Security system: the smart light bulb controller
Additional documentation

Bernardita Štitić

June 17, 2016
Spring / Summer

Objective of this document

The smart light bulb controller was an IoT system developed as a final project for the class EECS 113 – Processor Hardware / Software Interfaces at UCI (Spring 2016). For this report, we were asked to cover only certain topics in a specified number of pages so not all details could be thoroughly explained. Consequently, this document was prepared with the purpose of providing additional documentation to the original report "Security system: the smart light bulb controller – EECS 113 - Final project". A copy of this report was given to Calplug.

Index of topics

I. Introduction

The device that was built and coded with Python is a smart light bulb controller which is similar to a security light. It has a light sensor and a motion sensor so that only if the area is dark enough and movement is sensed, a light bulb will be turned on for approximately 30 seconds using a mechanical relay controller (relay + amplifier). After this time period, the light bulb will be turned off automatically and the environment will be sensed again using the sensors.

Additionally, this system goes one step further than a typical security light – it can be incorporated into an IoT system. This means that by pressing a switch that activates an interrupt service routine, the light bulb can be turned on and off through Wifi using the MQTT protocol (CloudMQTT) for emergency/security purposes. It is then possible to use this Wifi control to deactivate the interrupt service routine and go back to sensing the environment. More details about this will be given later.

II. Bill of Materials (BOM)

For this project, the required materials are the following:

- **Raspberry Pi 3 Model B (RPi):** a microcontroller unit (MCU); the one used for this project had an SD card with Raspbian installed. For this MCU, it is also required to have a power adapter (a 5V 2A in this case) to power it up. Finally, it is also needed to have a GPIO cable and a breakout board to run the Python code for the prototype.
- **Breadboard:** the board used to build the simple prototype.
- **Light dependent resistor (LDR/photocell):** also called a light sensor, an LDR outputs an analog signal that represents the resistance of a light variable resistor. Particularly, the value decreases as light intensity increases and vice versa. The one used for this project also came with a potentiometer.
- **PIR (Passive Infra Red) motion sensor:** a sensor which reacts to infrared heat, consequently indicating movement. The one used for this prototype outputs a digital signal using 3.3V logic, so that it can be directly connected to the breakout board.
- **ADC:** a MCP3008 chip was used to read the analog outputs coming from the LDR.
- **Tactile switch:** a tactile switch (with 4 terminals) was used to activate the interrupt service routine.
- **10K resistor:** used with the tactile switch to create an appropriate voltage drop.
- **Mechanical relay controller:** this included a relay and an amplifier inside it. It is capable of turning on the device connected to it (a light bulb in this case) if it receives 3.3V. It can also turn it off if it is given 0V.
- **Light bulb:** with a base, it's the component that will be turned on or off.

- **Power strip:** this will power up the prototype, since the Raspberry Pi's adapter and the relay will be connected to it.
- **Jumper wires:** these are used to make connections within the prototype.
- **Male to female wires:** these are used to connect the sensors to the MCU or the ADC.
- **Wood board:** to mount the prototype and isolate it electrically.
- **Tape:** to mount and stick the parts of the prototype to the board.

Also, it is important to emphasize that a special open source library for Python called the Paho library (MQTT client library) was installed on the Raspberry Pi as well as a piece of python code that sends the IP address to your email upon booting. This is so that you can access the RPi using PuTTy. More details about this will be given later.

III. Connections (wiring)

This section describes, in detail, how every component was wired in the project. Sometimes, it was necessary to use more than one jumper cable to achieve a determined connection. For example, when connecting 3.3 V to the LDR, it was necessary to use a jumper cable to connect the 3.3V pin to a certain point of the breadboard, and then use another one to connect that point to the VCC pin of the LDR.

1. ADC

For the ADC, the following diagram, which is shown in Figure 1, was used. **The image was taken from**: http://jeremyblythe.blogspot.com/2012/09/raspberry-pi-hardware-spi-analog-inputs.html.

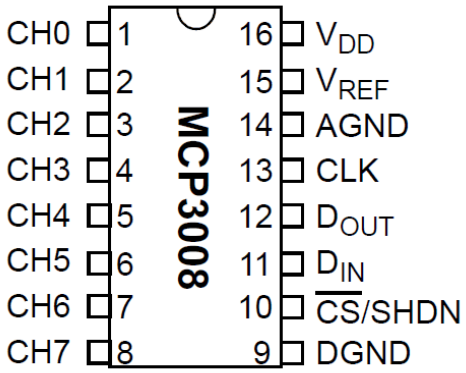

Figure 1. Pins and channels of the MCP3008

Taking Figure 1 into consideration, the wiring is as follows:

- CH0 -> Analog OUT (LDR)

- VDD -> 3.3V (RPi)
- Vref -> 3.3V (RPi)
- AGND -> GND (RPi)
- CLK -> GPIO #18
- Dout -> GPIO #23
- Din -> GPIO #24
- Cs / SHDN -> GPIO #25
- DGND -> GND (RPi)

2. LDR

For the case of the LDR, the connections are shown in Figure 2 below. Here, it is important to clarify that the Analog OUT pin goes to CH0 of the ADC (as indicated in the previous section), GND goes to the GND of the RPi and VCC goes to the 3.3V pin of the RPi.

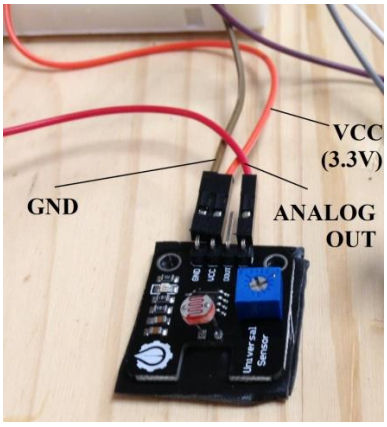

Figure 2. LDR and its connections

3. PIR motion sensor

For the PIR motion sensor, the connections are shown in Figure 3 on the next page. Here, it is important to clarify that the Dig Output (Digital output) of the PIR is connected to GPIO # 16 directly (since the PIR has 3.3 V logic), GND goes to the GND of the Rpi and VCC goes to the 5V pin of the RPi.

In case the pins of the PIR are not labeled on the outside, please remove the round cap. The labels indicating which one is the GND pin and which one is the 5V pin should now be visible. Usually, the Dig OUT pin is the one in the middle.

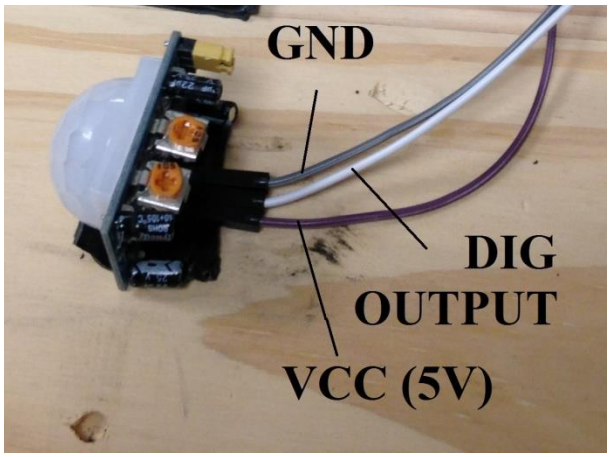

Figure 3. PIR motion sensor and its connections

4. Mechanical relay controller

For the mechanical relay controller (relay + amplifier), the connections are shown in Figure 4. Additionally, just like before, it is important to note here that the VCC cable is connected to GPIO # 12 of the RPi. When configured as an output and taken to the "True" state, the GPIO pin will feed 3.3V to the relay using this cable. This is configured in the code of the project. Finally, the GND wire should be connected to the GND pin of the RPi.

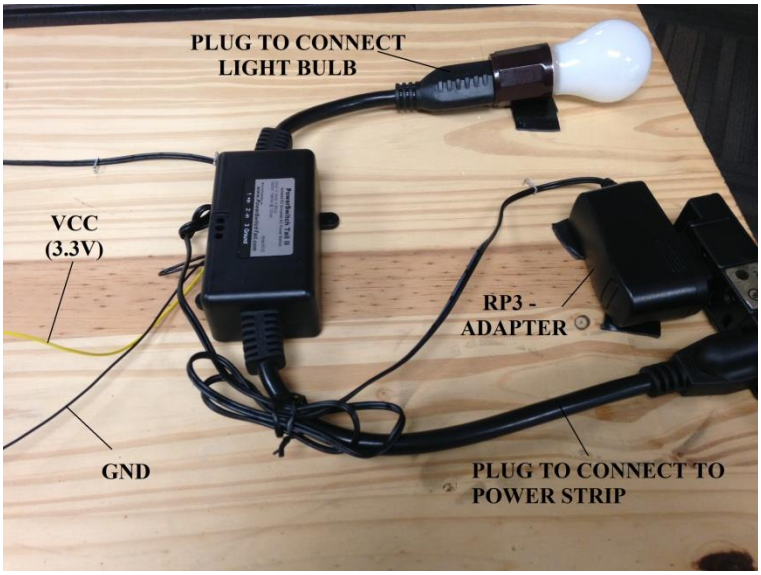

Figure 4. Mechanical relay controller and its connections

5. Tactile switch

For the tactile switch, the diagram used in Figure 5 was used. **The picture was taken from Assignment #5 of the class EECS 113 – Processor Hardware / Software Interfaces, Spring 2016**. To see it, go to EEE (eee.uci.edu), go to the class websites of

Spring 2016, click on Electrical Engineering & Computer Science, then click on EECS 133 LEC A and, finally, click on Weekly assignments (one of the tabs at the top).

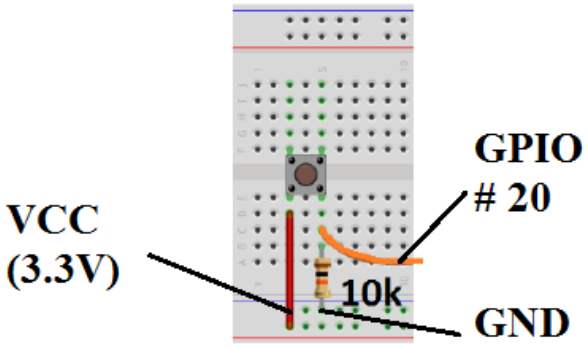

Figure 5. Tactile switch and 10K resistor: electrical connections

6. Other connections

The remaining connections are basically the wires used to power up the device and the RPi. These can be seen in Figure 6, which is a picture of the project on the wooden board. Strong tape was used for this purpose (mounting).

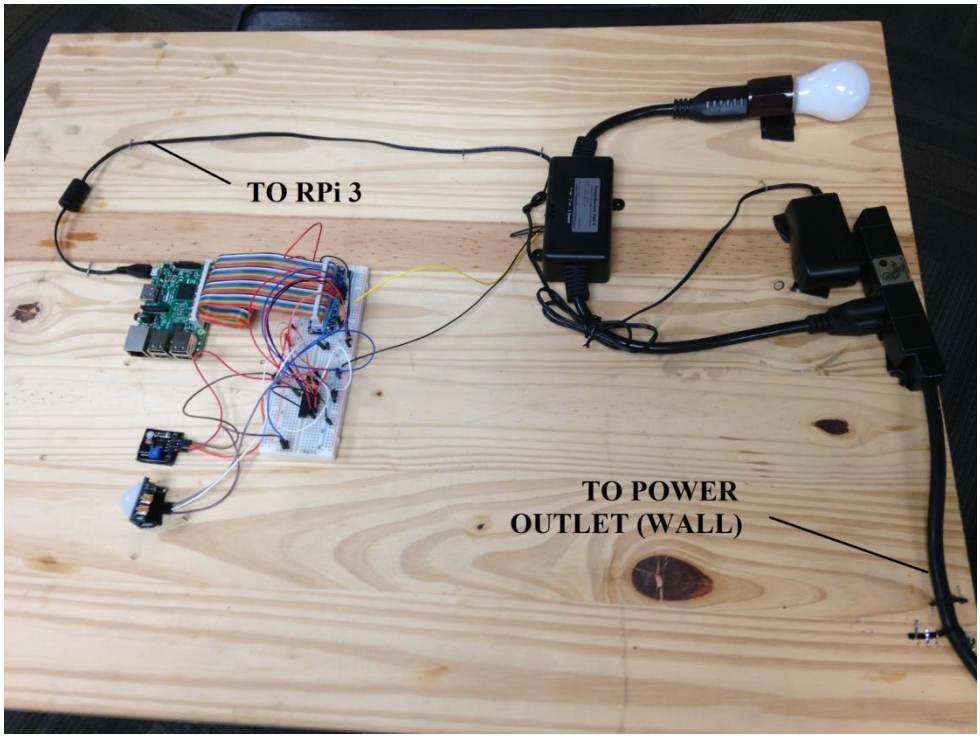

Figure 6. Remaining connections (smart light bulb controller)

IV. Sensors, logic and actuators

A general diagram of the project is illustrated in Figure 7. It shows the components that are classified as sensors and as actuators as well as the logic of the device. It must be emphasized that out of all the sensors used for this project, the PIR is the primary one and the rest are considered to be secondary.

While is it is true that the LDR (for sensing brightness) with the ADC and the PIR (for sensing motion) are sensors, there are also other sensors that can be classified as such in this project. The first two (the PIR and the LDR+ADC) are classified as physical sensors (passive sensors) since they physically exist and generate outputs based on conditions that surround them. They take energy from the environment and generate a signal. Due to this, the tactile switch can also be classified as a physical sensor.

However, it can be argued that the smart light bulb controller possesses another type of sensor that can be classified as a logical sensor. This is the MQTT control, since it receives input through the PIR and LDR+ADC, processes the signals and publishes their outputs to the cloud. Software is required to do this. The information can then be seen on an interface at cloudmqtt.com. For this, one must have an account on the website first.

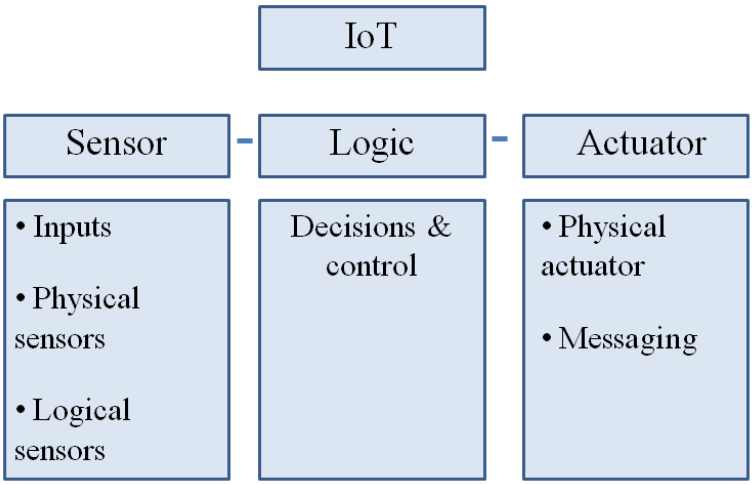

Figure 7. Diagram of the smart light bulb controller

The logic (the decisions and control) of this project will be described later in this report. A specific section will be dedicated to this as well as to explaining the key topics and messages that control the behavior of the device via MQTT control. At this point, however, it is worth mentioning that decisions and control are realized using Wifi (physical layer), MQTT (command and control) and algorithms (in the Python code).

Finally, it is important to explain which components classify as actuators. The mechanical relay controller (relay + amplifier) is a physical actuator. It physically exists

and generates an output based on a particular input (in this case, a DC 0V or 3.3 V voltage signal coming from the RPI). However, just like before, the MQTT control can be classified as an actuator as well. This is because depending on the message that is published to the cloud when the interrupt service routine is activated, one can remotely turn the light bulb on or off using MQTT.

V. Algorithms

This project has three algorithms. They're implemented using both code (in Python) and MQTT control. These are:

- Motion controlled security light: turn on the light bulb for 30 seconds when it's dark and there is movement. This requires a threshold for the LDR, which was determined to be 800 (tested with bright light, dim light and in dark). Therefore, **only** when the output of the PIR is 1, and the output of the LDR is above 800, the light bulb will be turned on. Also, the 30 seconds were created with two for loops in Python (for i in range (0, 5000): for j in range (0, 15000) in the code). The duration of the light was tested and measured using the stopwatch app of an Iphone 5c.
- Read sensor values remotely with the MQTT: this is achieved by first reading the output of the PIR and LDR. Then, there are instructions in the code so that these are published to the CloudMQTT console using the Paho library.
- Control actuators remotely: this is achieved by first pressing the switch (one must wait for a few seconds because it depends on which part of the While True loop the program is in) to activate the interrupt service routine. Then, using the CloudMQTT console, it is possible to turn the light bulb on / off using Wifi. This will be detailed later on.

VI. CloudMQTT: creating a console

On the next page, in Figure 9, the terminal (or console) that was used for this project is illustrated. To be able to access it, one must first create an account at cloudmqtt.com. To do this, go to cloudmqtt.com, click on "Plans" at the top and then select the CuteCat plan (click on "try for free"). Then, create your account.

Once this is done, go back to cloudmqtt.com. This time, click on "Control Panel" at the top. Login using your email and password and you should be taken to the control panel. There should be two rectangles with information that look like the ones shown in Figure 8 on the next page.

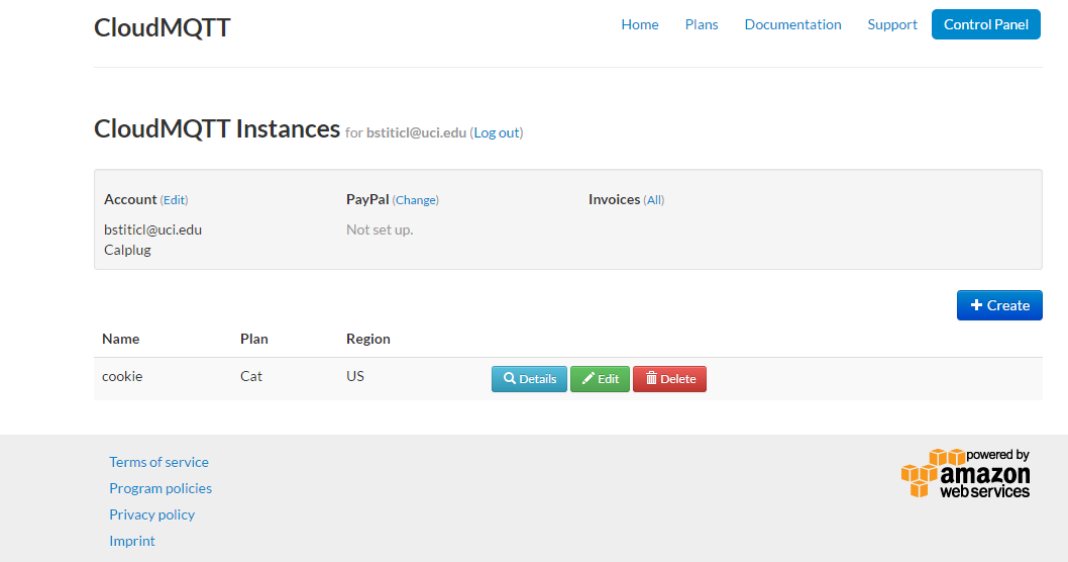

Figure 8. Control panel - CloudMQTT

Now, one should click on "Details". Here, all the information related to your account will be displayed. This is the information that needs to be in the code to be able to activate the Wifi and MQTT control. Next, click on "Websocket UI" at the top to see the MQTT terminal. It's now shown below in Figure 9 as it was said at the beginning of this section.

Figure 9. CloudMQTT – terminal (console)

If the details about the account wish to be seen again, one has to click on "Overview" at the top. Additionally, in Figure 9, it can be seen that one can type a "Topic" and a "Message", click on "Send" and see the topic and message under "Received messages". This action (of making messages appear on this webscoket) is known as **publishing to the cloud**.

VII. Sending and receiving the IP address upon booting & changing Wifi

Remember that to be able to configure the following instructions, or see the IP address of the RPi for the first time, it is necessary to connect the RPi 3 using an HDMI cable to a monitor. You will also need a keyboard for this. If Raspbian was installed correctly, the desktop with the raspberry should appear. Here, it is possible to obtain the IP address by placing the cursor on the Wifi logo. However, what is desired is to access the RPi remotely using Putty later on, which is why we need to skip the monitor part and get the IP address using your email.

The RPi 3 I left in Calplug has a piece of code named "startup_mailer.py" in the home directory of the RPi. Please, open it, and edit the email address you want the IP address to be sent to. Also, edit the information of the email that will be used to send the IP address. This email address (the 'from'), however, MUST be GMAIL. For this one, also, enter your password (in the code). The IP address is needed to access the RPi using PuTTy and WinSCP (it's the "host name").

If, for whatever reason, you're using another RPi which was not mine, you will need to setup this part yourself. For this, connect the RPi to a monitor and register it so that it can connect to the UCI mobile wifi. **The following instructions were taken from Assignment #5 of the class Processor Hardware / Software Interfaces.** To see it, go to EEE (eee.uci.edu), go to the class websites of Spring 2016, click on Electrical Engineering, then on EECS 113 LEC A, and then finally click on Weekly Assignments. These instructions will be quoted in "" to indicate that the information is from an external source. You will need to put your account information in the next code.

"Copy the following startup_mailer.py script into the /home/pi directory.

```
__author__ = 'Cody Giles'
__license__ = "Creative Commons Attribution-ShareAlike 3.0 Unported License"
__version__ = "1.0"
__maintainer__ = "Cody Giles"
__status__ = "Production"

import subprocess
import smtplib
from email.mime.text import MIMEText
import datetime
import socket
import traceback
import time

def connect_type(word_list):
    """ This function takes a list of words, then, depeding which key word, returns the corresponding
    internet connection type as a string. ie) 'ethernet'.
    """
```

```
        if 'wlan0' in word_list or 'wlan1' in word_list:
                con_type = 'wifi'
        elif 'eth0' in word_list:
                con_type = 'ethernet'
        else:
                con_type = 'current'
        return con_type

# Change to your own account information
# Account Information
to = 'youremail@gmail.com' # Email to send to. (CAN BE ANYTHING)
gmail_user = 'youremail@gmail.com' # Email to send from. (MUST BE GMAIL)
gmail_password = 'yourpassword' # Gmail password.
tries = 0
while True:
        if (tries > 60):
                exit()
        print str(tries) + "\n"
        try:
                smtpserver = smtplib.SMTP('smtp.gmail.com', 587, timeout=30) # Server to use.
                break
        except Exception as e:
                tries = tries + 1
                #logging.error(traceback.format_exc())
                print e.__doc__
                print e.message
                time.sleep(1)

smtpserver.ehlo() # Says 'hello' to the server
smtpserver.starttls() # Start TLS encryption
smtpserver.ehlo()
smtpserver.login(gmail_user, gmail_password) # Log in to server
today = datetime.date.today() # Get current time/date

arg='ip route list' # Linux command to retrieve ip addresses.
# Runs 'arg' in a 'hidden terminal'.

p=subprocess.Popen(arg,shell=True,stdout=subprocess.PIPE)
data = p.communicate() # Get data from 'p terminal'.

# Split IP text block into three, and divide the two containing IPs into words.
ip_lines = data[0].splitlines()
split_line = ip_lines[1].split()

# con_type variables for the message text. ex) 'ethernet', 'wifi', etc.
ip_type = connect_type(split_line)

"""Because the text 'src' is always followed by an ip address,
we can use the 'index' function to find 'src' and add one to
get the index position of our ip.
"""
ipaddr = split_line[split_line.index('src')+1]

# Creates a sentence for each ip address.
my_ip = 'Your %s ip is %s' % (ip_type, ipaddr)
```

```
# Creates the text, subject, 'from', and 'to' of the message.
msg = MIMEText(my_ip + "\n")
msg['Subject'] = 'IPs For RaspberryPi on %s' % today.strftime('%b %d %Y')
msg['From'] = gmail_user
msg['To'] = to
# Sends the message
smtpserver.sendmail(gmail_user, [to], msg.as_string())
# Closes the smtp server.
smtpserver.quit()
```

Then, make it executable:

cd ~

sudo chmod +x startup_mailer.py

Open the Cron configuration file:

sudo crontab –e

Then enter the following line and save:

@reboot python /home/pi/code/startup_mailer.py

Then reboot, and you should receive an email from RPi with its IP address:

sudo reboot".

Now, you should be able to get an email with the IP address. Using your laptop, which should be connected to the same network that the RPi is, go to PuTTy and put the IP address where it says "host name". **Remember that every time you change the wifi, you will need to connect the RPi to the new network using a monitor**. After this, you can simply boot it and use PuTTy on your laptop.

VIII. Installing the Paho library

To be able to run the code that was made for this project, and to be able to use the MQTT protocol, it is imperative to install the open source library "Paho" by Eclipse. It is an MQTT client library designed especially for Python. To do this, there are two possible ways. Both are described at: https://eclipse.org/paho/clients/python/ and in the next paragraphs. **Note that you must be able to access the RPi using PuTTy or have a monitor connected to it**.

The first way is to use the pip tool. For this, enter the following command:

pip install paho-mqtt

For this project however, the second way was chosen. Using PuTTy, enter the following:

git clone https://github.com/eclipse/paho.mqtt.python.git

cd org.eclipse.paho.mqtt.python.git

python setup.py install

This last line however (the python setup.py install) may need to be run with sudo if using Linux, which was the case for this project.

IX. Code: general structure, topics and messages

The code of this project was left on my RPi 3 in Calplug. The name of the file is "project2.py". It begins by importing the required libraries. Here, the Paho library is imported. After this, 5 functions are defined. The first four are for the MQTT protocol. Specifically, the second one, on_message, was modified so that the system can react differently according to the message and topic sent to the cloud from the laptop. **The final one is the readadc function taken from Assignment #6 from the class EECS 113 – Processor Hardware / Software Interfaces** to be able to read an analog input using the ADC. This will be further explained in section 11.

Then, certain configurations are made including the setup of the GPIO pins and the breakout board. With these declarations, the required variables for the code are also declared. Two of these are lists to optimize the program. After this, the MQTT must be configured with your account's id, password, port terminal and host name (it is necessary to create an account at cloudmqtt.com). Immediately after this, the subscriptions to the topics used for this device are made. The template for connecting, publishing, subscribing and the MQTT functions were taken from the website of CloudMQTT. This will be further explained in the next section (section 10).

After this, the program enters the main code – a 'while True' loop. Inside, it enters another while loop that meets a condition (that will be false if the switch for the ISR is pressed). Firstly, the switch is checked and then a delay is introduced. After this, the PIR and the LDR are read (this last one using the ADC). Their outputs are published to the cloud (digital and analog). If the combination of the outputs is the desired one (movement and darkness), the light bulb will be turned on for 30 seconds.

If the switch is pressed, then the code will enter the mqttc.loop() where the device will only check messages sent to the cloud from the laptop (this is done by modifying a variable). The topics and messages that will make the device do something while the interrupt service routine is in operation are the following:

- With the topic 'SOS', one can turn the light bulb on (message '1') /off (message '0').

- With the topic ‘Sense’, one can go back to reading from the sensors (and the variable that keeps the program in the mqttc.loop() will be modified so that it goes back to the while True loop).
    - If this happens, the interrupt service routine will be deactivated (one must press the switch again to activate it). Here, the code will post to CloudMQTT the following information:
        - If there is movement (1) or not (0), using the topic “Movement”.
        - If it’s dark (0) or not (1), using the topic "
        - Using the topic “Light_raw”, the analog output of the LDR will be posted as a message.
- With the topic ‘Over’, the program will exit the giant while True loop, execute GPIO.cleanup() and finish the program.

X. Code: configuring the MQTT

As it was mentioned before, there is a template that was used to publish, subscribe, check the messages and, in general, configure the MQTT in the code so that the program can publish and subscribe. This piece of code will allow the user to use CloudMQTT to see the messages, too. The template was taken from: https://www.cloudmqtt.com/docs-python.html, and goes as follows. Again, quotation marks “” are used to indicate that this comes from an external source.

“import mosquitto, os, urlparse

```
# Define event callbacks

def on_connect(mosq, obj, rc):

    print("rc: " + str(rc))


def on_message(mosq, obj, msg):

    print(msg.topic + " " + str(msg.qos) + " " + str(msg.payload))


def on_publish(mosq, obj, mid):

    print("mid: " + str(mid))


def on_subscribe(mosq, obj, mid, granted_qos):

    print("Subscribed: " + str(mid) + " " + str(granted_qos))
```

```
def on_log(mosq, obj, level, string):
    print(string)

mqttc = mosquitto.Mosquitto()

# Assign event callbacks
mqttc.on_message = on_message
mqttc.on_connect = on_connect
mqttc.on_publish = on_publish
mqttc.on_subscribe = on_subscribe

# Uncomment to enable debug messages
#mqttc.on_log = on_log

# Parse CLOUDMQTT_URL (or fallback to localhost)
url_str = os.environ.get('CLOUDMQTT_URL', 'mqtt://localhost:1883')
url = urlparse.urlparse(url_str)

# Connect
mqttc.username_pw_set(url.username, url.password)
mqttc.connect(url.hostname, url.port)

# Start subscribe, with QoS level 0
mqttc.subscribe("hello/world", 0)

# Publish a message
mqttc.publish("hello/world", "my message")
```

```
# Continue the network loop, exit when an error occurs

rc = 0

while rc == 0:

    rc = mqttc.loop()

print("rc: " + str(rc))”
```

To make this work properly so that you can see the topics and messages using your account (and your console), there are a few things to configure. In the following part of the previous code:

```
“# Connect

mqttc.username_pw_set(url.username, url.password)

mqttc.connect(url.hostname, url.port)”
```

Put your username between ‘’ (single quotation marks) where it says url.username. Then, put your password between ‘’ where it says url.password. Then, where it says url.hostname, put 'm12.cloudmqtt.com' (in this case, I’ve already included the single quotation marks for you). Finally, where it says url.port, put your port number **without** single quotation marks. This is because Python is expecting a number here and not a string.

Remember that all this information is available once you login using your cloudmqtt account at cloudmqtt.com. Also, for the port, there are two more ports called “SSL Port” and “Websockets Port”. **Don’t use these**; use only the first port, which is labeled “Port”.

XI. Code: using the ADC to read analog inputs

As it was mentioned before, the readadc function to read analog inputs using the ADC (MCP3008 chip) was taken from **Assignment #6 from the class EECS 113 – Processor Hardware / Software Interfaces, Spring 2016**. To see this document, go to the class websites of 2016 (eee.uci.edu), click on Electrical Engineering, then EECS 113 LEC A and finally click on Weekly Assignments.

The piece of code that allows you to use the ADC to read analog inputs is the following. It’s quoted using quotation marks “” to indicate that this is not something I coded and that it comes from an external source.

```
“

# Written by Limor "Ladyada" Fried for Adafruit Industries, (c) 2015

# This code is released into the public domain
```

```
import time
import os
import RPi.GPIO as GPIO

# read SPI data from MCP3008 chip, 8 possible adc's (0 thru 7)
def readadc(adcnum, clockpin, mosipin, misopin, cspin):
        if ((adcnum > 7) or (adcnum < 0)):
                return -1
        GPIO.output(cspin, True)
# To LEDs SPI Interface ADC Chip: MCP3008 Potentiometer
        GPIO.output(clockpin, False) # start clock low
        GPIO.output(cspin, False) # bring CS low
        commandout = adcnum
        commandout |= 0x18 # start bit + single-ended bit
        commandout <<= 3 # we only need to send 5 bits here
        for i in range(5):
                if (commandout & 0x80):
                        GPIO.output(mosipin, True)
                else:
                        GPIO.output(mosipin, False)
                commandout <<= 1
                GPIO.output(clockpin, True)
                GPIO.output(clockpin, False)

        adcout = 0
        # read in one empty bit, one null bit and 10 ADC bits
        for i in range(12):
```

```
        GPIO.output(clockpin, True)
        GPIO.output(clockpin, False)
        adcout <<= 1
        if (GPIO.input(misopin)):
            adcout |= 0x1

    GPIO.output(cspin, True)

    adcout >>= 1 # first bit is 'null' so drop it
    return adcout"
```

Also, from the previously mentioned document (**Assignment #6**), the configuration for GPIO pins 18, 23, 24 and 25 is made explicit in terms of code. The following piece of code is coherent with the connections that were described at the beginning of this document in section 3:

```
"# SPI port on the ADC to the Cobbler
SPICLK = 18
SPIMISO = 23
SPIMOSI = 24
SPICS = 25
# set up the SPI interface pins
GPIO.setup(SPIMOSI, GPIO.OUT)
GPIO.setup(SPIMISO, GPIO.IN)
GPIO.setup(SPICLK, GPIO.OUT)
GPIO.setup(SPICS, GPIO.OUT)"
```

The SPICLK, SPIMISO, SPIMOSI and SPICS variables were put in an array in the code of the project (the "project2.py" file).

XII. Trade-offs and limitations of the design

There are two important trade-offs/limitations in this project. The first one is the speed of publishing. The program is designed so that the Raspberry Pi 3 publishes to

CloudMQTT what the PIR and the LDR are sensing. For the first case, it's a string that's either 1 (movement) or 0 (no movement) under the topic 'Movement'. For the second case, it will post both the analog output under the topic 'Light_raw' and its logical value with the topic 'Light'. Therefore, if the analog input is above 800 the logical value will be 0 (no light). The contrary happens if it's below 800.

One cannot have real time sensing (with publishing) and a suitable speed on the CloudMQTT terminal. This means that if one seeks real time sensing, the messages will be sent extremely fast. On the contrary, one can add delays (with for loops) to make the messages appear with a more suitable speed, but the sensing activity will not be realistic. Therefore, a delay (with for loops) was added but suitable enough so that the sensing and publishing tasks were still efficient.

The second and final trade off/limitation was due to the Paho library for the MQTT. This library makes it so that if one wants to send a message to the cloud and have the device act upon it, only this activity can be done. This means that it's not possible to publish the results of the sensors and read messages to react at the same time. This is why the Wifi control of the light bulb is activated through an interrupt service routine using a tactile switch for security purposes.

XIII. PIR, LDR and mechanical relay controller: limitations

The main limitation of the sensors is that they output a signal with a certain delay. However, this delay is so that the output is almost instantaneous, so a delay with for loops was introduced in the code to publish at a reasonable speed. Also, another limitation could be the output voltage logic of a sensor. In the case of the PIR, it was 3.3V, but in the case of the LDR it depended on the value chosen to feed it (5V or 3.3V). To obtain reasonable outputs 3.3V was chosen, since if 5V were used the ADC would read values that would stay high. Finally, a mechanical relay takes up more space than a solid state relay. However, this last one has to be connected directly to AC current so it's not as safe. Therefore, the mechanical one was chosen with a big wood board to mount it.

XIV. Future work

Future work of this project is concerned with integrating other sensors and actuators. For example, there should be a current sensor, a solid state relay, another switch and an IR input and output (or an IR transceiver). Also, the project should be in a project box instead of being mounted on a wooden board.

Additionally, this project should be able to control other devices that use the MQTT protocol. This was simulated using the light bulb – the unit that was built is meant to be a central control unit. This is the broader 'vision' of the project, since its objective is to contribute toward energy savings. For now, it is called the smart light bulb controller and efficiently serves its purpose.