Bring ideas to life
VIA University College

Green house project

Rokas Barasa 285047

Arturas Maziliauskas 285051

Software engineering technology

6th semester

23/11/2021

Table of content

1 Introduction

Smart greenhouses are becoming more popular with development of IOT devices because they require minimal user interaction and on-site work.

This project is made with purpose to learn relevant technologies used behind IOT devices.

2 Analysis

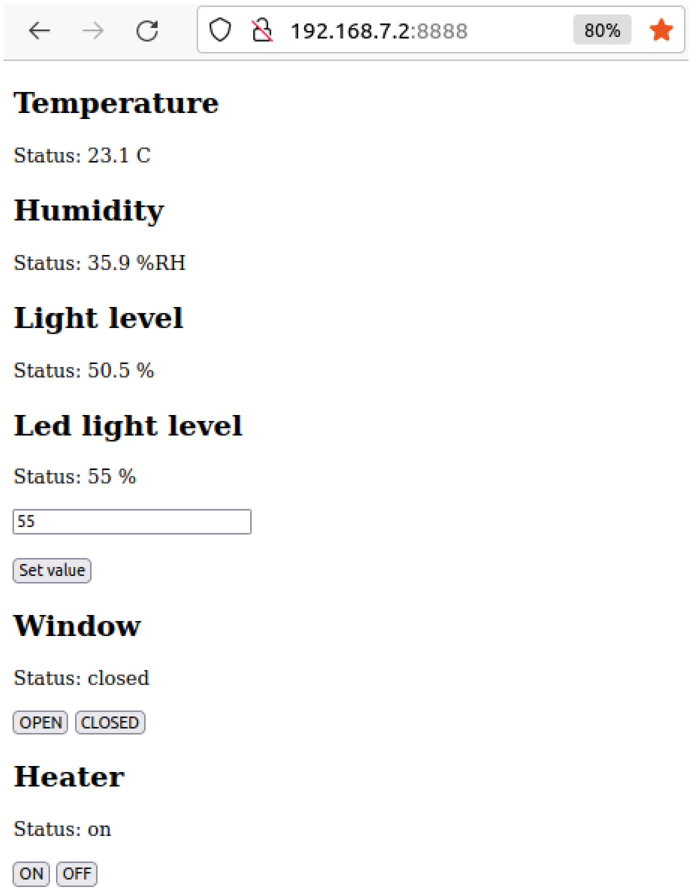
Requirements were given to implement:

1. All user interaction must be via a web interface
2. Administrator action must be possible remotely, via an ssh-connection
3. User must be able to read the temperature in the greenhouse
4. User must be able to read the humidity in the greenhouse
5. User must be able to read the light intensity in the greenhouse
6. User must be able to see window position(open/closed)
7. User must be able to see heater status (on/off)
8. User must be able to set a light intensity level in the greenhouse
9. User must be able to open and close the window in the greenhouse
10. User must be able to activate/deactivate the heater in the greenhouse

3 Design

3.1 Front-end

The user can interact with the greenhouse through a web client that communicates with node.js server. The server returns an html file when the user connects which has the basic user interface and JavaScript functions that call the server and handle user input.


192.168.7.2:8888

80%

Temperature

Status: 23.1 C

Humidity

Status: 35.9 %RH

Light level

Status: 50.5 %

Led light level

Status: 55 %

55

Set value

Window

Status: closed

OPEN CLOSED

Heater

Status: on

ON OFF

The client is configured to call the server every 5 seconds to get the latest state of the greenhouse which returns data about:

- Led light level
- Light level in the greenhouse
- Window status
- Temperature in the greenhouse
- Humidity in the greenhouse

- Heater status

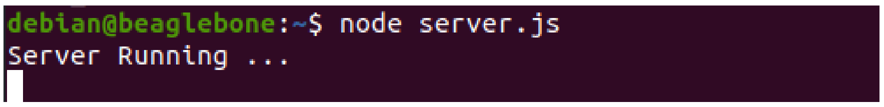
The user can change the status of window (open, closed) or heater (on, off) and set the specific intensity level (0-100%) of led lights in the greenhouse. These functions send out a signal to the back end to perform the functionality. The client is constantly waiting for events from the back end. After each of the change status the user after some time will receive the response from server and based on the response from server it will update the UI with the changed status.

3.2 Sockets.io

The connection between the client and the server is handled by a JavaScript library called Socket.io. It creates a two-way connection between the client and the server though which both can send data to each other. This means that the client does not have to poll the server constantly to see if it has the latest data. Both must maintain the connection by pinging each other instead.

3.3 Back-end

The backend handles the requests from the client. Depending on the client's request, it calls a specific functionality of the C++ executable that interacts with the hardware of the greenhouse.



```
debian@beaglebone:~$ node server.js
Server Running ...
```

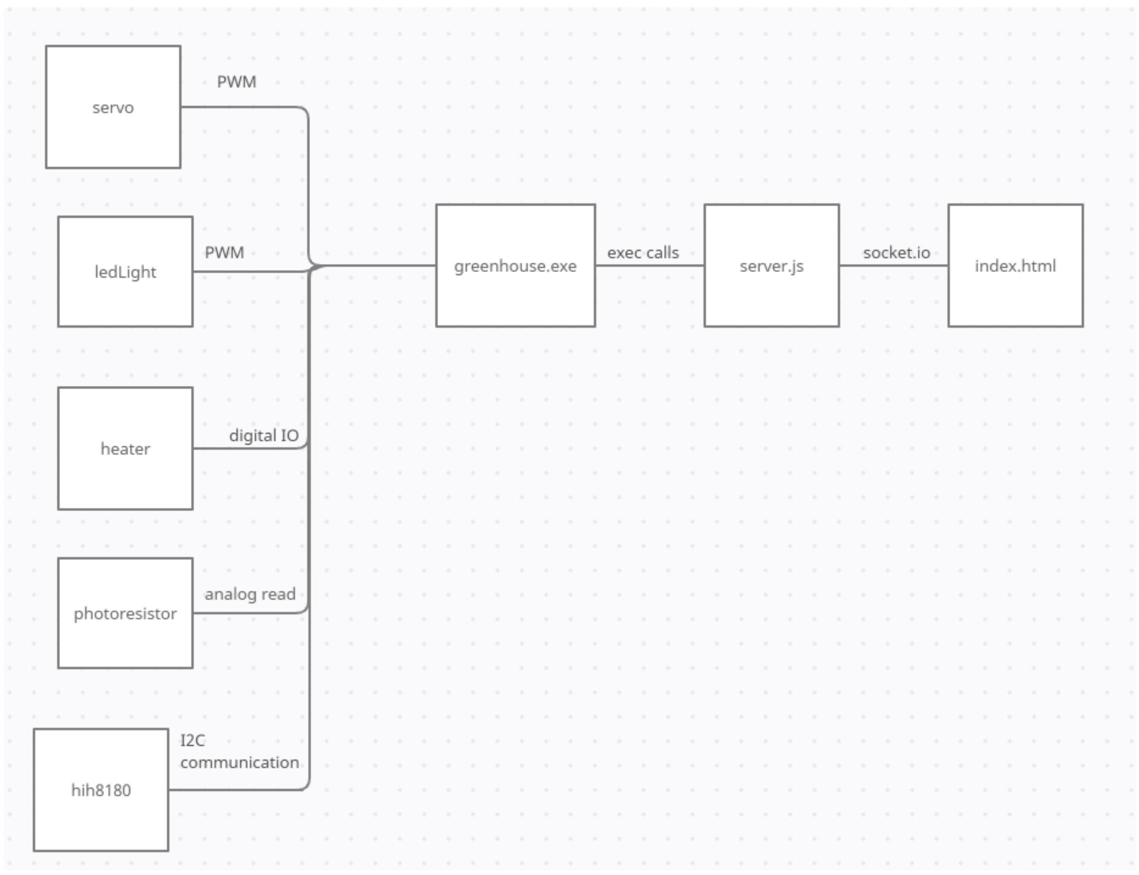
The main functionalities are reading the state of the greenhouse for the client and forwarding the client specified state of peripherals to the greenhouse.

Reading the current state of the greenhouse uses synchronous calls to the C++ executable as all data must be gathered before being sent back to the client. The data to the client is sent back as a JavaScript dictionary file containing each of the peripheral statuses.

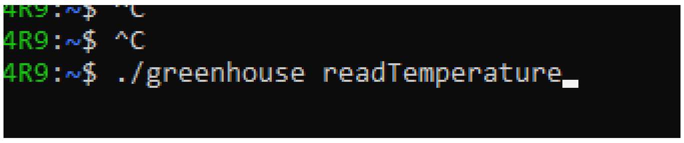
Setting a client specified state in the greenhouse uses asynchronous calls as they are always affecting only one peripheral at a time. After each set state call a read same state call is made to give a response to the client that the state did indeed change.

3.4 C++ executable

The greenhouse executable file is responsible for managing PWM, digital io, analog io and I2C communication to control and read the hardware depending on the selected options.



The executable checks what functionality the user wants to perform and then directs the call to the appropriate function. The function then interacts with the appropriate peripheral to read or modify the status of it.



```
4R9:~$ ^C
4R9:~$ ^C
4R9:~$ ./greenhouse readTemperature
```

When calling the executable at least one parameter must be passed to indicate which functionality to perform. Functionality which modifies the status of the greenhouse needs to have a second parameter passed to declare what state the greenhouse peripheral needs to be in. The function then interacts with only the requested functionality and changes it.

Various helper functions are implemented to accompany this process. Almost all functionality is managed through the editing files that the Debian OS provides.

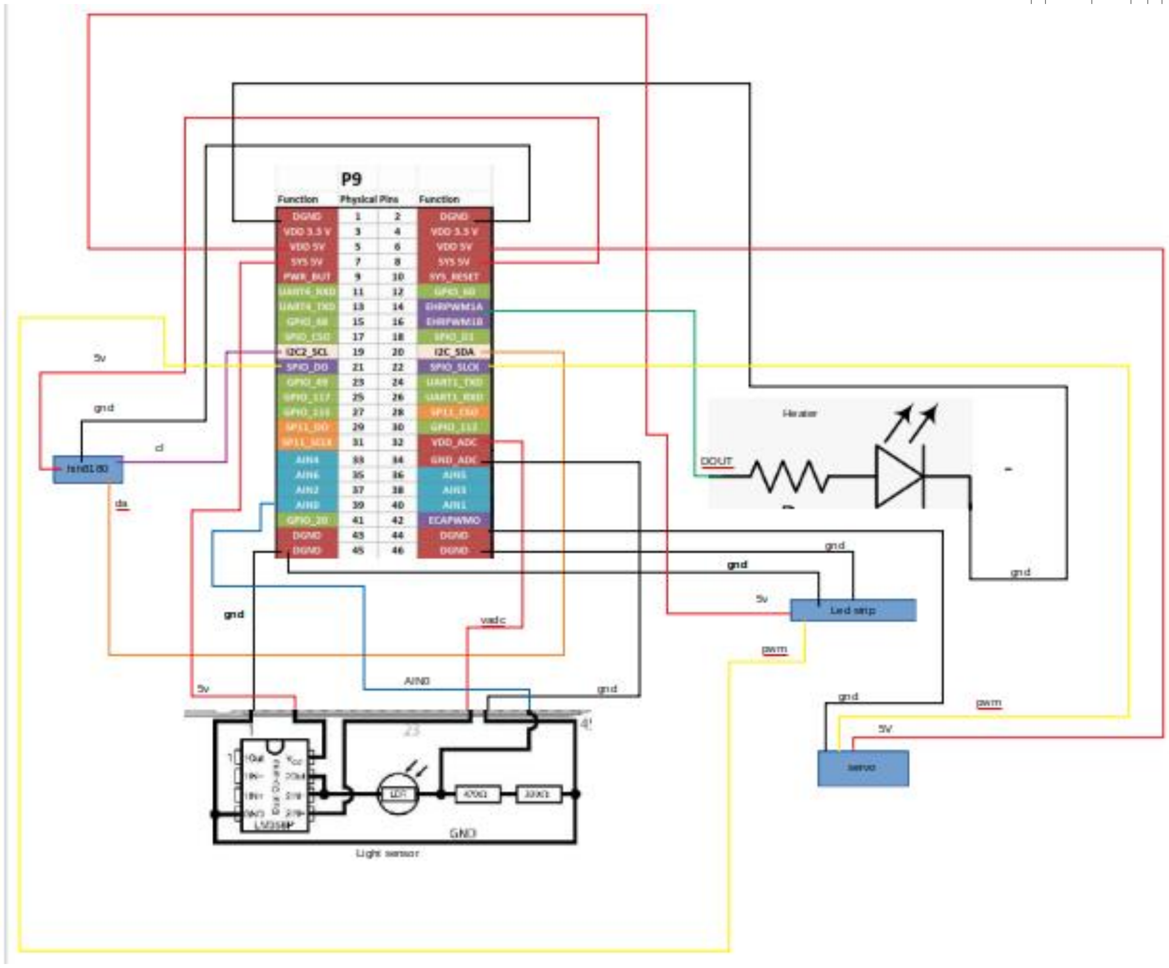
3.5 Beagle bone

The greenhouse hardware is run on a Beaglebone black which uses the firmware image - Debian 10.3 (2020-04-06 4GB SD IoT). The C++ application uses files defined in the OS to control gpio pins on the board.

4 Implementation

4.1 Circuit diagram

All the hardware is connected according to this circuit.



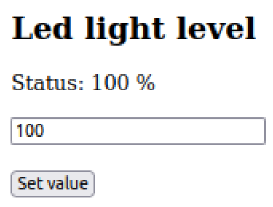
4.2 Connections

This table specifies the connections between beagle bone physical pins and sensor pins

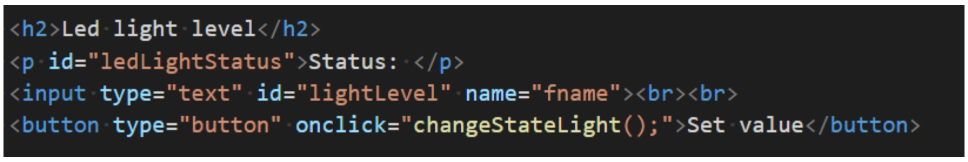
Sensor Pin	BBB physical pin on P9
hih8180 5v	8
hih8180 gnd	2
hih8180 cl	19
hih8180 da	20
LED stripe gnd	44
LED stripe gnd	46

LED stripe 5V	5
LED stripe PWM	21
Servo PWM	22
Servo GND	44
Servo 5V	6
Heater DOUT	14
Heater gnd	1
Light sensor circuit gnd	45
Light sensor circuit gnd	34
Light sensor circuit 5v	7
Light sensor circuit AIN0	39
Light sensor circuit VADC	32

4.3 Example case: User changing led light level

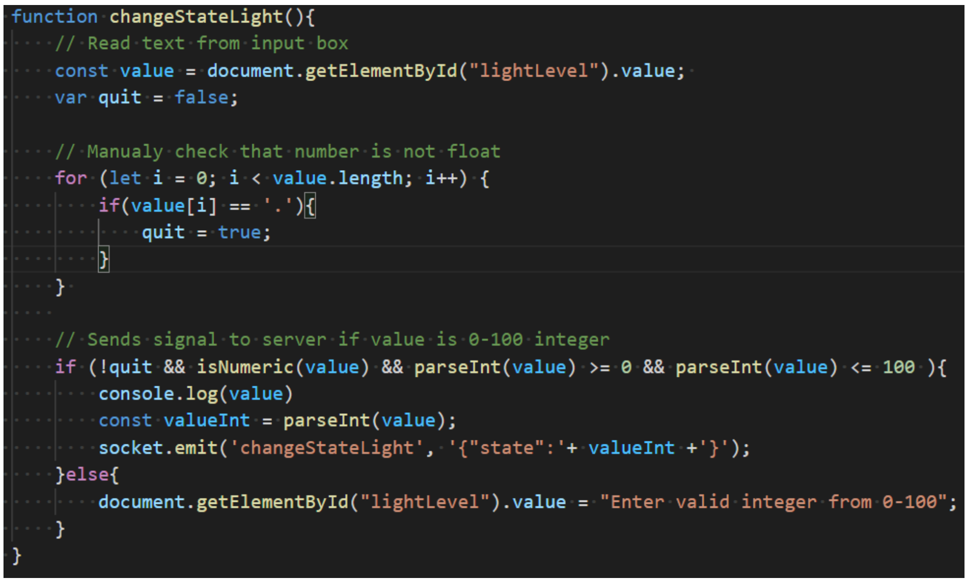


At the base of the changing of led light level functionality is the user putting the correct value and pressing the button “Set value”.



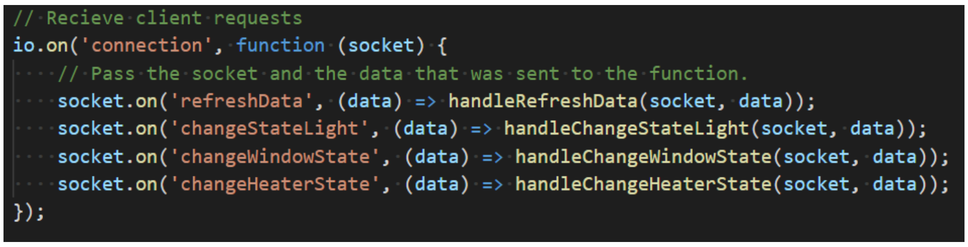
```html
<h2>Led light level</h2>
<p id="ledLightStatus">Status: </p>
<input type="text" id="lightLevel" name="fname"><br><br>
<button type="button" onclick="changeStateLight();">Set value</button>
```

The html tag has specified which function has to handle the event in this case it is the “changeStateLight”.



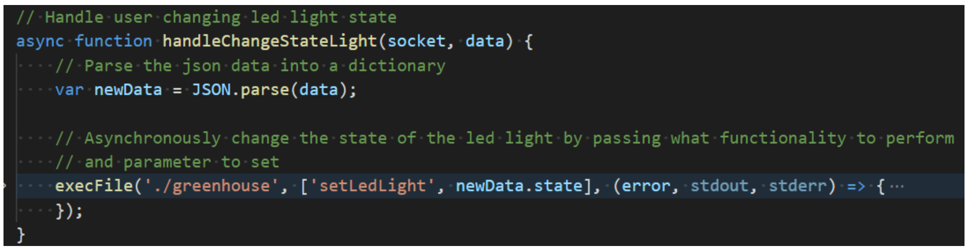
```javascript
function changeStateLight(){
    // Read text from input box
    const value = document.getElementById("lightLevel").value;
    var quit = false;

    // Manualy check that number is not float
    for (let i = 0; i < value.length; i++) {
        if(value[i] == '.'){
            quit = true;
        }
    }

    // Sends signal to server if value is 0-100 integer
    if (!quit && isNumeric(value) && parseInt(value) >= 0 && parseInt(value) <= 100 ){
        console.log(value)
        const valueInt = parseInt(value);
        socket.emit('changeStateLight', '{"state":'+ valueInt +'}');
    }else{
        document.getElementById("lightLevel").value = "Enter valid integer from 0-100";
    }
}
```

The function then checks that the input is an integer from 0 to 100. If the value is correct the client sends the data to the server using the “emit()” function with a specific tag of “changeStateLight”. If the value was wrong, it will give user feedback on how to correct that.



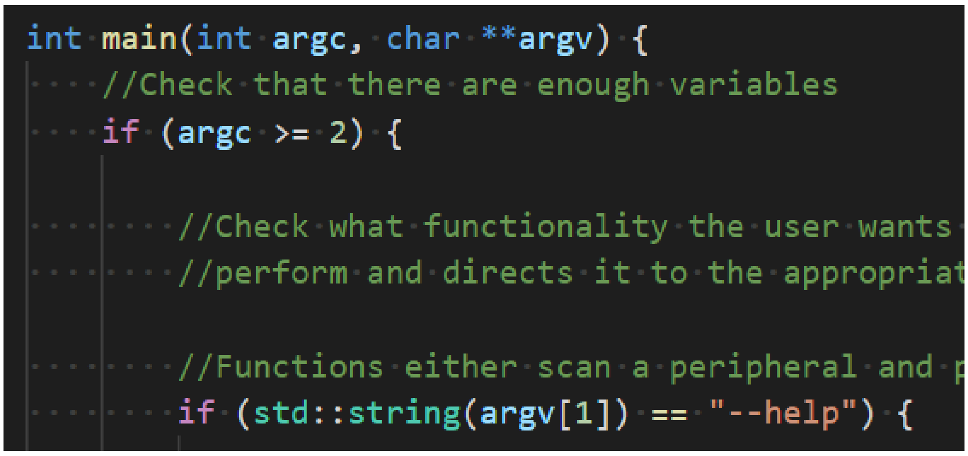
```
// Recieve client requests
io.on('connection', function (socket) {
    // Pass the socket and the data that was sent to the function.
    socket.on('refreshData', (data) => handleRefreshData(socket, data));
    socket.on('changeStateLight', (data) => handleChangeStateLight(socket, data));
    socket.on('changeWindowState', (data) => handleChangeWindowState(socket, data));
    socket.on('changeHeaterState', (data) => handleChangeHeaterState(socket, data));
});
```

The server then directs the request to the appropriate function that handles the request while at the same time passing the socket that sent the request and the data that was sent.

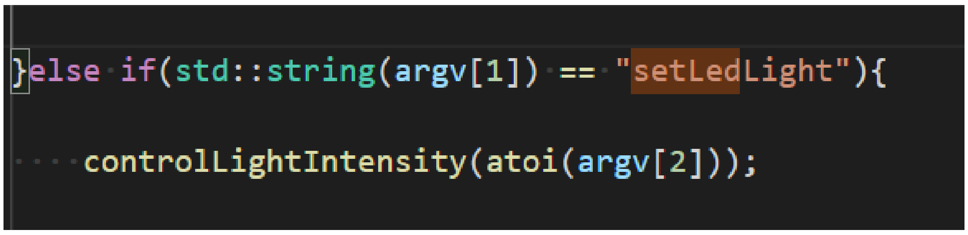


```
// Handle user changing led light state
async function handleChangeStateLight(socket, data) {
    // Parse the json data into a dictionary
    var newData = JSON.parse(data);

    // Asynchronously change the state of the led light by passing what functionality to perform
    // and parameter to set
    execFile('./greenhouse', ['setLedLight', newData.state], (error, stdout, stderr) => {···
    });
}
```

The server then parses the data that was sent and forwards the request to the C++ executable in the same directory as the server using an asynchronous execute call. In this case the functionality that is needed form the executable is "setLedLight” the data that the client sent is passed as well. The server will get the returned output after the executable is finished what it was asked to do.

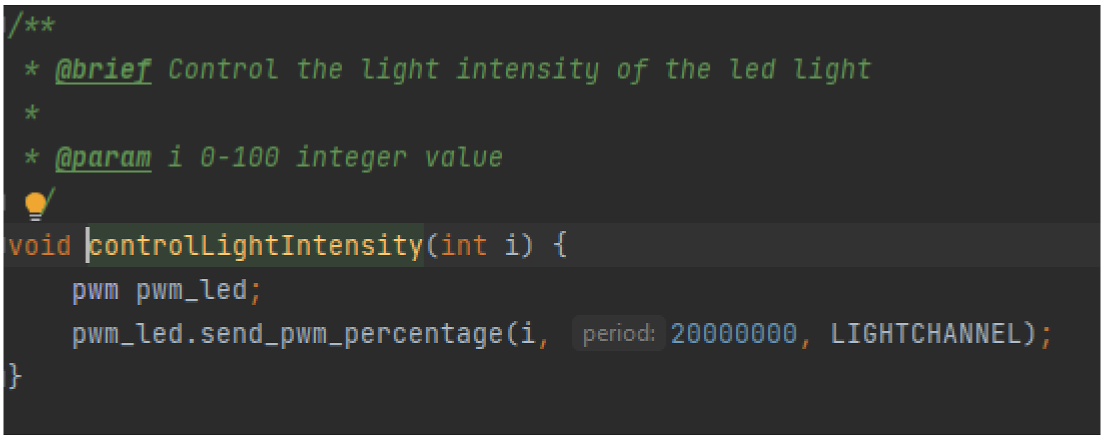


```
int main(int argc, char **argv) {
    //Check that there are enough variables
    if (argc >= 2) {

        //Check what functionality the user wants
        //perform and directs it to the appropriat

        //Functions either scan a peripheral and p
        if (std::string(argv[1]) == "--help") {
```



```
}else if(std::string(argv[1]) == "setLedLight"){

    controlLightIntensity(atoi(argv[2]));
```

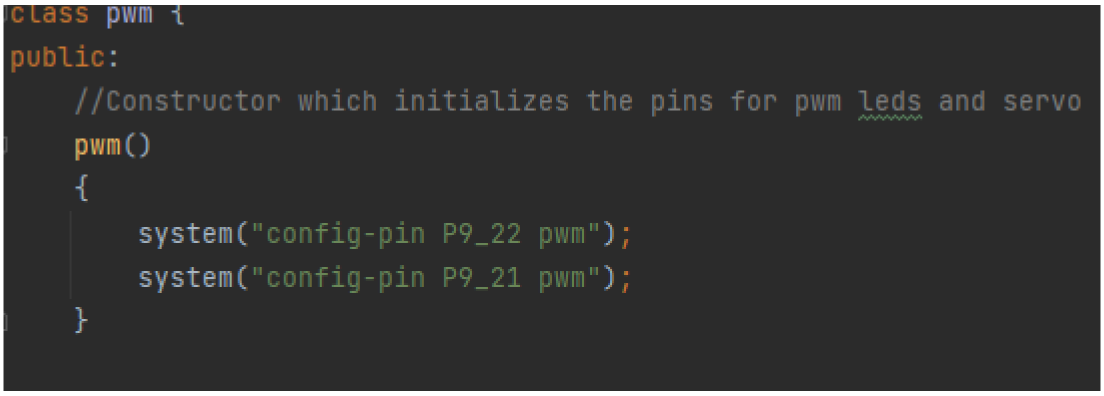
The C++ executable then checks what parameters it got passed, checks the functionality parameter that was passed and directs the call and the second argument to the function that modifies the led light level.



```
/**
 * @brief Control the light intensity of the led light
 *
 * @param i 0-100 integer value
 */
void controlLightIntensity(int i) {
    pwm pwm_led;
    pwm_led.send_pwm_percentage(i,  period: 20000000, LIGHTCHANNEL);
}
```

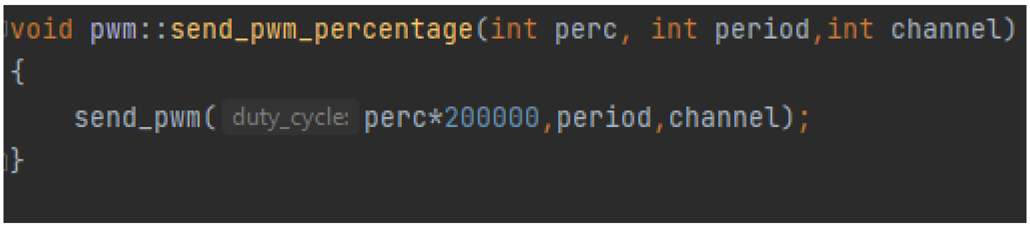
Object of PWM is then created and its send_pwm_percentage function is called which takes percentage parameter, hardcoded period and lightchannel which controls the PWM channels.

During the creation of the PWM object pins 22 and 21 are initialized as PWM.

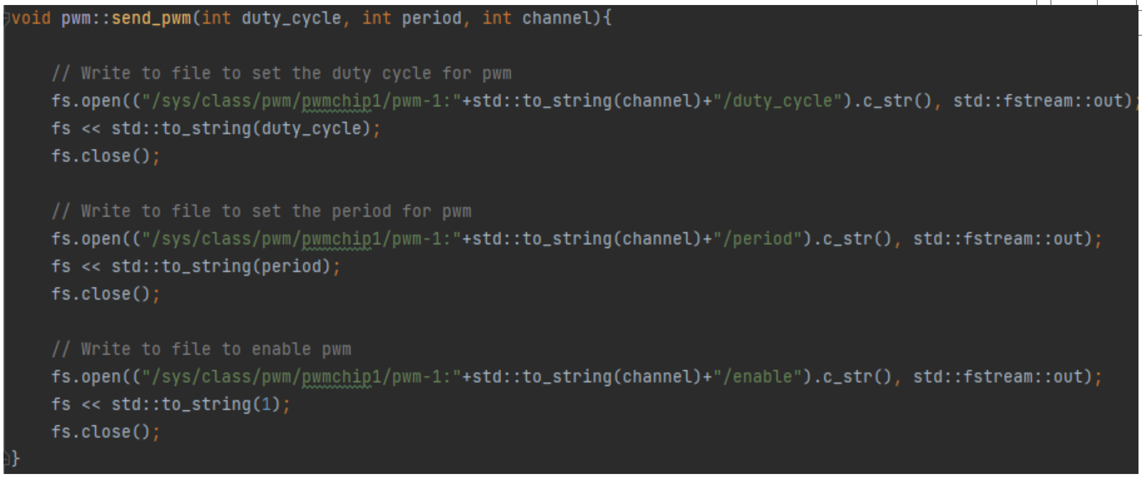


```
class pwm {
public:
    //Constructor which initializes the pins for pwm leds and servo
    pwm()
    {
        system("config-pin P9_22 pwm");
        system("config-pin P9_21 pwm");
    }
```

The helper function is called to convert the percentage to duty cycle and pass it to send_pwm function

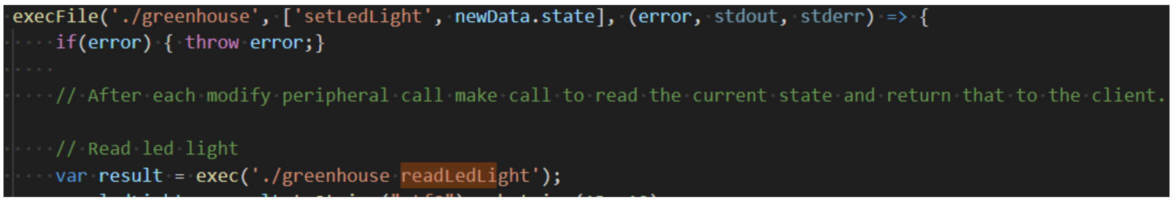


```
void pwm::send_pwm_percentage(int perc, int period,int channel)
{
    send_pwm( duty_cycle: perc*200000,period,channel);
}
```



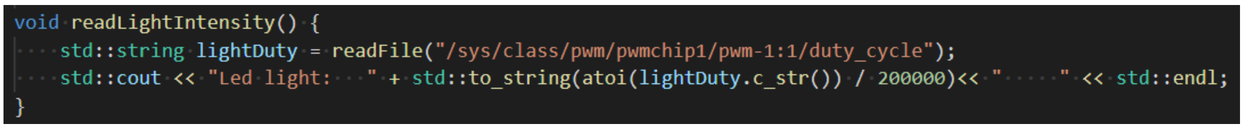
```
void pwm::send_pwm(int duty_cycle, int period, int channel){

    // Write to file to set the duty cycle for pwm
    fs.open(("/sys/class/pwm/pwmchip1/pwm-1:"+std::to_string(channel)+"/duty_cycle").c_str(), std::fstream::out);
    fs << std::to_string(duty_cycle);
    fs.close();

    // Write to file to set the period for pwm
    fs.open(("/sys/class/pwm/pwmchip1/pwm-1:"+std::to_string(channel)+"/period").c_str(), std::fstream::out);
    fs << std::to_string(period);
    fs.close();

    // Write to file to enable pwm
    fs.open(("/sys/class/pwm/pwmchip1/pwm-1:"+std::to_string(channel)+"/enable").c_str(), std::fstream::out);
    fs << std::to_string(1);
    fs.close();
}
```

This function opens and writes passed parameters to the duty cycle, period and enable files. This is the end of the changing set led level functionality in the C++ executable.



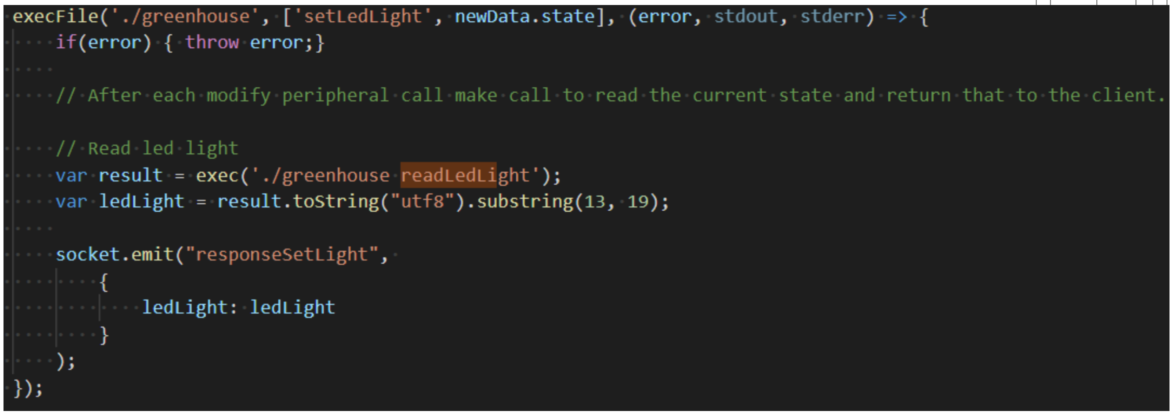
```
execFile('./greenhouse', ['setLedLight', newData.state], (error, stdout, stderr) => {
    if(error) { throw error;}

    // After each modify peripheral call make call to read the current state and return that to the client.

    // Read led light
    var result = exec('./greenhouse readLedLight');
```

The server then gets a call back without errors which signals it to read the status of the led light. This is done using a synchronous call to executable functionality of “readLedLight” and not second parameter.



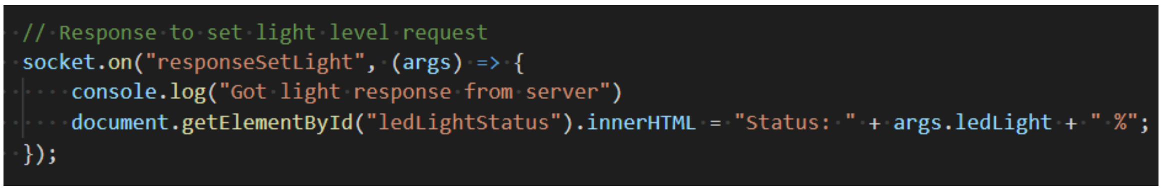
```
void readLightIntensity() {
    std::string lightDuty = readFile("/sys/class/pwm/pwmchip1/pwm-1:1/duty_cycle");
    std::cout << "Led light:   " + std::to_string(atoi(lightDuty.c_str()) / 200000)<< "     " << std::endl;
}
```

The executable directs the call to a function that prints out the current percentage of the led light level.



```
execFile('./greenhouse', ['setLedLight', newData.state], (error, stdout, stderr) => {
    if(error) { throw error;}
    
    // After each modify peripheral call make call to read the current state and return that to the client.

    // Read led light
    var result = exec('./greenhouse readLedLight');
    var ledLight = result.toString("utf8").substring(13, 19);
    
    socket.emit("responseSetLight",
        {
            ledLight: ledLight
        }
    );
});
```

The server then reads and parses the console output that the executable gave and send it back to the client through the socket that was passed to the function.



```
// Response to set light level request
socket.on("responseSetLight", (args) => {
    console.log("Got light response from server")
    document.getElementById("ledLightStatus").innerHTML = "Status: " + args.ledLight + " %";
});
```

The client then sets the response it got from the server as the status of the led light.

5 Test

5.1 Manual testing

Manual testing was used to test the web application. Boundary value analysis was used in order to check how the software would react.
One minor bug was found whenever the application is just started window position is “windo” this is probably due file being empty and misreading of the printed string.

6 Results and Discussion

The greenhouse application and it’s hardware work well.

.

7 Conclusions

While making this project we learned important knowledge about developing IoT software in linux, Beaglebone black hardware platform, embedded communication protocols and interaction with embedded peripherals through gpio pins.

8 Appendices

The purpose of your appendices is to provide extra information to the expert reader.

List the appendices in order of mention.

Examples of appendices

- Source code
- User Guide
- Circuit diagram
- Domain model diagram

Bring ideas to life
VIA University College