UNIVERSITY OF SOUTHERN DENMARK

Automated Plant Watering System for Embedded Linux (EMLI)

Group 15

Loris Giannatempo
logia23@student.sdu.dk

Nathan Savard
nasav22@student.sdu.dk

GitHub: https://github.com/LorisGiann/Embedded-Linux-Proj

June 5, 2023

1 Introduction

The goal of this project was to create an automatic plant-watering system for an in-home garden. The system consists of three main components:

- **Raspberry Pi**: The main processing unit for the system which handles WiFi communications and data processing
- **Raspberry Pico**: Microcontroller used to manage the plant. It allows us to interface with the Pump and the sensors. It's connected via USB to the Raspberry Pi.
- **ESP 8266**: Microcontroller used as a wireless remote. It has 3 LEDs: green, yellow, and red to display the system status. It also has a button to manually water the plant. It connects to the Raspberry Pi over WiFi

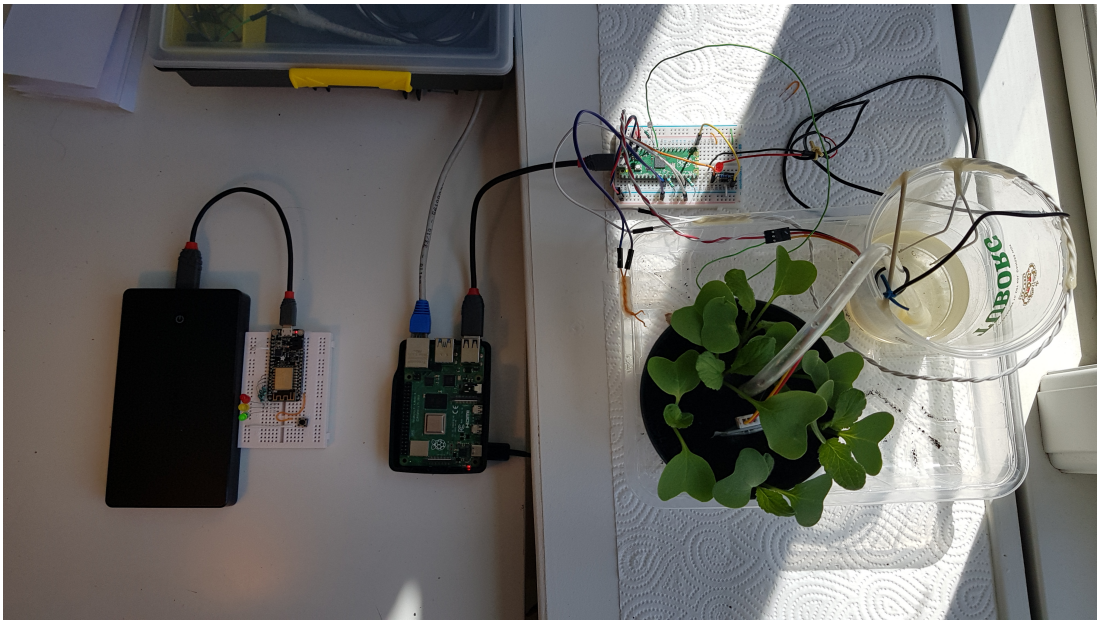

Figure 1: Overall system

The system also integrates a set of sensors to read and respond accordingly:

- **Light Sensor**: A Light Dependent Resistor (LDR) to detect how much light the plant is receiving.
- **Soil Moisture Sensor**: Detects the soil's moisture to check if it is too dry or too wet. The provided sensor consists of 2 contacts to be inserted in the soil to measure the resistance (See product page [1]). However, this sensor didn't work properly in our case, since a maximum value of around 2.7v was always red. This is probably due to the high sensitivity of the device, causing the output to be saturated in highly conductive soils. Sticking 2 inox steel contacts into the soil and reading the values with the help of a partitioner initially delivered better results. However, it looks like the resistance keeps decreasing over time, ultimately causing a low correlation to water changes in the soil. Energizing the device for only a fraction of a second while taking the measurement also didn't help a lot. So, a contactless capacitive soil moisture sensor was used instead (see product page [2]). The sensor is inserted in a small plastic bag (to make it waterproof) and placed under the soil. It provides an analog voltage which is inversely proportional to the soil moisture (so, minor modifications to the Pico code were still necessary).

Figure 2: Capacitive soil moisture sensor being used

- **Water Contact Sensors**: Detects the presence or absence of water. Used to determine if the pump reservoir is empty or if water has spilt outside the plant
- **Raspberry Pi**: Internal sensors in the Pi monitor the system's health and its processing capabilities.

The overall system structure can be seen in Figure 3

2 Solution Approach

As a general approach to the project, we decided to create a set of "modules", each one performing a small action (like an actor-based model). These nodes communicate using the MQTT communication protocol. Each module has its own bash script, all started by a launcher script which is in turn managed and started at boot by Systemd.

2.1 WIFI Communication

Because the system is required to communicate wirelessly between an ESP chip and the Raspberry Pi, we set up the Raspberry Pi as a wireless access point using the `nmcli` tool available on Ubuntu. This allows the ESP to connect to the Raspberry Pi and send/receive messages over WiFi, without the need for an external router.

2.2 MQTT Communication

The system uses MQTT to communicate between modules. In our final system, we also modified the ESP code to directly send and receive MQTT messages, so a consistent communication protocol is used system-wide.

When designing the communications (and the system as a whole), a coherent topic structure was defined to support multiple devices (a.k.a. plants) if needed. As such, each topic is preceded by an identifier for the associated plant, such as *plant0*. Therefore, when a *soil_moisture* reading for the first plant is generated, then the full topic would be *plant0/soil_moisture*.

2.3 System configuration and management

The lifecycle of each module is managed by the launcher script. The responsibilities of this script include:

- Launching of module processes: A separate process group is created for each module, and the PGID is memorized for later cleanup operations. In fact, in this way, each of the module subprocesses also receive a SIGTERM signal when the system is about to be stopped.
- Set up the logging infrastructure: Each module process writes to a temporary named pipe that is read by an instance of a script that manages logs. The script automatically handles the rotation and deletion of log files, so that disk usage does not increase too much over time.
- Provides the modules with the relevant configuration information. This information are defined in an internal data structure that looks like this:

```
PLANTS=()
# ------ first plant -------
declare -A PLANT
PLANT["name"]="plant0"
PLANT["dev"]="/dev/ttyACM0"
PLANT["soil_thresh"]=40
PLANTS+=PLANT
# ------ second plant -------
declare -A PLANT
PLANT["name"]="plant1"
PLANT["dev"]="/dev/ttyACM1"
PLANT["soil_thresh"]=60
PLANTS+=PLANT
```

  Basically, this is an array of plants, where each plant is composed of the following fields:

  - *name*: the identifying plant name, which corresponds to the MQTT topic prefix in which information regarding that plant are published in
  - *dev*: the device file for the Raspberry Pico associated with the plant
  - *soil_thresh*: threshold under which the soil moisture is considered low.

  The script automatically launches and initializes as many instances of the modules based on how many records are present in the array. This is to ensure scalability: new and heterogeneous plants can easily be added to the system if needed.

The launcher script is in turn managed by Systemd. This way the system is started at boot, but only after other services (like the Mosquitto MQTT broker) are ready.

2.4 Modules

As already partially mentioned, for each plant we instantiate a set of "modules" each one responsible for a limited number of actions. We can summarize the interaction existing between these modules with the schematic of Figure 3.

Each module is implemented by a corresponding bash script. For a quick reference, consult Table 1.

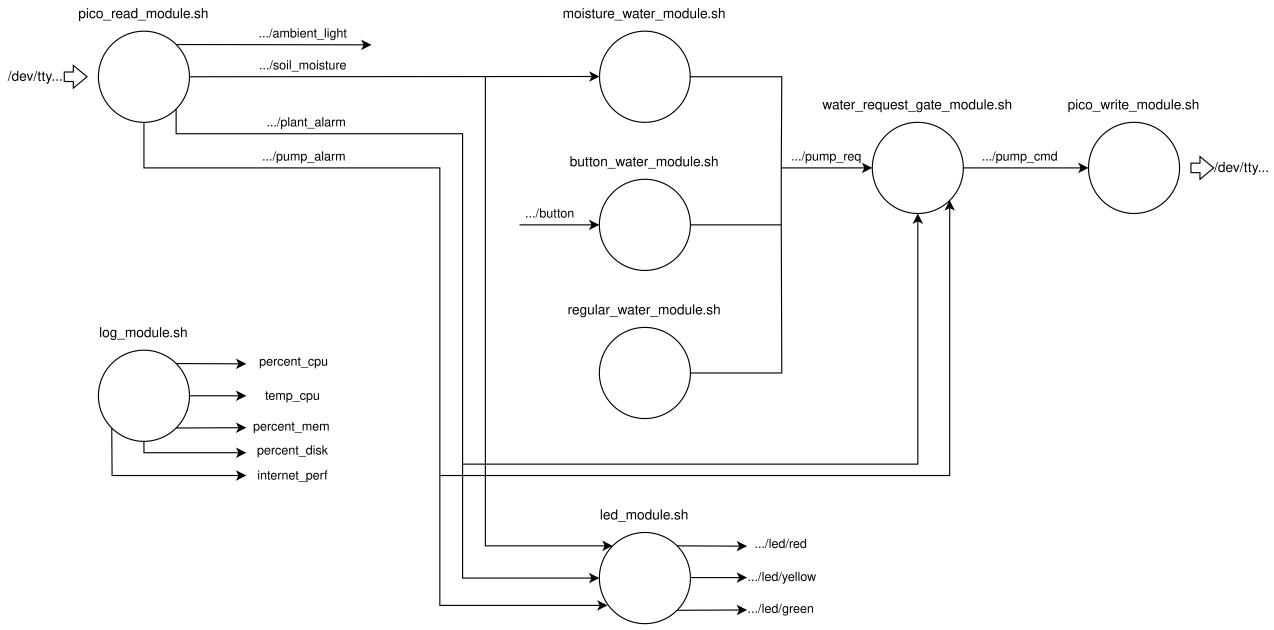

Figure 3: Project modules and structure

Module	Script name	Description
Pico Read Module	pico_read_module.sh	Refer to 2.4.1
Pico Write Module	pico_write_module.sh	Refer to 2.4.2
Water Request Gate Module	water_request_gate_module.sh	Refer to 2.4.3
Regular Water Module	regular_water_module.sh	Refer to 2.4.4
Moisture Water Module	moisture_water_module.sh	Refer to 2.4.5
Button Water Module	button_water_module.sh	Refer to 2.4.6
LED Module	led_module.sh	Refer to 2.4.7
LOG Module	log_module.sh	Refer to 2.4.8

Table 1: Information Read from Pico

Most of the scripts are structured as message consumers, meaning that they listen for incoming MQTT messages on the relevant topics, elaborate the message, update the internal status and produce new messages (or do some other actions) if needed.

Because of this, many of them feature a structure similar to the following one:

```
mosquitto_sub -h localhost -p 1883 -t $TOPIC1 -t $TOPIC2 -d -u $USER -P
    ↪ $MQTT_PASSWD -F "%t %p" |
grep -vE --line-buffered '^Client|^Subscribed' |
while read TOPIC MESSAGE
do
    #echo "From \"$TOPIC\": \"$MESSAGE\""

        if [[ $TOPIC == $TOPIC1 ]] ; then
                ...
        elif [[ $TOPIC == $TOPIC2 ]] ; then
```

```
                ...
        else
                echo "ERROR: not recognized topic!" | tee /dev/stderr
        fi
done
```

As we can see the script consists of 3 commands chained together by the pipe operator:

- *mosquitto_sub*: this program is used to subscribe to the specified topics (*-t <topic>* options). Each output line of this command is formatted as "*<topic>* *<message>*" through the use of the *-F "%t %p"* options
- *grep*: used to filter out some debug / connection info messages
- *while read loop*: the read command waits for a new line from stdin (which is the output of the previous *grep* command in this case) unless EOF is received. When a new message arrives, the topic from which the message originated is placed in the *TOPIC* variable, while the content is assigned to the MESSAGE variable. Fitting logic is then executed (typically an else-if structure is used to handle differently messages from different topics)

The following subsections go into the details of the functionality of each module. You may want to keep an eye on the schematic of Figure 3 to better understand how the single module integrates within the system as a hole.

2.4.1 Pico Read Module

The **Pico Read** module is used to read information from the pico and send that information out over MQTT for other modules to use. It does this by reading directly from the device file in the /dev directory (which is kept static by configuring Udev) to obtain the lines being sent over the serial port. For this to work, the setup script executes the following lines:

```
    sudo usermod -a -G dialout $USER
    sudo stty -F /dev/ttyACM0 115200 -ixon -ixoff #set the Baud rate and other
       ↪  configs for the serial device
```

The module then publishes the information on the following topics:

Value	Description	Topic
Ambient Light	% of ambient light read by the photoresistor (theoretical range: 0 to 100)	*ambient_light*
Soil Moisture	% of soil moisture read by the moisture sensor (theoretical range: 0 to 100)	*soil_moisture*
Plant Alarm	A flag that is set to 1 if there is water spilt outside the plant (0 means no alarm)	*plant_alarm*
Pump Alarm	A flag that is set to 1 if the pump water level is low (0 means there's sufficient water)	*pump_alarm*

Table 2: Information Read from Pico

2.4.2 Pico Write Module

The **Pico Write** module sends information to the Pico by writing to the device file in /dev. Specifically, the module is used to send the pump activation command over the serial interface, which is done by

writing the "p" character to the device. This is done whenever a message is received on the *pump_cmd* topic, to which the **Water Request Gate** module writes to.

2.4.3 Water Request Gate Module

The **Water Request Gate** module is used to prevent the plant from being watered when there is an ongoing alarm. This module reads from both the *plant_alarm* and *pump_alarm* topics to check whether it is safe to water the plant. If a **Water Module** (described in 2.4.4, 2.4.5, or 2.4.6) requests for the plant to be watered, and there are no alarms raised, then the request will be passed to the **Pico Write** module by publishing on the *pump_cmd* topic.

2.4.4 Regular Water Module

The **Regular Water** module requests for the plant to be watered at regular intervals. Specifically, it will make a request every 12 hours. This is done by sending a request to the *pump_req* topic.

2.4.5 Moisture Water Module

The **Moisture Water** module is used to water the plant if the soil moisture drops below the configured threshold. It does this by reading the soil moisture from the *soil_moisture* topic, and requesting the plant to be watered over the *pump_req* topic if necessary. Once a water request is sent, the module waits an hour before issuing any new request, limiting the pump activations to a maximum of once per hour.

2.4.6 Button Water Module

The **Button Water** module is used to water the plant if the user manually requests it by pressing the button on the ESP. The module checks the *button* topic, which informs about the number of times the button has been pressed since the last message was sent. If a value greater than 0 is received, and the last activation request has been performed more than 2 seconds ago, then this module requests the plant to be watered by sending a message on the *pump_req* topic. Conversely, if a request was performed less than 2 seconds ago, this module simply ignores the message.

2.4.7 LED Module

The **LED** module communicates with the ESP by directly sending MQTT messages to the appropriate *led* topics. The ESP starts an MQTT client that subscribes to those topics (*led_red*, *led_yellow*, and *led_green*).

The module first reads the alarm topics *plant_alarm* and *pump_alarm*. If either alarm is raised, then it publishes on *led_red*. If no alarm is triggered, it checks for the moisture sensor value over the *soil_moisture* topic. If the value is below the desired soil moisture threshold, then it publishes on *led_yellow*. If otherwise no problem has been found, a message is published on *led_green*.

2.4.8 LOG Module

The log module is used to monitor the system's health. It does this by reading various system metrics on the Raspberry Pi, and publishing them to MQTT. These status readings include:

Signal	Description	topic
CPU Usage	The percent of the CPU that is currently in use	*percent_cpu*
CPU Temperature	The current temperature of the CPU in Celsius	*temp_cpu*
Memory Usage	The percentage of the RAM memory in use on the system	*percent_mem*
Disk Usage	The percentage of the root partition space currently occupied	*percent_disk*
Internet Performance	The speed of the internet connection in Mbit/s	*internet_perf*

Table 3: System Status Log Values

2.5 Database

We used InfluxDB as a time series database on the Raspberry Pi to store the values logged over time. Values are fed in from the MQTT topic to the database by properly configuring Telegraf. Telegraf subscribes to the specified MQTT topics and opens a connection to the database to insert data whenever a message is received. The database can then be queried by Grafana to graphically display the stored values.

2.6 Graphical User Interface

Our system uses a graphical interface to display the status of the system as well as system health statistics as described in Section 2.4.8. These values are displayed over a duration of time to show how the system behaves.

To display this information, we used Grafana, which features an HTTP server through which values and graphs can be displayed on a browser. Grafana is able to read values from a database, the component that records the history of the system.

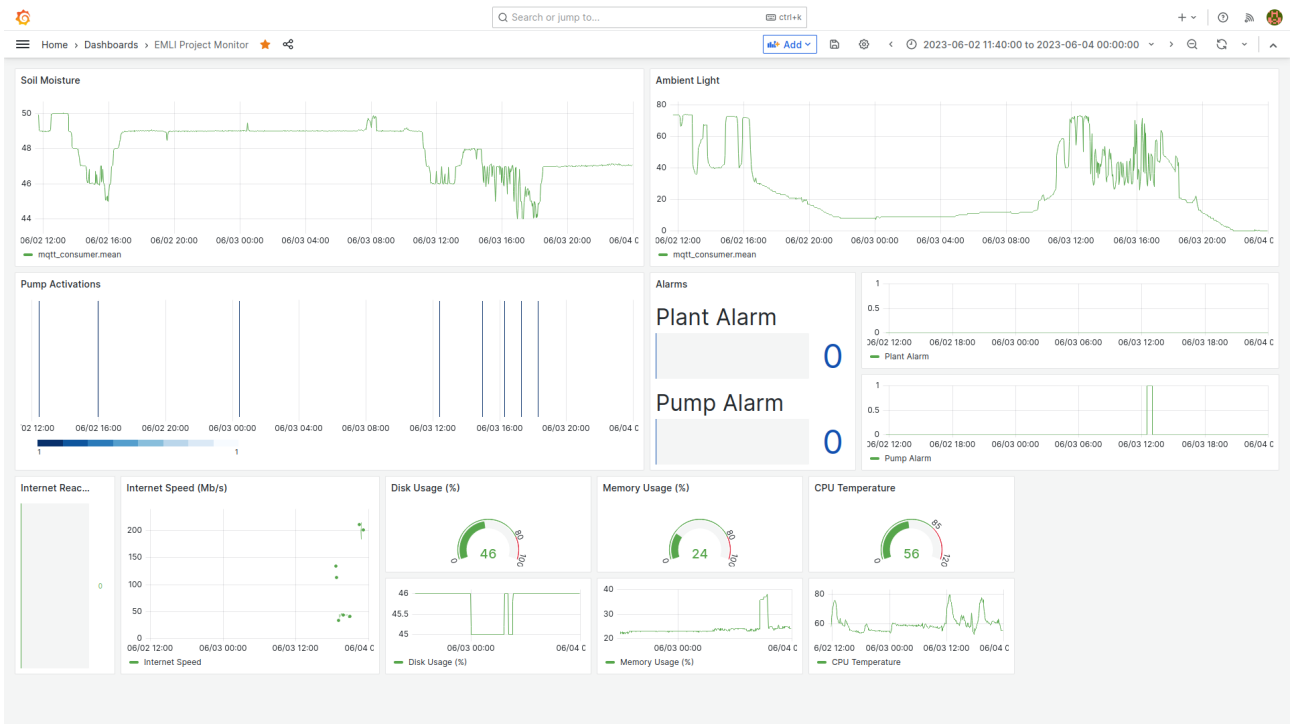

Figure 4: Grafana UI, with the system performing over the course of a couple of days

2.7 Network Security

To increase the security of our system, we implemented a handful of security measures. Those include:

- **SSH keys**: We configured the system to require ssh login with ssh keys. This prevents unwanted users from attempting a brute-force attack on a password login.

```
#on your PC:
mkdir ~/.ssh/rpi4_EMLI/
cd ~/.ssh/rpi4_EMLI/
ssh-keygen -t ed25519
ssh-copy-id pi@10.0.0.10
echo "Host 10.0.0.10" >> ~/.ssh/config
echo " PreferredAuthentications publickey" >> ~/.ssh/config
echo " IdentityFile ~/.ssh/rpi4_EMLI/id_ed25519" >> ~/.ssh/config

#on RPI
echo "PasswordAuthentication no" | sudo tee /etc/ssh/sshd_config
sudo systemctl restart sshd
```

  This however requires making a backup of the ssh key, otherwise system access could be easily lost! An alternative less secure but still good solution would be to use `fail2ban`.

- **Firewall**: We set up a firewall on the Raspberry Pi system to limit network traffic to specific ports required for services to run. This includes ssh (over port 22), MQTT (port 1883) and Grafana (3000). By blocking communication over all other ports, we prevent unwanted inlets, limiting the exposition of the system. This was done using `ufw` (Uncomplicated Firewall) which can be used on Ubuntu installations.

```
sudo ufw allow 22
sudo ufw allow 1883
sudo ufw allow 3000
```

- **Secure Storage**: The system is stored in a locked dorm room with no shared roommates, making it difficult to access the physical system.

- **Unique ID and Password** For the wifi access point and other systems and services that require passphrases, unique IDs and Passwords should be chosen so that they could not be easily guessed (like "pi/raspberry"). However, for some services (like ssh), we felt confident leaving Pi as the user, since access was restricted to ssh keys mode only.

3 Tests and Results

While creating the system we ran many tests to ensure that it was working as intended

3.1 Isolated Tests

While creating and combining the many different parts of the system, we ran tests at intermediate points to ensure that everything would properly interface when put together.

For scripts interacting with MQTT, it was important to ensure the correct input and output. To test the single modules, we attached a subscriber to the relevant topics to visualize the messages and ensure

that the right output was shown. Similarly, messages can be sent manually on the input topics to trigger actions and test the behaviour of the module.

```
$ mosquitto_pub -h localhost -p 1883 -t plant0/button -m 0 -u pi -P
   ↪ raspberry #Example of message sending
$ mosquitto_sub -h localhost -p 1883 -t plant0/soil_moisture -t plant0/
   ↪ plant_alarm -u $USER -P $MQTT_PASSWD -F "%t %p" #Example of
   ↪ subscription
```

To test the system behaviour as a whole, the same approach can be used but by starting more modules at a time. Furthermore, a methodology for simulating the Raspberry Pico has been developed, thanks to the use of `socat`. A pair of device files are created, one of which is the device file that corresponds to the Pico and that the system will use for operation. The other is the interface through which the terminal can write and read from. With the following commands, we obtain an interactive shell through which we can pretend to be the Pico serial device.

```
#SETTING THE TERMINAL PRETENDING TO BE THE DEVICE
#Create the device file /dev/ttyACM0, and another device that is the other
   ↪  end we use to emulate what the device would send and receive
socat PTY,raw,echo=0,link=/dev/ttyACM0 PTY,raw,echo=0,link=
   ↪ devttyACM0_socat &
sleep 1
#This other end (devttyACM0_socat) is the one we now connect our terminal.
   ↪  We can write things pretending to be the device, and read the
   ↪ things the device would receive
cat devttyACM0_socat & PID=$! ; cat | while read LINE ; do echo $LINE >
   ↪ devttyACM0_socat; done
kill $PID
```

This way we can enter a list of “red" values (like for example “0,0,80,20”) and see how the system reacts - for example when the system issues the pump command, a.k.a. when the “p” character is shown on the console. All of this opens the possibility for more convenient and extensive tests, even without the need for physical hardware.

3.2 Integration Tests

After the system was completed, we ran a set of operational tests to ensure the system met all the necessary requirements.

Those tests included:

- Manually controlling the pump by pressing the button multiple times and at different intervals.
- Manually triggering the alarms by placing water where it should not be, or by removing water from the pump tank. We ensured that the proper flags were triggered, and that pump requests would be stopped as required.
- We ensured the remote functionality by running the system through each possible status while observing the remote’s led output. We ensured the red light indicated an alarm, and had a higher priority over the other status options.
- We observed the Grafana output to ensure the readings were being properly logged. We then tested the system during this monitoring to ensure the values behaved as expected.

- We ensured the system could properly restart by running through multiple restarts, without requiring additional commands from the user

Ambient Light Test: To test the range of values from the ambient light sensor, we tested it during the day and night. We recorded the measurement in bright daylight and with the sensor covered at night. The results are shown in Table 4

Input	Value
daylight	74
darkness	0

Table 4: Light Sensor Values

Water Moisture Test: To test the moisture sensor, we tested it in completely dry and well-watered soil. We then added water to the dry soil until it was "suitable" for plants in order to find the threshold used by the **Soil Moisture** and **LED Module**. The results are shown in Table 5

Dry Soil	**Wet Soil**	**Threshold Value**
35	56	45

Table 5: Moisture Sensor Values

System Startup Test: To ensure the system could resume operations even after a loss of power or internet, we cut off access temporarily to these sources and found that the system could properly resume after the disruption.

4 Conclusion

Through this project, we were able to successfully create an automated system for plant-watering. We were able to integrate a set of software tools including MQTT, a database, a UI, and network communications to create a working system that a user can interact with. The system also implemented many hardware components that communicate together to function.

The project was a fun exploration of the applications of embedded Linux design. We hope to take this knowledge further into the future to create many more useful and functional projects.

References

[1] *Sparkfun soil resistive sensor*. URL: `https://www.sparkfun.com/products/13322`. (accessed: 03.06.2022).

[2] *BerryBase soil capacitive sensor*. URL: `https://www.berrybase.de/en/analog-capacitive-soil-moisture-sensor`. (accessed: 03.06.2022).