need to turn down the blue LED of this mode and change the value of the variable *currentStateMode* into the value '0'.

- **Mode3 or Scheduling of Days**

This state is entered when the variable *currentStateMode* is assigned the value of the constant *mode3* in the detection of the command in the main state '0'. Once it is verified that the current mode is *mode3*, the microcontroller will keep retrieving and processing the luminosity data coming from the sensor. The same process that has been explained previously is followed. This time though, we will be detecting 7 samples which belong to each of the 7 days of the week. Here as well as the other two previous states, the time delayed after the ignition flash signal is received is **376ms** and afterwards the samples are stored every **239ms**. Once the 7 samples are stored, the resulting programmed days that have been sent by the uWater application, are displayed through the console by validating each sample if its value is below or above the threshold. This is done by checking one by one the detected value of each sample and displaying in the console the correspondent day to that sample. Before leaving the state and returning to the main one, both blue LEDs that where lit previously are turned off, the value of the variable *currentStateMode* is put back at '0' and finally a Boolean flag called *programedDay* is raised so that later on, when the *mode4* is detected, the HS will now whether the days have already been programmed or not.

- **Mode4 or Check HS**

This is the last mode or state that the uWater system offers to the user. The HS enters this state when the variable *currentStateMode* is assigned the value of the constant *mode4* in the detection of the command in the main state '0'. Once it is verified that the current mode is *mode4*, the microcontroller will check one by one the set of Boolean flags *synchronizedDate, programedDay* and *programedTime.* Depending on the value of the flag the information will be displayed on the console window or not. For example, if the *synchronizedDate* is 'true', then the current date that was synchronized previous to the check is prompted in the console, meanwhile if there has not been any synchronization of the calendar yet, a message informing the user that it has not been programed yet will be displayed instead. The same conditions are applied to the other flags resulting in the output of the programmed data, if any, or not of the different functionalities.

5. Communication Protocol

In this section, we are going to present the communication protocol that our uWater system works with. When we talk about the uWater system, we mean both the uWater application and the uWater HS. As it has been explained in the previous pages, in this

system there is a clear separation between a sender actor and a receiver actor. The smartphone application is the sender element and the HS is the one receiving and interpreting the messages send by the first one. However, in some specific situations, the HS will be acting passively as a sender and the application will be the one receiving the information from the other. These situations are explained in the pages 15-21, with the state detection, and the state change validations. Apart from these specific and punctual moments, we can consider the system being as one directional communication channel, being the smartphone the one giving the orders and the HS obeying them. Our communication protocol is based on a combination of Visible Light Communication system (VLC), which is a data communications variant which uses visible light between 400 and 800 THz, and a binary code. Light pulses are used to send binary codes containing the information that we want to communicate from one side to the other. The light source is the torch light from the smartphone where the uWater application is installed, and the receiver will detect the signal through a luminosity sensor which is coupled with the microcontroller in the HS. The base element of this communication system is the pulse, which has a duration of **250ms**. A light pulse, meaning that during a pulse duration of 250ms, the torch light is lit and light is poured to the sensor, means that a '1' is being transmitted. On the other hand, if during a time pulse, the torch is turned off, or simply remains off without emitting no light to the sensor, it is considered as sending a '0'. With these basic rules, we can build our communication system which we will dissect in the following lines.

a) Main states protocols

As it has been explained before, for every state we have a unique binary code that both the sender and the receiver understand. Since we can switch between 6 different commands we will have a 4-bit binary code. Those 6 different states and their associated code sequences are presented below these lines:

STATE	BINARY CODE
OFF	1 0 0 0
ON	1 0 0 1
Synchronize Date	1 0 1 0
Schedule Days	1 1 0 0

Set Timing	1 0 1 1
Check HS	1 1 0 1

Table 3. Shows the codification of every functionality

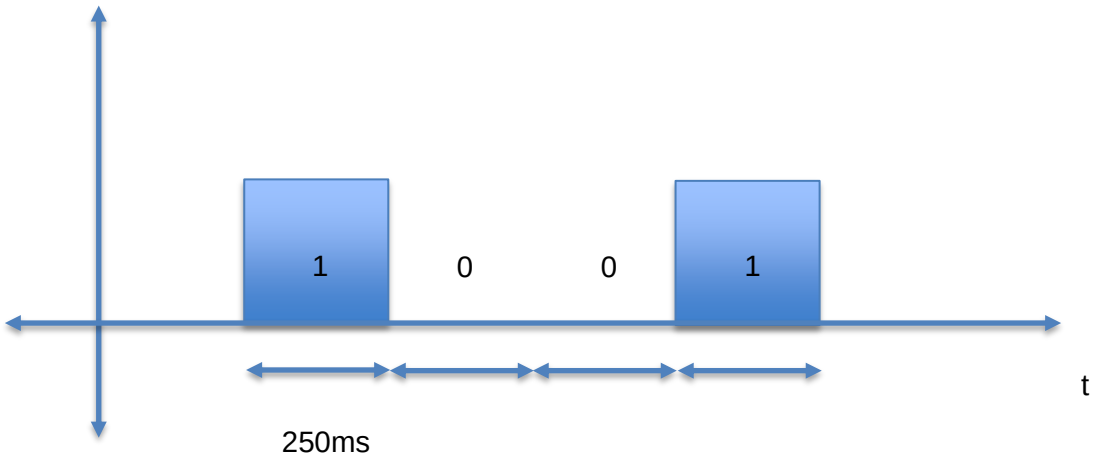


Fig. 19. Shows Switch ON sequence

b) Coding information

The other situation when information is transmitted from the application to the HS is when we want to synchronize the date from the application to the HS, when we set the start and end hour of the irrigation system and finally when we choose the days. From these cases, the first two share the same protocol and the case of the week days is the same containing only 7 samples. The information is transformed from decimal to binary obtaining a binary sequence formed by 32 bits. Before sending this sequence, an ignition signal formed by a pulse light of 250ms is appended prior to the sequence itself. This is done, as it has been explained in the page 30, because the HS's luminosity sensor is capturing data every **10ms**, and once a sample that overtakes the threshold of **50 lux** is detected, the microcontroller will change to a sampling mode following a pattern to detect the samples for each pulse. Since we are going to send a longer sequence of bits than the sequences explained in the previous point, we need to calibrate the sampling rate from the first sample detected as light. We want to detect a sample for each pulse trying to fall within the centre of each pulse. The first sample will be retrieved after **376ms** have passed since the sensor detected a 'light'. This is because we want to skip the first pulse and fall straight into the centre of the second pulse which is the first bit of the sequence of the total 32 bits. If we consider that the ignition signal is detected the first 20ms of the

pulse, then we wait an entire pulse length time, 250ms, plus half a pulse, 125ms, and we will probably retrieve a sample contained within the inner centre of the first pulse.

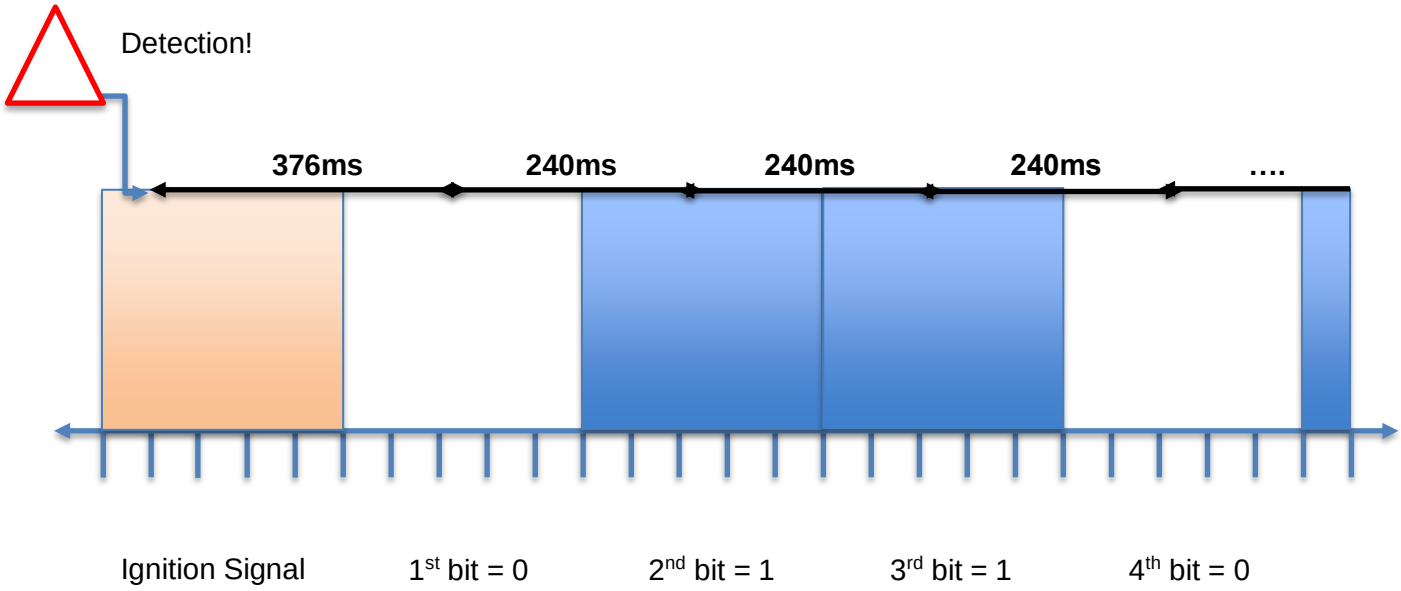


Fig. 20. Shows time lapses on the detection of the pulses

6. Results: uWater prototype

In order to complete the project delivery of our system we need to have a real and functional **HS** and a **smartphone application** that comply with all the requirements and features that have been explained so far.

First, we need to build the HS device. This first requirement is it needs to be **light,** and is achieved by using a light weight plastic box that measures 9x14.5x4 cm. We also need the box to be **isolated** from the external light conditions since we are working with a light-signal driven system which uses a luminosity sensor, and is sensible to light environment conditions. In that sense, the box is recovered with a black vinyl tape which acts as isolator. A narrow area of the top surface is left without tape cover so that the camera and torch light of the smartphone have a clear access to the inner LEDs and sensor. The HS has on the front side surface **1 green LED**, **1 red LED** and **2 blue LEDs**. On this same surface, a hole practised to its surface gives access to the microcontroller's USB connector. This USB can be plugged directly to the mains giving power to the HS through a typical USB smartphone plug charger like the iPhone one. The device can also be

opened and closed at will, which gives access to the electronic components. For the moment, the HS prototype needs to be connected to a laptop with the Arduino software so that we see the output through the console logs. As said before a future work would include a small LED screen and also a battery to make the system portable. In the *Annex,* in the results section, a set of pictures are depicted showing how our HS looks like.

6.1. uWater Test software

In order to determine the robustness and flaws of our communication system we need to submit this prototype to a battery of tests. This part is a key piece of our project since it will provide the information we need to later on improve our system. The start point is that depending on what is the closeness between the luminosity sensor and the mobile's torch light, the caption of the transmitted signal will succeed or will fail. The reason for this to happen is due to the phenomenon of glare or overload of light, where the amount of light irradiated towards the sensor makes it impossible to distinguish between the pulses containing light and the lightless ones. This is why we need to make a study of the correct boundaries or range where our system behaves correctly and to know which are these boundaries to take them into account in the design improving process of the prototype for the HS. For this purpose, a specific software for both mobile and the microcontroller, have been designed and implemented. Next, we are going to briefly present and explain both.

6.1.1. uWater smartphone test app

This testing application is really simple in design and takes some of the core functionalities of the uWater application and applies them into a more specific role which is to trigger the battery of tests. What we want to submit under test is the correct function of our communication protocol. To do so we have designed an application with a single view containing an action button in the centre, which by taping on it triggers a method contained in the very same *ViewController* called *StartTest*. This method contains a *for* loop of 100 times and inside this loop *flashButtonSendSequence* method is called, which will be explained on the following lines. This method is the one in charge of accessing the torch light, generating the sequence to send and finally sending it. Since we want to send a sequence of 8 digits and we want to repeat this process a hundred times we need to generate a simple and easy to do sequence. That's why we decided to go for the following sequence: 0 1 2 3 4 5 6 7

For every sequence, an ignition signal is triggered before the sequence itself prior to the sequence.

6.1.2. uWater HS test software

In this case, the approach is similar to what we did to the mobile test application, taking the parts already created from the uWater HS final software and applying them into this more specific task. What we want is to detect, store and decode the sequences that the mobile will be sending to the Arduino over and over again. Once the sequence is decoded, the result will be prompted to the console indicating the lux level that was detected at first, the number of the sequence detected and the decoded message detected. An additional variable is added to count the number of correct detections in order to calculate the percentage of success of the battery of 100 tests launched.

Here we show some pictures of how the prototype of the HS device looks like.

Fig. 21. The picture above showcases the Home System with the USB plugged.

Fig. 22. Front side of the HS containing the four LEDs.

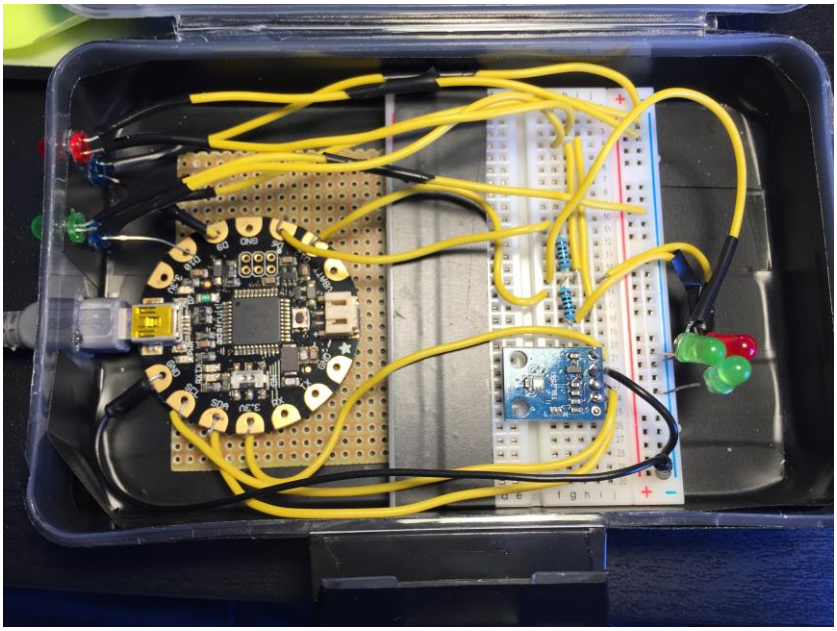


Fig. 23. Detail of the circuit inside the HS.

Fig. 24. Top side of the HS with detail of the access area.

Fig. 25. Detail of the interaction between the smartphone and the HS.

6.2. Test Results

A set of 10 different tests, each one of them containing 100 sequences that are transmitted and decoded, have shone some results which we are going to proceed to present here. We have two actors in our system, the smartphone acting as the sender, and the microcontroller-sensor system as the receiver. The first actor here, the smartphone, uses the torch light to communicate, by means of light pulses, a message which is codified in binary where the time length of every pulse is **250ms.** The reason to take this measure is due to an iterative process of try and failure. Tests showed that with this value, the percentage of successful transmission of data was almost of 100%. Trying to reduce this time led to no success so we decided to go with the 250ms. The second actor, the microcontroller-sensor system, is configured to retrieve information with the lower resolution possible which translates also in a greater conversion velocity. Another parameter which is key here as well is the sample rate. With regard on this matter, the approach that was taken is widely explained and detailed in the *Communication Protocol* section, and the values taken for the sample rate are driven by our approach over the system. Then, which factors play an important role on our system? Basically, the relative position among the two parts of the system and the environmental conditions. These

tests are very important once the design and building of the device is done, because it will help us understand what improvements and adjustments can be done. The difference between each test configuration was given by these two factors. The configuration of the elements playing a role in this test is the following. The HS device laying on a table and the smartphone on top of it in the correct position. Figure 25 shows this configuration. From these tests, here there are the conclusions extracted:

1. **Consistency:**

The first result we can extract from our tests is that given a certain configuration on how the two actors are located under a given environment conditions, the results will be consistent all along the test run. This can be extracted from the fact that for our set of 10 battery test, all of them where 100% successful, except one being 99% which we can almost consider it also a 100%. This means that regardless of the environment light conditions or the position of the smartphone on the top surface of the HS box, the transmitted messages will be well decoded with a 100% of certainty. Such a cohesion in the results give us the impression of the robustness and reliability of the system.

2. **Boundary values:**

In order to learn what are the constraints of our system we are going to analyse the overall outcome of the test results. For each test, we are going to pay more emphasis to the lowest and highest luminosity values detected on the ignition signal. This will give us a first glance of the boundaries for each set of tests, and in the end, we will extract the conclusions. On *TestPrototype1* all the sequences where decoded successfully. The min. lux value is **51 lux** and the max. value is **215 lux**. The most repeated lux value detected is also **215 lux** with a total of 31 repetitions. On *TestPrototype2* the min. lux value was **55 lux,** the max. was **283 lux** and the whole set of sequences were well detected and decoded. On *TestPrototype3* the min. luminosity value detected is **51 lux**, and the max. is **580 lux**. Although the lowest value is just one point above the threshold, we can see that the higher one increases considerably compared to the two previous examples. In this test, the desired situation was to be as closer one element to the other as we could, while the transmitted signals were still able to be decoded. On the *TesPrototype4* we got **53 lux** as the minimum value, **131 lux** the maximum value, and all of the sequences analysed were successfully decoded. For *TestPrototype5,* the minimum luminosity value is **52 lux** and the maximum is **387 lux** with a total amount of 56 repetitions. Taking the values for the five first different set of test batteries, we have that the lower luminosity value of the ignition signal that has been decode is **51 lux,** and for the opposite case, we have **580**

lux. It is important to understand that the first value that is detected is really important since it will be the moment from where the other ones will be sampled following communication protocol. These 5 different test batteries were set changing the position of the smartphone on top of the HS device in order to get different lectures and hence extract some conclusions. For the following battery of tests, we changed the environmental light conditions by switching on more lights on the room and at the same time carrying the position of the device's torch light as well. The first one of these was *TestPrototype6* where the min. was **51 lux** and the max. **232 lux**. We can see here that the values are very similar to those that were obtained with the room with no light at all. On *TestPrototype7*, min. value was **60 lux**, and a max. **235 lux**. Apart from the lower luminosity value being slightly higher compared to the ones we have been detecting, there is also one value that failed and gave a result of *01234564* instead the desired sequence *01234567*. Our guess is that the smartphone locked itself due to inactivity and thus this sequence failed. For the left battery of tests, we wanted to check another scenario where the HS was under a daylight environment on open air. The first thing we noticed was that with the first adapted threshold level of 50 lux, the HS started detecting samples as if they belonged to a message being transmitted. This is because the environment light was above the threshold we set before of **50 lux**. The solution was to slightly level it up again, to a value of **175 lux**. With this *threshold* value, the following set of battery tests were performed. On *TestPrototype8* the min. is **176 lux**, and max. is **327 lux**. The percentage of successful decoded sequences was of 100%. Rising the level of the threshold didn't affect at the communication protocol at all under daylight conditions. Next step was to take our HS back to a dark room and perform again test it with the new level of **175 lux.** For *TestPrototype9* a 100% of success is achieved and min. value is **184 lux** and max. is **213 lux**. Finally, in *TestPrototype10* we submit the sensor to an environment with medium luminosity levels, on a room with natural daylight but not in the open air. The lower value was **176 lux**, meanwhile the highest one was **251 lux**, this last one with 59 repetitions. In this case the percentage of success fully decoded sequences was again of 100%.

We can extract some conclusions here. First is that the we can assure that the system communication protocol is **consistent**. The second conclusion we can take is that the boundaries of our communication system are limited by the lower threshold of detection which is set by us to **175 lux**, and the higher boundary would be around **600 lux**, when submitted to a direct light of the torch light.

7. Budget

- List of the elements of the elements used for the prototype and the cost:

a) Arduino - Adafruit FLORA microcontroller - 12,51€
b) TSL2561 digital luminosity/lux/light sensor - 12,43€
c) 2 x Blue LED - 0,186€
d) 2 x Red LED - 0,184€
e) 2 x Green LED - 0,180€
f) Protoboard 400 contacts Breadboard - 1,41€

- Design and prototyping costs

The main tasks for the assembly of the prototype are:

- Mount and connect the different parts of the electronical circuit
- Encapsulate it inside an already isolated box, like this we prevent the task of isolating it with the black vinyl tape.

An estimation time for the assembly of the prototype is less than thirty minutes for a human being. This, having the instructions and all the material ready and prepared, meaning that every cable and piece is numbered and has the correct length, value and size in order to facilitate the mounting of the prototype. Taking into account that the average working hours is 8, it means that in a work day a total of approximately 16 devices could be mounted and ready to be packaged and delivered by one person.

- Financial viability

Financial viability is concerned with private profitability and is based on financial flows which relate across several dimensions:

- Technical feasibility. The project can be implemented as planned, using the proven technologies and without reasonable technical risks.
- Legal feasibility: There exist no legal barriers with the kick off of this project.
- Environmental and social sustainability: The project complies with all environmental standards.

8. Conclusions and future development

The main objective of this project, which conceived the achievement of a fully functional communication system between two actors, has been achieved. The system that had been designed, developed and finally assembled works as it was intended to do from the initial design. The result of this project is a real prove of concept of an irrigation system, in the form of a prototype, which offers a **reliable** and **affordable** solution to the consumers.

The conclusion of the project is due to a **multidisciplinary** application of different methodologies. They include the design and development of a smartphone application and a microcontroller, applying different techniques such as **coding** in different languages, **caption** and **analysis** of an **image** and **data** from a sensor, or mounting and assembling of an **electronic circuit**. By merging all of these different disciplines, the succeed is translated on the final prototype and the knowledge gained in the process.

The **communication protocol** used for the transference of data results to be **effective**, proving that by means of resources which are available to everyone, such as the torch light from a smartphone, it is possible to share information. Although for the moment it is mainly **unidirectional**, a future work on a **bidirectional** communication approach would add more accuracy and reliability to the system.

Nevertheless, the system can be improved in many ways by adding more **features**, scaling it for a **multiplatform** mobile use, **increasing the data rate** and thus **decreasing the waiting times**, enabling the **bidirectional** communication, or substituting the current hardware pieces used in the prototype like the Arduino or the luminosity sensor, by less expensive ones such as a simple microcontroller or a LED acting as a receiver. This last action would even help to decrease the production costs and the consumer price. Also, a further work would be to include a **LED screen** in the HS system in order to visualize the operations performed. This indispensable requirement is also of great importance if the system wants to be commercialized.

9. References

[1] Kil-Sung Park, Sun-Hyung Kim, Young-chang Kang. "Development of transceiver using Flashlight and camera in VLWC". *Advanced Science Technology Letters Vol.77*, UNESST 2014, Republic of Korea. pp. 23-32. doi: 10.14257

[2] Developing iOS Apps[Online] Available: https://developer.apple.com/library/content/referencelibrary/GettingStarted/DevelopiOSAppsSwift/index.html [Accessed: February 2017- October 2017].

10. Annex

- **Functional Design Specification Overview**

The following tables are related what it is explained in the functional design section, describing the functional design for each of the different Graphical User Interface views that the uWater application shall have, such as buttons, texts and their locations.

GUI 2.1 Figure 4: Main Menu Screen – General Requirements	
GUI 2.1.1	Screen name is Main Menu
GUI 2.1.2	Contains 5 action buttons
GUI 2.1.2.1	Action button with text *Switch Off.* The button is located in the left top corner of the view and it is smaller than the main action buttons. This action will send a light sequence and instantiate the GUI 1.1
GUI 2.1.2.2	Action button with text *Synchronize Calendar*. Located on the top column of buttons. This button, together with the other three main action buttons is bigger than the first one. This action will send a light sequence and instantiate another GUI containing the synchronize date view.
GUI 2.1.2.3	Action button with text *Schedule Days*. Located below the previous button. This action will send a light sequence and instantiate another GUI containing the scheduling days view.
GUI 2.1.2.4	Action button with text *Set Timing*. Located below the previous button. This action will instantiate another GUI containing the setting of time view.
GUI 2.1.2.5	Action button with text *Check HS*. Located below the previous button. This action will send a light sequence.

Table 4. Requirements for Main Menu screen

For the first three different modalities that the application gives to the user, a different GUI needs to be implemented following distinct approaches.

GUI 3.1 Figure 5: Synchronize Calendar Screen – General Requirements	
GUI 3.1.1	Screen name is Launch Screen
GUI 3.1.2	Contains 2 action button and 1 label text
GUI 3.1.2.1	Action button with text: *Main Menu* Located in the top left corner of the view, redirects the user to the GUI 2
GUI 3.1.2.2	Label with the current day text. Located in the centre of the GUI.
GUI 3.1.2.3	Action button with text: *Synchronize Date* Located on the bottom centre of the GUI. This action will send a light sequence and instantiate afterwards GUI 2

Table 5. Requirements for Synchronize Calendar screen

GUI 4.1 Figure 6: Schedule Days Screen – General Requirements	
GUI 4.1.1	Screen name is Schedule Days
GUI 4.1.2	Contains 9 action button and 1 label text
GUI 4.1.2.1	Label with the text: *Select Irrigation days*. Located in the top centre of the GUI.
GUI 4.1.2.2	Action button with text: *Main Menu* Located in the top left corner of the view, redirects the user to the GUI 2
GUI 4.1.2.3	Action button with text: *Program Days.* Located on the bottom centre of the GUI. This action will send a light sequence and instantiate afterwards GUI 2
GUI 4.1.2.4	7 different action buttons with the text for each day of the week Located on the centre of the GUI below the label text. Each button action selects the day by tapping on it, and stores it to send it later with the action button 4.1.2.3

Table 6. Requirements for Schedule Days screen

GUI 4.1 Figure 7: Set time Screen– General Requirements	
GUI 5.1.1	Screen name is Schedule Days
GUI 5.1.2	Contains 2 action button, 2 label texts and 2 roulette time picker items
GUI 5.1.2.1	Action button with text: *Main Menu* Located in the top left corner of the view, redirects the user to the GUI 2
GUI 5.1.2.2	Label with the text: *Start time*. Located in the top centre of the GUI.
GUI 5.1.2.3	A roulette 4-digit picker Located below the previous label
GUI 5.1.2.4	Label with the text: *End time*. Located in the centre of the GUI below the previous picker
GUI 5.1.2.5	A roulette 4-digit picker Located below the previous label
GUI 5.1.2.6	Action button with text: *Program Selected Time* in the bottom centre of the view. This action will send a light sequence and instantiate afterwards GUI 2

Table 7. Requirements for Main Menu screen

The tables below are the rest of the CDC belonging to the classes used on the smartphone uWater app.

ProgramViewController	
send flash sequence to Switch Off	ViewController DayViewController TimePickerViewController DateViewController

Table 8. CRC of ProgramViewController

DayViewController	
selection of days go back to main menu send flash sequence to program data	ProgramViewController

Table 8. CRC of DayViewController

TimePickerViewController	
selection of time go back to main menu send flash sequence to program data	ProgramViewController

Table 9. CRC of TimePickerViewController

DateViewController	
go back to main menu send flash sequence to program data	ProgramViewController

Table 10. CRC of DateViewController

- **Circuit & flow diagrams**

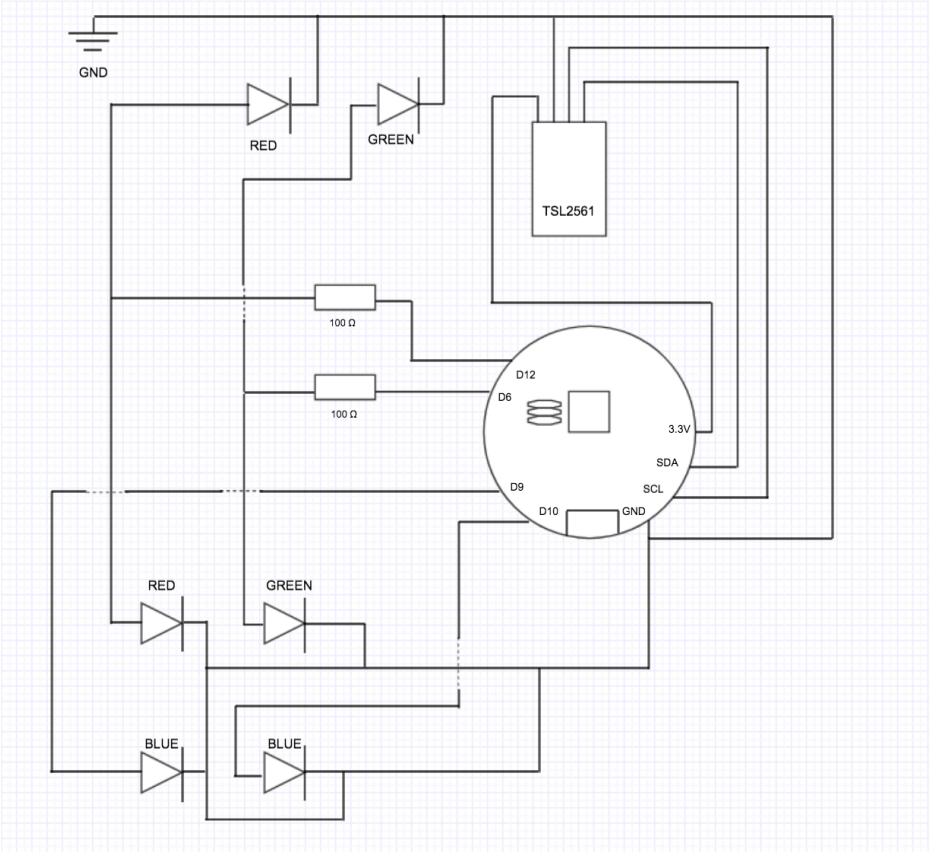
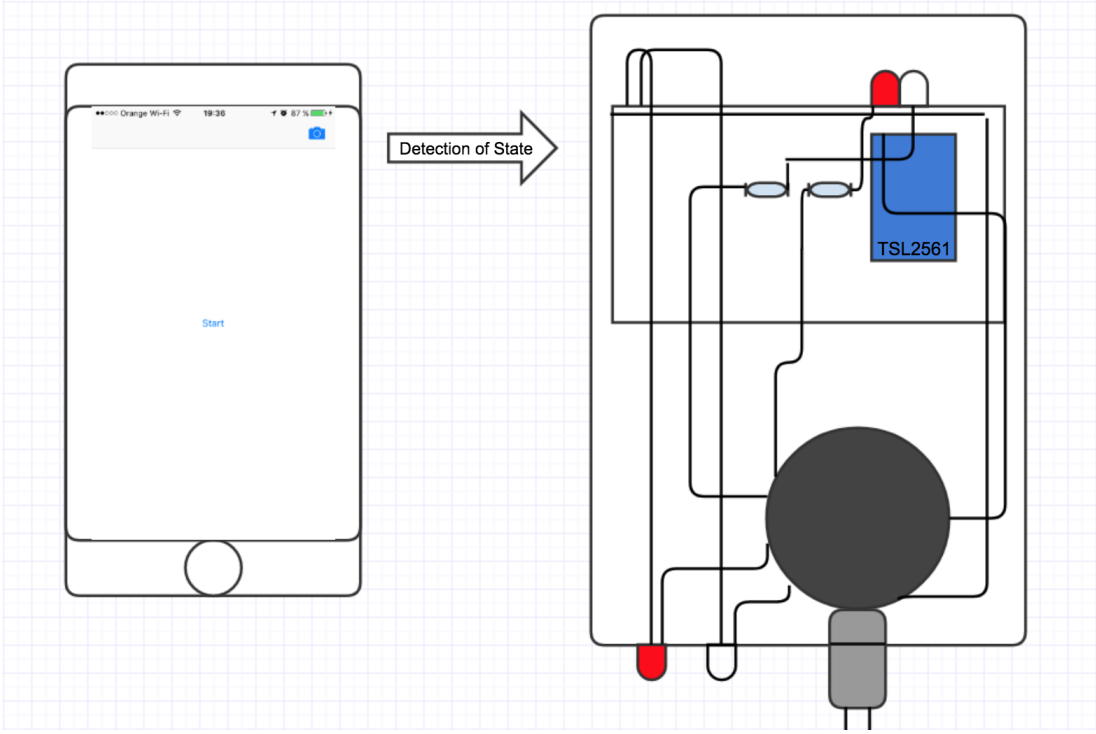
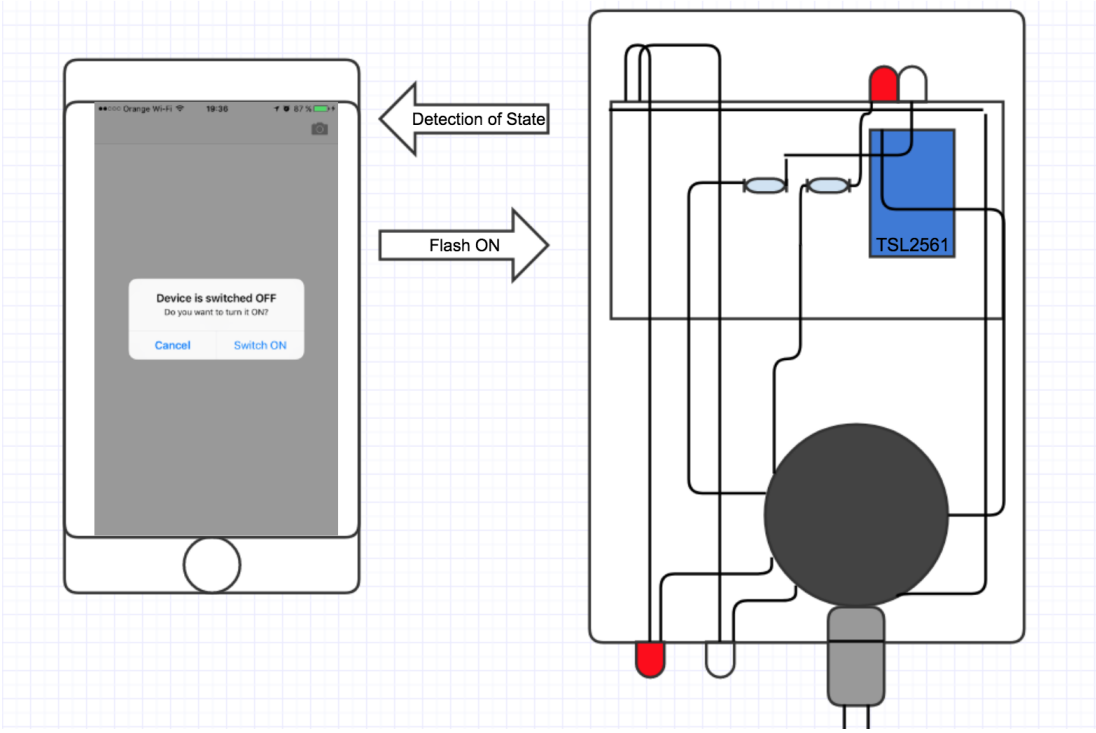


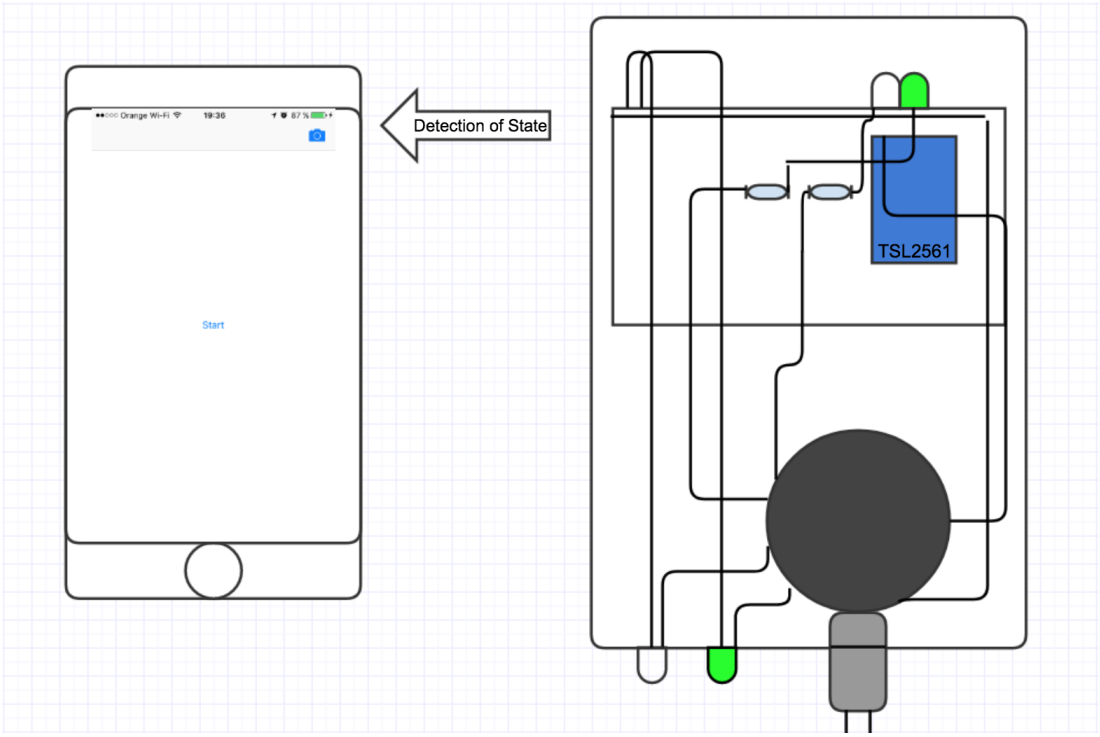
Fig. 26. Electrical circuit of the HS

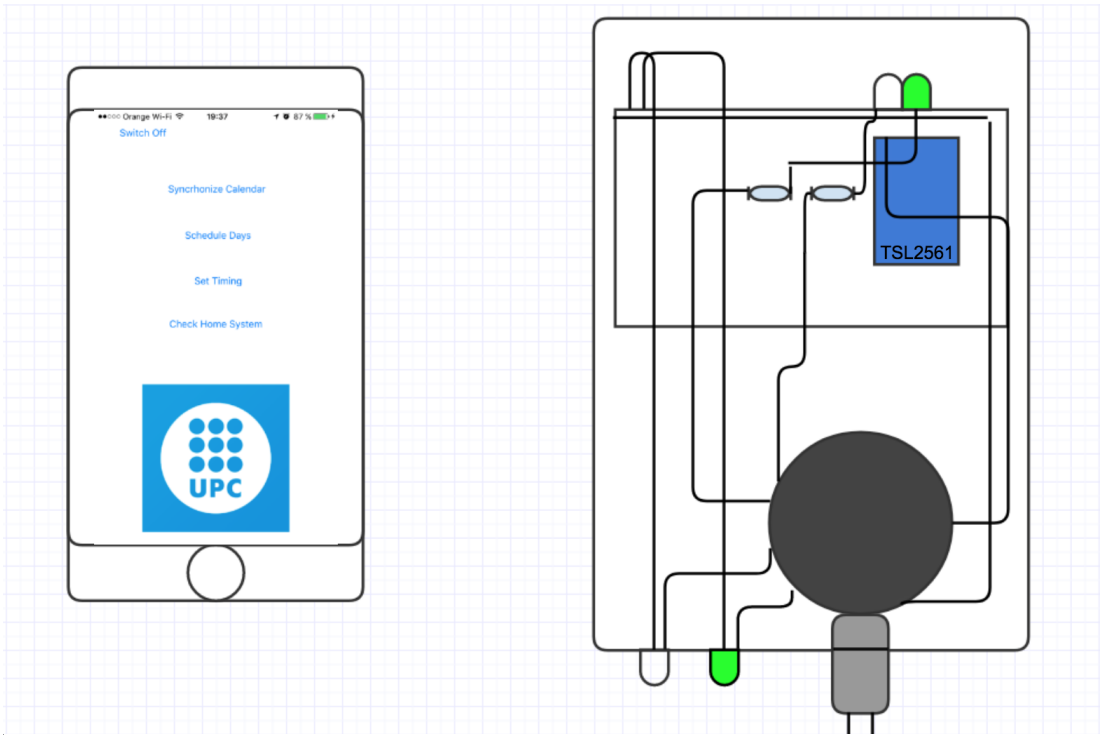
Diagram flows at launch of the application

a) HS is OFF. Detection, Switch ON command and successful switch.

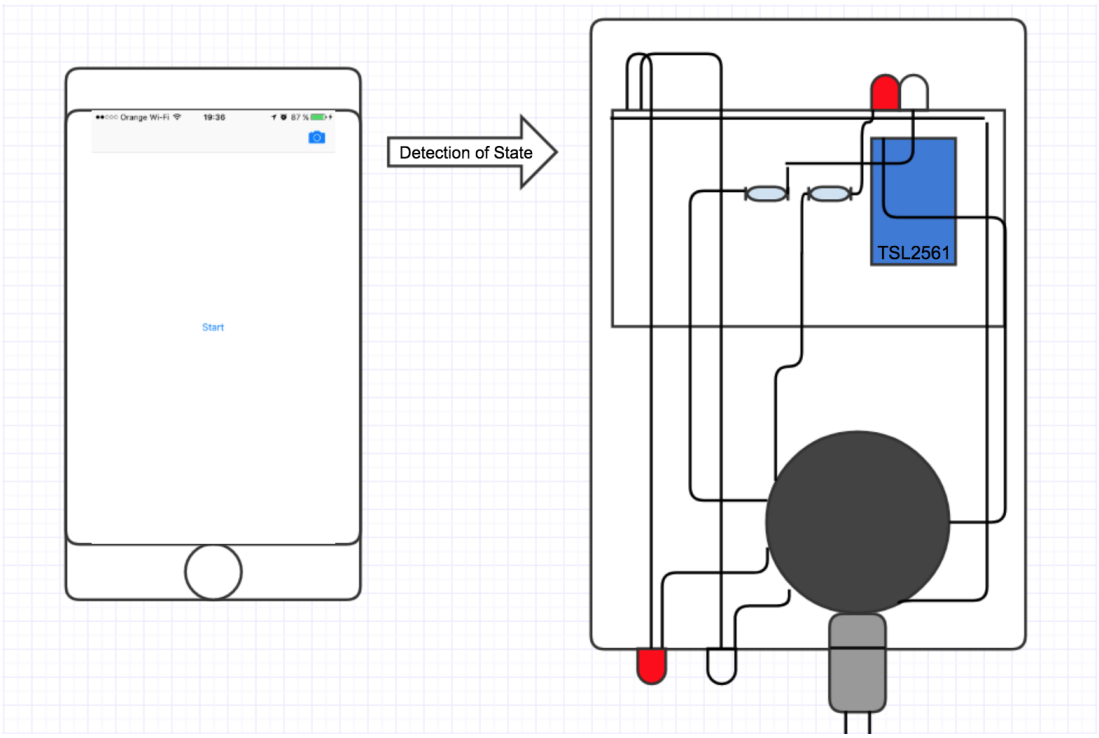


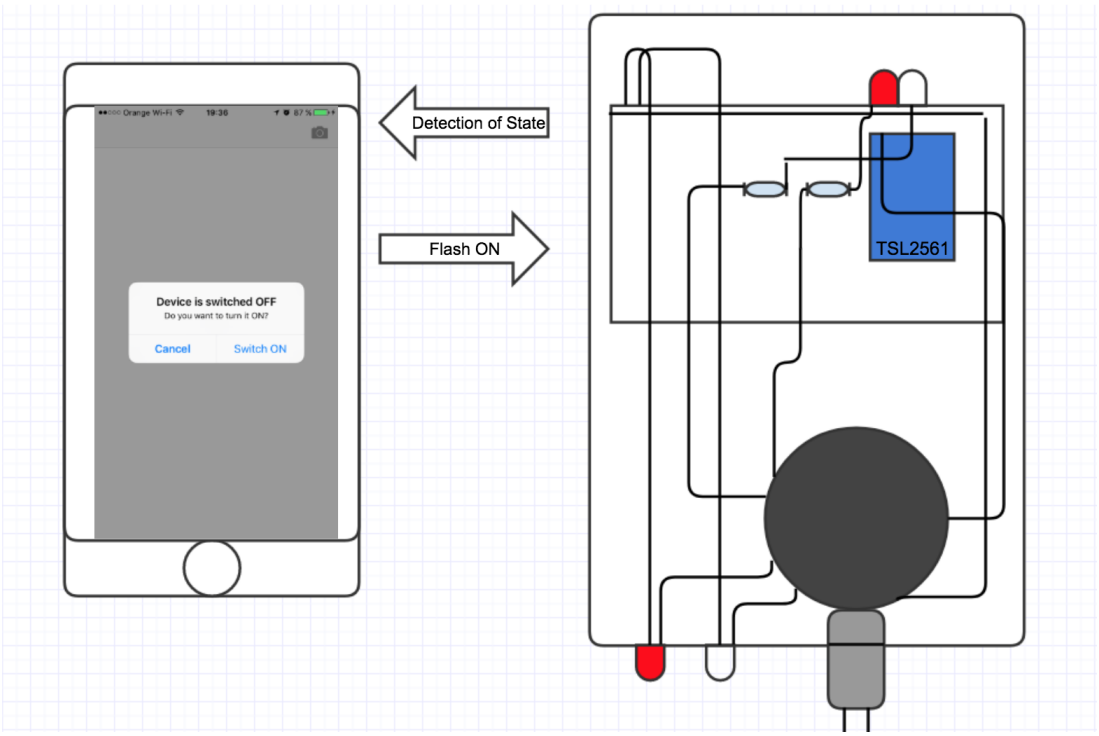


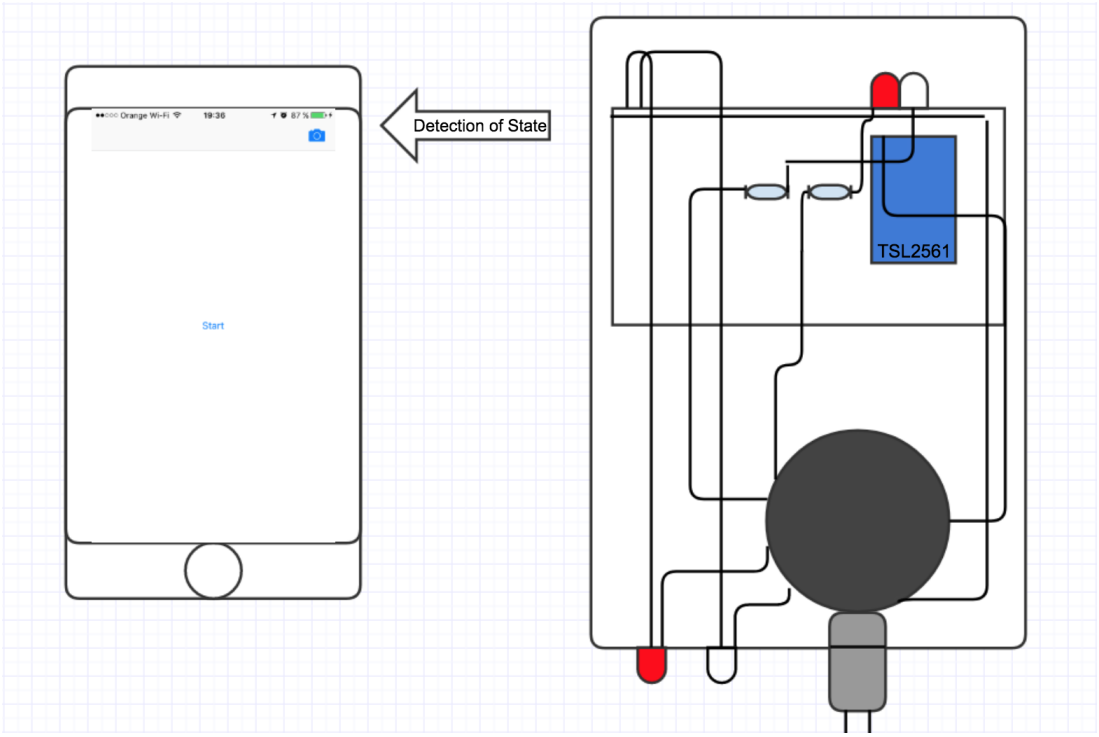

Orange Wi-Fi 19:36 87 %
Start
Detection of State
TSL2561

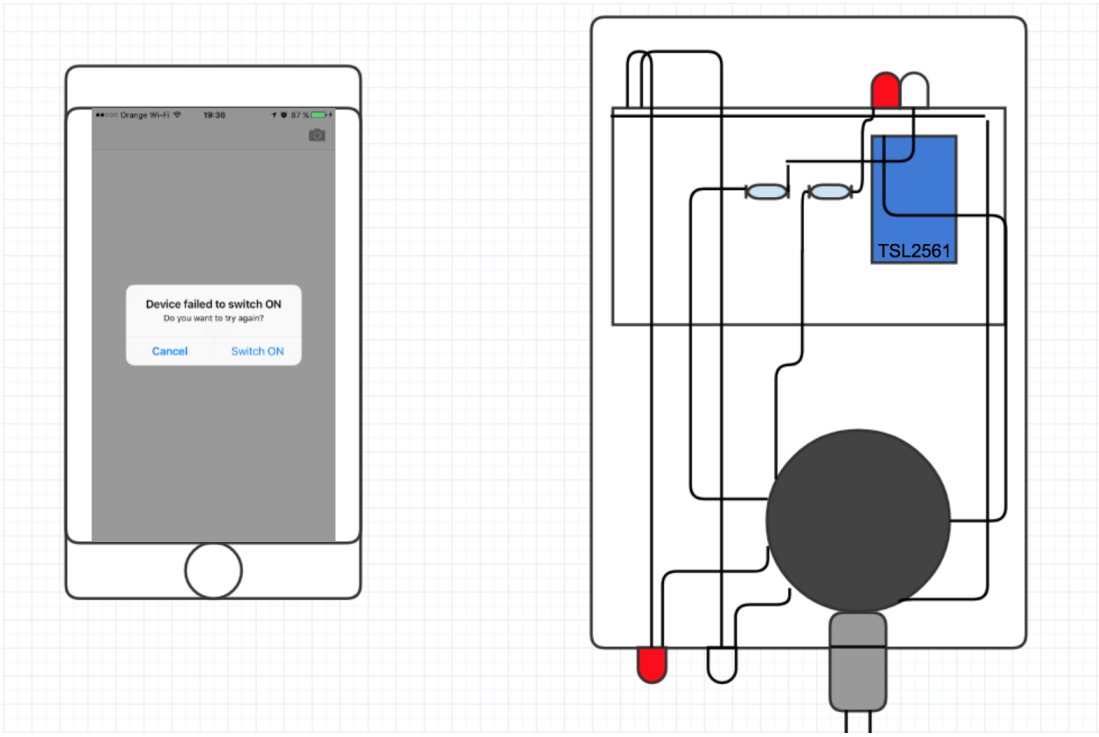

Orange Wi-Fi 19:37 87 %
Switch Off
Syncrhonize Calendar
Schedule Days
Set Timing
Check Home System
UPC
TSL2561

b) HS is OFF. Detection, Switch ON command and unsuccessful switch.

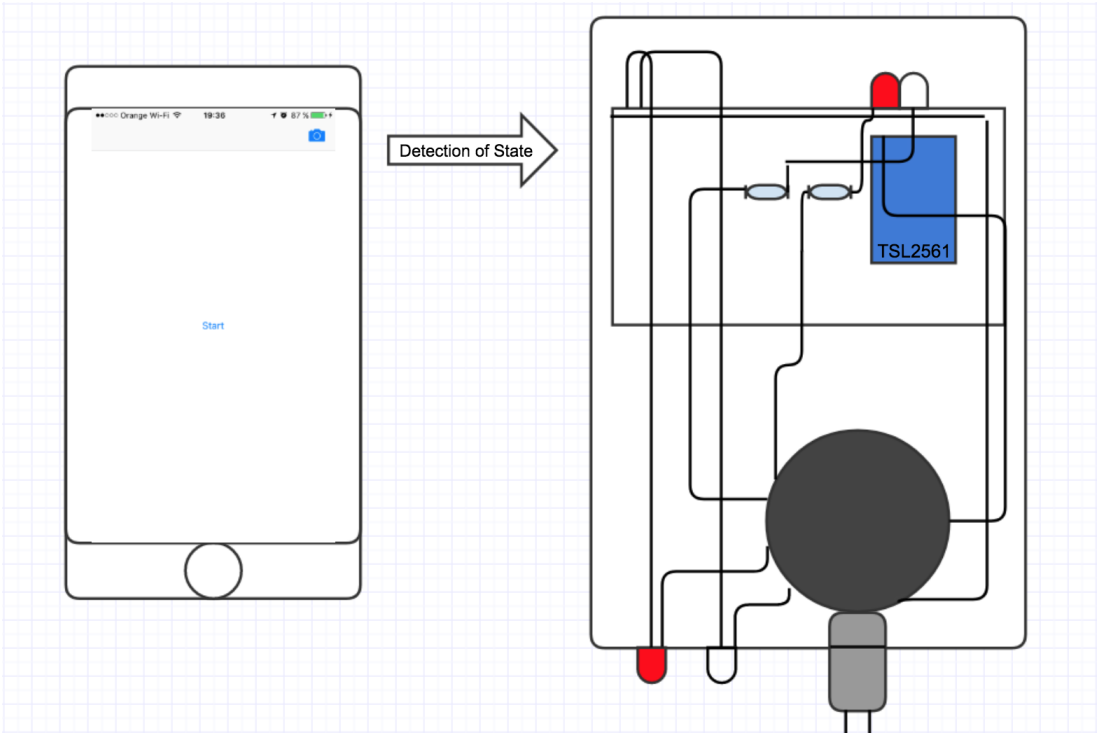


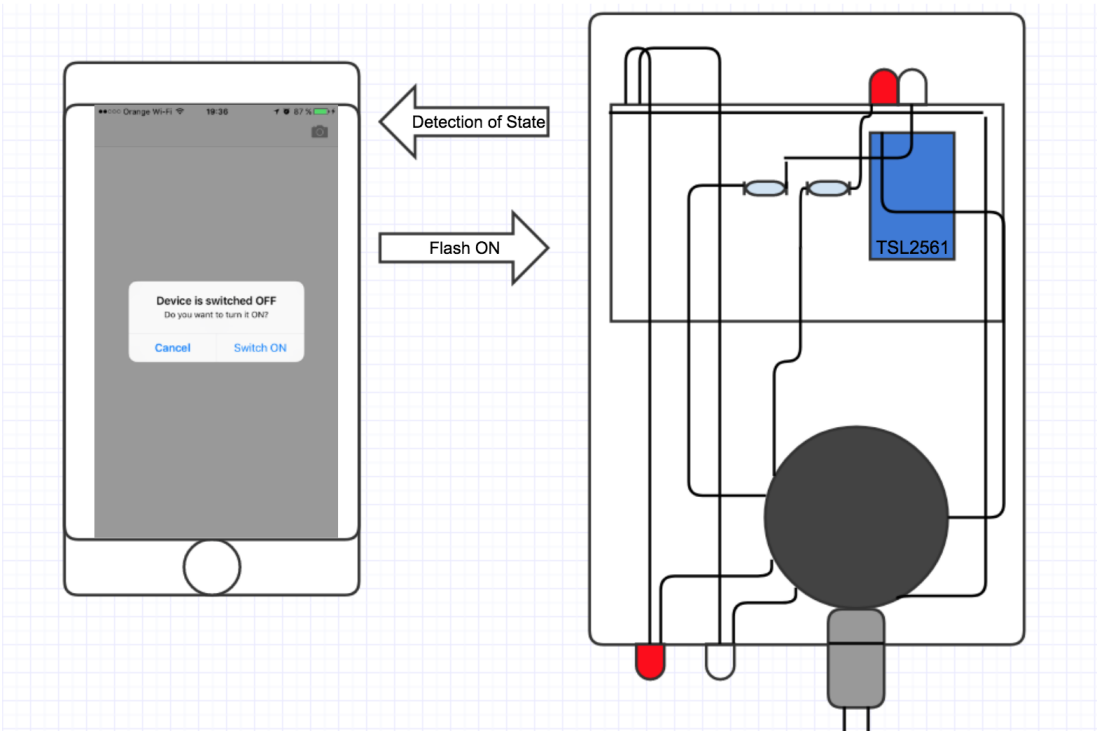


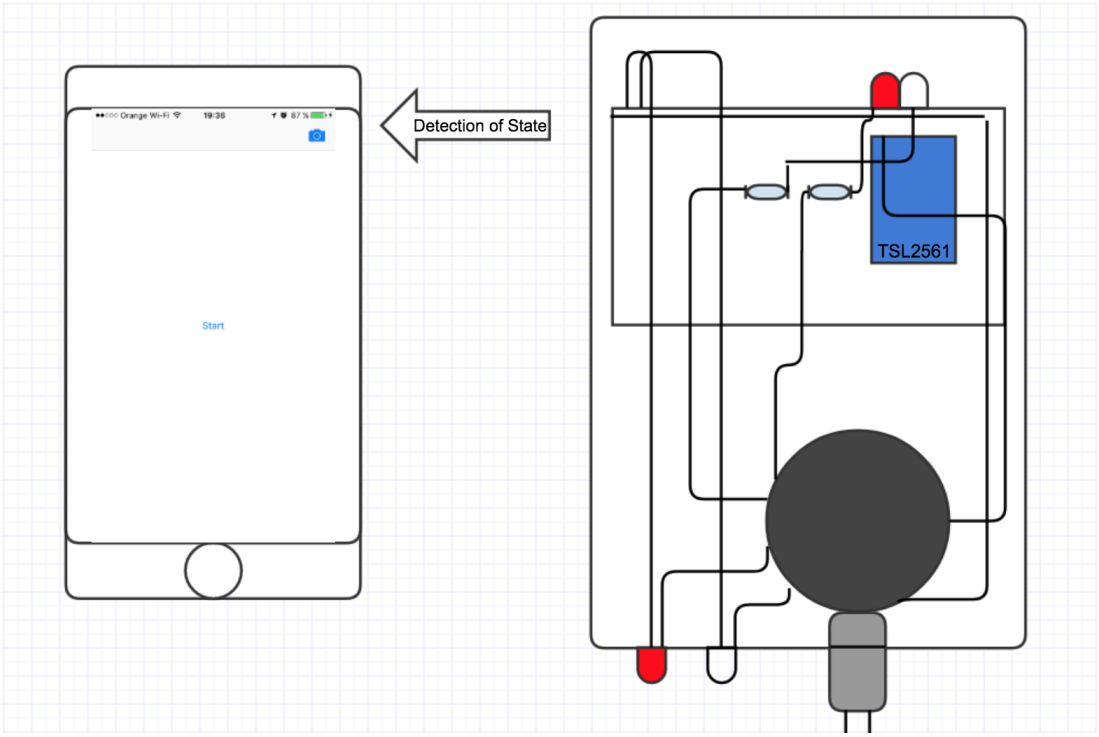

Orange Wi-Fi 19:36 87 %
Start
Detection of State
TSL2561

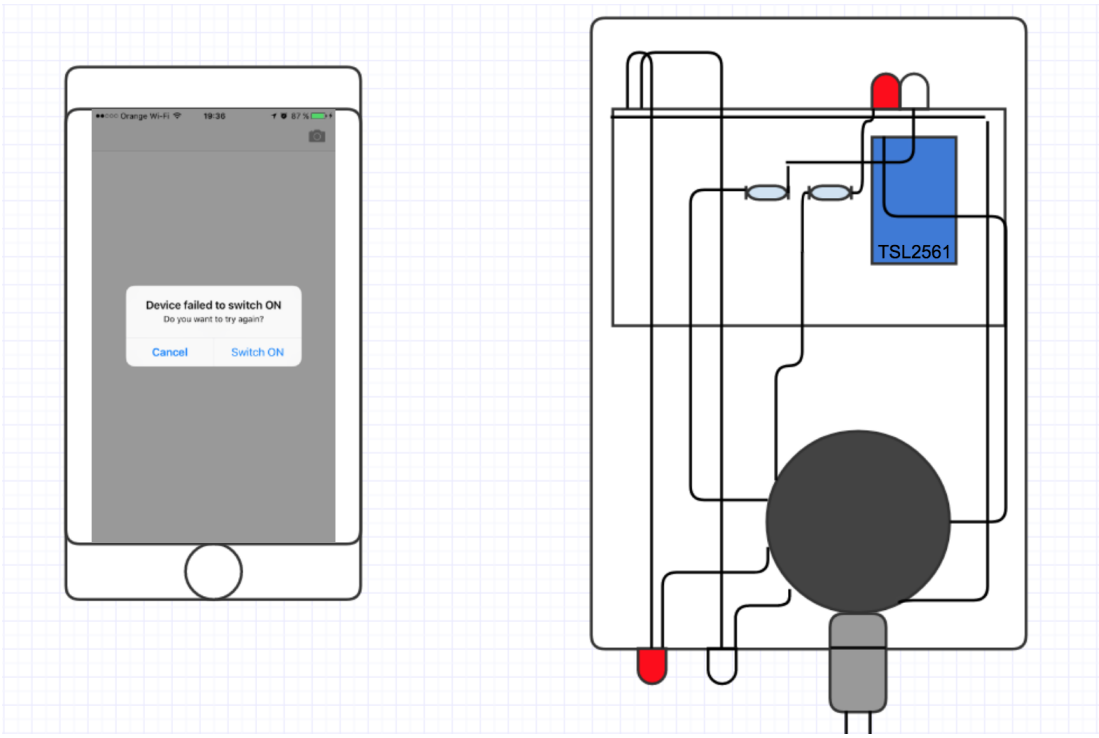

Orange Wi-Fi 19:36 87 %
Device failed to switch ON
Do you want to try again?
Cancel
Switch ON
TSL2561

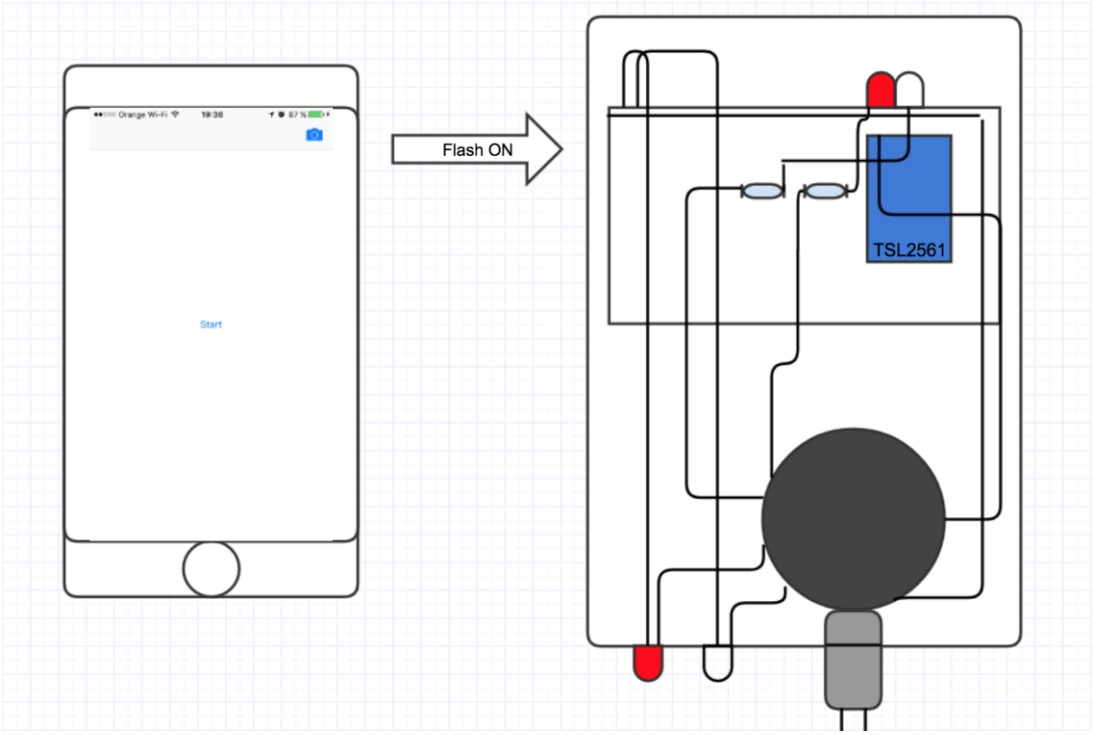
c) HS is OFF. Detection, Switch ON command, unsuccessful switch and successful retrial.

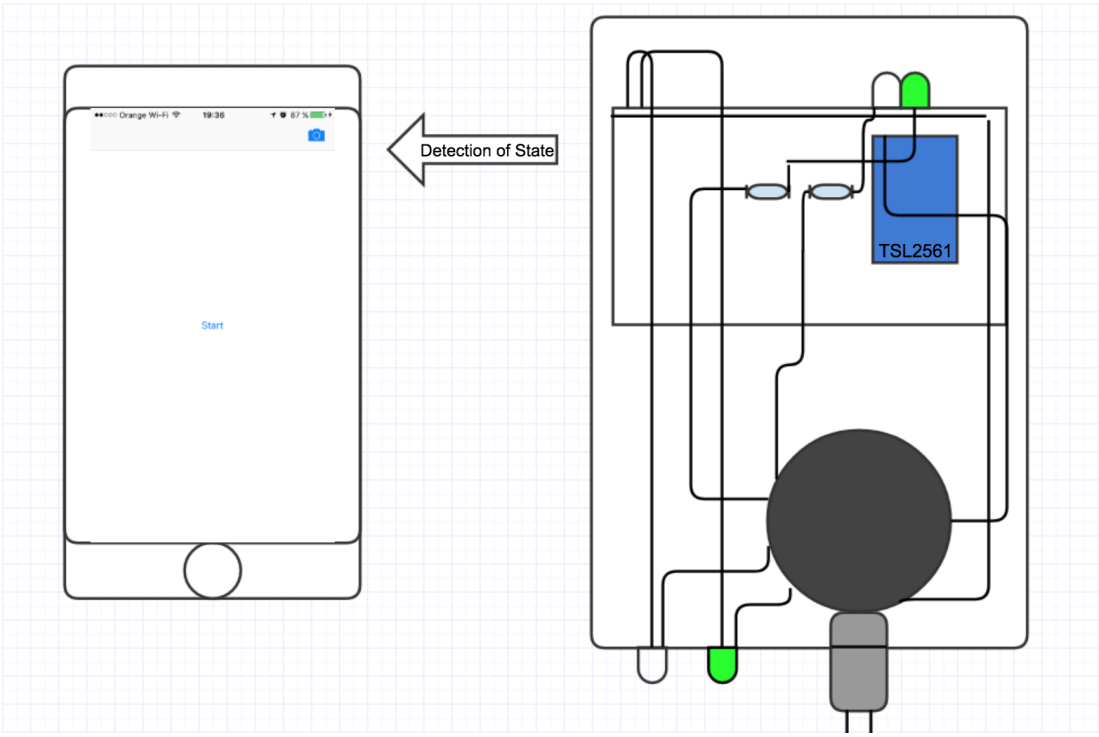


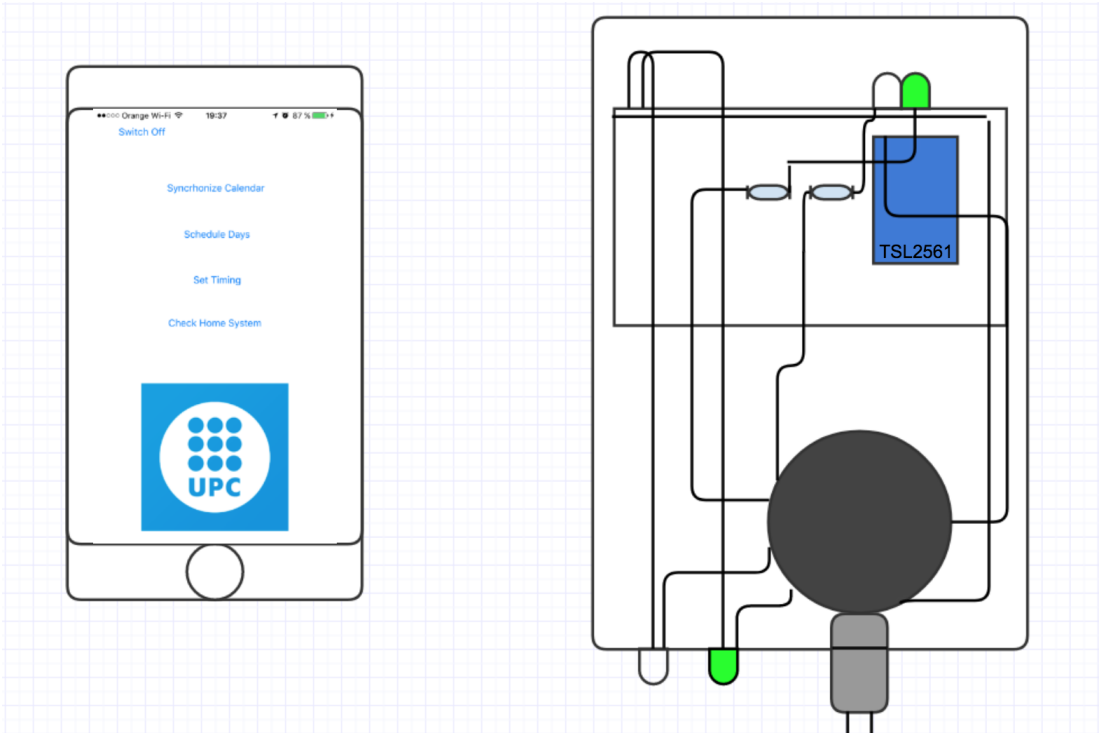



Start
Detection of State
TSL2561

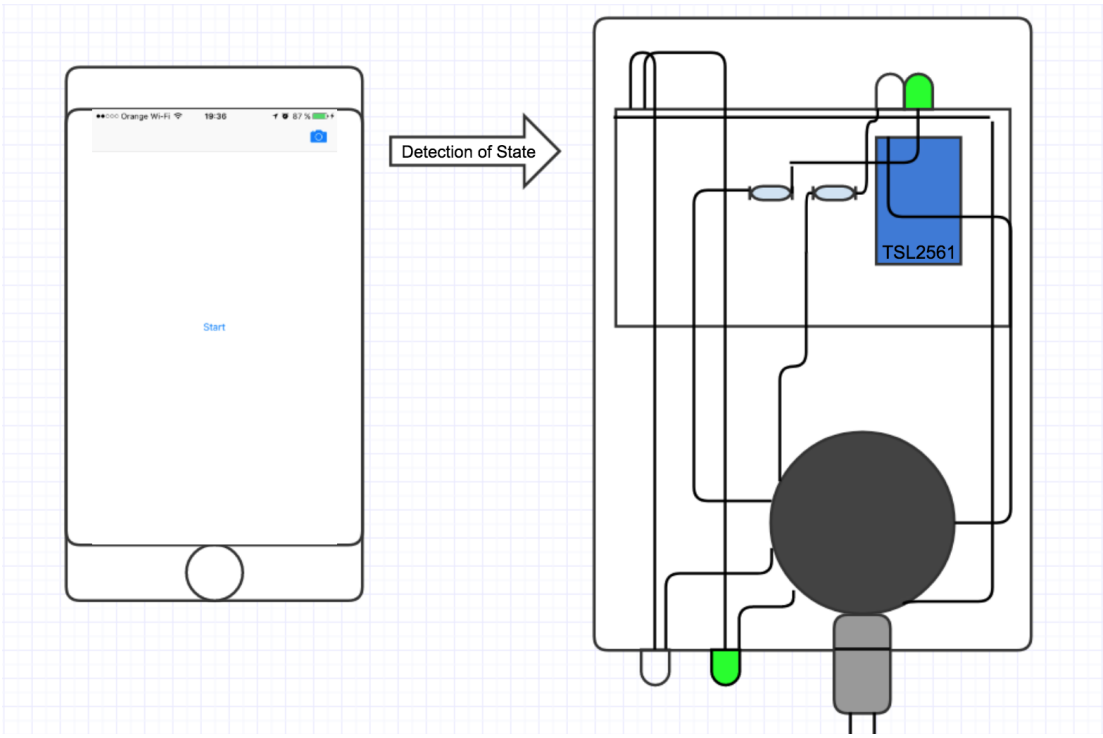

Device failed to switch ON
Do you want to try again?
Cancel
Switch ON
TSL2561

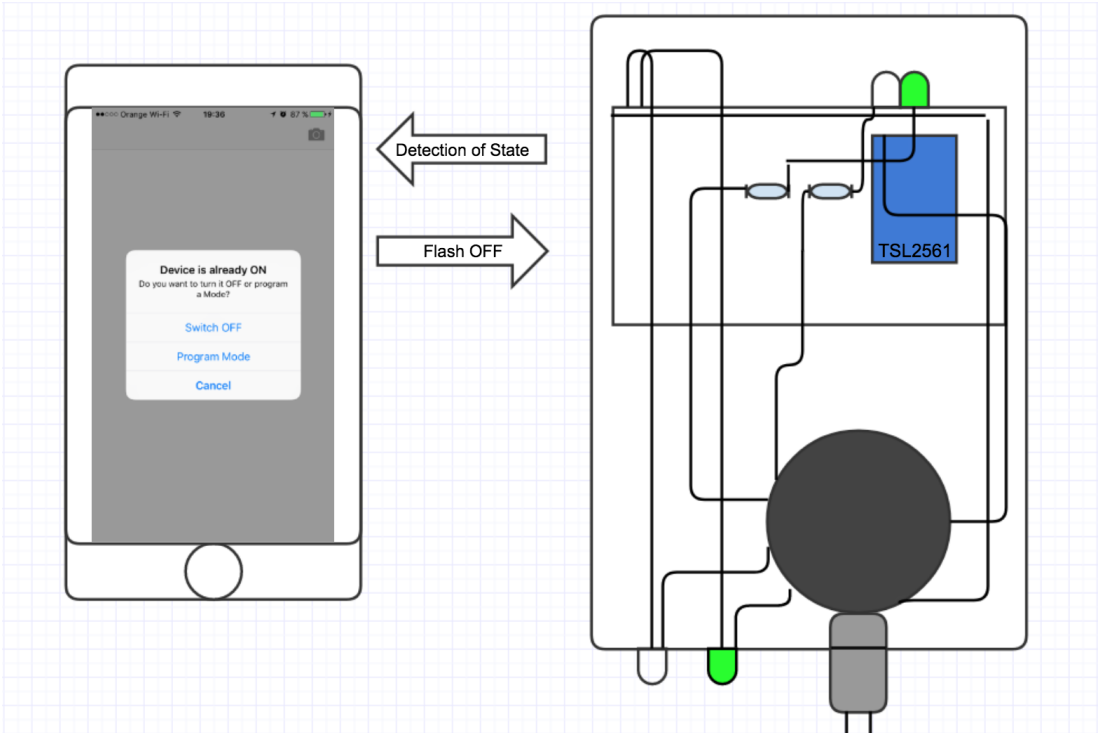

Start
Flash ON
TSL2561

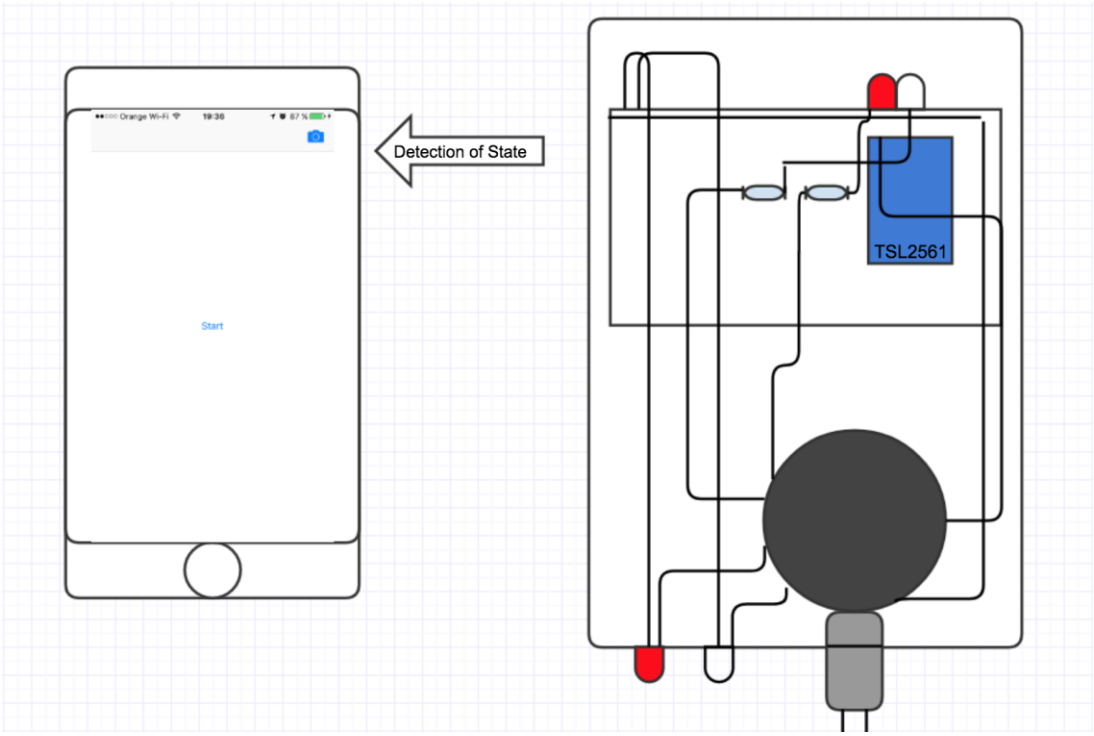

Start
Detection of State
TSL2561

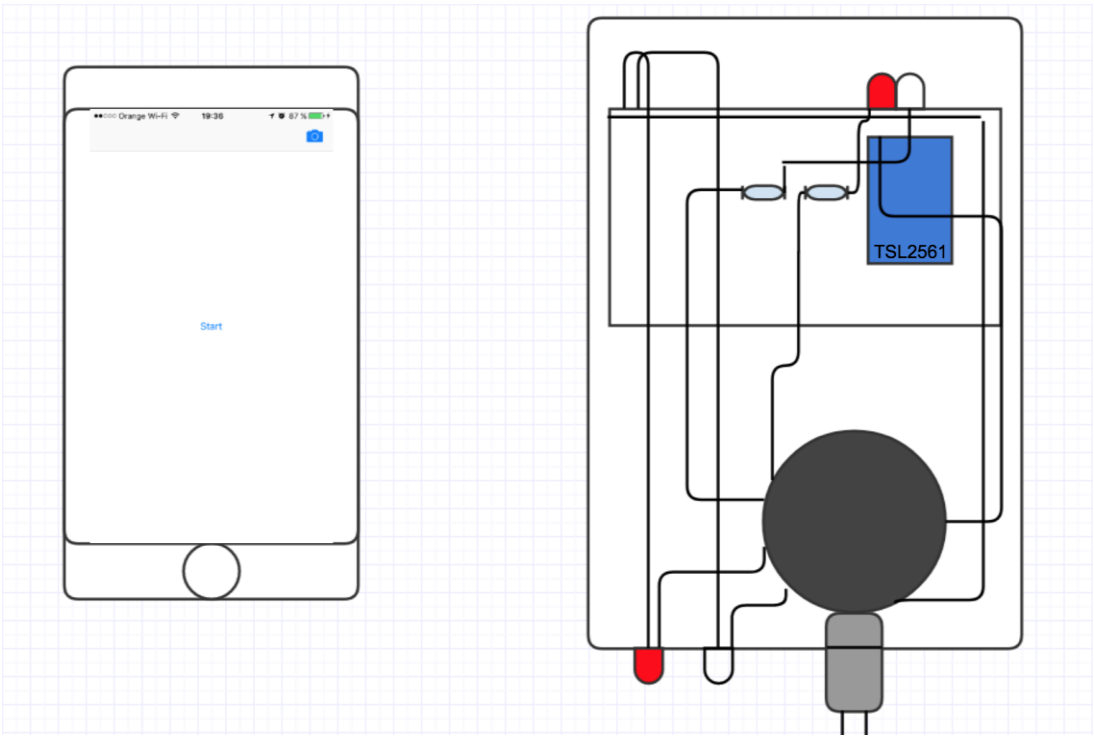


d) HS is ON. Detection, Switch OFF command and successful switch.

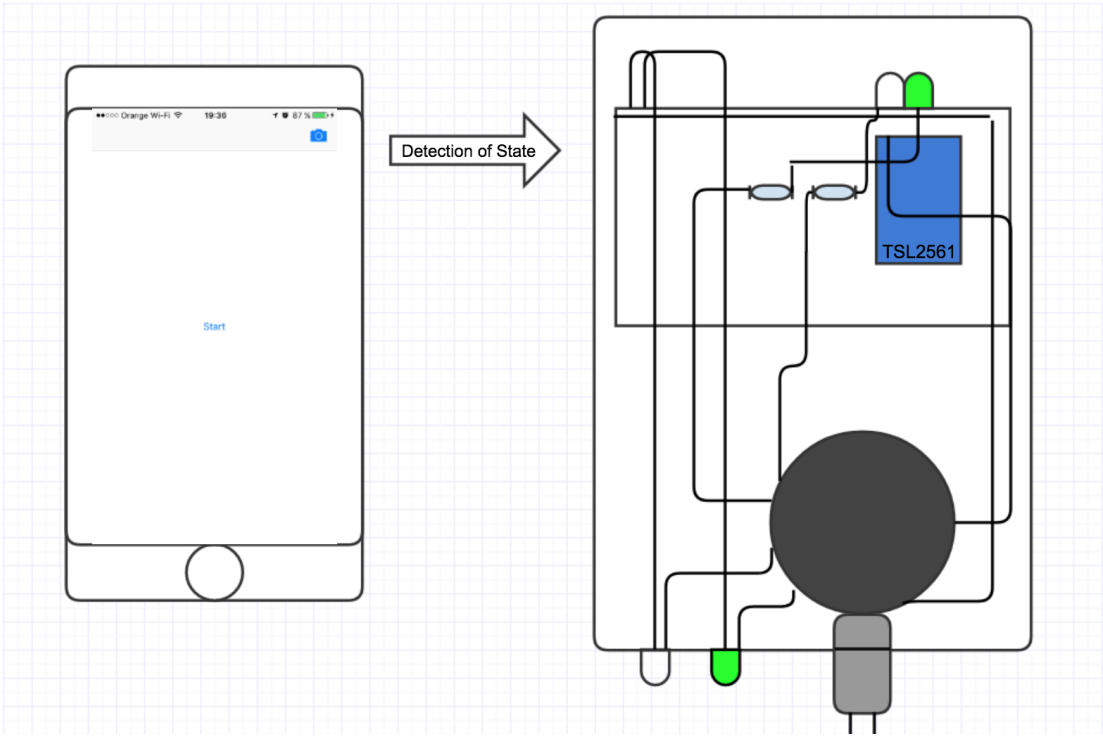


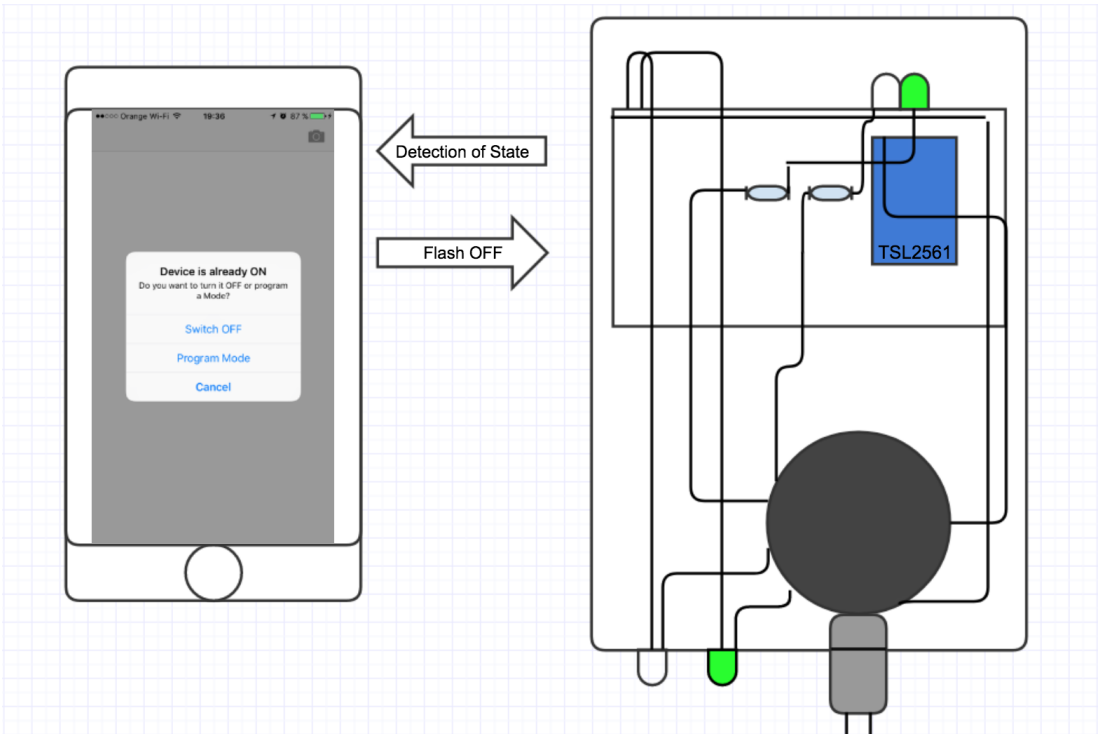

Device is already ON
Do you want to turn it OFF or program a Mode?
Switch OFF
Program Mode
Cancel
Detection of State
Flash OFF
TSL2561

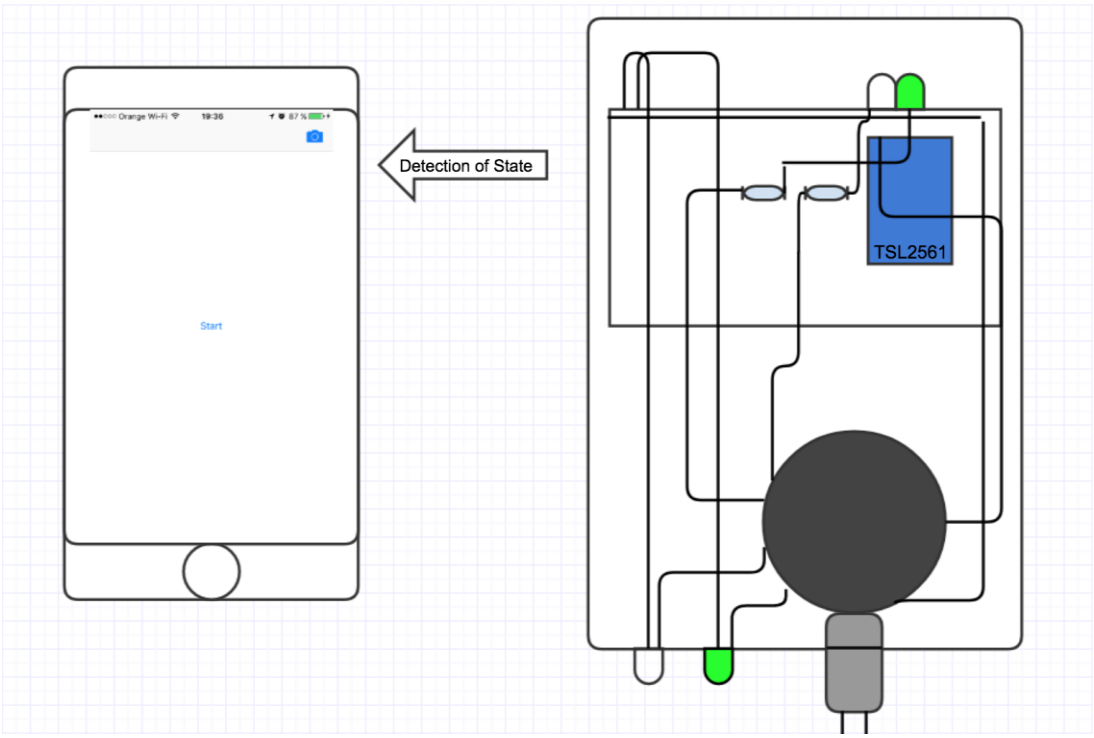

Start
Detection of State
TSL2561

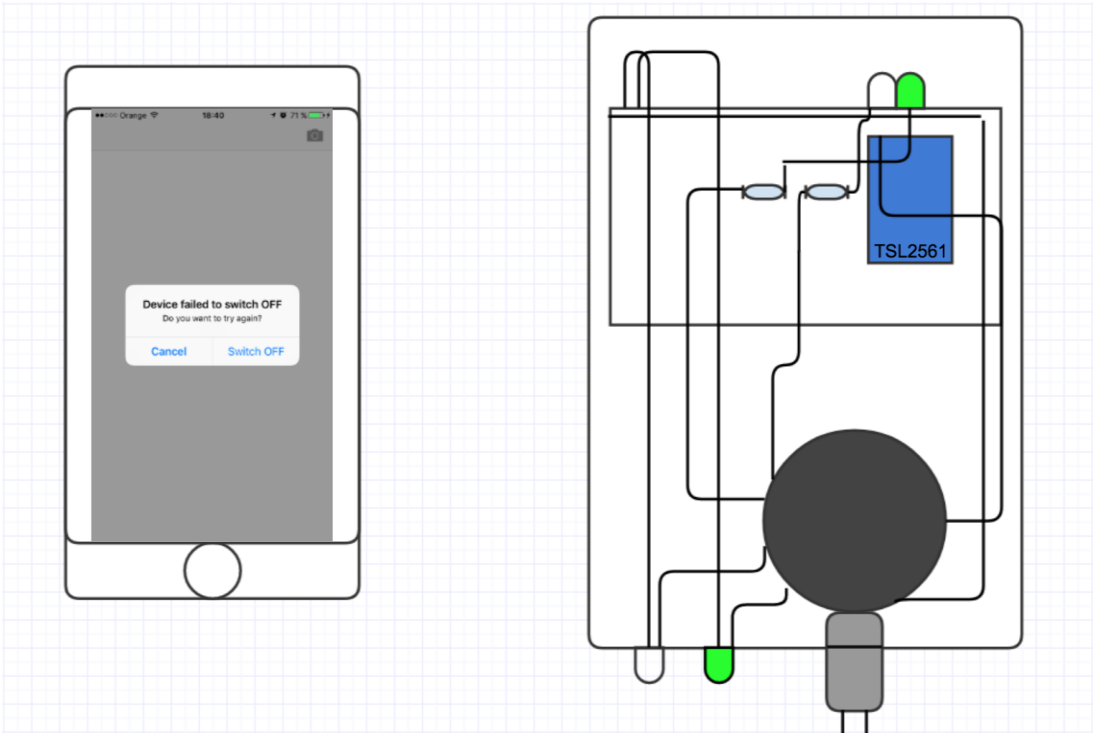


e) HS is ON. Detection, Switch OFF command and unsuccessful switch.

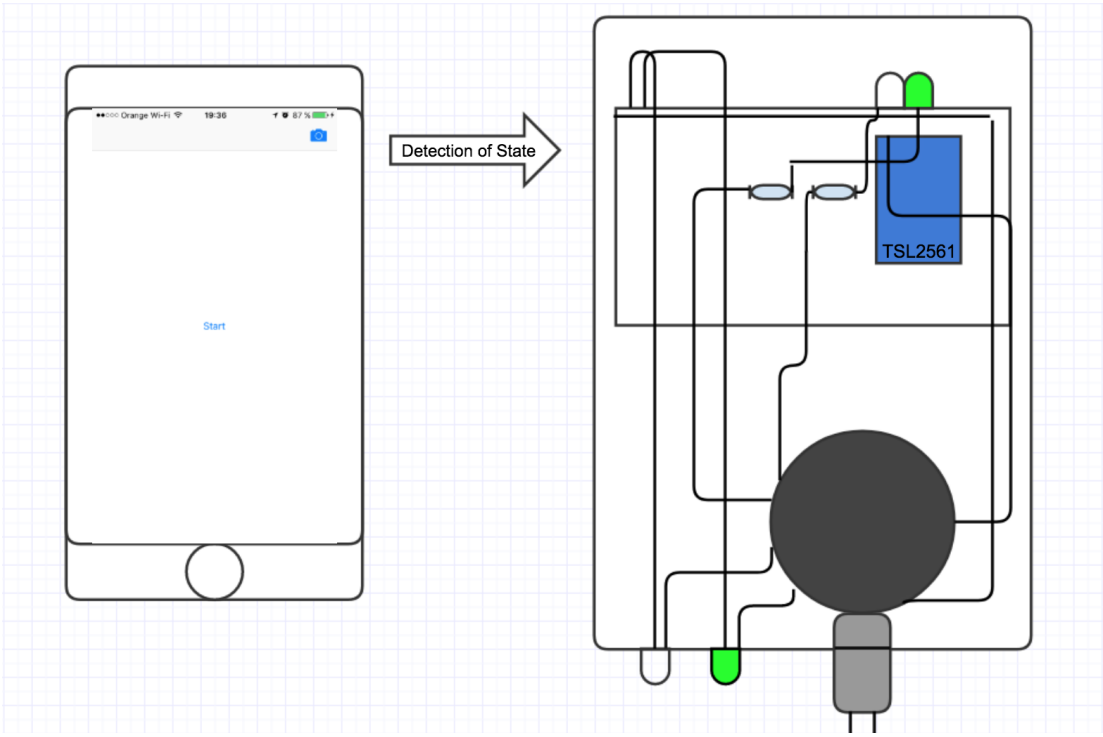


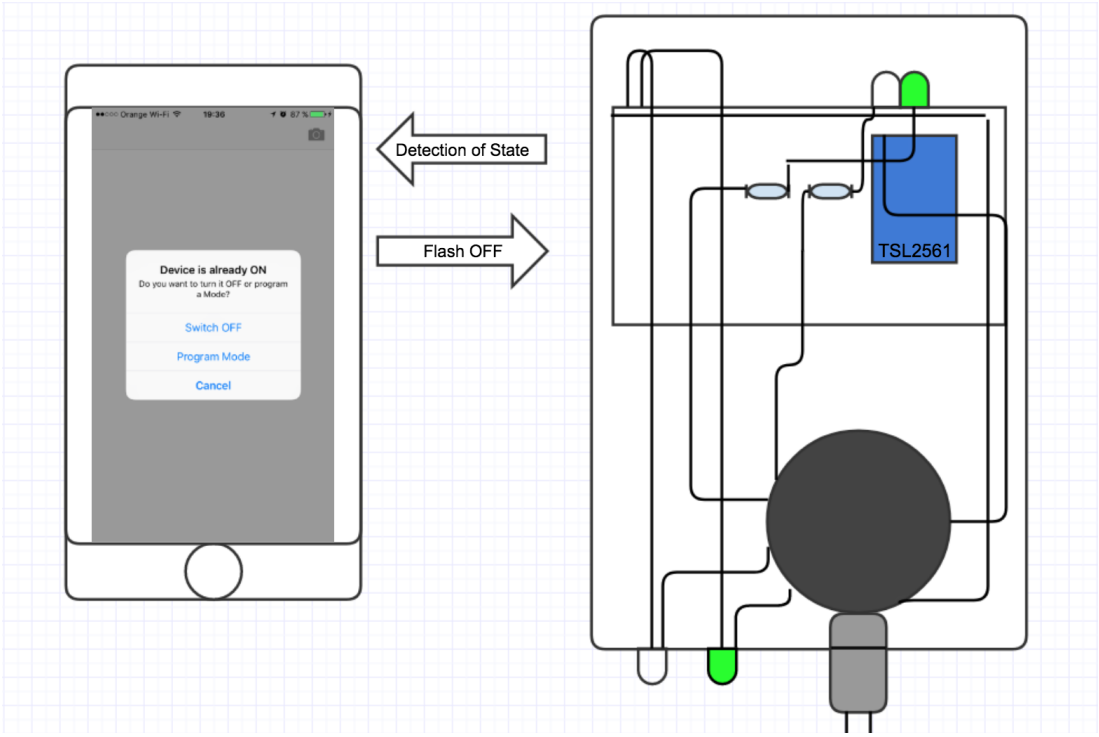

Device is already ON
Do you want to turn it OFF or program a Mode?
Switch OFF
Program Mode
Cancel
Detection of State
Flash OFF
TSL2561

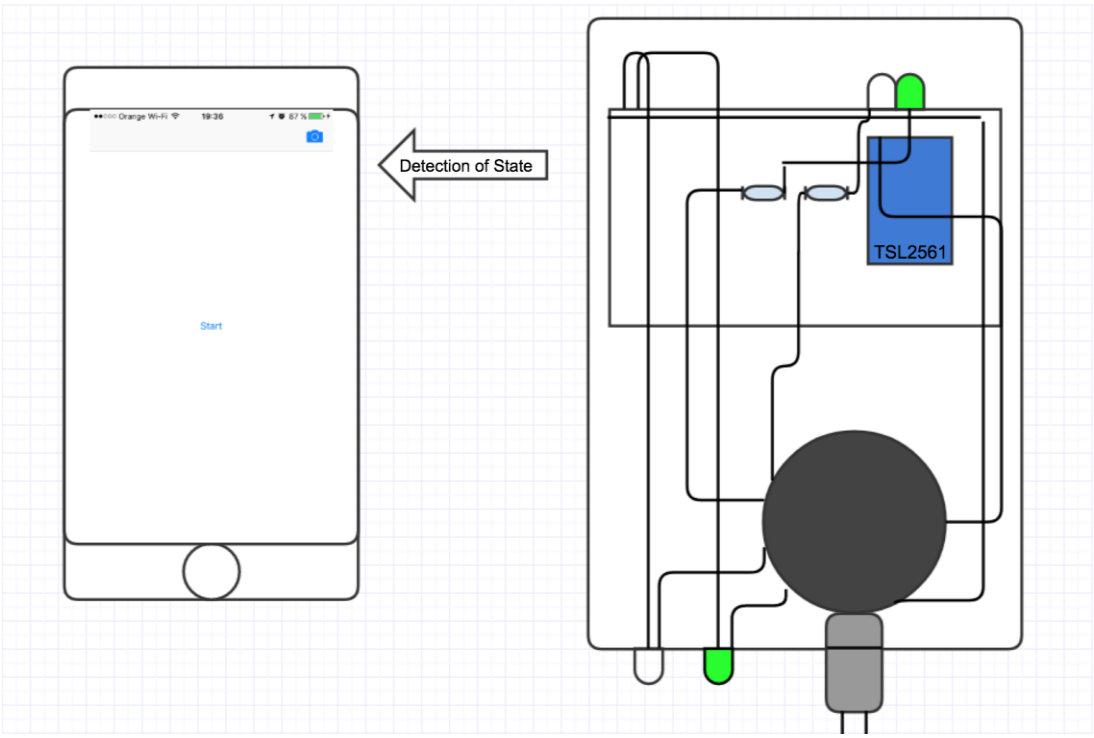

Start
Detection of State
TSL2561

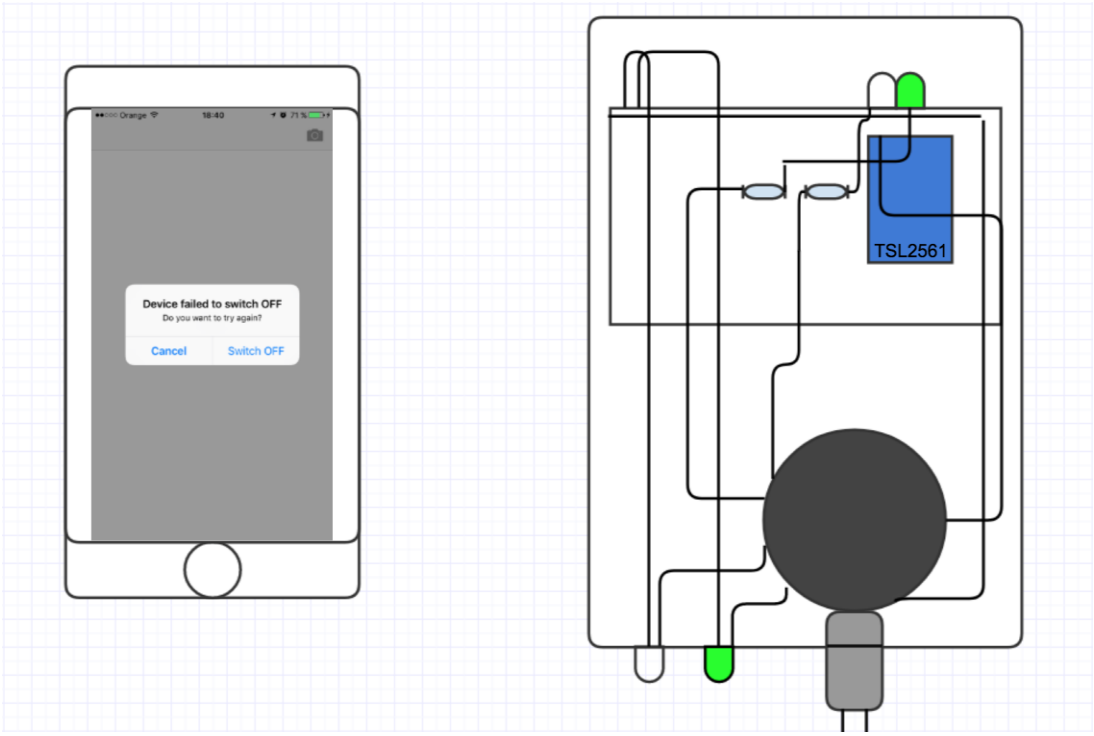


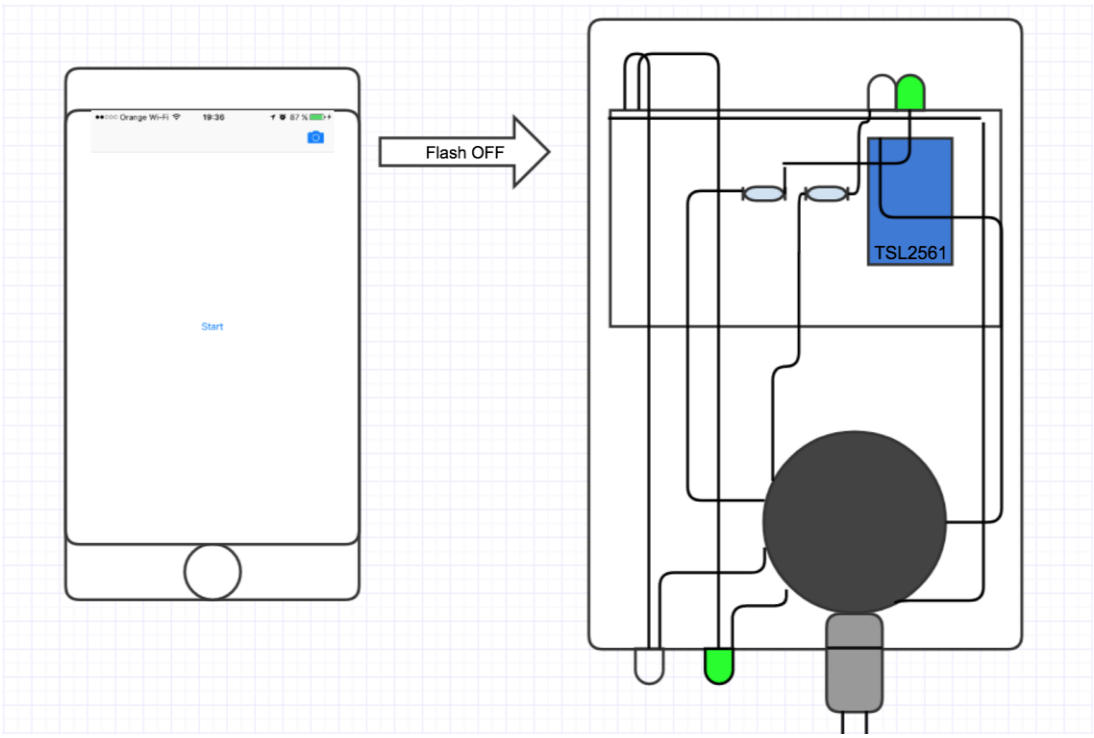
f) HS is OFF. Detection, Switch ON command, unsuccessful switch and successful retrial.

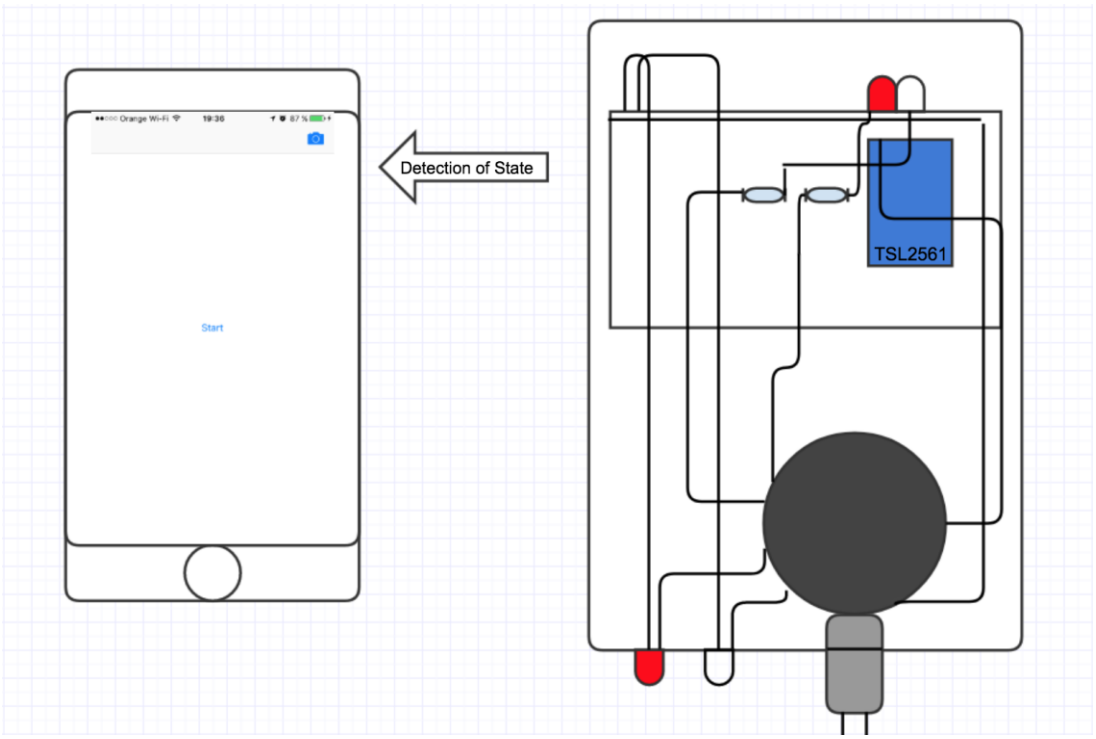


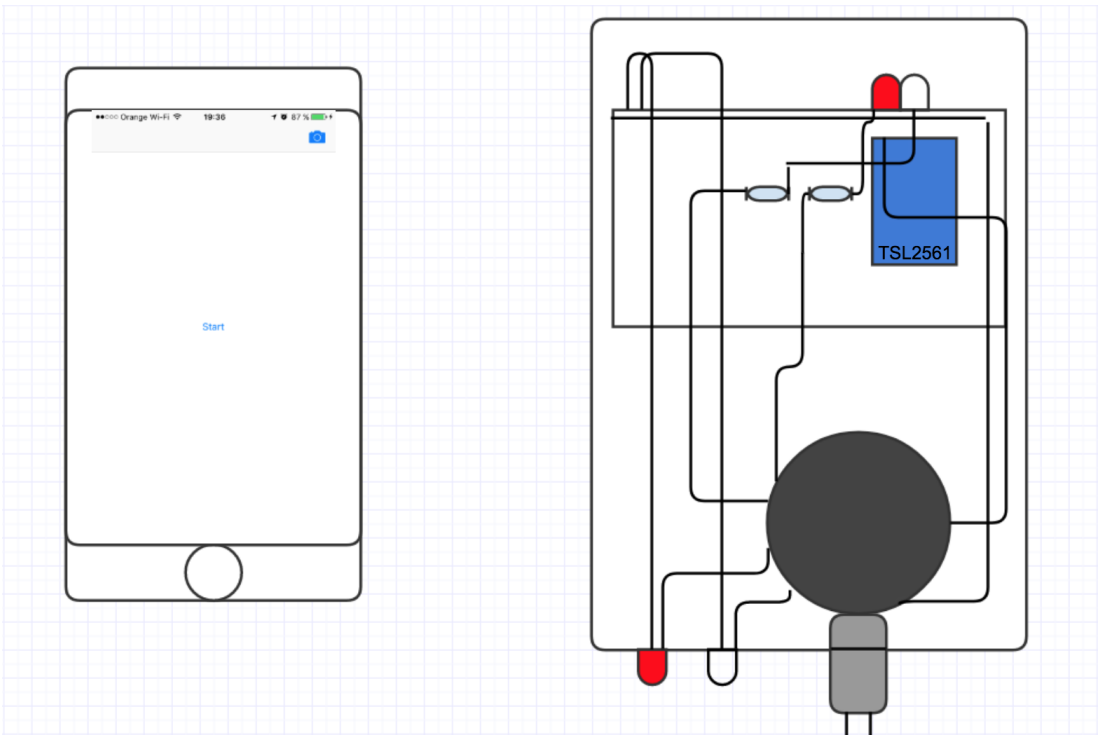

Device is already ON
Do you want to turn it OFF or program a Mode?
Switch OFF
Program Mode
Cancel
Detection of State
Flash OFF
TSL2561


Start
Detection of State
TSL2561


Device failed to switch OFF
Do you want to try again?
Cancel
Switch OFF
TSL2561


Start
Flash OFF
TSL2561


Start
Detection of State
TSL2561


Start
TSL2561

11. Glossary

A list of all acronyms and the meaning they stand for:

cm centimetre

ms milliseconds

HS Home System

lux International System Unit for luminosity level

GUI Graphical User Interface

MVP Minimum Viable Product

VLC Visible Light Communication