IoT Project

Automatic Watering System

Aarhus University in Herning

5. Semester Bachelor of Engineering in Electronics

Course: E5IOT

Date of submission: December 13, 2017

Supervisor: Morten Opprud Jakobsen

Authors:

Student number: 201505078	Name: Narenth Veluppillai
Student number: 201404978	Name: Mohammad Almaanaki

Contents

Preface

This report is written as a documentation for our IoT project on the course E5IOT at Aarhus University Herning. The report is written by a team of two students currently studying on 5th semester at the Electronic Engineering education. The project is an open source project with all source code and files published online for the open source community. It gives opportunity for other developers to make a similar project, get inspiration or to optimize the project further. The project is based on the stated requirements from the E5IOT course. The audience of the report is targeted to be our lecturers and our fellow students, nonetheless other developers.

Introduction

When we buy bigger plants/trees in Bilka or somewhere else, we usually buy them when they are approximately between 5 months - 1.5-year-old, depending on the tree-type. They are normally pre-grown at a plant nursery, where each plant is taken care of before being shipped to a seller. The process of watering the plants is done easier by automating it. This can be achieved by putting a wireless MCU connected to a moisture sensor in the soil of the plant, where the MCU will order a valve to open as soon as the soil moisture is below level. This allows water to flow to the plant. Some nurseries are placed outside under the sky. Here it is convenient that the MCU is connected to the internet, where a weather forecast from a cloud can give the MCU information whether there will be rain at that day or not. If it will rain some hours later, the system will not provide water for the plants, but instead benefit from the free water. In figure 1 is a plant nursery. On a larger scale with more than a thousand trees, this system could come into great effect.

Figure 1 Plant nursery - https://d2gg9evh47fn9z.cloudfront.net/800px_COLOURBOX4982271.jpg

Another idea for this project is to provide a user with an online interface, for future development, so that they can trace water usage as a financial plot. The system is not aimed to be a 100 % accurate. The relevance is to get an understanding and be able to track and automate the system, so that it will provide water to plants by itself, and lastly give the user a graphical relationship on screen.

Project description

We have developed an IoT automatic watering system throughout this project. The project is based on the theory we have learned throughout the course E5IOT this semester. We have been using the development environment IDE, Atom, from Particle, to generate code for the Photon. The project is open source, shared on Github, https://github.com/Narenth04/E5IOT-Automatic-watering-system, so other people have the opportunity build and optimize it them self.

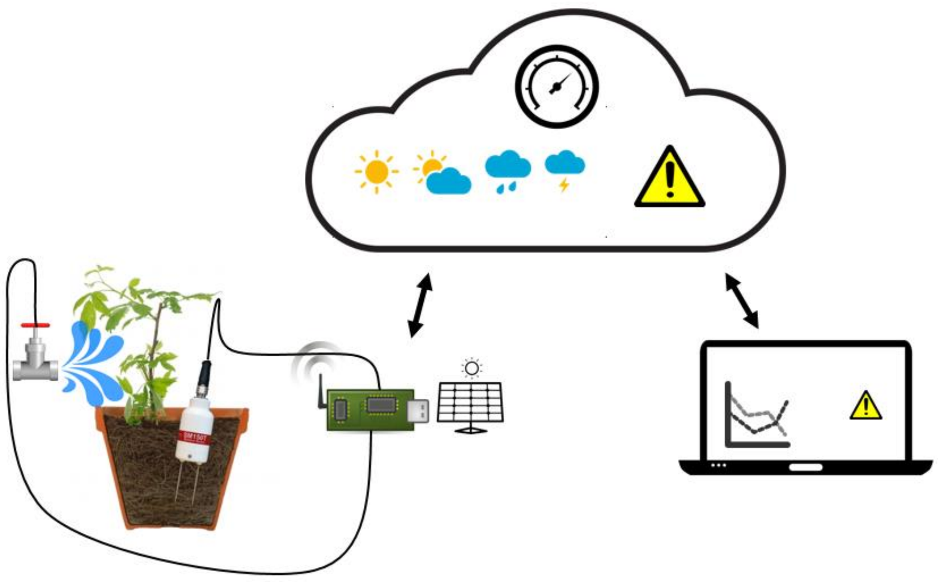


Figure 2 - An illustration of the automatic watering system

Requirement analysis

The following chapter describes the stated requirements for the project and what we are going to do for fulfilling them.

1. The device must be able to connect to the internet

1.1 Internet connection shall be via WIFI

1.2 The device should preferably be able to connect to AU's "AU Gadget network"

We are using the Particle Photon device that has built in WiFi which we connect to the AU gadget network with. The AU gadget network is an encrypted network that only allows authorized devices to connect to it. The devices have gained authorization because they are registered with MAC address to the network.

2. Your device must be able to read data from a connected sensor, local to the device

We need a sensor connected to our platform device for fulfilling the requirement. An obvious solution for that is a soil moisture sensor that measures the humidity of the soil, then we will know whether the soil needs water or not.

3. Your device must be able to control an actuator

For fulfilling this requirement, we should be able to control an actuator from our device. The actuator we have chosen is a solenoid valve that will allow water to the plants. When the soil moisture sensor detects that the soil is dry and there is no rain coming, then the valve shall it open for the water to pass through.

4. Your device must use data from a web service, to augment "what it does", this could be weather data, traffic data, stock prices, twitter feeds, emails, rss-feeds, or something different.

For fulfilling this requirement, we should be able to access the internet and use some relevant data for our system. We are making an automated watering system and therefore we need to access weather data for obtaining whether it is going to rain or not. The way we want to do this is by connecting to an online service that provides us with the current a rain situation.

5. Your software and hardware design must be shared

We have created a public Github repository where we share our project files. It contains everything from source code to flowcharts.

By sharing our code on Github it both allows other students and teachers to track the process of our project, and it allows us as team to work on the same file at the same time without overwriting each other's work. Other than that, by sharing the source code also makes the project open source, so people who are interested can make a similar IoT project, and maybe also optimize it further.

Selecting technical platform (Narenth)

The technical platforms describe which modules we have chosen to work with in the project.

Particle Photon

The embedded device we are working with is the Particle Photon. The Particle Photon is a small Wifi development board made for creating internet of things projects. It is possible to use Particles own online IDE for developing and we can choose whether to flash it over the air or through USB. One of the reasons we have chosen this platform is because of the amount support there is available for it. Particle has its own library with support and a public community where developers can share ideas and solutions.

In the future would it for our automatic watering project be obvious to use a Particle Electron instead. The Electron is a cellular connected IoT device, that could be great to use at tree nurseries where there typically is no WiFi connection.

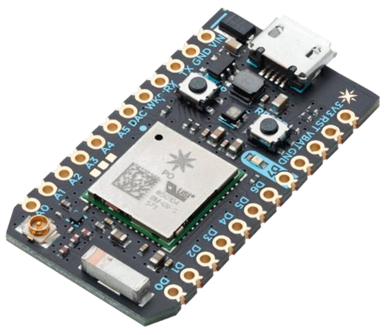


Figure 3 The Particle Photon IoT platform

Soil moisture sensor

We need a sensor for measuring the moisture in the soil of the plants. To that purpose we have bought a soil moisture sensor that act as a variable resistor. The sensor has two large pads that works as probes and acting like a variable resistor. The more water there is in the soil, the better conductivity would there be between the pads which leads to a lower resistance and a higher output voltage. By this way we are able to measure the moisture level in the soil.

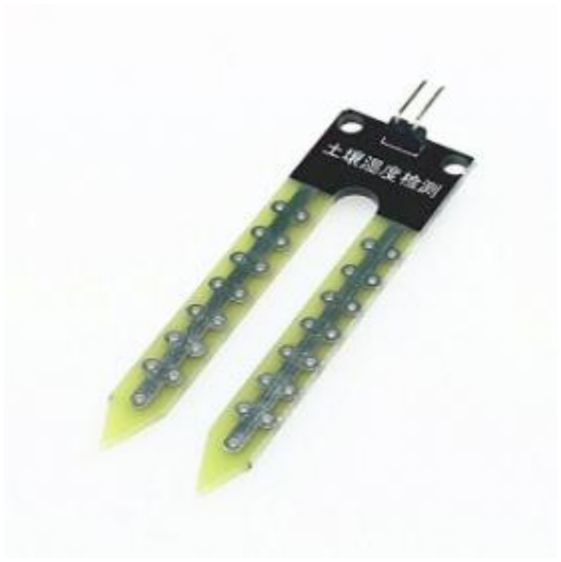
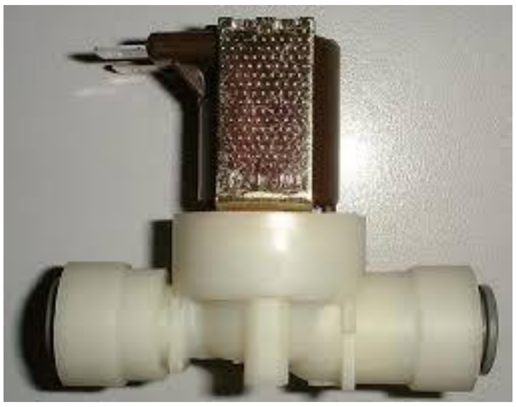


Figure 4 - The image is showing the soil moisture sensor we are working with.

Valve

We need a valve for controlling the water supply for the system. Thus, we are working on a prototype of the system, have we just chosen a valve we have found in the school lab. The main purpose of it is to open and lock for the water supply, depending on the soil moisture level and the weather forecast. The valve operates in a simple way; by applying a voltage to it, will it open for the water. It would be connected to the Photon device, so the device itself can control the water flow.



Figur 5 - The Invensys valve we are using

System design (Mohammad)

In the following chapter we will discuss the system architecture design.

Hardware

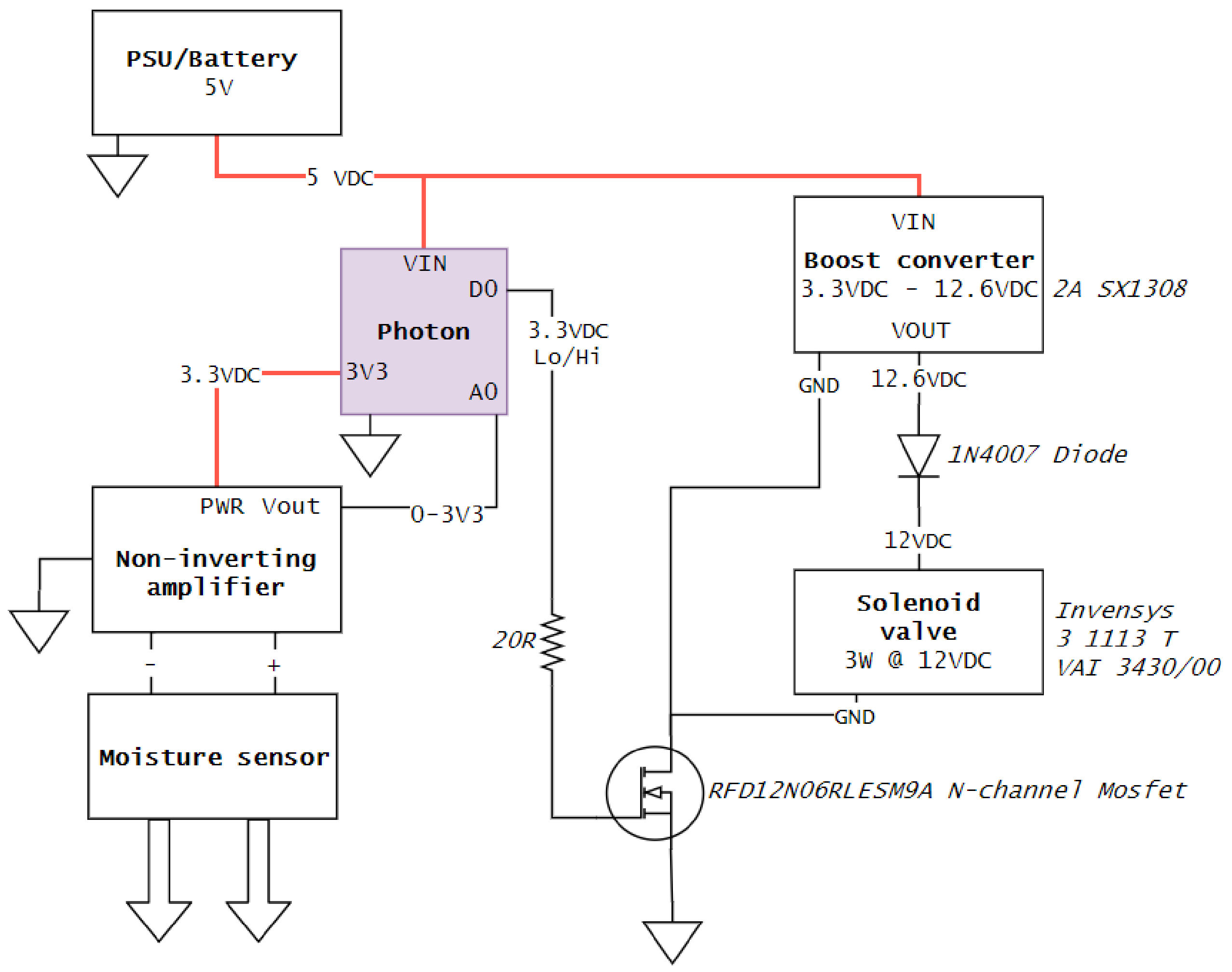


Figure 6 Hardware diagram

Either a PSU or a battery is used to provide 5 VDC. The photon gets the supply voltage across the VIN pin, which can tolerate up to 5.5 VDC. The non-inverting amplifier gain controls the sensitivity of the moisture sensor setup. The moisture sensor setup provides a downgoing voltage, from 3.3 VDC towards 0V, the more moisture there is around the moisture sensor legs. Since we are using a solenoid valve that needs 12 VDC to allow water to pass, we must use a boostconverter to go from 5 VDC supply to 12.6 VDC. 0.6 VDC goes across the diode and 12 VDC is provided to the solenoid valve’s supply pin. The diode ensures that the solenoid does not generate a backward going spike, that will potentially damage the boost converter. This is due to the inductance in the solenoid. The solenoid and boost converter are only driven, when the mosfet has a high gate voltage from D0. That way, no quiescent current in running through the boost converter logic IC.

Software

The architechture of the system functionality and synchronization is described in the ASM figure XX (algorithmic state machine) chart.

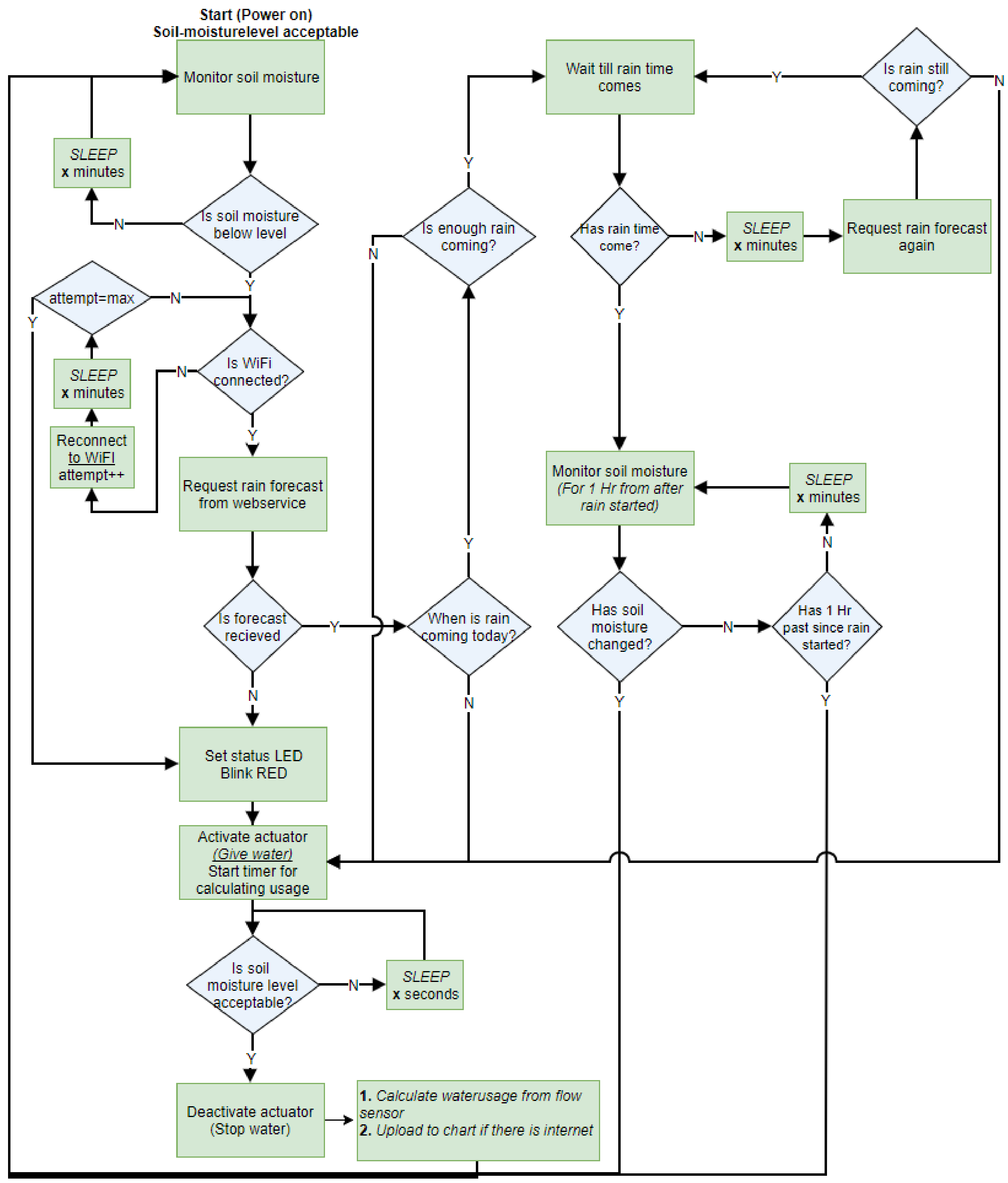

Start (Power on)
Soil-moisturelevel acceptable
Monitor soil moisture
SLEEP x minutes
Is soil moisture below level
N
Y
attempt=max
N
Y
SLEEP x minutes
Reconnect to WiFI attempt++
Is WiFi connected?
N
Y
Request rain forecast from webservice
Is forecast recieved
Y
N
Set status LED Blink RED
Activate actuator (Give water) Start timer for calculating usage
Is soil moisture level acceptable?
N
SLEEP x seconds
Y
Deactivate actuator (Stop water)
1. Calculate waterusage from flow sensor
2. Upload to chart if there is internet
Is enough rain coming?
Y
N
When is rain coming today?
Y
N
Wait till rain time comes
Has rain time come?
N
SLEEP x minutes
Request rain forecast again
Is rain still coming?
Y
N
Monitor soil moisture (For 1 Hr from after rain started)
Has soil moisture changed?
N
Y
Has 1 Hr past since rain started?
N
Y
SLEEP x minutes

Figure 7 ASM chart

In the ASM chart, above to the left, we start the Photon with a newly watered plant. It keeps monitoring the soil moisture state. If the moisture is acceptable, the system will shut off WiFI, the microcontroller and sleep. It will start from the complete beginning as a shut off microcontroller, which is fine in this case, since we are already at the start. The aim with the system sleep state is only to save power. As it can be seen, if the forecast is not received or if WiFi is not available, the system will continue to run and provide the plants with water, but set a blue notification LED on the board. This LED could be response trigger to the cloud for a fault condition. Unfortunately, we didn’t get far enough to implement the functionality.

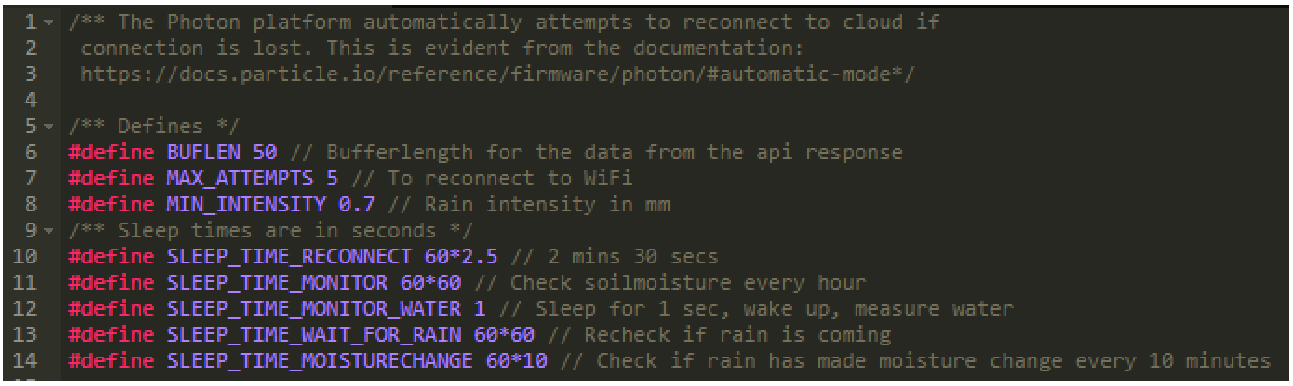
One convenient functionality, when WiFi is not working, is that it will take some minutes to trigger a fault condition. Imagine that the user had shut off their router, if they used the photon, and were powering it up within the next couple of minutes. The software would at least wait a couple of minutes, before triggering an error. Of course, if the error is triggered, it should in the future of this system be automatically reversible, as soon as WiFI is reconnected. The system must not be limited by error conditions and shall continue to provide the plants with water, even if no connection to the cloud is provided.

Implementation

The following chapter describes how the states from the ASM chart are implemented. Only relevant code is described. The whole sourcefile can be found on our github repository, referenced in the project description. Furthermore, the webhook API calling method is described.

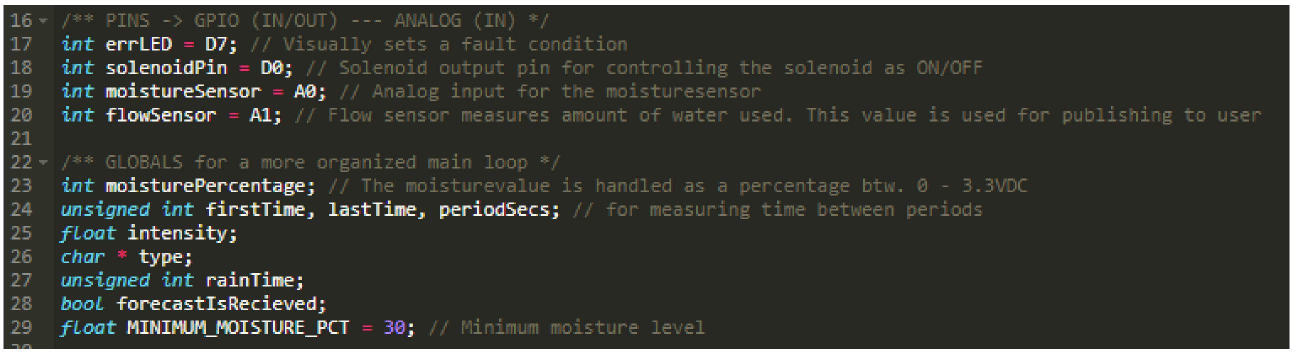
Software (Mohammad)

The software builds on the exact ASM flowchart under the system design. A good flowchart can make code and functionality implementation much more structured and easier. The code is written in one big file, in C form, but with some c++ types used, such as String. At the beginning of the code, 1-14 (code line numbers), there are many defines. Those provide an easy twitching and teaching “interface” for other people, that are interested in this open source project. The different sleep times dictate how long the board shall sleep in the different states.



```
 1  /** The Photon platform automatically attempts to reconnect to cloud if
 2   connection is lost. This is evident from the documentation:
 3   https://docs.particle.io/reference/firmware/photon/#automatic-mode*/
 4
 5  /** Defines */
 6  #define BUFLEN 50 // Bufferlength for the data from the api response
 7  #define MAX_ATTEMPTS 5 // To reconnect to WiFi
 8  #define MIN_INTENSITY 0.7 // Rain intensity in mm
 9  /** Sleep times are in seconds */
10  #define SLEEP_TIME_RECONNECT 60*2.5 // 2 mins 30 secs
11  #define SLEEP_TIME_MONITOR 60*60 // Check soilmoisture every hour
12  #define SLEEP_TIME_MONITOR_WATER 1 // Sleep for 1 sec, wake up, measure water
13  #define SLEEP_TIME_WAIT_FOR_RAIN 60*60 // Recheck if rain is coming
14  #define SLEEP_TIME_MOISTURECHANGE 60*10 // Check if rain has made moisture change every 10 minutes
```

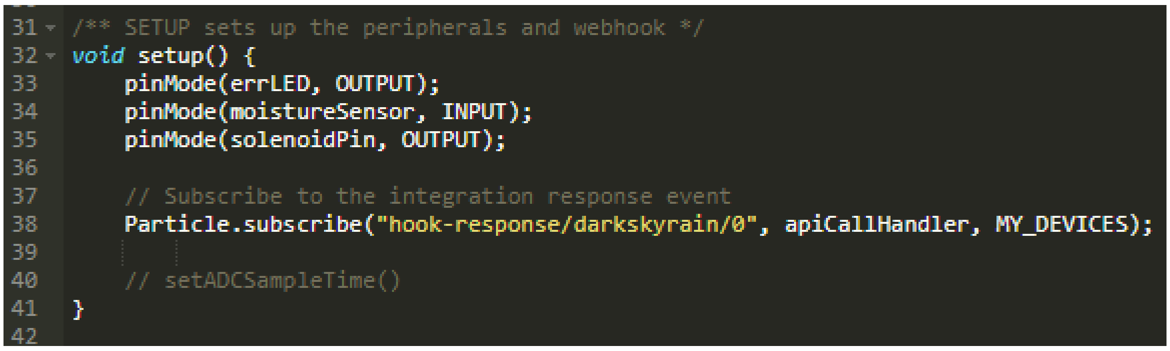
As can be seen from line 20, there is another analog input pin to be set up. The idea is to use a flow sensor, to measure flow, e.g. water output per time. While this measure is happening, one can measure the time, in which water is being poured on the plant. Multiplication of flow/time and time reveals the water volume poured on the plant. This can later be sent to the cloud graphing tool, so the user can benefit from the information. Depending on what is important for the user of the system, it is possible to make a financial plot of water usage.



```
16 /** PINS -> GPIO (IN/OUT) --- ANALOG (IN) */
17 int errLED = D7; // Visually sets a fault condition
18 int solenoidPin = D0; // Solenoid output pin for controlling the solenoid as ON/OFF
19 int moistureSensor = A0; // Analog input for the moisturesensor
20 int flowSensor = A1; // Flow sensor measures amount of water used. This value is used for publishing to user
21
22 /** GLOBALS for a more organized main loop */
23 int moisturePercentage; // The moisturevalue is handled as a percentage btw. 0 - 3.3VDC
24 unsigned int firstTime, lastTime, periodSecs; // for measuring time between periods
25 float intensity;
26 char * type;
27 unsigned int rainTime;
28 bool forecastIsRecieved;
29 float MINIMUM_MOISTURE_PCT = 30; // Minimum moisture level
```

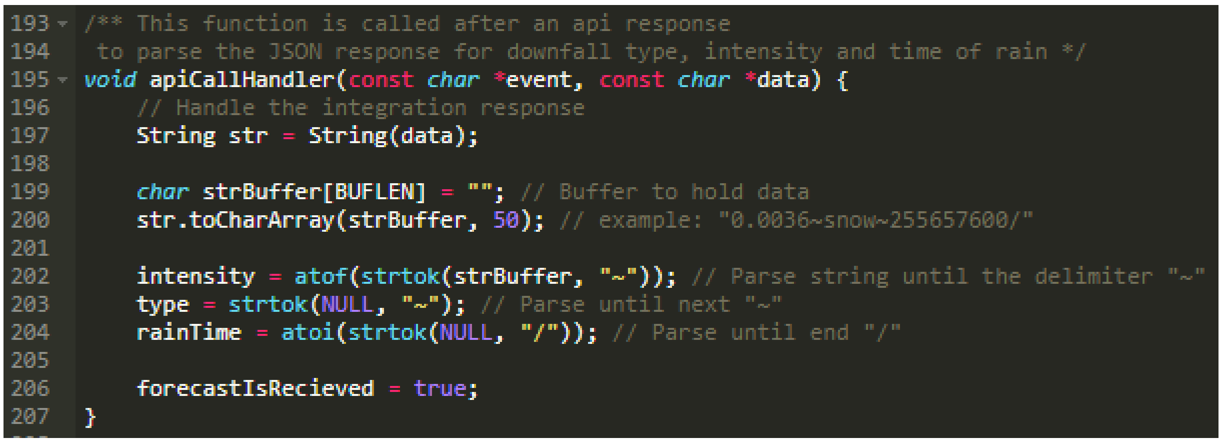
From line 23 – 29, we see partly global values, that are to be used in functions as well as the main loop, and a minimum percentage, that the moisture can get at in the plant soil.

In the setup, line 38, we subscribe to an integration response. The nested JSON response from the Darksky API is filtered using moustache.



```
31 /** SETUP sets up the peripherals and webhook */
32 void setup() {
33     pinMode(errLED, OUTPUT);
34     pinMode(moistureSensor, INPUT);
35     pinMode(solenoidPin, OUTPUT);
36
37     // Subscribe to the integration response event
38     Particle.subscribe("hook-response/darkskyrain/0", apiCallHandler, MY_DEVICES);
39
40     // setADCSampleTime()
41 }
42
```

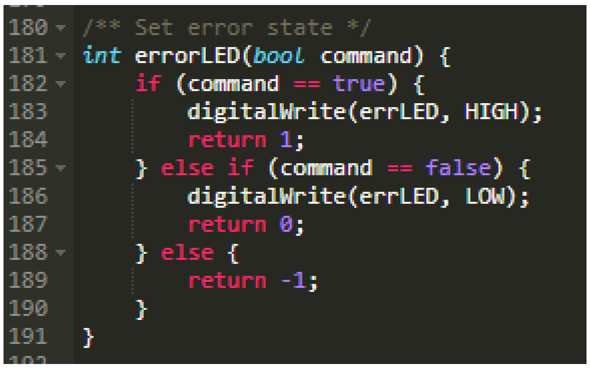
When the filtered response is returned, it calls the *apiCallHandler* function.



```
193 /** This function is called after an api response
194  to parse the JSON response for downfall type, intensity and time of rain */
195 void apiCallHandler(const char *event, const char *data) {
196     // Handle the integration response
197     String str = String(data);
198
199     char strBuffer[BUFLEN] = ""; // Buffer to hold data
200     str.toCharArray(strBuffer, 50); // example: "0.0036~snow~255657600/"
201
202     intensity = atof(strtok(strBuffer, "~")); // Parse string until the delimiter "~"
203     type = strtok(NULL, "~"); // Parse until next "~"
204     rainTime = atoi(strtok(NULL, "/")); // Parse until end "/"
205
206     forecastIsRecieved = true;
207 }
```

The filtered response, or *data*, is casted to a string type, and with *strtok* we extract the intensity of the data, which is the first part of the string, until we hit the delimiter, ~. When it hits the delimiter, it converts the char array to a float value with *atof.* The same goes for the type of precipitation and the time at which it will rain at its maximum, during the day.

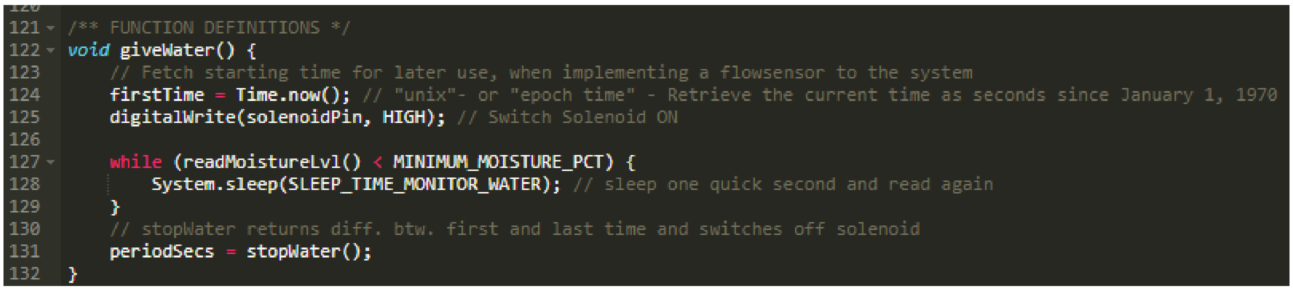
forecastIsRecieved = true is used in the main loop for error handling.



```
180 /** Set error state */
181 int errorLED(bool command) {
182     if (command == true) {
183         digitalWrite(errLED, HIGH);
184         return 1;
185     } else if (command == false) {
186         digitalWrite(errLED, LOW);
187         return 0;
188     } else {
189         return -1;
190     }
191 }
```

For now, as seen from line 181 – 191, the lost WiFi connection is only handled by a LED lighting blue.

The giveWater function, line 122 below, activates a mosfet, that powers the solenoid valve. When it opens, water pours out. The first thing is to remember the time *firstTime* since we opened the valve. This time format is also called Unix time.



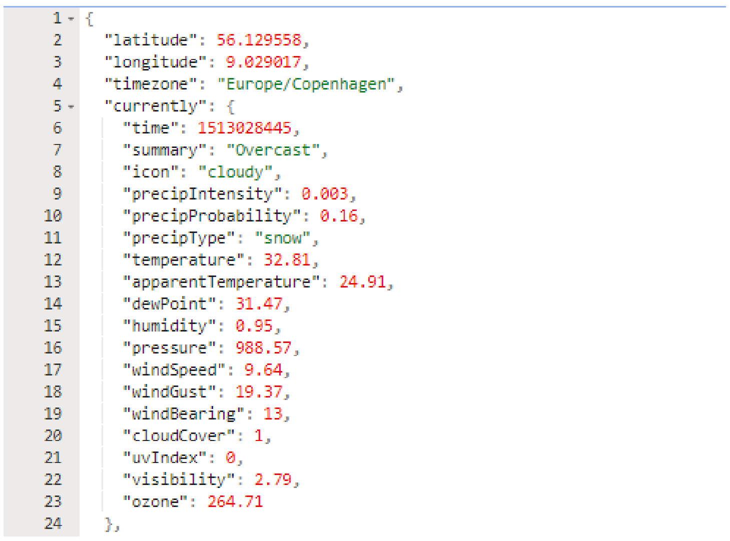
```
121 /** FUNCTION DEFINITIONS */
122 void giveWater() {
123     // Fetch starting time for later use, when implementing a flowsensor to the system
124     firstTime = Time.now(); // "unix"- or "epoch time" - Retrieve the current time as seconds since January 1, 1970
125     digitalWrite(solenoidPin, HIGH); // Switch Solenoid ON
126 
127     while (readMoistureLvl() < MINIMUM_MOISTURE_PCT) {
128         System.sleep(SLEEP_TIME_MONITOR_WATER); // sleep one quick second and read again
129     }
130     // stopWater returns diff. btw. first and last time and switches off solenoid
131     periodSecs = stopWater();
132 }
```

Webhook (Narenth)

We are using a webhook in the Particle platform as a webservice which connects to an only weather API that gives us the complete current weather situation for now and the coming time, for the desired location we choose.

The API we are using is called Dark Sky API which provides us with the weather data in a JSON format. It is possible to make 1000 API calls per day with this API, by using beyond this amount will it cost 0,0001$ for each API call. When creating an account at Dark Sky API, we will get a unique API key for use when we set it up with our particle platform. By using this API key, the Dark Sky system also keeps tracking of our usage and amount of API calls.

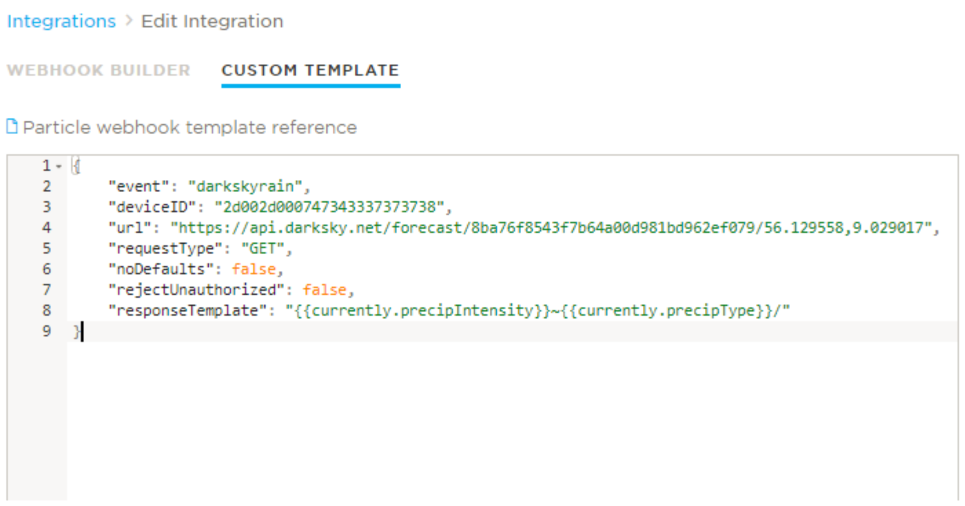
We can create and administrate webhooks on the online Particle console. And it is here we are setting up our webhook. Another option is to set up the webhook through the CLI (command line interface). When creating a new webhook, we should give it an event name, URL and request format. Every time we call the event name will it start the webhook. The URL is for the address of the JSON string and the request type is for whether we are “getting” or “posting” data, which in our case is “get” because we are getting weather data from the webhook.



```
{
  "latitude": 56.129558,
  "longitude": 9.029017,
  "timezone": "Europe/Copenhagen",
  "currently": {
    "time": 1513028445,
    "summary": "Overcast",
    "icon": "cloudy",
    "precipIntensity": 0.003,
    "precipProbability": 0.16,
    "precipType": "snow",
    "temperature": 32.81,
    "apparentTemperature": 24.91,
    "dewPoint": 31.47,
    "humidity": 0.95,
    "pressure": 988.57,
    "windSpeed": 9.64,
    "windGust": 19.37,
    "windBearing": 13,
    "cloudCover": 1,
    "uvIndex": 0,
    "visibility": 2.79,
    "ozone": 264.71
  },
```

Figure 8 – The image shows the data from our weather API URL. The data is in JSON format.

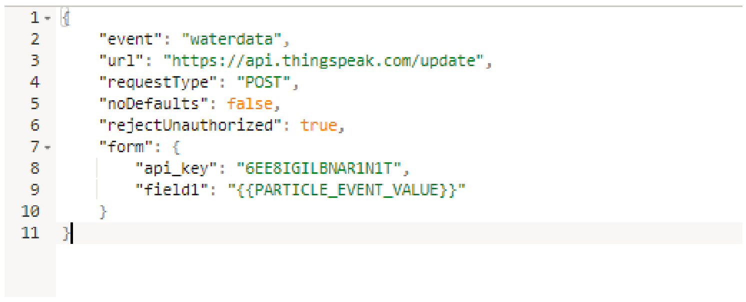
The data from the weather API contains a lot of information about the weather, but we are only interested in two values; precipitation intensity which shows us the intensity of the precipitation, and the precipitation type which tells us whether the precipitation is rain, snow, or sleet. Therefore, we need to filter out the relevant information from the JSON string. We will do this by using a response template on our webhook, where we are using mustache templates to fetch out the desired values.


Integrations > Edit Integration
WEBHOOK BUILDER
CUSTOM TEMPLATE
Particle webhook template reference

```
{
    "event": "darkskyrain",
    "deviceID": "2d002d000747343337373738",
    "url": "https://api.darksky.net/forecast/8ba76f8543f7b64a00d981bd962ef079/56.129558,9.029017",
    "requestType": "GET",
    "noDefaults": false,
    "rejectUnauthorized": false,
    "responseTemplate": "{{currently.precipIntensity}}~{{currently.precipType}}/"
}
```

Figure 9 - This is where we set up our webhook, with event name, URL and request type. We can also see how we have used mustache to get the desired values on our response template.

Another webhook we are using, is an API called ThingSpeak. ThingSpeak is an open IoT platform for collecting and analyzing data. We are using the API for collecting the watering data, so we can keep track of the water usage. The ThingSpeak API is created by Mathworks which gives us the possibility to analyze and visualize the data in Matlab. When dealing with this API we are using the request type "post" because we are sending information to the webhook. This API needs also unique API key before we can send information to the webhook.

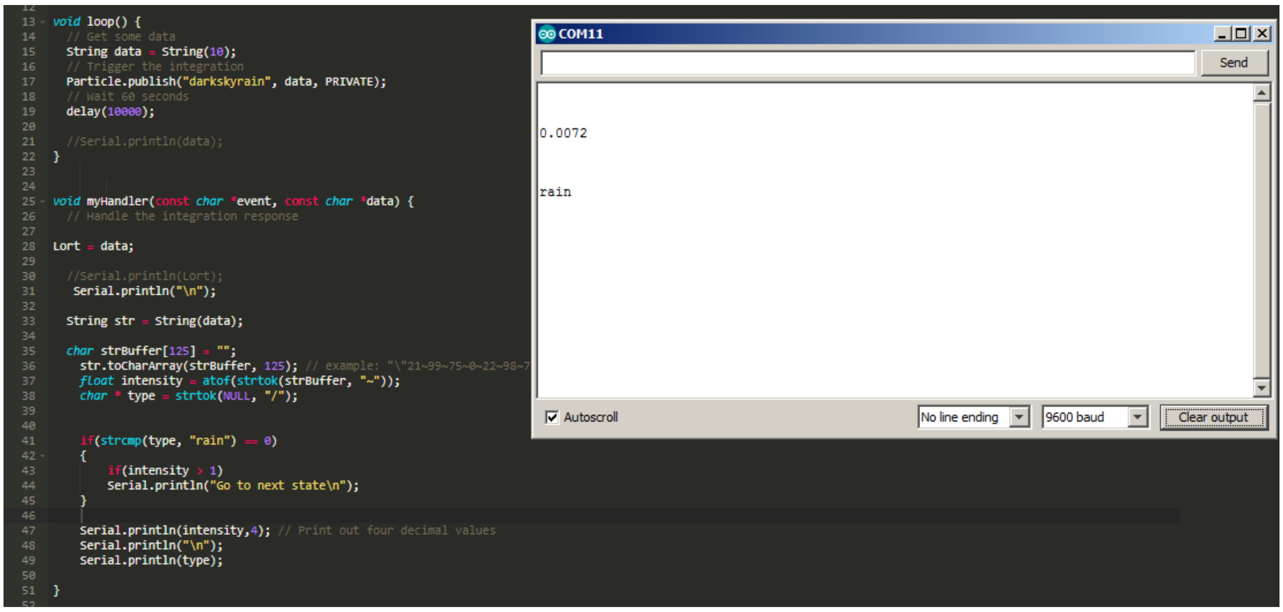


```
{
    "event": "waterdata",
    "url": "https://api.thingspeak.com/update",
    "requestType": "POST",
    "noDefaults": false,
    "rejectUnauthorized": true,
    "form": {
        "api_key": "6EE8IGILBNAR1N1T",
        "field1": "{{PARTICLE_EVENT_VALUE}}"
    }
}
```

Figure 10 - The image shows how we have set up our ThingSpeak API

Test/verification

We have managed to set up and receive information from our webhook. Furthermore, also implemented our moisture sensor and valve. But we have not been able to test the complete setup, because the system still is a prototype which is not capable of being tested outdoor, due to the low durability of the system. We have only managed to test the separate modules individually. We have documented our working webhook that receives weather data and fetches the relevant data (precipitation intensity and type):



```
void loop() {
  // Get some data
  String data = String(10);
  // Trigger the integration
  Particle.publish("darkskyrain", data, PRIVATE);
  // Wait 60 seconds
  delay(10000);

  //Serial.println(data);
}


void myHandler(const char *event, const char *data) {
  // Handle the integration response

Lort = data;

  //Serial.println(Lort);
   Serial.println("\n");

  String str = String(data);

  char strBuffer[125] = "";
    str.toCharArray(strBuffer, 125); // example: "\"21~99~75~0~22~98~7
    float intensity = atof(strtok(strBuffer, "~"));
    char * type = strtok(NULL, "/");


    if(strcmp(type, "rain") == 0)
    {
        if(intensity > 1)
        Serial.println("Go to next state\n");
    }

    Serial.println(intensity,4); // Print out four decimal values
    Serial.println("\n");
    Serial.println(type);

}
```

```
0.0072


rain
```

Figure 11 - The image shows the source code and our terminal where we can read that we will have 0,0072-millimeter rain.

As we can see in the above source code, are we subscribing and publishing to the webservice, afterwards are we printing out the received data. We have parsed the relevant data by using mustache at the JSON string. This test verifies that our webhook is correctly set up and that we are receiving the exact data.

Furthermore, have we tested our humidity sensor separate by placing it in some soil and measured the output.

Conclusion

Since the system testing would require finished hardware modules and protection from rain, when used outside under the rain, the complete system was not tested outside. As for most of the individual modules, such as pouring water and stopping that process, it is a matter of a simple GPIO pin being set HIGH or LOW. We see it as redundant testing. The system can successfully send a request to the Darksky API, and parse for the relevant information. The code for this project and intentions are now open sourced, for inspiration to other people, for making an improvement or what else would be beneficial. The requirements of the project were successfully fulfilled, as can be seen in the requirement analysis.