



Internet-of-Things Plant Monitoring and Management System

FULL REPORT

by

Sean Dunbar

This Preliminary Report is submitted in partial fulfilment of the requirements of the B.Sc.
Honours Degree in Networking Applications & Services (DT080B) of the
Dublin Institute of Technology

February 18th, 2017
Supervisor: Frank Duignan
School of Electrical and Electronic Engineering

Declaration

I, the undersigned, declare that this report is entirely my own written work, except where otherwise accredited, and that it has not been submitted for a degree or award to any other university of institution.

Signed:

A handwritten signature in black ink, appearing to read "Sean Dunbar". The signature is written in a cursive, flowing style.

Date:

18/02/2017

Acknowledgements

I would like to thank Frank Duignan primarily for providing the assistance I needed with the electronics aspects of this project, mainly the LDR usage. Without this help these areas would have been seriously lacking. I also want to thank him for taking the time to meet with me over the project and discuss various parts of it to make it the best I could.

Abstract

This report attempts to develop a small, IoT-style system for measuring, monitoring and managing a plant pot. This would be achieved by identifying the requirements of the system and designing an implementation. This involves comparing multiple options for each individual component and discussing the reason behind making choices in each case. Finally, the security aspect will be reviewed and the viability of powering the system by solar panels will be determined.

The project was ultimately successful despite some setbacks and logistics issues. The system appropriately measures and reports data to a time-series database. These entries are then transformed into a graph viewable via a web interface.

Further research and refinement could be made following this, such as reducing cost and size, as well as constructing a weatherproof, solar-powered enclosure for the entire system.

Table of Contents

Introduction.....	1
Objectives and Approach	1
Identification of a suitable Single Board Computer (SBC) and development environment.....	1
Identification of a suitable software components	1
Select, source and interface with sensors	1
Select, source and interface with pump system	1
Develop SBC application in tandem with software components.....	2
Investigate security aspects of chosen software solution	2
Test and analyse performance	2
Background of Technologies Chosen	3
Hardware & Interfacing.....	3
ESP8266/NodeMCU	3
Comparison with other technologies	4
Arduino Nano.....	4
Technical Specs	4
Raspberry Pi Zero W	5
Technical Specs	5
WeMos D1 mini.....	5
Technical Specs	5
I ² C Moisture Sensor	6
Comparison with other technologies	6
I ² C Standard	6
Pump	8
Light Detection	8
Data Storage.....	9
InfluxDB.....	9
Why a TSDB?	9
Comparison	9
Data Presentation.....	10
Grafana	10
Comparison	10
Chronograf.....	10
Kibana	10
Custom Application	10
Operating System	11
UnRAID	11
Comparison	11
Docker.....	11
Comparison	11
Testing.....	12
InfluxDB/Grafana.....	12
I ² C Moisture Sensor.....	16
Design	19
Hardware Design	19

Software Design.....	21
Implementation.....	22
Sending Data to InfluxDB	22
Testing time spent awake.....	25
Displaying data in Grafana	26
Code Refinement.....	27
Pump	27
System Performance	28
Performance Criteria	28
Evaluation Procedures.....	28
Sensor Reliability	28
Power Usage	28
Test Results	28
Sensor Reliability	28
Power Usage	29
Solar Power Viability	29
Testing Methods	29
Test Results	30
Security	30
InfluxDB.....	30
Grafana	30
External Access	31
LetsEncrypt.....	31
Nginx	31
Implementation	31
Discussion	33
Conclusions.....	33
Future Alternatives	33

Figure 1, LoLin NodeMCU [2].....	3
Figure 2, Arduino Nano Technical Specs [6]	4
Figure 3, Raspberry Pi Zero W Technical Specs [7]	5
Figure 4, WeMos D1 Mini specs [8]	5
Figure 5, I2C Data Transfer example [10]	7
Figure 6, I2C Data Transfer with 7-bit addresses.....	7
Figure 7, I2C Data Transfer with 7-bit addresses, Repeated START [10].....	7
Figure 8, LED Diagram	8
Figure 9, LDR Diagram	8
Figure 10, InfluxDB Container Creation	12
Figure 11, Grafana Container Creation.....	13
Figure 12, Grafana Data Source Creation	13
Figure 13, Crude testing setup, not pictured: Plant Pot and NodeMCU.	16
Figure 14, Example Output from Example Script	17
Figure 15, Testing of LDR	18
Figure 16, NodeMCU V1.0 pin definition [23]	19
Figure 17, Breadboard Layout.....	20
Figure 18, Schematic Layout.....	20
Figure 19, Basic Flowchart of Software	21
Figure 20, UDP stats from InfluxDB Web Interface	24
Figure 21, Diagnostic info from device	24
Figure 22, Final Grafana Dashboard.....	26
Figure 23, USB / DC / Solar Lithium Ion/Polymer charger - v2 [26]	29

Introduction

The objective of this report is to research, design and implement a self-contained plant monitoring system, wherein the system monitors the soil moisture level, temperature and light level. It then outputs these statistics to a database, from which a graph is generated based on the supplied values and displayed over the web. The system will also be capable of self-watering, where a pump will be activated which will draw water from a reservoir, this will be triggered by the reading from the soil moisture sensor.

Objectives and Approach

The objective of this project is to develop a sensor platform that will report soil moisture, temperature and light level back to a central server, which will display these statistics in a graph over a web interface. This will involve determining what single board computer (SBC) to use, along with sensors, communication methods, data storage methods and methods for displaying the stored information. The SBC will also be able to drive a solenoid for watering the plant.

The key objectives that must be achieved are as follows:

Identification of a suitable Single Board Computer (SBC) and development environment

This will involve determining which SBC to use and what development environment to use from the choices available. This will involve comparing features between a handful of SBCs, namely IO features and price. Then, a development environment must be chosen. Depending on the SBC there might only be one choice available, but for those with a choice, the environment or language with most familiarity or least issues will be chosen.

Identification of a suitable software components

This will require that a server-side data storage mechanism is decided on and implemented, this must be accessible by both the SBC and the graphing web service. This will be chosen based on ease of setup, ease of use and integration with graphing service. A graphing web service must also be decided on and implemented, this will follow the same criteria as the database for selection.

Select, source and interface with sensors

This will involve finding sensors that are capable of interfacing with the SBC and have their data pulled as necessary. The sensor or sensor package must be capable of at least measuring temperature, light level and soil moisture level. These will be chosen based on ease of use, complexity of wiring (A simple all-in-one solution is more desirable) and price.

Select, source and interface with pump system

This will require finding a suitable pump and method of controlling it. The pump must be small, quiet and suitable for deployment in multiple environments. The method for activating the pump will be chosen based on suitability, reliability and price.

Develop SBC application in tandem with software components

This will be the actual design of the software for the SBC as well as the setup of the server-side components, the database and graphing web service. The software will be designed to do the following:

- Wake up every minute
- Pull data from the sensors
- Send collected data to database
- Turn on pump for X seconds (goal is 100ml of water per activation)
- Return to sleep mode

The graphing service will need to be set up with appropriate queries to best represent the data.

Investigate security aspects of chosen software solution

This will involve investigating some aspects of security with the software, these will mainly focus on the database and graphing server and the data in transit from the SBC. This step will involve setting up hardening the server where possible, so that the SBC can still transmit data to the server.

Test and analyse performance

This will involve testing and analysing performance, along with determining viability of solar power as an option. This will mean testing all the sensors for both accuracy and precision and testing the pump is operating within reasonable limits for both ml of water pumped and power usage. The power usage of the system with and without the pump will also be measured, to use for determining if solar power is a viable option.

Background of Technologies Chosen

The first steps in this project were to determine suitable hardware and software. The hardware implementation needed to be capable of several things:

1. Connect to a network
2. Read in sensor data
3. Send this data to a database
4. Activate a pump on a trigger condition
5. Sleep or delay for a set time

Hardware & Interfacing

ESP8266/NodeMCU

The ESP8266 is a miniscule, low-cost System on a Chip (SOC) with integrated WiFi. It features an 802.11b/g/n capabilities and is directly programmable, but can also interface with other SBCs such as an Arduino. [1]



Figure 1, LoLin NodeMCU [2]

The NodeMCU is a Microcontroller Platform which features an ESP8266 at its core. It is designed to be an easily interfaceable development platform for the ESP8266, providing easier access to GPIO, I2C, PWM and One-Wire. [3] It is scriptable via Lua, but is also able to be used with MicroPython [4] and the Arduino IDE [5].

Comparison with other technologies

There are 3 main competing choices with the NodeMCU for this project: the Arduino Nano, Raspberry Pi Zero and the Wemos D1.

Arduino Nano

The Arduino Nano is another popular small SBC that is popular among hobbyists. Despite its price being lower than the NodeMCU, it was ultimately not chosen due to its lack of WiFi which was crucial to this project.

Technical Specs

Microcontroller	ATmega328
Architecture	AVR
Operating Voltage	5V
Flash Memory	32KB, 2KB used by bootloader
SRAM	2KB
Clock Speed	16MHz
Analog I/O Pins	8
EEPROM	1KB
DC Current per I/O Pins	40mA
Input Voltage	7-12V
Digital I/O Pins	22
PWM Output	6
Power Consumption	19mA
PCB Size	18 x 45mm
Weight	7g
Product Code	A000005

Figure 2, Arduino Nano Technical Specs [6]

Raspberry Pi Zero W

The Raspberry Pi Zero is another popular SBC, being much more powerful than any other SBC considered. Ultimately it was decided against due to lack of availability, larger form factor and power usage.

Technical Specs

Dimensions	65mm x 30mm x 5mm
SoC	Broadcom BCM2835
CPU	ARM11 at 1GHz
RAM	512MB
Wireless	2.4GHz 802.11n WLAN
Bluetooth	4.1 and LE
Power	5V via micro USB
Video and Audio	Mini-HDMI
Storage	MicroSD Card
Output	Micro Usb
GPIO	40 pings
Pins	Run mode, RCA Composite

Figure 3, Raspberry Pi Zero W Technical Specs [7]

WeMos D1 mini

This SBC is also based on the ESP8266, just like the NodeMCU, and it is also able to use the Arduino IDE and the NodeMCU's Lua firmware. It is cheaper than the NodeMCU and offers the pinouts needed to use the sensors. However, it was not chosen as there was some more familiarity with the NodeMCU, the NodeMCU appeared to be the more mature product, and a NodeMCU was available on hand for testing before the WeMos D1 mini would have been able to arrive from China. Had there been enough time to have on delivered, or if there was no NodeMCU to test early into the project, this SBC would have been chosen.

Technical Specs

Microcontroller	ESP-8266EX
Operating Voltage	3.3V
Flash	4MB
Clock Speed	80MHz/160MHz
Analog I/O Pins	1
PCB Size	34.2mm x 25.6mm
Weight	10g

Figure 4, WeMos D1 Mini specs [8]

I²C Moisture Sensor

The chosen moisture sensor was the I²C Soil Moisture Sensor by Catnip Electronics. It is an open source hardware solution that provides measurement of soil moisture level, light level and temperature. It uses the I²C protocol for communication. [9]

The sensor is powered by 3.3v – 5v, it will be used at 3.3v to use as little power as possible. It is measured at 7.8mA when measuring at 3.3v, compared to 14mA at 5v. [9]

The moisture sensor and light sensor are “relative” readings, meaning the info is not some absolute value and only makes sense in relation to other measurements. This means a baseline must be found for the moisture reading to activate the pump. The temperature sensor returns an absolute value, meaning it returns a value that makes sense by itself. In this case, it's in tenths of a degree Celsius, where a reading of 123 equates to 12.3°C [9]

The version of the sensor used was the Rugged version, this came with wires pre-soldered and the PCB itself coated in epoxy and rubber. This was chosen so that the sensor could be exposed to weather and be used outside with minimal issue.

Comparison with other technologies

The DHT11 sensor is a non-I²C alternative to fill in the temperature niche, with the added measurement of humidity. It uses its own proprietary protocol for communication. It would then be possible to use a cheap capacitive sensor such as the hundreds available on AliExpress. An LDR can then be used to measure the light levels.

The reason this approach was not taken was two-fold, first, the NodeMCU only has one analogue input, which means using something like a multiplexer to use the capacitive sensor and LDR at the same time. The second reason was the I²C Moisture Sensor gave the opportunity to learn and work with the I²C standard.

This approach would have one huge advantage though, price. The sensor stick itself is €22 for the rugged version, while the DHT11 can be gotten for around \$1, the capacitive sensor for less than \$1 and the LDR for less than \$1 also.

I²C Standard

The I²C standard provides a communication protocol for serial communication between devices. It uses two pins for communication, SCL and SDA. Communications start with a pre-amble and end with a post-amble. An addressing scheme is used, 7-bits long. I²C uses a master and slave system, where the master system sets the clock rate via SCL. Unlike SPI, I²C can support multiple masters interconnected to multiple slaves. [10]

I²C transmits one bit of data for each clock pulse. Each command starts with what's known as a START condition and ends with a STOP. For each of these the clock must be high, it is the transition of SDA from high to low that determines START and STOP. [10]

Data is sent in 8-bit bursts, which must then be followed by an ACK bit. A clock pulse must be generated for each bit. If the slave does not send an ACK it means that transmission has ended or the device is not ready for transmission, it is then up to the master to generate the STOP condition or repeated START. [10]

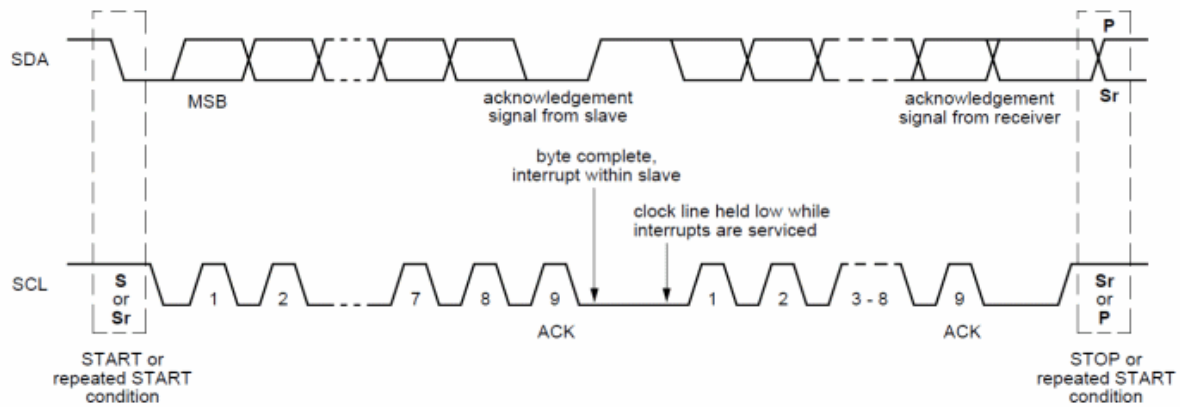


Figure 5, I2C Data Transfer example [10]

When using 7-bit addresses, the method used is slightly different. Communication starts with the START condition as usual, which is then followed by the address bits, plus a R/W bit, finally followed by an ACK bit. R/W bit is used to determine the direction of data, whether the slave is being written to or read from. When communication is finished, a STOP condition is generated as it is above. [10]

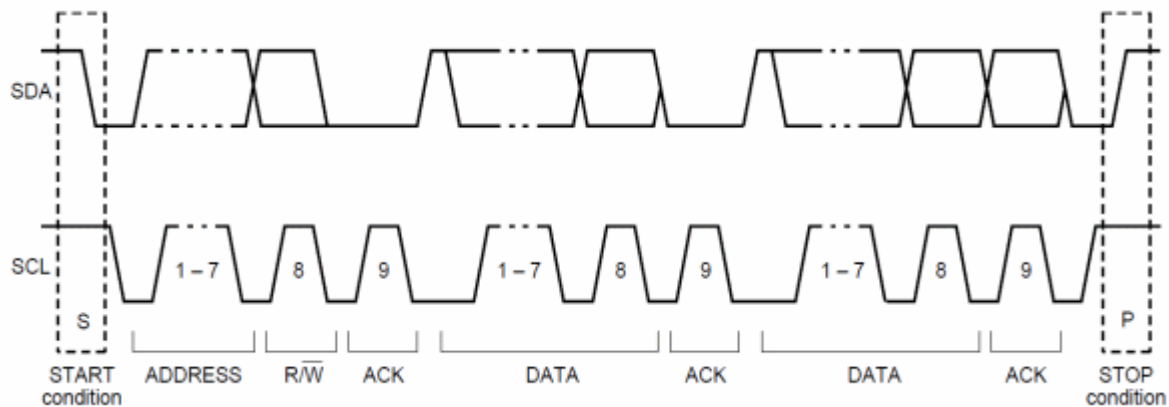


Figure 6, I2C Data Transfer with 7-bit addresses

When writing, the slave acknowledges all bytes sent; when the master is finished, it sends the STOP condition. When reading however, the master acknowledges all data bytes read instead, but of course still sends the STOP condition. If the Master needs to change from reading to writing, such as writing a register address to a slave and reading from it, it will send the repeated START condition instead of STOP and then start communicating again. [10]

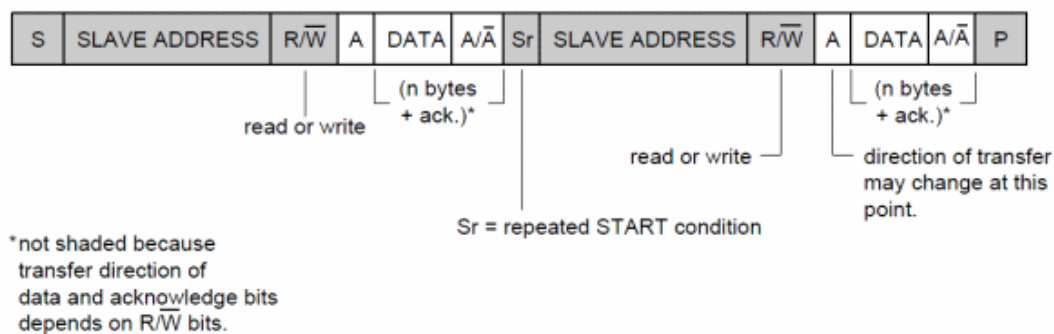


Figure 7, I2C Data Transfer with 7-bit addresses, Repeated START [10]

Pump

Unfortunately, due to logistics issues, an appropriate pump was unable to be sourced. The pump was intended to be a small submersible pump, placed into a small water tank. This pump would then be activated when necessary to water the plant once moisture goes below a predefined threshold.

As an alternative to this, to illustrate the effect of enabling a device under a trigger condition, an LED was enabled in its place. This was achieved with a simple design:

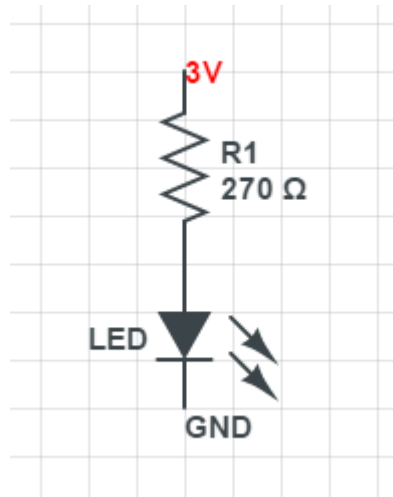


Figure 8, LED Diagram

Light Detection

During later testing, it was discovered the LDR on the sensor stick was obscured by the waterproof coating. This meant that a change was needed in design and a replacement had to be found. With the help of Frank Duignan, a simple LDR + resistor setup was devised as follows:

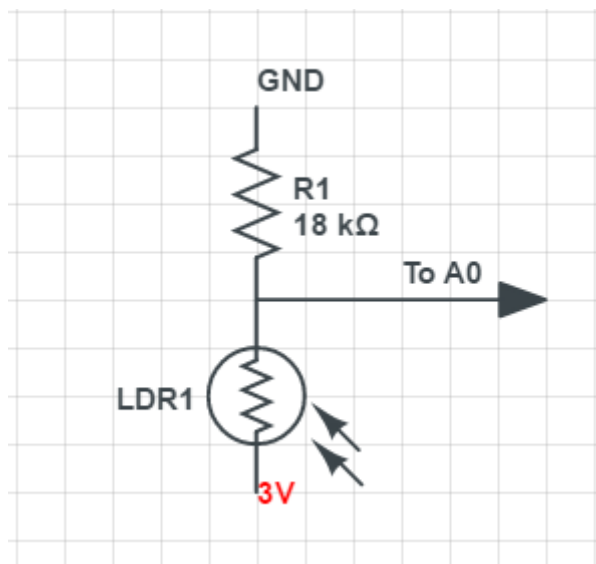


Figure 9, LDR Diagram

Data Storage

InfluxDB

InfluxDB is a Time-Series Database (TSDB) with a query language like Structured Query Language (SQL). As a TSDB, it is designed for storing metrics and events from sensors and other data sources. [11]

It features a simple HTTP API for access, which is written to and queried with GET requests. Data is stored in a chosen Database using several key/value pairs, notably measurement, fields and tags. Measurement is used as an identifier for the entry, describing the type of data the entry contains. Fields are used to store data in identified fields, e.g. storing temperature data in a field with the key 'Temperature'. Tags are like fields, but are optional and indexed. InfluxDB encourages use of these, as they are indexed and thus much faster to query than fields. [12]

Being a TSDB, all entries have a timestamp, if the timestamp value is left out of the entry, the entry will use the servers local Unix Epoch timestamp. [13]

Why a TSDB?

A time series database was chosen for this project based on the intended application. Typical Relational databases such as PostgreSQL or MySQL store data in "tables". These tables share some sort of relationship between the values and are identified with a primary key. There was also the choice of a NoSQL database such as Redis or MongoDB, which do not store data in tables but as simple files.

These types of databases work for most types of data, it would have been possible to use them in this case, however TSDBs offer a few advantages that are well suited to this project:

- Built-in operations to transform or aggregate a series
- Customisable retention policies (for example, to discard any data over 72 hours old)
- Out-of-the-box compatibility with Grafana and similar presentation software

For these reasons and for the ability to learn a new technology that is gaining rapid popularity [14], InfluxDB was chosen.

Comparison

There are a handful of other TSDBs that can be used with Grafana, such as openTSDB.

OpenTSDB, like InfluxDB, is a scalable, distributed Time-Series Database with an HTTP API. [15] The main reason it was not chosen was due ease of use of InfluxDB. OpenTSDB does not have something like InfluxDB's UDP Line protocol, which trivialises sending data from a device that may not have perfect connectivity most of the time.

Data Presentation

Grafana

Grafana is an open source tool for metric analytics and visualisation. It interfaces with many “Data Sources”, such as InfluxDB, Graphite and OpenTSDB. It uses a user-defined query to aggregate and display data from the selected data sources. Grafana has support for “Organization” groups, which group users by what dashboards and data they can access. for the purposes of this project only one organization will be used. [16]

Grafana is accessible via a web interface, by default on port 3000. Data sources are defined through this interface and dashboards are created for them. Each dashboard is constructed from several different panels, organized into rows.

Panels are usually Graphs, which graph one or more series of data using a set query and timeframe, but can also be “Singlestat”, which displays a single series of data into a single number, such as the minimum or average, “Dashlist”, which allows for linking to other dashboards within Grafana, “Table”, which displays a series in a table instead of a graph, and “Text”, which simply displays entered text. [16]

Comparison

There are 3 alternative approaches to this aspect, Chronograf, Kibana and a custom-built user application.

Chronograf

Chronograf is another open source monitoring tool, very similar to Grafana. This was created by InfluxData, the same people who created InfluxDB. So why choose Grafana over Chronograf? Functionally and visually they are very similar, along with the ability to set up queries, so the reason is mainly documentation, while Chronograf is well documented by InfluxData, Grafana has a vast amount of user documentation.

Another main reason is to distance the visualisation from the data storage backend. Chronograf is designed to be more tightly integrated around InfluxDB and the “TICK” stack (Telegraf, InfluxDB, Chronograf, Kapacitor).

Kibana

Kibana is part of the “ELK” software stack (ElasticSearch, LogStash, Kibana). Again, it performs very similar functions to Grafana and Chronograph, but is designed more for use in the ELK stack, like how Chronograf is designed for use in the TICK stack. For the same reasons as Chronograf, Kibana was decided against in favour of Grafana

Custom Application

This would require developing a web frontend which takes in the queries and transforms then into a graph. It was felt that this would be far too difficult to implement given the level of skill required.

Operating System

UnRAID

UnRAID is a commercial Linux Distribution designed for servers. It is intended to be used to create a software analogue to Redundant Array of Independent Disks (RAID). It accomplishes this by using 1/2 Parity disks to store information for any number of data disks. Data is not striped or mirrored across drives, meaning read and write performance are dependent on the drive being accessed at the time. [17]

UnRAID also provides a simple WebGUI for creating and maintaining Docker containers, which is how it will be used in this project. The reason for this is that an UnRAID server was already available at the start of the project and there was a lot of familiarity with it.

Comparison

Any operating system would be a suitable choice here, as InfluxDB and Grafana run on Windows and Linux systems. The reason UnRAID was chosen is because it was available, is familiar and offers an easy way to deploy and maintain Docker containers.

Docker

Docker provides a way to deploy applications independent of the Operating System or hardware beneath it. It approaches this by using “containers”, pre-made images containing the code/application, runtime, system tools, libraries and settings. These containers are sandboxed when deployed and OS agnostic, allowing for the same container image to be used on any OS if it runs the Docker service. [18]

Docker containers are not the same as Virtual Machines, as they do not contain an entire operating system, however, Docker can be used on a Virtual Machine.

Comparison

The primary alternative to using Docker containers is to install these applications directly to the base operating system. This was decided against for several reasons, foremost the simplicity of installation. Docker Containers are also easier to migrate, as one can back up the configuration data (stored all in one folder) and move it painlessly to another install of the same container image.

Testing

InfluxDB/Grafana

An InfluxDB Docker container and a Grafana Docker container were set up on a home server. These Docker containers were installed on an UnRAID server as follows:

The screenshot shows the 'Add Container' interface in UnRAID. The 'Template' dropdown is set to 'Influxdb'. The 'Name' field contains 'Influxdb'. The 'Overview' section shows the 'Configuration' tab selected, displaying 'Container Volumes' as '/var/lib/influxdb Persistent data storage location' and 'Container Ports' as '8083 Web admin interface' and '8086 HTTP API port'. The 'Repository' field is set to 'influxdb:latest'. The 'Network Type' is set to 'Bridge'. The 'Privileged' checkbox is unchecked. The 'Host Port 1' is set to '8083' with 'Container Port: 8083'. The 'Host Port 2' is set to '8086' with 'Container Port: 8086'. The 'Host Path 1' is set to '/mnt/user/appdata/influxdb' with 'Container Path: /var/lib/influxdb'. There are 'Edit' and 'Remove' buttons for each port and path. At the bottom, there are 'Apply' and 'Done' buttons.

Figure 10, InfluxDB Container Creation

This creates a Docker container using the official InfluxDB image, with the ports 8083 and 8086 being bound to it. 8083 is intended to provide a Web GUI to the database, but is deprecated as of 1.1 and disabled by default [19], 8086 is the port used by other services for querying or writing to the database. By default, it is unsecured any anyone can access it.

Add Container Basic View

Template: my-Grafana

Name: Grafana

Overview: Grafana is an open source, feature rich metrics dashboard and graph editor for Graphite, Elasticsearch, OpenTSDB, Prometheus and InfluxDB.

Configuration

Container Volumes: /var/lib/grafana Persistent data storage location

Container Ports: 3000 Web admin interface

Environment Variables: GF_SERVER_ROOT_URL The url to which you will be navigating to get to the grafana dashboard. Typically your ip or hostname. GF_SECURITY_ADMIN_PASSWORD Your password to use with the admin user. The default is user admin with password admin.

CLICK ADVANCED VIEW AND SET THE ENVIRONMENT VARIABLES

After the docker container is running check out the datasource documentation at <http://docs.grafana.org/datasources/overview/> and then the getting started guide at <http://docs.grafana.org/guides/gettingstarted/>. A convenient datasource would be InfluxDB, which can also be found in the community apps.

Repository: grafana/grafana

Network Type: Bridge

Privileged: OFF

Host Port 1: 3000

Host Path 1: /var/lib/grafana

Key 1: http://192.168.0.11

Key 2: admin

Buttons: Apply Done

Figure 11, Grafana Container Creation

Grafana is set up through the same interface, it uses the official Grafana image and is bound to port 3000, which provides the graphing service aspect of this project. It has 2 keys passed into it, one is the IP address or domain name of the server installed on and the other is the default admin password. For the purposes of testing this was left at default, but will be explored further in the security testing part of this report.

Edit data source

Name: testing

Type: InfluxDB

Http settings

Url: http://192.168.0.11:8086

Access: direct

Http Auth

Basic Auth: ☐ With Credentials: ☐

InfluxDB Details

Database: test

User: root Password: ****

Default group by time: example

Buttons: Save & Test Delete Cancel

Figure 12, Grafana Data Source Creation

Once Grafana is installed and logged into, the first step is to add a data source, as seen on the left. The Data Source is given a name, testing in this case, and a type, InfluxDB in this case.

The URL of the database is then given. Because this is a Docker container, we cannot use localhost and must give the full IP Address of the server.

Finally, the database is selected, Grafana automatically fills in the default user of root:root, but InfluxDB by default has no authentication, so this is not used.

A simple script was written in Python to send random data within set ranges to the database every minute. This was intended to simulate the SBC reading the sensors and sending data. The script uses the InfluxDBClient class from the influxdb package. The method InfluxDBClient.write_points() is used, as this allows writing to multiple time series names [20].

```
import time
import random

from influxdb import InfluxDBClient
client = InfluxDBClient('192.168.0.11', 8086, '', '', 'test')
client.create_database('test')
```

The modules time and random are imported for logging to console when data is posted and for generating the data later. The IP address, port, username, password and database name are passed in to the InfluxDBClient object. create_database() is used to create the test database we are storing this data in, if it does not exist.

```
while True:
    json_body = [
        {
            "measurement": "Temperature",
            "fields": {
                "value": round(random.uniform(10.0, 25.0), 1)
            }
        },
        {
            "measurement": "Moisture",
            "fields": {
                "value": random.randrange(100, 400, 10)
            }
        }
    ]
    client.write_points(json_body)
    print("Posted at " + time.strftime("%Y-%m-%d %H:%M:%S",\
        time.gmtime()))
    time.sleep(60)
```

Inside a never-ending loop, we create a list of dictionaries, representing JSON. Inside this list, we store two dicts, both with the keys 'measurement' and 'fields'. Measurement is used to identify what this data represents, while fields contains another dict, representing what fields the data has. In this case, only value is used, but time can also be set here. If not set here, the server will use local time for the data point.

With the data successfully stored in the "test" database, a new dashboard is created in Grafana connected to this. Two graphs are then created to display the data. They use the following queries

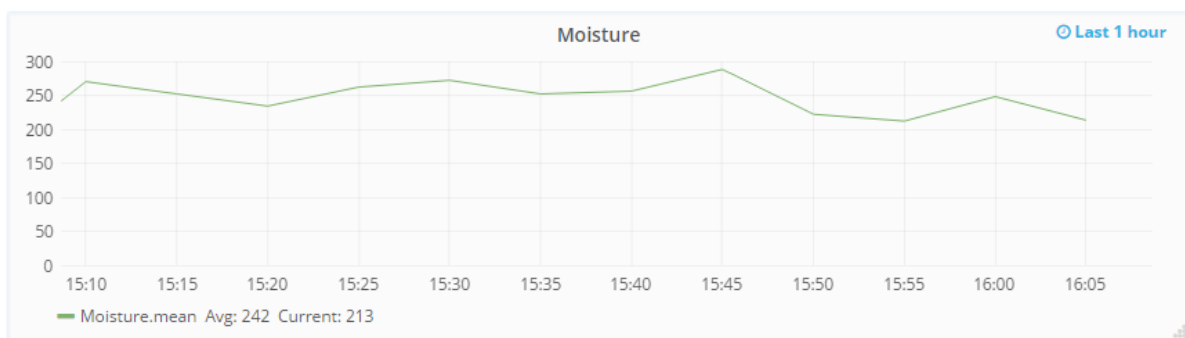
▼ A	FROM	default	Moisture	WHERE	+	
	SELECT	field (value)	mean ()	+		
	GROUP BY	time (5m)	fill (0)	+		
	ALIAS BY	Naming pattern			Format as	Time series ▼

This results in the query:

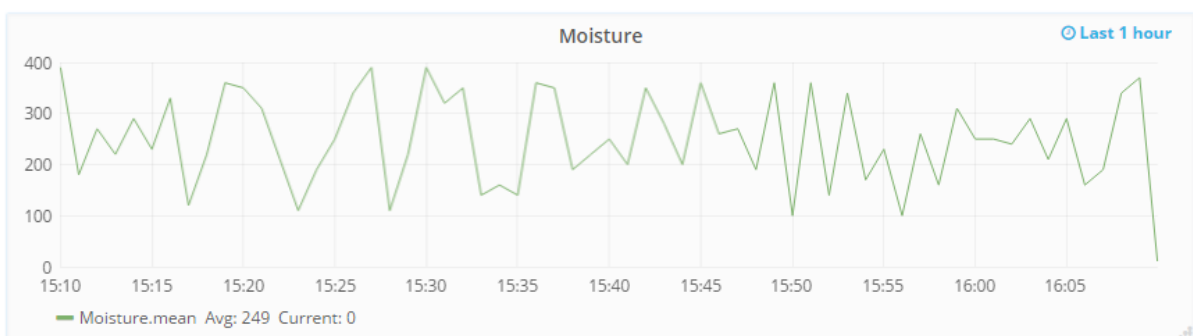
```
SELECT mean("value") FROM "Moisture" WHERE $timeFilter GROUP BY
time(5m) fill(0)
```

This query plots the mean value of the last 5 minutes' worth of Moisture data. Fill(0) means it will start the graph at the value 0 instead of auto-fitting the graph to the data.

This results in the following graph after 1 hour of data has been received:



To illustrate the effect of the time() grouping, this is the same data with time(1m):



I²C Moisture Sensor

The Sensor Stick was connected to the NodeMCU via a breadboard. Due to the breadboards imperfections, the NodeMCU will not fit directly into it. The 3.3v pin is connected to the VCC of the sensor stick and the SDA/SCL pins are connected for I²C to function.

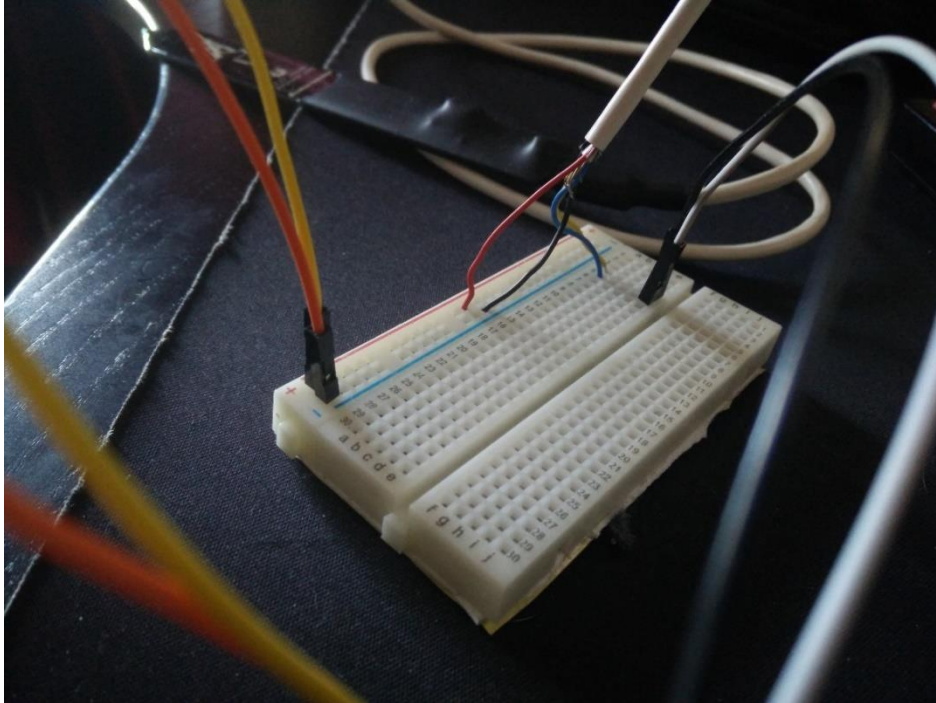


Figure 13, Crude testing setup, not pictured: Plant Pot and NodeMCU.

The NodeMCU is then connected to the computer via USB. The example script from the I2CSoilMoistureSensor library is uploaded to the NodeMCU via the Arduino IDE.

The example script, available from Apollon77 on the GitHub repository for the I2CSoilMoistureSensor Library [21], sets up the Sensor and prints out the address and firmware version to serial. After this, it enters a loop, where it prints out the soil moisture capacitance, temperature and light level. It then sleeps for a moment before repeating again. The sleep command drastically lowers the power usage of the sensor.

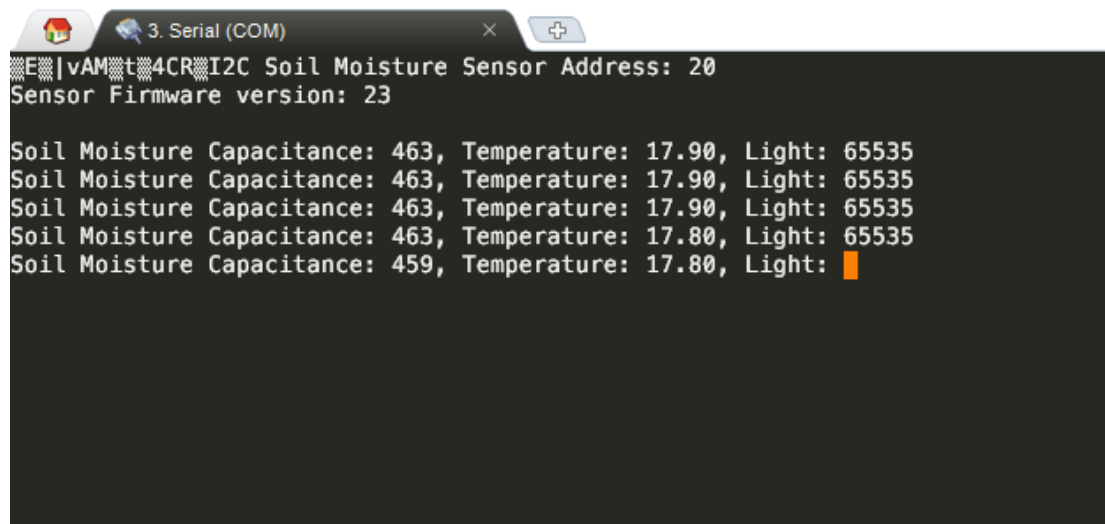


Figure 14, Example Output from Example Script

This is some example output when the sensor is placed in a plant pot, using the example script from the I2CMoistureSensor Arduino library. As can be seen, there is some slight variance with the capacitance reading.

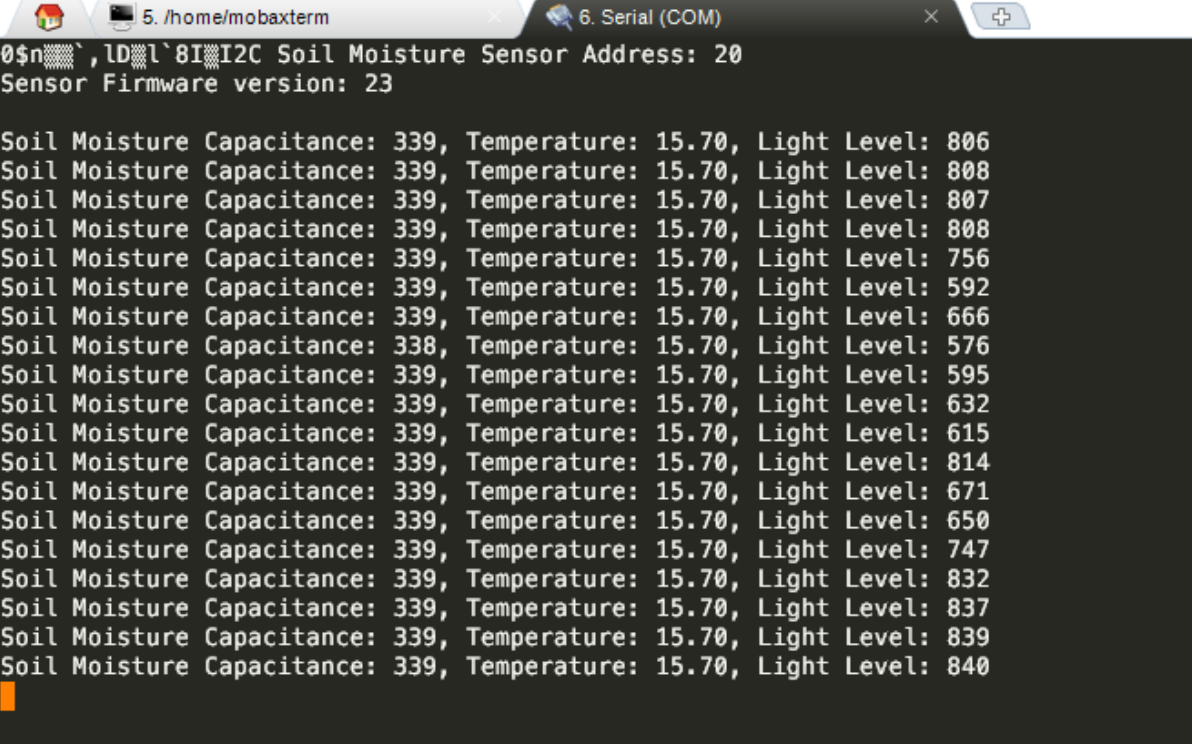
During testing, it was also found that the light sensor is obscured in the rugged version, attempts were made to contact the manufacturer to discover if this was a defective sensor or if it was in fact blocked. There was no response, so an alternative implementation had to be found.

It was decided to use a simple LDR combined with the single Analog pin to measure light level, this was set up on the breadboard following the design outlined in the previous section.

A modification was then made to the test script, replacing the built-in light sensor method with a simple analogue read:

```
Serial.print(", Light Level: ");  
Serial.println(analogRead(A0));
```

Fortunately, this worked perfectly, although the range was restricted to a 10-bit number (0 to 1023), this was more than adequate.



The screenshot shows a terminal window with two tabs: '5. /home/mobaxterm' and '6. Serial (COM)'. The active tab displays the following text:

```

0$n` ,lD`l`8I`I2C Soil Moisture Sensor Address: 20
Sensor Firmware version: 23

Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 806
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 808
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 807
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 808
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 756
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 592
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 666
Soil Moisture Capacitance: 338, Temperature: 15.70, Light Level: 576
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 595
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 632
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 615
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 814
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 671
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 650
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 747
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 832
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 837
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 839
Soil Moisture Capacitance: 339, Temperature: 15.70, Light Level: 840

```

Figure 15, Testing of LDR

It was then necessary to test the deep sleep function of the NodeMCU. This requires D0 to be wired to the RST pin first and foremost [22]. The testing sketch is then appended as follows:

```
ESP.deepSleep(SLEEP_DELAY_IN_SECONDS * 1000000, WAKE_RF_DEFAULT);
delay(500);
```

SLEEP_DELAY_IN_SECONDS is a constant defined at the top of the sketch, it is set to 5 so there is less waiting between tests. This Deep Sleep function powers off everything but the RTC, which will generate a short pulse out D0 when the time has run out. As this is connected to the RST pin, it forces the device to reboot. This means that setup() will be ran again each time the device starts up.

Design

Hardware Design

The hardware will be connected via a simple breadboard. As it needs no soldering, this allows for quick troubleshooting and amendments to be made. All power will come from the 3v3 out pins on the NodeMCU. Pin D0 will also be wired to the RST pin, as this allows the ESP8266 to return from Deep Sleep.

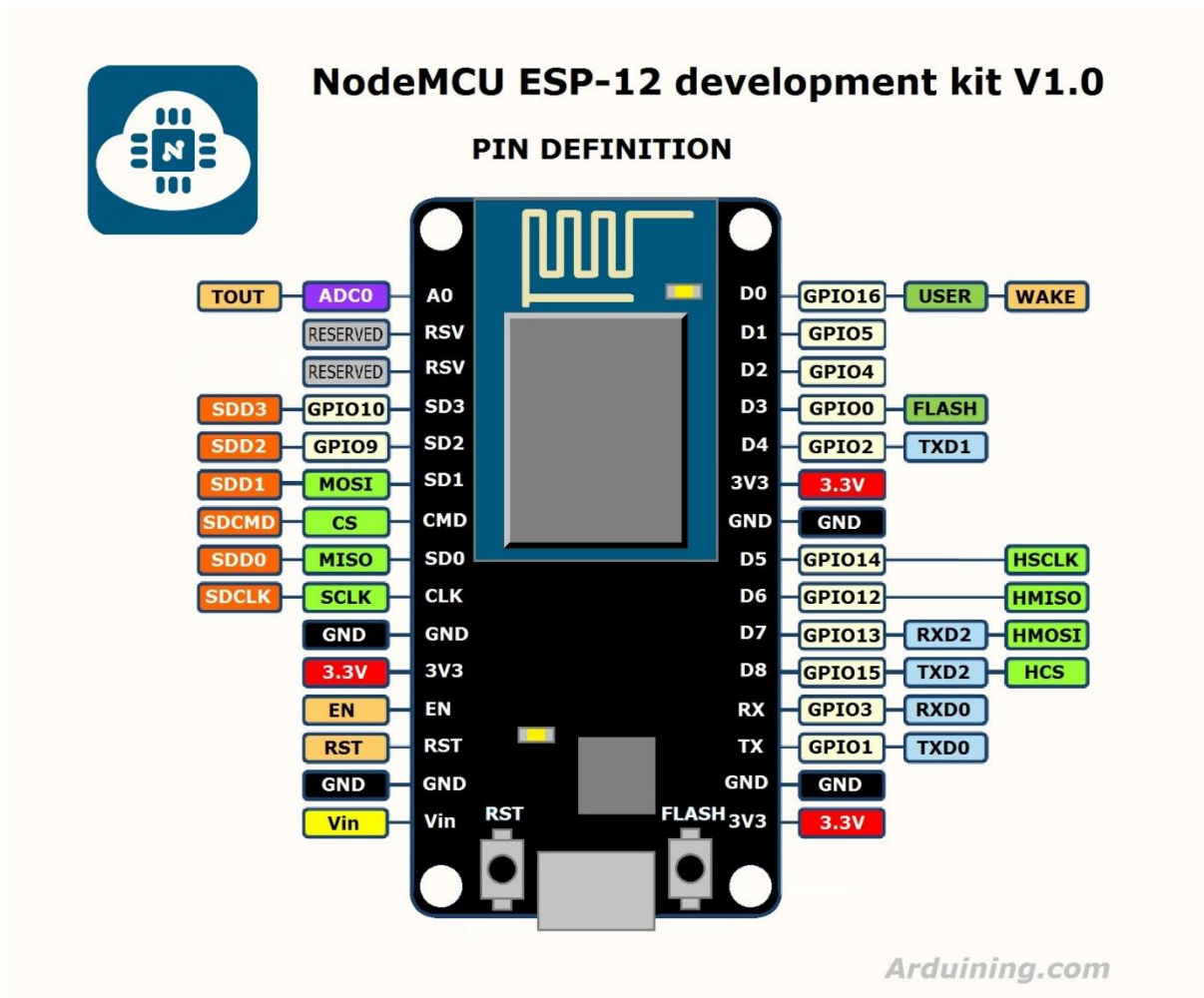


Figure 16, NodeMCU V1.0 pin definition [23]

Using this pin definition, a drawing is created detailing the intended layout of the final breadboard.

To aid in the design of this project, the tool Fritzing was used. Using this tool, it was possible to create the following breadboard layout.

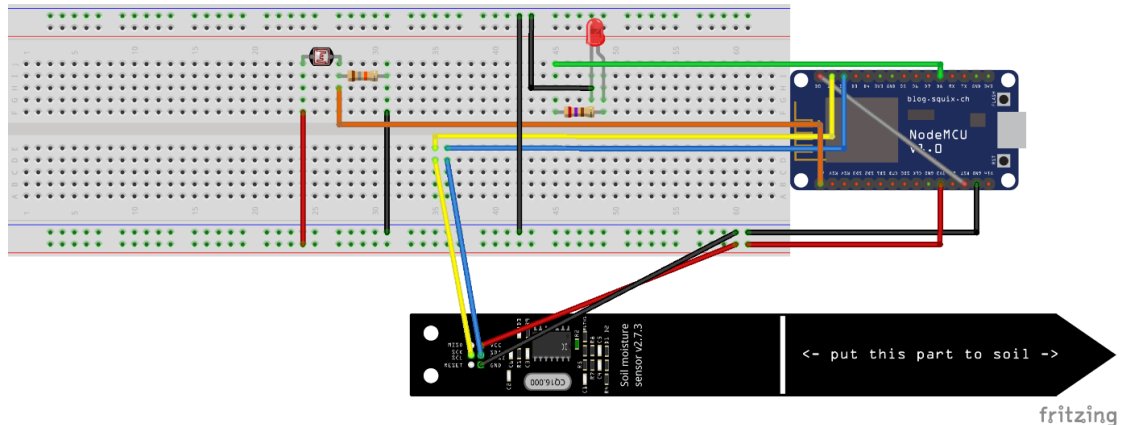


Figure 17, Breadboard Layout

This layout has the accompanying schematic as follows.

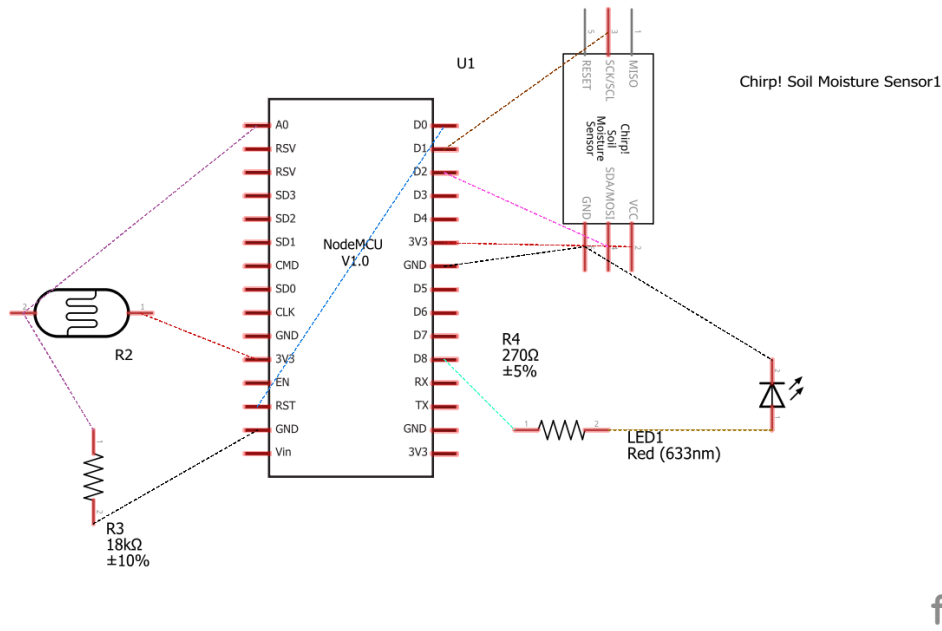


Figure 18, Schematic Layout

As can be seen in the designs above, D1 will be connected to SCL, with D2 connected to SDA. This allows I²C to be used between the NodeMCU and the Moisture Sensor.

D8 is connected to the 270Ω Resistor, which is then connected to the LED. D8 is used as pins D5, D6, and D7 showed a “flicker” on the LED when the NodeMCU was booting up.

A0 is connected to the junction point between R2, a photocell LDR, and R3, an 18kΩ resistor. This allows for a reading of the relative light level.

The Deep Sleep function is enabled by connecting D0 to RST. This is needed, as Deep Sleep disables all but the Real-Time Clock and Deep Sleep Watchdog Timer, which is loaded with the value entered in the code. Once the DSWDT is ran down, an “interrupt” is generated, which involves a single pulse out of D0. When wired up to RST, this enables the NodeMCU to reset itself after entering deep sleep. This drastically cuts down on power usage.

Software Design

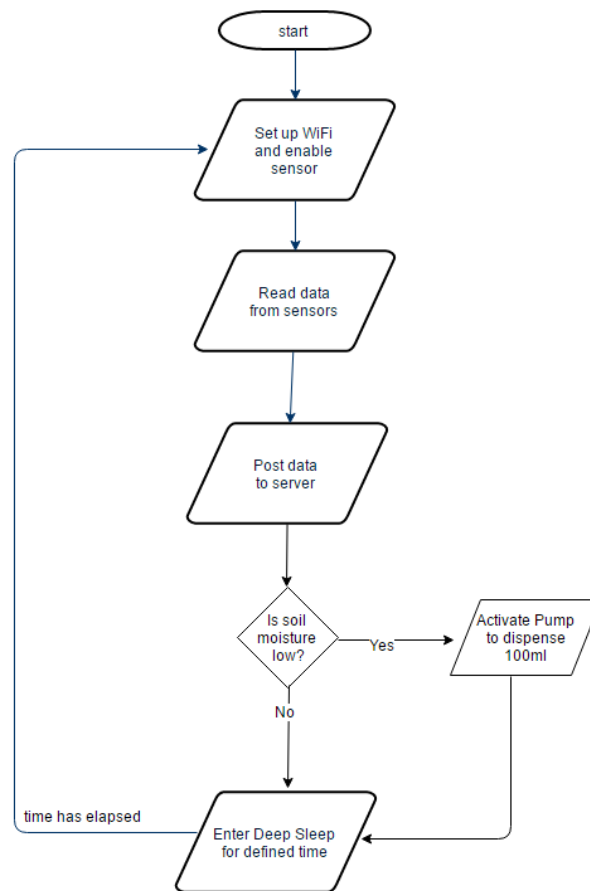


Figure 19, Basic Flowchart of Software

The software will ultimately follow this simple flow, as the only decision to make is whether the soil capacitance is low enough to trigger the pump. The software will be implemented using the Arduino IDE as it was the language with most familiarity.

Implementation

Sending Data to InfluxDB

With the collection of data completed in the test sketch, it was time to test sending it to InfluxDB. It was decided to use InfluxDB's UDP Line Protocol. In a nutshell, this involves sending a datagram with the content in the format of [key] [fields] [timestamp] [24]. To cut down on complexity, the timestamp entry is omitted, as the server will append its own timestamp if the query does not contain one.

The connection to WiFi and using UDP to send the data requires the following variables to be set up:

```
#define SLEEP_DELAY_IN_SECONDS 60
const char* ssid      = "Dunbar";
const char* password = "xxxxxx";
IPAddress ip(192, 168, 0, 35);
IPAddress gateway(192, 168, 0, 1);
IPAddress subnet(255, 255, 255, 0);
byte host[] = {192, 168, 0, 11};
int port = 8089;
WiFiUDP udp;
```

The connection is then started with the following two lines:

```
WiFi.config(ip, gateway, subnet);
WiFi.mode(WIFI_STA);
WiFi.begin(ssid, password);
Serial.print("Connecting to ");
Serial.println(ssid);
```

A while loop is used to delay until the connection is completed:

```
while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
}
```

Once this connection is complete, the following is printed to the Serial:

```
Serial.println("");
Serial.println("WiFi connected");
Serial.println("IP address:");
Serial.println(WiFi.localIP());
```

This concludes the Wi-Fi setup, so we then start the sensor with `sensor.begin()`; Inside `loop()`, we collect the data and send it off to the server:

```
void loop() {  
  while (sensor.isBusy()) delay(50); // available since FW 2.3  
  send_moisture();  
  send_temp();  
  send_light();  
  sensor.sleep(); // available since FW 2.3  
  ESP.deepSleep(SLEEP_DELAY_IN_SECONDS * 1000000, WAKE_RF_DEFAULT);  
  delay(500);  
}
```

As written in the sensor testing sketch, this polls for the sensor to not be busy, then runs 3 functions, each one collecting different data. After this data is collected and sent, the sensor is set to sleep and then the NodeMCU is set to deep sleep for the constant defined at the top of the sketch.

The content of the `send_moisture()` function is as follows:

```
Serial.println("Sending Moisture Level");  
String line, capacitance;  
capacitance = sensor.getCapacitance();  
line = String("moisture value=" + capacitance);  
Serial.println(line);  
send_line(line);
```

Here, the sensor is polled for the capacitance reading, this is then added to a String which follows the format outlined in the Line protocol used by InfluxDB. The function `send_line()` is then called, which is as follows:

```
Serial.println("Sending datagram");  
udp.beginPacket(host, port);  
udp.print(line);  
udp.endPacket();  
delay(250);
```

This begins the construction of a datagram, destined for the host and port defined earlier. The line is then “printed” to the UDP object, followed by the `endPacket()` function. A delay is then called to allow for the datagram to be properly sent.

Before this can all come together, some extra work is needed to be done on the InfluxDB server. First, UDP must be enabled on a port (8089) and a database must be defined, all Line entries are to the same database defined in the config file:

```
[[udp]]
  enabled = true
  bind-address = ":8089"
  database = "plant"
```

With this enabled, the database must now be created, by entering CREATE DATABASE plant in the InfluxDB web admin interface, which is also enabled in the config. We can then verify the datagrams are being received through the SHOW STATS query.

udp
bind::8089

batchesTx	batchesTxFail	bytesRx	pointsParseFail	pointsRx	pointsTx	readFail
8503	2	475678	0	25645	25637	0

Figure 20, UDP stats from InfluxDB Web Interface

With the device running, diagnostic data is then printed to serial as expected

```
ZL...LS...i:h...
Connecting to Dunbar
...
WiFi connected
IP address:
192.168.0.35
Sending Moisture Level
moisture value=446
Sending datagram
Sending Temperature
temperature value=20.70
Sending datagram
Sending Light Level
light value=933
Sending datagram
...
Connecting to Dunbar
...
WiFi connected
IP address:
192.168.0.35
Sending Moisture Level
moisture value=446
Sending datagram
Sending Temperature
temperature value=20.70
Sending datagram
Sending Light Level
light value=934
Sending datagram
```

Figure 21, Diagnostic info from device

Testing time spent awake

The rough time spent awake was recorded to determine power usage of the device. With no static IP assigned, the WiFi took on average 825ms to connect, this was then followed by a 250ms delay, along with three 100ms delays, for a total of 1375ms. The IP was then hardcoded as shown in the examples above, this drastically shortened the connection time down to 200ms, making a total of 750ms spent awake. It is possible to reduce this further by redesigning the Line posting method, down to a single packet.

```
void post_line(){
  String line, capacitance, temperature, light;
  capacitance = sensor.getCapacitance();
  line = String("moisture value=" + capacitance + "\n");

  temperature = sensor.getTemperature()/(float)10;
  line += String("temperature value=" + temperature + "\n");

  light = analogRead(A0);
  line += String("light value=" + light);

  Serial.println("Sending datagram");
  udp.beginPacket(host, port);
  udp.print(line);
  udp.endPacket();
  delay(100);
}
```

Replacing the three send functions with this shaves off another 200ms, making the active time roughly 550ms. Posting every 60 seconds, this means the device will be active only 0.91% of the time.

Displaying data in Grafana



Figure 22, Final Grafana Dashboard

The following queries are used for these graphs:

```
SELECT median("value") FROM "moisture" WHERE $timeFilter GROUP BY time(5m) fill(0)
```

Light Level and Temperature of course do not use 'FROM "moisture"' and instead use their respective data sets.

What this query achieves is it extracts the median value of all the time series entries in a 5 minute "slot" and outputs this to the graph. Fill(0) is used to provide a constant floor of 0 for the graph. This keeps the graph consistently scaled and easier to interpret.

Code Refinement

With a few days of testing, it was found that the `delay()` call after deep sleep was entirely unnecessary, and did not in fact seem to be doing anything.

The delay after start sensor was also removed, in favour of the 50ms delay loop already in place in `loop()`. This resulted in a further saving of 250ms, as it was found that the loop was never entered after a test of 100 restarts.

During testing, it was discovered that on occasion the Sensor Stick would not wake up properly from sleep, leading to erroneous readings until the entire system was unplugged and reset. After doing some research, an issue was found out the GitHub for the `I2CSoilMoistureSensor` library [25] detailing the same bug and discussing how to fix it. However, since the sensor will be turned off anyway, making the actual sleep call redundant, it was decided to remove the call to the sensor to enter sleep entirely. After 168 hours' runtime, the issue did not present itself again.

Pump

It was hoped that it would be possible to use a pump for this project, but unfortunately due to logistics issues one was unable to be sourced in time. In its place is a simple LED, as seen on the breadboard schematic above. This is used in tandem with some code to emulate a pump activation, simply blinking the LED on when the soil moisture reading is low. The code for this is simple and is as follows:

```
if (sensor.getCapacitance() < 400){
  Serial.println("Low Moisture");
  digitalWrite(15, HIGH);
} else {
  Serial.println("High Moisture");
  digitalWrite(15, LOW);
}
```

This basic code would write out to GPIO15 (which is D8 on the nodeMCU) and set it high if the moisture was below a predetermined level, along with printing this to the Serial output. If it was above this level, it would set it low (redundant as it starts low) and print High Moisture to the Serial output.

System Performance

With the implementation now mostly complete, it was time to test the systems performance.

Performance Criteria

There are 2 major areas that will be tested, the reliability of the sensors, which includes the precision and accuracy of the readings along with how likely they are to fail.

The second major area will be power usage, this will be used to extrapolate the lifetime on a battery as well as potentially solar powering the device.

Evaluation Procedures

Sensor Reliability

This meant testing the sensor for both failures and accuracy of the readings. As the light and moisture readings are relative, it is only possible to test them for failures, as the readings only make sense when compared to other readings taken at different times.

For the temperature, an IR spot sensor will be used to measure the temperature on the stick and compare it to results posted by the sensor.

Power Usage

This will involve determining the average power usage while active, which can then be divided by the time spent awake vs asleep to roughly determine power usage. This will be achieved by measuring with a USB power tester, designed to test charging rates for phones, it will be more than appropriate for this task.

Test Results

Sensor Reliability

As mentioned above, the sensor proved to have some issues after a couple days of testing, with false readings being read by the NodeMCU related to putting the sensor to `sleep()`. This was resolved by simply removing the call to `sleep()` entirely, as the sensor will be powered off when the NodeMCU goes into `DeepSleep` anyway.

It was also found that the moisture values would sometimes degrade slower than expected, however, when the device is uprooted and replaced or “jiggled” in place, lower values are often returned, it was found this was somewhat dependent on the plant pot itself, poorly draining pots seemed to retain patches of moisture instead of draining properly. When placed in an appropriately sized pot for the plant with proper drainage, the issue was less severe, resulting in a drop in the value read of roughly 5-15, instead of the 50-60 or so seen with a poorly draining pot.

It was also found that distance from the base of the plant had an impact on readings, with the areas closer to the plant base retaining moisture for longer. In some cases, this was a difference in reading of 60.

Power Usage

By connecting the NodeMCU to a bench power supply and an ammeter, it was possible to determine the load during various states of operation.

Under normal, idle conditions, such as when attempting to connect to WiFi, the full system drew 80mA. This would spike to roughly 125mA when polling the sensor. Under deep sleep, this dropped to a miniscule 10mA.

Under normal conditions, the NodeMCU takes 200ms to connect, and a further 100ms with the sensor enabled, this equates to 200ms using 80mA, 100ms using 125mA and 59.7 seconds using 10mA.

This means that 99.5% of the time, the sensor will be using only 10mA, 0.33% using 80mA and 0.17% using 125mA. When averaged out, this means an average power usage of 10.4mA. Conversely, if the delay after starting the sensor was not removed, this results in an increase of 0.5mA once averaged out.

With a 2000mAh battery, assuming the battery can be used to its full capacity with no loss, a runtime of 183 hours and 29 minutes is calculated.

Solar Power Viability

In this section, the possibility of powering the system via a solar panel and a LiPo battery will be assessed. The reasoning behind this is to facilitate the system being placed in an outdoors environment, away from where it would otherwise have easy access to power.

Testing Methods

Using the average power usage calculated above, a suitable battery will be determined, along with a solar panel of appropriate power. The daily generation of power will be calculated by assuming the solar panel will output it's full rating under sunshine hours and that the power usage calculated above would be consistent.

For testing purposes, a 6W 6V Solar Panel will be considered. A 3.7v LiPo battery with a capacity of 2000mAh will also be considered for testing. A component would be needed to both charge the battery from the panel and power the system, such as the USB / DC / Solar Lithium Ion/Polymer charger - v2 from AdaFruit.

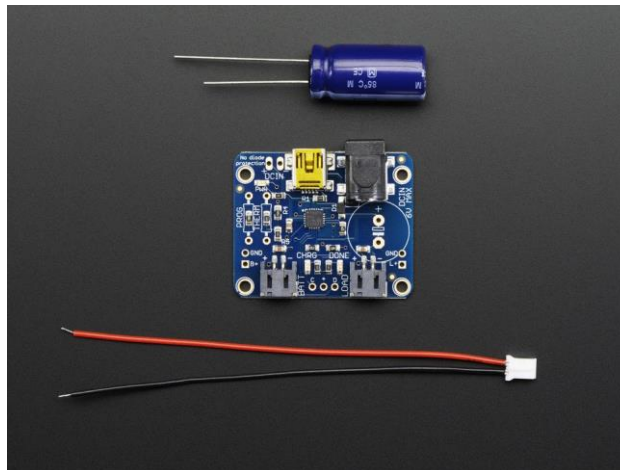


Figure 23, USB / DC / Solar Lithium Ion/Polymer charger - v2 [26]

Test Results

According to the met office of Ireland, Ireland has between 5 – 6.5 hours of sunshine a day. [27]. Using a 6W/6V panel, we will assume that it will output at full power during this time. The solar panel outputs 1A, meaning it should charge the 2000mAh cell fully in 2 hours of sunshine, generating between 5000mAh and 6500mAh of charge per day. This is more than adequate, considering the system will use around 250mAh per day.

This offers a considerably safe choice for powering the system, under normal conditions it is unlikely that it will output 100% power during the 5 – 6.5 hours, but at anything above 5% power output, it outputs more charge than the device will use.

Security

With testing complete, the security aspects of the project must be discussed. This will involve determining potential weaknesses in the NodeMCU, along with weaknesses with InfluxDB and Grafana.

InfluxDB

InfluxDB can utilise a user-based permissions system, needing a username and password to perform queries on the database. However, the UDP Line posting protocol does not use this at all. This means that anyone or anything can post to it without issues.

The way to secure this would be to place the sensors on their own subnet, with an ACL allowing only those sensors to post to the database.

Grafana

Grafana can also use a user-based permissions system, meaning accounts can be set up purely for viewing data, with the admin account used only for editing data. Grafana also has the option of allowing signups, but this might want to be disabled depending on who the Grafana install is intended to be shared with.

Grafana also has an organisation system, which is a way of grouping users with similar permissions, this is intended to restrict access to certain dashboards for some groups and providing a way for them to self-manage access within the group via Organisation Admins.

External Access

It is possible to use LetsEncrypt with Nginx to enable external access to Grafana. Using the LetsEncrypt Docker, set up with a subdomain cert for grafana.sdunbar.me, we can provide a HTTPS cert for the public facing Grafana web interface.

LetsEncrypt

LetsEncrypt is a free and automated open Certificate Authority, run by the Internet Security Research Group. LetsEncrypt provides a free way to create short-lived SSL/TLS certs for websites at no user cost. Any number of certs can be created and renewed. Although no wildcard certificates are available, any number of free subdomain-only certs can be created, which alleviates the problem. [28]

Nginx

Nginx is a HTTP and reverse proxy server, with mail and TCP/UDP proxy capabilities. [29] It gained huge popularity in Russia before gaining popularity elsewhere and is now used by nearly one third of the internet. [30]

Implementation

This requires creating a server{} block in the Nginx config file, which is done as follows,

```
server {  
    listen 443 ssl;
```

Here a listening service is established on port 443, using SSL

```
    root /config/www;  
    index index.html index.htm index.php;
```

This defines the “root” folder of the webpage being served.

```
    server_name grafana.*;
```

This is one of the more important parts of the config, this means that this specific block will only be run for any query with the URL grafana.sdunbar.me. This allows for multiple services to be run on independent subdomains.

```
    ssl_certificate /config/keys/letsencrypt/fullchain.pem;  
    ssl_certificate_key /config/keys/letsencrypt/privkey.pem;  
    ssl_dhparam /config/nginx/dhparams.pem;
```

This part defines the location of the cert, keys and Diffie-Hellman Key Exchange parameter files.

```
ssl_ciphers 'ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-DSS-AES128-GCM-SHA256:kEDH+AESGCM:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-DSS-AES128-SHA256:DHE-RSA-AES256-SHA256:DHE-DSS-AES256-SHA:DHE-RSA-AES256-SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-SHA256:AES256-SHA256:AES128-SHA:AES256-SHA:AES:CAMELLIA:DES-CBC3-SHA:!aNULL:!eNULL:!EXPORT:!DES:!RC4:!MD5:!PSK:!aECDH:!EDH-DSS-DES-CBC3-SHA:!EDH-RSA-DES-CBC3-SHA:!KRB5-DES-CBC3-SHA';
ssl_prefer_server_ciphers on;
```

This massive block is quite simple, it is a list of ciphers that the server will present as available to the client for establishing secure communications

```
client_max_body_size 0;
```

Setting this parameter to 0 disables the checking of the “body” size of requests, this would otherwise restrict the size of requests from the client.

```
location / {
    include /config/nginx/proxy.conf;
    proxy_pass http://192.168.0.11:3030;
}
```

This concludes setting up what is known as a “reverse proxy” in Nginx. This sets the location “/”, aka the root of the website being served, as whatever data is located at <http://192.168.0.11:3030>. In this example, said URL and port points to the local Grafana install.

This however only works if the user accesses <https://grafana.sdunbar.me/> and not http://, another block must be set up for this to automatically redirect.

```
server {
    listen 80;
    server_name *.sdunbar.me;
    return 301 https://$host$request_uri;
}
```

With this block in place, any request to any subdomain on port 80 (HTTP) will be redirected with error code 301 and towards the same address, but with HTTPS instead of HTTP. Error code 301 indicates that the server has Moved Permanently to the address returned along with it.

Discussion

With the tests complete, the sensor is working as well as can be expected. The issue with false readings was resolved, meaning that the sensor reports accurate results all the time now, so long as it is repositioned regularly at spots with the same distance from the base of the plant.

It was also found that a Solar Panel for power was more than adequate, with tests showing a cheap, 6W 6V panel providing more than 20 times as much power as was needed to just about match the consumption of the device. A 1W 6V panel coupled with a 1000mAh battery should be more than adequate to power the system nearly indefinitely.

When using appropriate user accounts for InfluxDB and Grafana, along with combining them with proper subnetting, ACLs and LetsEncrypt for external access, it was found to be more than adequate for security. It is unfortunate that the UDP Line protocol is not encrypted at all, however this is at least remediable on the network level by segmenting the sensors off from the main network.

Conclusions

The intent of this project was to measure the soil moisture level, temperature and light level of a plant pot, by using a small Single Board Computer, and then sending these metrics to an appropriate database with graphing, along with activating a pump if the plant pot was too dry.

After this was complete, the idea was to then determine power usage and decide if solar power was a possible power source for this.

Overall this project was successful in meeting these requirements, it is unfortunate the pump was not able to be obtained as it would have been interesting to see it in use and determine how long it needed to be ran for to pump 100ml per activation.

Future Alternatives

With the knowledge gained from completing this project, it would be very easy to improve on this even further. Ways this could be improved would be using a cheap radio transmitter instead of a board with WiFi, hopefully cutting down on power usage even further. This could involve having numerous sensors radio back metrics to a “base” station, which would then upload the data to the database.

A waterproof, solar-powered enclosure could be used in the system. This would allow for easy deployment in a garden environment.

It could also be used with a home irrigation system, using metrics about the soil to activate a solenoid for a sprinkler, this would tie in well with the Solar Panel idea and the radio transmitter idea, allowing for multiple remote systems independently activating sprinklers in certain areas of a garden.

References

- [1] AdaFruit, “ESP8266 WiFi Module,” [Online]. Available: <https://www.adafruit.com/product/2282>. [Accessed 28 February 2017].
- [2] BangGood, “LoLin V3 NodeMcu Lua WIFI Development Board,” [Online]. Available: <http://www.banggood.com/V3-NodeMcu-Lua-WIFI-Development-Board-p-992733.html>. [Accessed 23 March 2017].
- [3] NodeMCU, “NodeMCU Features,” [Online]. Available: http://nodemcu.com/index_en.html#fr_54747361d775ef1a3600000f. [Accessed 28 February 2017].
- [4] “Getting started with MicroPython on the ESP8266,” [Online]. Available: <https://docs.micropython.org/en/latest/esp8266/esp8266/tutorial/intro.html>. [Accessed 28 February 2017].
- [5] B. Thomsen, “ESP8266 Getting Started with Arduino IDE,” [Online]. Available: <https://bennthomsen.wordpress.com/iot/iot-things/esp8266-wifi-soc/esp8266-getting-started-with-arduino-ide/>. [Accessed 28 February 2017].
- [6] Arduino, “Arduino Nano,” [Online]. Available: <https://www.arduino.cc/en/Main/ArduinoBoardNano>. [Accessed 16 April 2017].
- [7] The MagPI Magazine, “INTRODUCING RASPBERRY PI ZERO W,” [Online]. Available: <https://www.raspberrypi.org/magpi/pi-zero-w/>. [Accessed 16 April 2017].
- [8] WeMos, “D1 mini,” [Online]. Available: <https://www.wemos.cc/product/d1-mini.html>. [Accessed 17 April 2017].
- [9] Catnip Electronics, “I2C Soil moisture sensor,” 16 June 2015. [Online]. Available: <https://www.tindie.com/products/miceuz/i2c-soil-moisture-sensor/>. [Accessed 16 March 2017].
- [10] i2c.info, “I2C Bus Specification,” [Online]. Available: <http://i2c.info/i2c-bus-specification>. [Accessed 13 March 2017].
- [11] InfluxDB, “InfluxDB README.md,” 18 January 2017. [Online]. Available: <https://github.com/influxdata/influxdb/blob/master/README.md>. [Accessed 14 March 2017].
- [12] InfluxDB, “InfluxDB Key Concepts,” [Online]. Available: https://docs.influxdata.com/influxdb/v1.2/concepts/key_concepts/#tag-key. [Accessed 14 March 2017].
- [13] InfluxDB, “InfluxDB: Writing Data,” [Online]. Available: https://docs.influxdata.com/influxdb/v1.2/guides/writing_data/. [Accessed 14 March 2017].
- [14] DB-Engines, “DB-Engines Ranking of Time Series DBMS,” DB-Engines, April 2017. [Online]. Available: <https://db-engines.com/en/ranking/time+series+dbms>. [Accessed 24 April 2017].
- [15] OpenTSDB, “How does OPenTSDB work?,” [Online]. Available: <http://opentsdb.net/overview.html>. [Accessed 23 March 2017].

- [16] Grafana, "Grafana: Basic Concepts," 2017. [Online]. Available: http://docs.grafana.org/guides/basic_concepts/. [Accessed 18 March 2017].
- [17] LimeTech, "Network-Attached Storage," [Online]. Available: <https://lime-technology.com/network-attached-storage/>. [Accessed 20 April 2017].
- [18] Docker, "What is a Container," 2017. [Online]. Available: <https://www.docker.com/what-container>. [Accessed 4 April 2017].
- [19] InfluxDB, "Web Admin Interface," 12 March 2017. [Online]. Available: https://docs.influxdata.com/influxdb/v1.2/tools/web_admin/. [Accessed 12 March 2017].
- [20] InfluxDB, "InfluxDB-Python," 21 February 2017. [Online]. Available: <https://github.com/influxdata/influxdb-python/blob/master/influxdb/client.py#L395>. [Accessed 13 March 2017].
- [21] Apollon77, "I2CSoilMoistureSensor," 3 March 2017. [Online]. Available: <https://github.com/Apollon77/I2CSoilMoistureSensor>. [Accessed 27 March 2017].
- [22] T. Foxworth, "MAKING THE ESP8266 LOW-POWERED WITH DEEP SLEEP," 21 March 2017. [Online]. Available: <https://www.losant.com/blog/making-the-esp8266-low-powered-with-deep-sleep>. [Accessed 12 April 2017].
- [23] A. Gerber, "Getting started with Espruino on ESP8266," 2016. [Online]. Available: <http://crufti.com/getting-started-with-espruino-on-esp8266/>. [Accessed 27 April 2017].
- [24] T. Persen, "How to send sensor data to InfluxDB from an Arduino Uno," 29 September 2015. [Online]. Available: <https://www.influxdata.com/how-to-send-sensor-data-to-influxdb-from-an-arduino-uno/>. [Accessed 14 April 2017].
- [25] simonrobb, "After calling sensor.sleep(), can't read value #8," 7 March 2017. [Online]. Available: <https://github.com/Apollon77/I2CSoilMoistureSensor/issues/8>. [Accessed 17 April 2017].
- [26] Adafruit, "USB / DC / Solar Lithium Ion/Polymer charger - v2," [Online]. Available: <https://www.adafruit.com/product/390>. [Accessed 4 May 2017].
- [27] Met Eireann, "Sunshine and Solar Radiation," [Online]. Available: <http://www.met.ie/climate-ireland/sunshine.asp>. [Accessed 4 May 2017].
- [28] Let's Encrypt, "About Let's Encrypt," 2017. [Online]. Available: <https://letsencrypt.org/about/>. [Accessed 26 April 2017].
- [29] nginx, "nginx," [Online]. Available: <https://nginx.org/en/>. [Accessed 18 May 2017].
- [30] Netcraft, "April 2017 Web Server Survey," April 2017. [Online]. Available: <https://news.netcraft.com/archives/2017/04/21/april-2017-web-server-survey.html>. [Accessed 18 May 2017].

Appendices

Test Sketch

```
#include <I2CSoilMoistureSensor.h>
#include <ESP8266WiFi.h>
#include <Wire.h>

#define SLEEP_DELAY_IN_SECONDS 60

I2CSoilMoistureSensor sensor;

void setup() {
  Wire.begin();
  Serial.begin(9600);
  sensor.begin(); // reset sensor
  delay(250); // give some time to boot up
}

void loop() {
  while (sensor.isBusy()) delay(50); // available since FW 2.3
  Serial.print("Soil Moisture Capacitance: ");
  Serial.print(sensor.getCapacitance()); //read capacitance register
  Serial.print(", Temperature: ");
  Serial.print(sensor.getTemperature()/(float)10); //temperature register
  Serial.print(", Light Level: ");
  Serial.println(analogRead(A0));
  sensor.sleep(); // available since FW 2.3
  ESP.deepSleep(SLEEP_DELAY_IN_SECONDS * 1000000, WAKE_RF_DEFAULT);
  delay(500);
}
```

Final Sketch

```
#include <I2CSoilMoistureSensor.h>
#include <ESP8266WiFi.h>
#include <WiFiUDP.h>
#include <Wire.h>

#define SLEEP_DELAY_IN_SECONDS 60

I2CSoilMoistureSensor sensor;

byte host[] = {192, 168, 0, 11};

// the port that the InfluxDB UDP plugin is listening on
int port = 8089;

const char* ssid      = "Dunbar";
const char* password = "newcastle1";
IPAddress ip(192, 168, 0, 35);
IPAddress gateway(192, 168, 0, 1);
IPAddress subnet(255, 255, 255, 0);

WiFiUDP udp;

void setup() {
  Wire.begin();
  Serial.begin(9600);

  Serial.println();
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.config(ip, gateway, subnet);
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);

  int counter = 0;

  while (WiFi.status() != WL_CONNECTED) {
    delay(10);
    Serial.print(".");
    counter++;
  }
```

```
Serial.println("");
Serial.println("WiFi connected");
Serial.print("Time to connect: ");
Serial.print(counter * 10);
Serial.println("ms");
Serial.println("IP address: ");
Serial.println(WiFi.localIP());

sensor.begin(); // reset sensor
;delay(250); // give some time to boot up

pinMode(15, OUTPUT);
}

void loop() {
  while (sensor.isBusy()){
    delay(50); // available since FW 2.3
    Serial.println("Waiting for Sensor");
  }
  if (sensor.getCapacitance() < 400){
    Serial.println("Low Moisture");
    digitalWrite(15, HIGH);
  } else {
    Serial.println("High Moisture");
    digitalWrite(15, LOW);
  }
  post_line();
  ESP.deepSleep(SLEEP_DELAY_IN_SECONDS * 1000000, WAKE_RF_DEFAULT);
}

void post_line(){
  String line, capacitance, temperature, light;
  capacitance = sensor.getCapacitance();
  line = String("moisture value=" + capacitance + "\n");

  temperature = sensor.getTemperature()/(float)10;
  line += String("temperature value=" + temperature + "\n");

  light = analogRead(A0);
  line += String("light value=" + light);

  Serial.println("Sending datagram");
  udp.beginPacket(host, port);
  udp.print(line);
  udp.endPacket();
  delay(100);
}
```

Influx_test.py

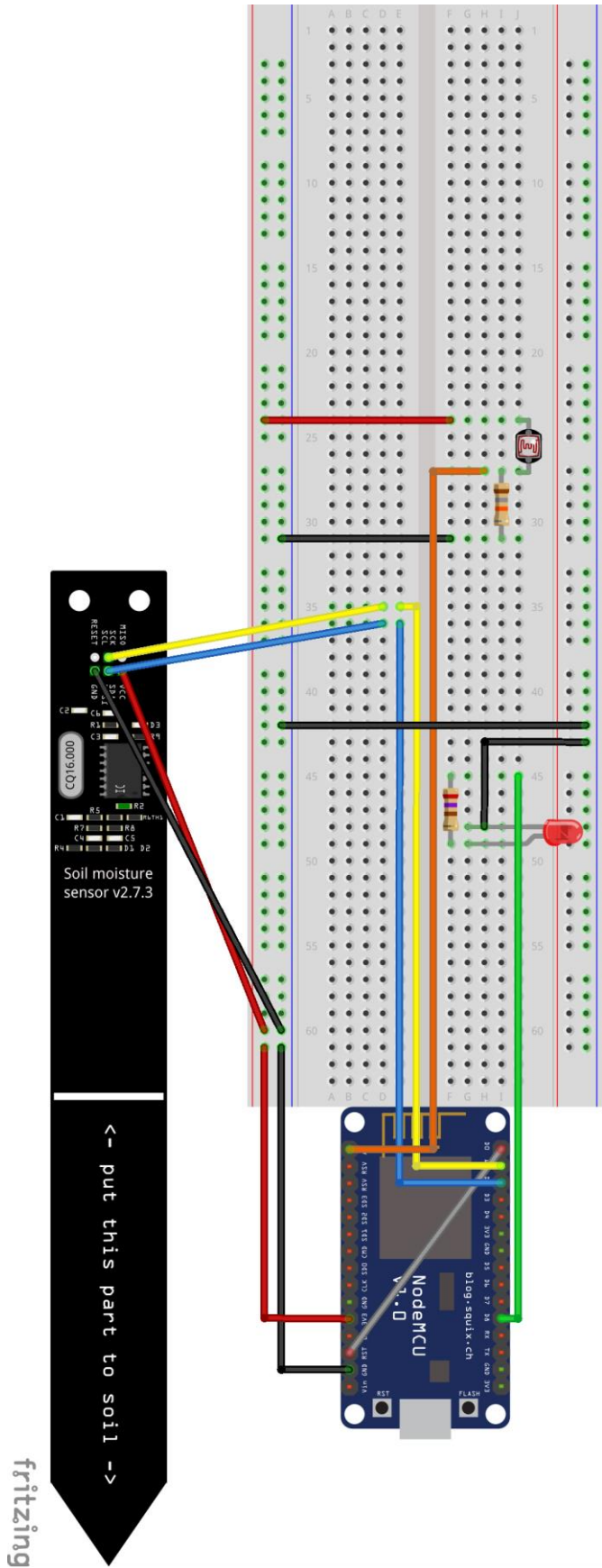
```
import time
import random

from influxdb import InfluxDBClient

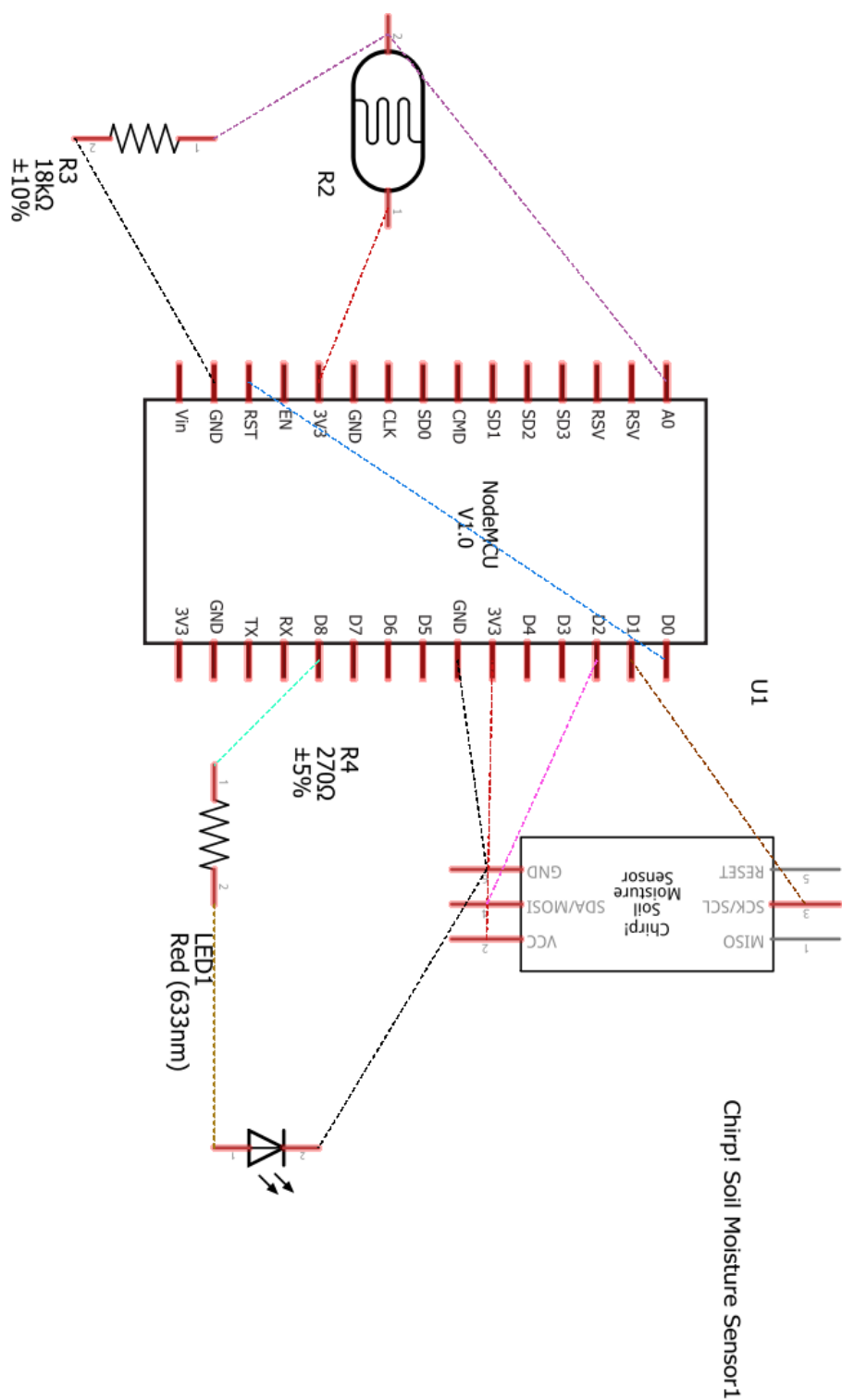
client = InfluxDBClient('192.168.0.11', 8086, '', '', 'test')
client.create_database('test')

while True:
    json_body = [
        {
            "measurement": "Temperature",
            "fields": {
                "value": round(random.uniform(10.0, 25.0), 1)
            }
        },
        {
            "measurement": "Moisture",
            "fields": {
                "value": random.randrange(100, 400, 10)
            }
        }
    ]
    client.write_points(json_body)
    print("Posted at " + time.strftime("%Y-%m-%d %H:%M:%S",\
        time.gmtime()))
    time.sleep(60)
```

Breadboard



Schematic



fritzing

[illegible]