EE3L11

Project SUNRISE: Control for Solar Powered e-bike Charging Station

Group D, Team ODROID

Delft University of Technology

T.B. Hosman, 4275691
Job van Staveren, 4317386

Tuesday 14th June, 2016
Version 1.0

TU Delft Delft University of Technology

SUNRISE

Challenge the future

Abstract

In the summer of 2016, a solar-powered charging station for e-bikes (both wired and wireless) and e-scooters will be built on the campus of Delft University of Technology. This station is also meant to provide a testing facility for future projects. The station has a number of systems that generate data: the power electronics (including the solar panels, batteries and grid-connection), a weather station, solar panel temperature sensors and the connected vehicles.

The SUNRISE (Smart Unified Networking Rig for an Integrated Solar E-bike charger) project is a BSc final project that should log and display all this data through an administrator page, a website and a local display. It should also be able to manage e-bike users that would like to connect to the station.

Team ODROID was responsible for the local data management and charger control. The team managed to fetch all necessary data from the power electronics, temperature probes and weather station with C-code running on an ODROID C1+. Furthermore, the developed system was designed from the start to be robust (since it will be running 24/7) and has been thoroughly tested for possible malfunctions. The system has also been made resistant against power and internet outages. Moreover, the system can register e-bike cable connections, which it uses to fully control the chargers. It also provides data necessary to display system information on a local display. Finally, the entire system is remotely controllable through a remote desktop.

Contents

1 Introduction

This project is part of a solar-powered e-bike charging station that will be built on the campus of Delft University of Technology. This station consists of power electronics to manage power between the solar panels, the station batteries, the grid and all the connected vehicles. The station is able to charge e-bikes both wired and wirelessly, as well as e-scooters. The station also has a weather station to monitor the outside climate. Furthermore, the system has a number of temperature sensors to monitor the temperature levels of the solar panels. A concept of the station can be found in Figure 1.

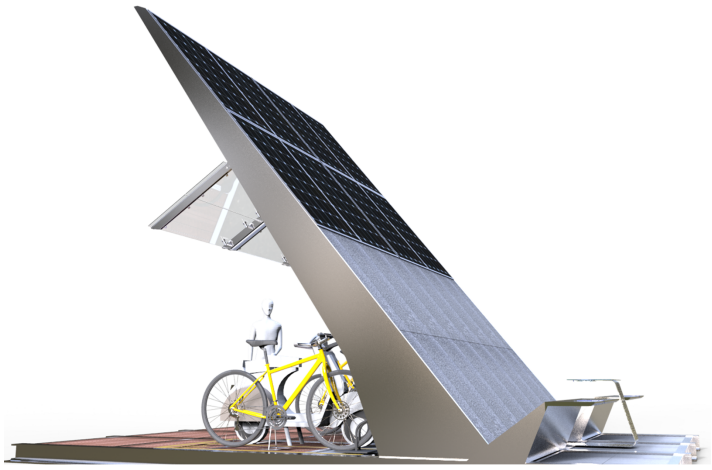


Figure 1: Concept of the solar powered e-bike charging station [1]

The SUNRISE project (Smart Unified Networking Rig for an Integrated Solar E-bike charger) was proposed as a final BSc project. The result of this project should manage the data generated by the power electronics, weather station and temperature probes through a local mini computer and log the information on a TU Delft server. This mini computer should also control a local display that shows live information about the system. SUNRISE should also be able to handle requests of registered users that want to charge their vehicle at the station. The input is processed through a webpage hosted by the university, which should trigger the station to turn on one of the chargers. The webpage will then comunicate to the user what charger to use, after which the user will be able to monitor the state of their battery through the webpage. Finally, the SUNRISE project should have an administrator dashboard to show the status and data of the system, and a media webpage that will display some basic information to the media. An overview of the entire system can be found in Figure 2.

1.1 Problem definition

Team ODROID of the SUNRISE project group will focus on the local mini computer, which does all the data acquisition, as well as the control of the local display. It should also be able to control the chargers and register any charger that gets disconnected. It is important that there is a focus on robustness, because the system should run 24/7. A more detailed description can be found in Section 2.

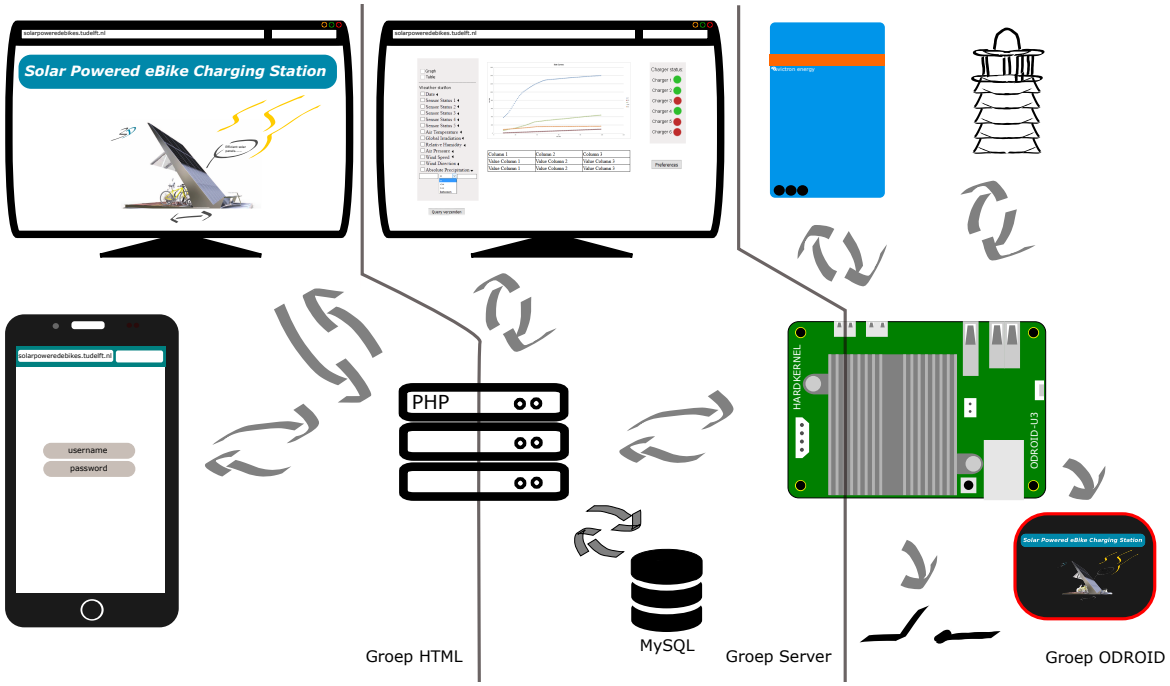


Figure 2: An overall view of the entire SUNRISE system

1.2 State-of-the-art analysis

In the last few years some work has already been done regarding the project as described in Section 1. As a consequence, the work to be done for this design can partially be built on these results.

A major contribution was made in the development of C-code that is able to perform a few conceptual tasks [2]. For example, this code is able to establish a connection with some devices connected to the ODROID and can read a register from those devices. However, it was not tested with all devices (e.g. temperature sensors) and it was not known what registers were read. The code also contains some functions to send data to the server and database, as well as a function to control the local display at the charging station. Beside the code, the report contained general information about the devices and C-code [2]. It also contains a short description of the functions and protocols being used.

The Victron power electronics system, the weather station and the temperature probes all communicate using Modbus. For C there is a library called *libmodbus* [3] that can be used to ease communication. By using this library, the communication can be done at a much higher level using the standard Modbus communication routines. Moreover, this library contains support for both Modbus RTU and Modbus TCP which are needed to read the registers from the Victron power electronics system (Modbus TCP) and the weather station and temperature probes (Modbus RTU).

Another C library that is available for working with the ODROID is the *WiringPi* library [4]. This library is used to easily assign logic values to the GPIO pins of the ODROID.

1.3 Overview

This thesis will start off with the general requirements of the system in the programme of requirements (Section 2). It will then give a general description of all the code modules, as well as the choices made regarding these modules (Section 3). This section will begin with a code overview (Section 3.1) after which it will go into more detail regarding the weather station (Section 3.2), Victron power management system (Section 3.3), solar panel temperature sensors (Section 3.4), chargers (Section 3.5), local display (Section 3.6) and bash scripts (Section 3.7). The thesis will then go over the validation tests and robustness of the system in the results (Section 4) after which they will be discussed (Section 5). The thesis will conclude in Section 6, in which there will also be some future recommendations (Section 6.2).

2 Programme of requirements

In the high-level user and admin interface, status information about the chargers and power electronics is needed, as well as general information about the weather. To obtain this information, the ODROID must read this data from the corresponding components in the system. This data is then sent to the server from which it is eventually displayed in the user/admin interface. Another main task of the ODROID is to turn the chargers for the electric vehicles on and off. Finally, there is an external display placed in the charging station which shows some general information about the weather, as well as availability of charging points.

2.1 Functional Requirements

Functional requirements determine what the design must be able to do and which functionalities it must have. Based on the problem definition the following functional requirements are determined.
The design must be able to:

[1.1] Read the data from the weather station.
[1.2] Read the data from the Victron power electronics system.
[1.3] Read the temperatures from the temperature probes.
[1.4] Control the local display at the e-bike charging station.
[1.5] Show general information about weather and availability of the charging points.
[1.6] Turn the chargers on and off based on server and sensor information.

2.2 System requirements

Requirements that are not functional but are important for the utilisation are the system requirements [5]. In this case these requirements merely put constraints on the software and ODROID once they are installed in the e-bike charging station.
The system must:

[2.1] Be accessible via a remote desktop.
[2.2] Be able to recover itself after a power or internet failure.
[2.3] Run the software 24 hours a day, 7 days a week.

2.3 General Conditions

The general conditions are determined by the framework in which this project is carried out. Other general condition are a result of constraints from the components used for this project. The general conditions are:

[3.1] The system has to be controlled by a minicomputer (i.e. Raspberry Pi or ODROID).
[3.2] Connected devices are bound to a maximum polling rate.
[3.3] The sofware for the mini computer has to be written in C.
[3.4] The charging station can facilitate four e-bikes, one electric scooter and one electric vehicle that can be charged wirelessly.

3 Design and implementation

This section will go through all the designed modules used in the system. The mini computer that is used will be the ODROID C1+ [3.1], which was chosen due to its availability within the DC&S research group. The ODROID C1+ will now be referred to as ODROID.

3.1 Code overview

The C-code [3.3] that will manage all the data on the ODROID consists of several elements. An overview of how these elements and their data are managed can be found in Figure 3.

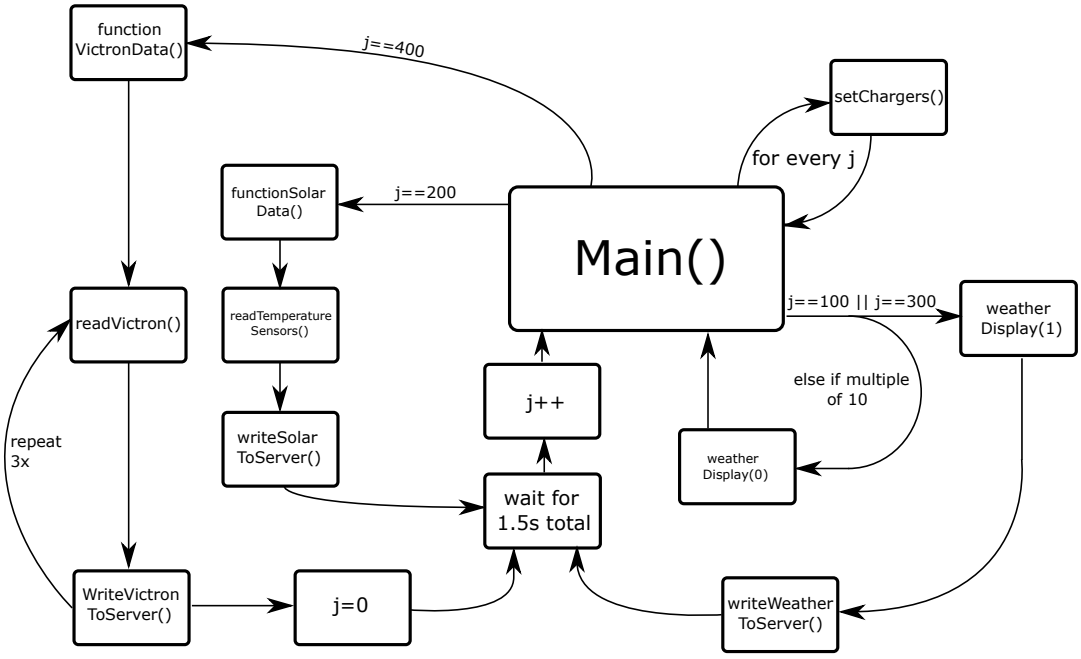


Figure 3: An overview of the C code

The entire program is built around a variable `j` that keeps track of time by incrementing every 1.5 seconds. This variable resets to zero every time it is reaches 400, creating a loop that restarts every 10 minutes.

The ODROID will decide what data to handle based on this variable. For every j, the ODROID will fetch the charger states from the server and set the chargers using the `setChargers()` function. Every 5 minutes (when `j` equals 100 or 300), data from the weather station is fetched and sent to the server (`weatherDisplay(1)`). Every 15 seconds (when `j` is a multiple of 10), data will also be fetched from the weather station, but only to refresh the local display (`weatherDisplay(0)`). To prevent this event from causing two data fetches to the weather station, it is only run if it does not conflict with `weatherDisplay(1)` (by using `else if`). Note that `weatherDisplay(0)` does not increment `j`, so it is possible that it runs together (before) the functions starting at `j` equal to 400 or 200.

Every 10 minutes (when `j` equals 200), data from the temperature probes is read. This data is also sent to the server. A similar loop is also run every 10 minutes (when `j` equals 400) which will read the Victron data. This loop will be run 3 times because there are three seperate chunks of data that will be read (further explained in Section 3.3). Every chunk of data is seperately sent to the server.

3.2 Weather station

One of the devices that is connected to the ODROID is the Lufft WS503-UMB weather station [6]. It is connected to an adapter that converts the Modbus RTU connection to an USB that can be connected to the ODROID (See Figure 4). This station is capable of measuring air pressure, relative humidity, solar radiation, air pressure, wind direction and wind speed. The data that this weather station processes is collected [1.1] and used to display local weather information to both the user [1.6] and the administrator. It could also be used to add functionality, such as disabling charging when the outside temperature is too low [7] or checking if the solar panels are functioning by comparing output power to solar radiation.

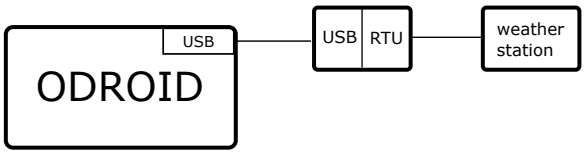


Figure 4: The connection of the weather station

The polling of the weather stations registers is set to roughly every 15 seconds. Although the maximum polling rate [3.2] is technically possible of going down to 10 seconds [8], it was decided that 15 seconds was more than enough to refresh the local display.

3.2.1 Register data

The weather station that is used has a total of 121 registers that can be read. Most registers, however, are merely values derived from another (e.g. US units). It is therefore decided that it is unnecessary to read all the data that can be found in the registers. The registers that are read by the ODROID are:

- Sensor statusses (registers 2 - 6)
- Relative humidity as percentage (register 13)
- Relative air pressure in hPa (register 17)
- Wind direction in degrees (register 18)
- Global radiation in W/m^2 (register 30)
- Air temperature in Celcius (register 34)
- Wind speed in m/s (register 45)

All data will be sent as a `double` array to the server. All data is decoded by casting and scaling functions before sending to the sever, this is done similar to the Victron data (see

Table 1: Sensor status coding

Register	High byte		Low byte	
	High half-byte	**Low half-byte**	**High half-byte**	**Low half-byte**
2	Air temperature buffer	Air temperature	~~Dew point buffer~~	~~Dew point~~
3	Rel. humidity buffer	Rel. humidity	~~Abs. humidity buffer~~	~~Abs. humidity~~
4	~~Mixing ratio buffer~~	~~Mixing ratio~~	Air press. buffer	Air pressure
5	Wind	Wind buffer	~~Precipitation~~	~~Compass~~
6	Global radiation buffer	Global radiation	~~Leaf wetness buffer~~	~~Leaf wetness~~

Section 3.3.4 and 3.3.5). Sensor status data is not sent decoded when cast to a `double`, however it is easier to send as a `double` because it can be placed in the array to be sent. The actual decoding (see Table 1) is done server-side. This method causes redundant sensor data (statusses with a strikethrough in Table 1) to be sent, but this is hardly of any impact on the server.

3.2.2 Read function

The `readWeatherStation` function was made to read registers from the weather station. It was designed to be flexible, so it is easy to change what registers to read. The first argument of the function, `registersToRead[]` is an integer array of a flexible length, where an element corresponds to the register number that has to be read. The second argument is the length of this array. An example of calling this function can be found in Script 1. The registers in this example correspond to the registers mentioned in Section 3.2.1. Note that `readWeatherStation` writes to an array that is initialized at the start of the program and globally writeable (further explained in Section 5.2).

Script 1: Read function

```
1    int registersToRead[] = {2, 3, 4, 5, 6, 34, 39, 13, 45, 30, 17, 18};
2    int length = sizeof(registersToRead)/sizeof(int);
3    if (readWeatherStation(registersToRead, length)){
4        ...
5    }
```

3.2.3 Robustness

A couple of checks are done in `readWeatherStation` to ensure that the system can run 24/7 [2.3]. This is done by returning zero whenever a check fails. The function that handles the data (see Section 3.1) will then check if a zero was returned. Whenever this happens, it will bypass updating the local display and the sending of data to the server. This is done in the `if`-statement of Script 1. These checks are inspired by the previously designed code [2].

One of the checks that is executed every call of the function is the code found in Script 2. In this statement, `rc` (the length of the array that is received from the weather station) is compared to `numberOfRegisters` (the number of registers that is requested). This ensures that all data is received, and that the modbus connection was not lost halfway transmitting. Either way, necessary information is printed and the modbus connection is closed. Memory is freed if necessary.

Script 2: Comparing rc to numberOfRegisters

```
1  if(rc==numberOfRegisters)
2  {
3      for (i=0; i<tabs; i++)
4      {
5          printf("OK, value %f \n",weatherFinalValues[i]);
6      }
7      modbus_close(ctx);
8      modbus_free(ctx);
9      return 1; //return success
10 }
11 else
12 {
13     printf("FAILED (nb points %d) \n",rc);
14     modbus_close(ctx);
15     modbus_free(ctx);
16     return 0; //return failure
17 }
```

Another check can be found in Script 3. This piece of code checks if the connection with the weather station has been successful.

Script 3: Checking if a connection can be established with the weather station

```
1      if (modbus_connect(ctx) == -1){
2          fprintf(stderr,"Connection failed: %s\n",modbus_strerror(errno));
3          modbus_free(ctx);
4          return 0;
5      }
```

Finally, Script 4 checks if all requested registers are available. This should generally not happen if the programmer knows what he is doing, so technically only the `stderr` notification is needed. However, a return value of zero is added to ensure that the rest of the code can be tested even when something went wrong in the weather station. Note that this piece of code also includes the line of code where filtering and scaling is done (line 12). Scaling done in this line is further explained in Section 3.3.5.

Script 4: Checking if the requested registers are available

```
1      for (i=0; i<tabs; i++)
2      {
3          // requested register exceeds range of registers that is available
4          if (registersToRead[i] > (nb_points-1))
5          {
6              fprintf(stderr,"Requested %d weatherstation register unavailable\n",
                   registersToRead[i]);
7              modbus_close(ctx);
8              modbus_free(ctx);
9              return 0; //return failure
10         }
11         //filter and scale
12         weatherFinalValues[i] = ((double)weatherCastValues[registersToRead[i]])/
               weatherScale[registersToRead[i]];
13     }
```

3.3 Victron system

The Victron system is a commercial device that will be placed in the e-bike charging station. It consists of all the necessary power electronics, except for the physical chargers for the e-bikes. Hence the system contains the inverter for the solar panels (BlueSolar), batteries to store the energy and power electronics to control the distribution of power (MultiPlus). It is also possible

Table 2: Part of register list of Victron Energy [10]

dbus-service-name	Description	Address	Type	Scalefactor	dbus-unit
com.victronenergy.vebus	Input voltage phase 1	3	uint16	10	V AC
com.victronenergy.battery	Battery voltage	259	uint16	100	V DC

to extract or deliver energy to the grid if needed. Inside the Victron there is a component called the Color Control GX [9] or CCGX, which is used as a simple panel that displays data and settings. All components in the Victron system are connected to this Color Control.

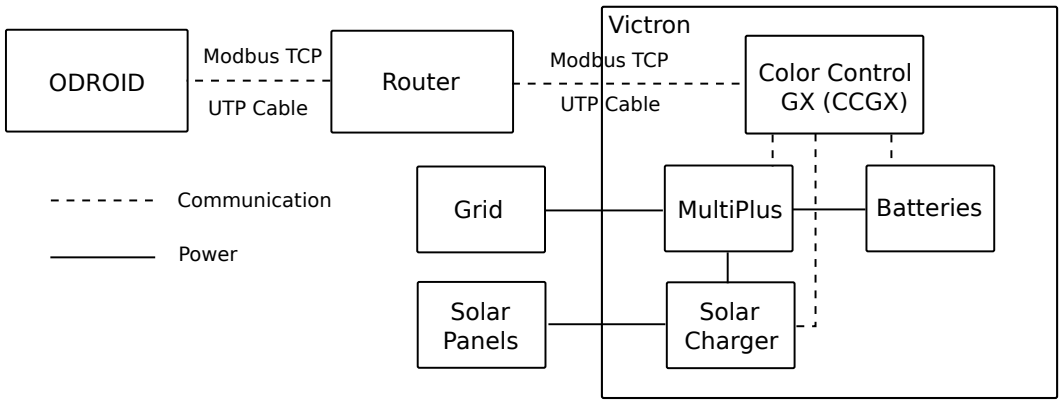


Figure 5: Overview of the Victron system

Figure 5 shows the configuration of the Victron system with the power connections, as well as connections for communication. In order to read data from the Victron system [1.2], it must be connected to a network. This is done using a UTP cable which is connected to a router. The ODROID is connected to this router as well. Using Modbus TCP the ODROID can communicate with the Victron system and read it's registers.

3.3.1 Reading the registers

Reading registers is mainly based on the register list made by Victron Energy [10]. Each entry in this list corresponds to a register that can be read.

Two entries from this list are shown in Table 2. The entire list contains 181 entries. Due to irrelevance some datafields of the two entries are omitted. It was decided that the registers 3-38 (MuliPlus), 259-303 (Battery) and 778-790 (Solar Charger) will be read. All other data will probably not be very relevant for this project. It is worth noting that the registers are not ordered from 0-180, but in blocks of register addresses. E.g. the first block has address 3 to 41, whereas the second block has address 259 to 304 (compensating for this offset is explained in more detail in Section 3.3.3).

Using the Modbus libary for C, it is only possible to read registers between two register addresses, hence the data is read in four seperate function calls. Using the Modbus library functions these registers can be read by specifying a range of addresses and a slave ID. This

slave ID is the ID of the seperate parts whithin the Victron system. On the CCGX monitor mounted on the Victron system a device instance number is shown for the connected devices. Using the second page of the register list [10] this device instance number can be converted to a unit ID, which is equal to the slave ID needed for the Modbus function. For example, the device instance for the batteries is 256, which corresponds to unit ID 247.

Interstingly, the CCGX detects the MultiPlus as two seperate units. One MultiPlus unit on device instance 0 and one on device instance 257. The meaning of this is further explained in Section 4.2.

3.3.2 Robustness

In order to keep the code robust [2.3], the code detects three different types of errors: an invalid register number is requested, the Modbus TCP connections failed or reading of a register failed (these are all similar to the checks explained in Section 3.2.3). For robustness it is desired not to stop the program if a failure occurs, hence it is chosen to let the functions return a 0 if a failure occurs, after which a check is done on the validity of the fetch.

Script 5: Code to detect if an error occured

```
1    /* READ VEBUS DATA FROM VICTRON AND SEND THEM TO THE SERVER */
2    if (readVictron(0,3,38))
3    {
4        writeVictron1ToServer(victronFinalValues); //Victron VEBUS data
5    }
```

Script 5 shows that if an error occured, and a 0 is returned, no data will be sent to the server. In this way there are no errors caused in the code that sends the data to the server and the program can keep running.

3.3.3 Flexibility and offset

In the future it could be desired to read other registers than the registers chosen for this design. To still be able to use the code, it was chosen to implement some flexibility. For example the file `scaleTables.c` contains the scalefactor and datatype of all registers instead of only those read in the current program. However, since the registers are not addressed as 0 to 180, a conversion is needed from the register address to the index of the vector containing scalefactor and datatype. This is done using a variable called `offset`.

Script 6: Select offset from registeraddress

```
1    /* Select correct offset */
2    int offset;
3    if(start >= 3 && start <= 41){offset = 3;}
4    else if(start >= 259 && start <= 304){offset = 220;}    //Batteries
5    /*Some more lines of else if(start >= x && start <= y){offset = z};*/
6    else if(start >= 2800 && start <= 2807){offset = 2625;}
7    else{
8        fprintf(stderr, "Invalid register start value for Victron request");
9        return 0;
10   }
```

Script 6 shows how the offset is selected from the register address. All addresses are mapped onto the range $0, 1, ..., 180$. Now this offset can be used for the scaling and casting purposes explained below. Also note that the last 4 lines return a 0 if the register address does not

exist. Furthermore, it is desired to determine the slave ID based on the requested register address. However, there are two MultiPlus instances hence the same register addresses can be read at different slave IDs. Therefore, the slave ID cannot be uniquely determined based on the requested register address. As a consequence the slave ID must be provided as function argument together with the addresses to be read.

3.3.4 Casting register values

All data received from the Victron system is represented as a `uint16_t`. However, not all the data is actually readable as a `uint16_t` (values that can be positive or negative). Based on the datasheet [10], data is cast from `uint16_t` to `int`. This can be done directly for data that is of the type `uint16_t` (See Figure 6).

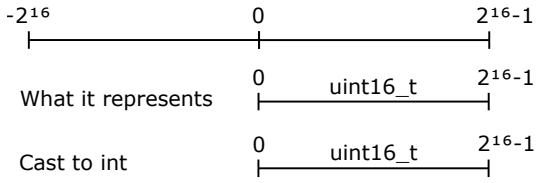


Figure 6: Casting a uint16_t to int can be done without misinterpreting the information

Since the data of type `int16_t` is also represented as a `uint16_t` when received, casting will cause the MSB to be misinterpreted as part of the value instead of the sign (See Figure 7). To prevent this, the data has to be cast twice. First to `int16_t`, and then to `int`.

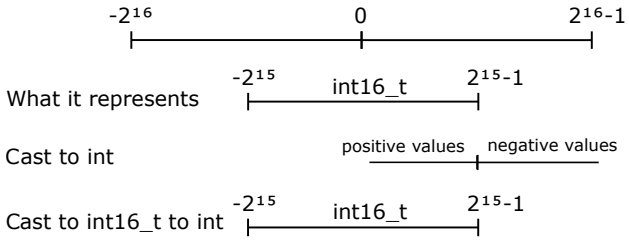


Figure 7: Casting the uint16_t to int causes a misintepretation of the sign bit. An extra cast to int16_t is needed.

The final casting diagram can be found in Figure 8. Any data that represents a string or `int32` is not needed and therefore discarded.

The implementation of this casting sequence can be found in Script 7. A seperate C-file (to keep things organized) holds the array which contains information about the type of data of a certain register. This array is made `extern` so it can be called upon by functions outside

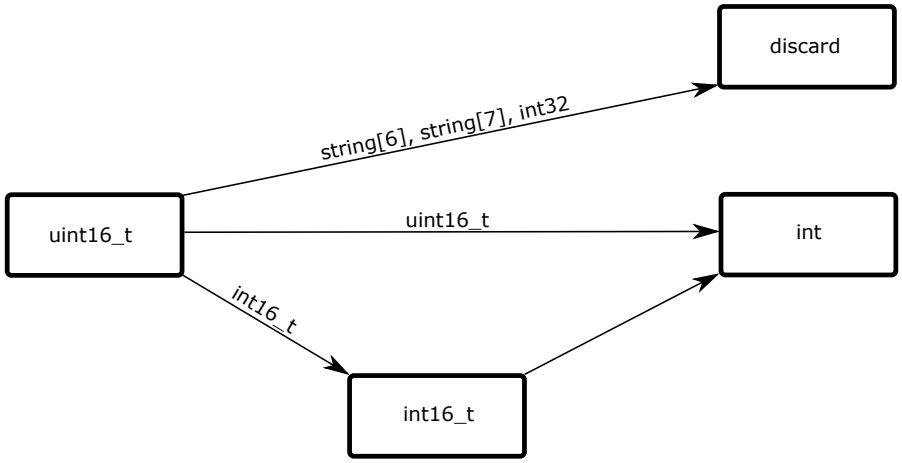


Figure 8: A diagram of how to cast certain data. The text close to the arrows describe how a certain register is logged in the Victron datasheet, the boxes show what type it is represented in. Note that 'discard' means that the value is not used in any further processing.

of this file. The array elements correspond to the Victron registers in the same order as the datasheet by Victron Energy [10]. The array is encoded, where '0' corresponds to a registers of type `uint16_t` (line 9), '1' corresponds to a register of type `int16_t` (line 13) and register values 2 to 4 correspond to values that are discarded (line 18). Discarded values are filled with zeroes so register values do not change of position in the array (line 22). The `for`-loop will loop through all received data, followed by an `if`-statement that checks how the register should be cast.

Script 7: Cast function for Victron data

```
1  /* CASTS VICTRON REGISTER VALUE TO CORRESPONDING TYPE (SEE VICTRON DATASHEET) */
2  int castVictronData(int start, int end){
3
4      int regCount = end-start+1;
5      int i;
6
7      for (i = 0; i < regCount; i++){
8          // register is of type uint16_t
9          if (victronType[start+i] == 0){
10             victronCastValues[i] = (int) victronReadValues[i];
11         }
12         // register is of type int16_t
13         else if (victronType[start+i] == 1){
14             int16_t res = (int16_t) victronReadValues[i];
15             victronCastValues[i] = (int) res;
16         }
17         // ignore registers of type int 32, string[6], string[7]
18         else {//if (VictronType[start+i] == 2 || VictronType[start+i] == 3 ||
               VictronType[start+i] == 4){
19             victronCastValues[i] = 0;
20         }
21     }
22     return 1;
23 }
```

3.3.5 *Scaling register values*

Scaling is done in a similar way as casting of the values (see Section 3.3.4). An array in a seperate C-file, called `victronScale[]`, contains all the scale factors and is made of type `extern double`. Register values are cast to `double` and then divided by their corresponding scale factor (line 3 in Script 8). Note that values of `victronScale` are called upon at an offset to ensure scaling by the correct value, this is explained in Section 3.3.3.

Script 8: Scaling function for Victron data

```
1    /* SCALE VICTRON VALUES BASED ON DATASHEET */
2    for (i = 0; i < regCount; i++){
3        victronFinalValues[i] = ((double)victronCastValues[i]) / victronScale[start-
             offset+i];
4        printf("tab_rp_registers %f, victronscale %f \n", victronFinalValues[i],
             victronScale[start-offset+i]);
5    }
```

3.4 Solar panel temperature sensors

There are four SFCS50-3-3-TB-5 temperature sensors [11] connected to the system to log temperatures of the solar panels [1.3]. The sensors are connected to an ADAM-4015 Data Aquisition Module [12], which is connected to an adapter that converts the Modbus RTU to USB so that it can be connected to the ODROID (See Figure 9).

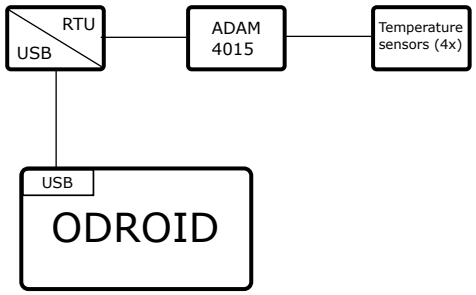


Figure 9: The connection of the weather station

3.4.1 *Casting and scaling of temperature values*

Temperatures received from the ADAM-4015 module are encoded as a `uint16_t`, but it was unknown how the module was programmed so it was yet to be determined what this value represents. To overcome this issue, The "AdamApax .NET Utility" [13] was installed to investigate the settings that the module was operating at. The settings that were found can be seen in Figure 10.

It was likely that the `uint16_t` values could be cast to `int`, and then be scaled between -50°C and 150°C. This would result in a temperature of $\frac{23823}{2^{16}-1} \cdot 200 - 50 = 22.70$°C. To confirm this, the settings were changed to output between -200°C and 200°C. The results can be found in Figure 11.

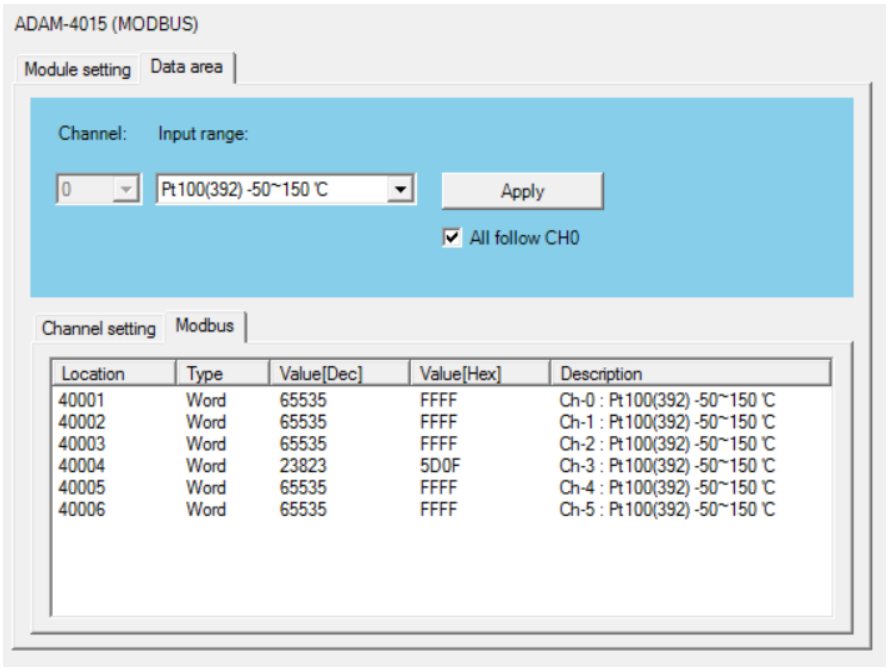


Figure 10: Settings of the ADAM-4015 Data Aquisition Module. Input range (top) shows the temperature range of -50°C to 150°C. The fourth connection shows the value that is used (23823), other sensors are not connected.

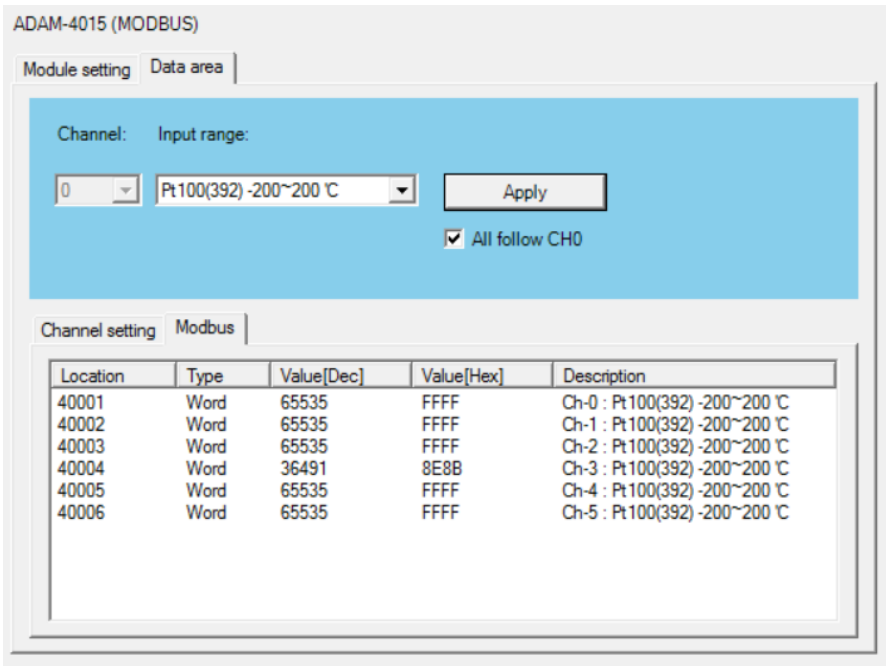


Figure 11: Alternative settings of the ADAM-4015 Data Aquisition Module. Input range (top) shows the temperature range of -200°C to 200°C. The fourth connection shows the value that is used (36491), other sensors are not connected.

This capture was done close after the image in Figure 10 and results in a temperature of $\frac{36491}{2^{16}-1} \cdot 400 - 200 = 22.73°C$, which is similar enough to the previous calculation to conclude that this calculation is indeed correct.

The temperature values are therefore cast using the function found in Script 9, after which they are scaled using the loop found in Script 10.

Script 9: Function used to cast temperature data

```
double* CastTempData(uint16_t *temperatures){

    //Cast the uint16_t array with temperatures to double
    double *double_temperatures;
    double_temperatures = (double *) calloc(NB_SENSORS, sizeof(double));

    int i;
    for(i = 0; i < NB_SENSORS; i++){
        double_temperatures[i] =(double)((int)((int16_t)temperatures[i]));
    }

    return double_temperatures;
}
```

Script 10: Function used to scale temperature data

```
for(i = 0; i < NB_SENSORS; i++){
    double tmp = cast_temperatures[i]/65535*200-50; //scale linearly between -50 and
        150 degrees
    scaled_temperatures[i] = roundf(tmp*100)/100; //round on two decimals
}
```

3.4.2 Hotswap

Due to the fact that the ADAM module gives values for ports that are available but not connected (65535) it is possible to implement a hotswap functionality to the temperature sensors. This means that it is possible to remove or add temperature sensors while the system is operational. The value of 65535 corresponds to 150°C and is not considered a realistic value for solar panel temperatures. This value will be sent to the server but will not be logged. It will be discarded instead. Whenever a sensor is added, however, this value will change and the server will know that a temperature sensor was added. Similarly, it will know when a sensor was disconnected by checking if the temperature value equals 150°C.

3.4.3 Robustness

A couple of checks are done in `readTemperatureSensors` to ensure that the system can run 24/7 [2.3]. This is done by returning zero whenever a check fails (for example, a failed connection). The function that handles the data (see Script 3.1) will then check if a zero was returned. Whenever this happens, it will bypass updating the local display and the sending of data to the server.

The function will first check if a connection can be established to the ADAM module, this is done similar to the check in `readWeatherStation` (see Script 3). After reading the registers containing the temperature data, the function will check if all registers were read using the code found in Script 11.

Script 11: Checking if the received data corresponds to the amount of requested registers

```
1 if(rc!=NB_SENSORS){
2     printf("FAILED (rc=%d) \n",rc);
3     modbus_close(ctx);
4     modbus_free(ctx);
5     return 0;
6 }
```

3.5 Chargers

3.5.1 Controlling the chargers

One of the main functionalities of the ODROID is to turn on the chargers if a user wants to charge an e-bike [1.6]. Turning on the chargers is done using a relais that is controlled by the ODROID. The relais can be turned on or off by using a signal from a GPIO pin.

At this moment the physical chargers that will be used in the charging station are not developed enough to be used for test purposes, hence another way to test the C-code must be designed. An easy way to check if the GPIO pins are turning the chargers on or off is by using LEDs that are connected to the GPIO pins. However, it is not desired to draw current for the LEDs directly from the GPIO pins. Since the driving current limit for the GPIO pins is 2-4 mA (not every pin has the same current limit) [14], it is neccesary to use an external driver for the LEDs. The easiest way of driving a LED is by using a voltage controlled switch. The voltage from the GPIO pins then determines whether the switch is open or closed. In electronic components, such a voltage controlled switch can be made by using a transistor [15]. To behave as a closed switch the transistor must be in saturation, because then the collector to emitter resistance is very low and the transistor almost acts as a short circuit. For the transistor to behave as an open switch, it must be in cutoff mode.

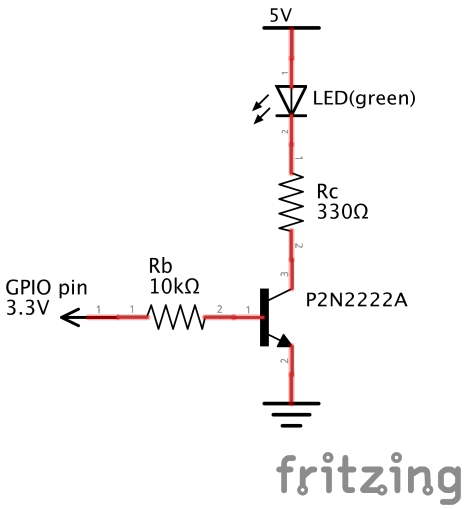


Figure 12: Circuit used to drive the LED. Made using Fritzing [16]

The transistor used for this circuit is a bipolar P2N2222A NPN transistor. This is a general purpose transistor that can be used for power amplification as well as switching applications requiring collector currents up to 500mA [17].

Figure 12 shows the circuit that is used. Resistor Rc is chosen such that the current through the LED is limited if the LED turns on, hence a maximum current of $\frac{5}{330} = 15mA$ can be drawn from the source. However, since there is also a voltage drop over the LED this current will be much smaller and well below the current limit for this LED.

In order to act as a buffer, the circuit should draw as less current as possible from the GPIO pins of the ODROID. This is achieved by choosing a very high base resistance Rb. However, a base resistance that is too high results in malfunctioning of the transistor because not enough current is fed into the base.

When the transistor is in saturation the base voltage is about 0.7V thus the base current is about $\frac{3.3-0.7}{Rb}$. To achieve a base current of 0.1mA a resistance Rb of 26kΩ is needed. However, after a test this resistance was found out to be too large. A smaller resistor of 10kΩ solved this problem.

The circuit shown in Figure 12 is built six times and then connected to the six GPIO pins of the ODROID that are used to control the chargers [3.4].

3.5.2 Detection of the charging cable

To increase the level of safety for the users the chargers should not be turned on when there is nothing connected to them. Therefore a charging cable detection system is designed. For each cable one additional GPIO pin is needed (of course the ground pin on the ODROID is also needed). In case of a connected charging cable, this GPIO pin is shorted to ground (inside the charging cable) and thus a logic 0 is set to this pin. In the other case (no connected charging cable), the pins keep floating. However, the ODROID can be programmed such that it internally connects a pull-up resistor to the pin. Now a logic 1 is detected whenever no charging cable is detected. Within the C-code this can be detected by using the `digitalRead()` and `pullUpDnControl()` functions.

Figure 13 depicts the circuit that is used for this purpose. Since there are no available charging cables yet, a switch was used in this circuit to simulate the behaviour of the cable. It was also chosen to include a 2.7kΩ resistor in series with the switch. As a testing circuit some precautions have to be taken to ensure that nothing can be damaged as a result of short circuits. This also results in a choice for the value of resistor R2. The lowest current limit of all GPIO pins is 2mA. Together with a maximum voltage of 5V on one of the pins, a 2mA current limit means that R2 must be at least 2.5kΩ.

But R2 may limit the short circuit current, it also influences the voltage at the pin. The voltage division on R1 and R2 makes that it is no longer possible to obtain 0V at the GPIO pin. With the switch in closed state, the voltage is $\frac{R2}{R1+R2} \cdot 3.3 = \frac{2.7}{62.7} \cdot 3.3 = 0.14V$. Altough this voltage is not zero, it is nevertheless low enough for the ODROID to detect it as a logic low or 0. With the switch in opened state, this voltage division does not occur and will thus not impose

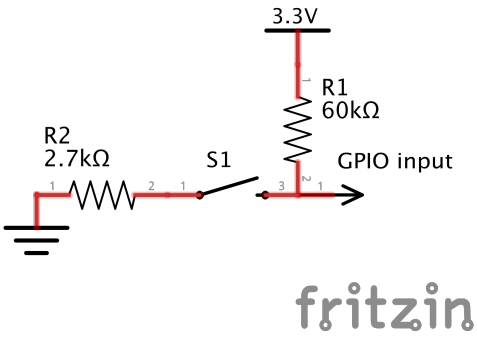


Figure 13: Circuit used to detect the connected cable using a GPIO pin [16]

problems with respect to the input voltage.

3.5.3 Control mechanism for the chargers

When a reservation is made in the end-user interface the state of the charger in the database is set to 1. Using the C-code on the ODROID this data can be retrieved from the server. When calling the function `getChargerState()` an array of length 6 is returned, containing the server state of all chargers. As explained in Section 3.1 the chargers are set every 1.5 seconds using the function `setChargers()`.

For the design of this control mechanism some additional functionalities were set up to improve user-friendliness and safety. These are:

- When a reservation is made on the website, but the owner of the e-bike does not show up within 60 seconds, the charging point is released. This ensures that it is not possible to claim charging points without actually charging.
- The chargers will only be switched on when a cable is connected for more than 6 seconds.
- In case of a cable that is being disconnected during charging, the chargers will turn off, but remain reserved for 15 seconds. A cable that is plugged out accidentally can now be replaced without having to login again.
- The admin-dashboard must be able to turn off a charger, even when a vehicle is connected.

To implement these functionalities, it can be seen that three different timers are needed and thus some form of memory to store these timers. Moreover, the chargers must refresh or reset every time the variable `j` from Figure 3 is incremented (every 1.5s). All these functionalities can now conveniently be combined in a Finite State Machine (FSM) that uses a change in `j` as the clock. Figure 14 depicts the finite state machnine that was designed for this purpose.

The diagram starts in the reset state, which is the OFF state. It only leaves this state when a reservation is made and the server says that a charger must be turned on. In that case the finite state machine goes into the USERTIMER state. Here it waits until a cable is connected. When no charging cable is connected within 60 seconds, the FSM goes back to the OFF state and sends

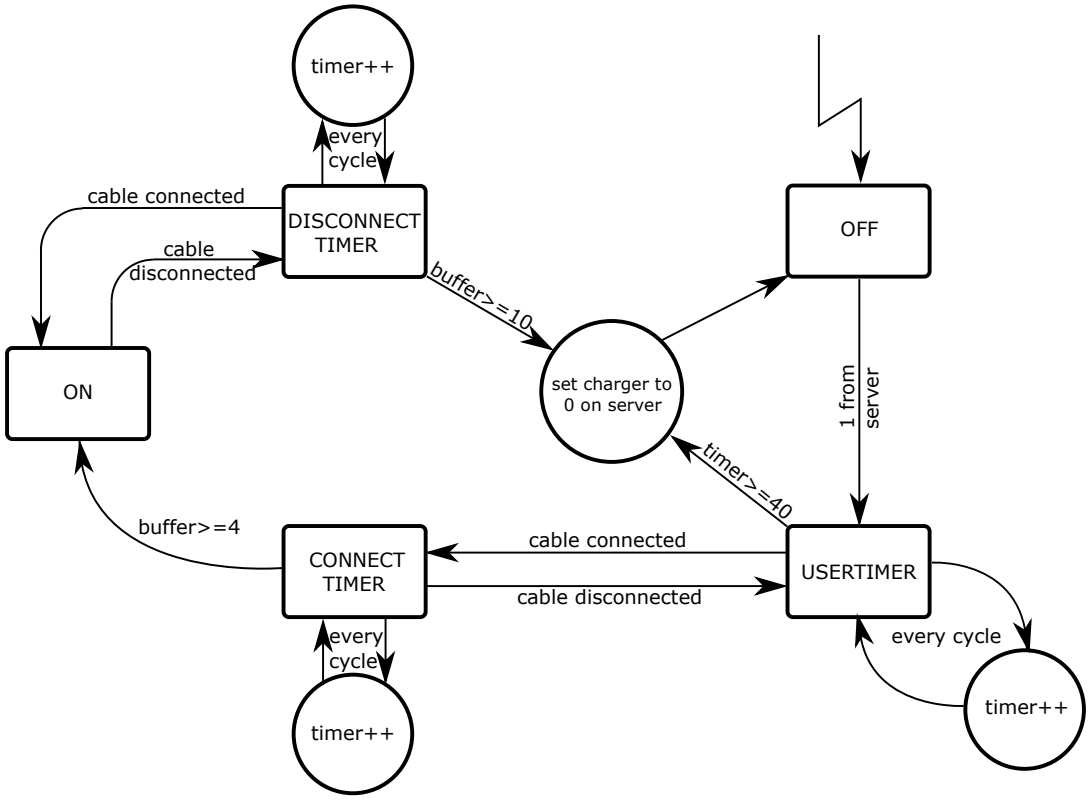


Figure 14: FSM to control the chargers

a message to the server that the charger can be freed for other users. In the other case (charging cable is connected within 60 seconds), the FSM goes to the CONNECTTIMER state. Here it waits 6 seconds until it moves on to the ON state where the chargers are switched on. However, if the cable is disconnected within these 6 seconds, the FSM returns to the USERTIMER state. Now the cable must be plugged in for another 6 seconds before the chargers are switched on.

Now suppose the e-bike is being charged and the finite state machine is in the ON state and the charging cable is disconnected. This can be due to two reasons: the e-bike is charged and leaves, or the e-bike is accidentally unplugged. Both cases will make the FSM move to the DISCONNECTTIMER state where the charger is turned off. If the cable is reconnected within 15 seconds, the FSM moves back to the ON state and the charger is turned back on again. On the other hand, after 15 seconds of no reconnection, the charger is left off and the FSM moves to the OFF state.

Finally, if the FSM moves to a next state, the timer of the corresponding state that is left, is reset. All three timers are reset when a charger is turned off on the admin-dashboard. The FSM then always moves to the OFF state.

To obtain a Moore machine, the output signals must only depend on the current state. Hence the chargers are turned on only in the ON state, whereas in all other states, the chargers

are off. Using the C library *WiringPi* [4] it is possible to easily write a logic 0 or 1 to the GPIO pins and effectively turn the chargers on or off [18]. The GPIO *WiringPi* pin numbers used for this purpose are 6, 10, 11, 12, 13 and 14 for the chargers and 21, 22, 23, 24, 26 and 27 for the cable detection [19]. Each of the six chargers is controlled by an FSM and hence 6 FSMs are neccessary to control all chargers. It is chosen to implement this by using a `for`-loop.

3.5.4 Internet failure failsafe

A failsafe was implemented which ensures that e-bikes are not connected for too long when internet is unavailable. This is important because the station cannot be monitored online during an internet outage. A timer was made that keeps track of how long the internet has been down for. It was decided that all chargers should be turned off whenever the internet is out for more than 2 hours. This time was chosen because it enables the station to finish the charging of any e-bike that is already connected, while also limiting potential risks of an unmonitored system.

Script 12: Internet failure failsafe

```
1     ...
2     static int noConnectionTimer;
3     ...
4
5     /*  SET THE PHYSICAL CHARGERS */
6     if(chargerStateServer)
7     {
8         ...
9         noConnectionTimer = 0;
10        ...
11    }
12    /* NO CONNECTION COULD BE ESTABLISHED */
13    else
14    {
15        /* ADD ONE TO TIMER */
16        if (noConnectionTimer) noConnectionTimer++;
17        else noConnectionTimer = 1; //starting without connection
18
19        /* CHECK IF CABLE IS UNPLUGGED */
20        for (i=0; i<6; i++){
21            if (digitalRead(detection_pins[i])) // Cable disconnected
22                ...
23        }
24
25        /* CHECK IF TIMER HAS BEEN OFF FOR 2 HOURS */
26        if (noConnectionTimer > 4800)
27        {
28            /* TURN ALL CHARGERS OFF */
29            for(i=0; i<6; i++)
30                ...
31        }
32    }
```

The implementation of this failsafe can be found in Script 12. In this function, the first `if` statement checks if the data fetch was successful. In this statement, the `noConnectionTimer` is set to zero. The `else`-statement is executed whenever connection was not successful. In this part, the `noConnectionTimer` is incremented (or set to one if the system just booted with no connection). The timer value is then checked if it is larger than $2 \cdot 60 \cdot 60 / 1.5 = 4800$ (one poll every 1.5 seconds), and the GPIO pins are set to low if it is. Note that the `for`-statements in the case of no connection manage the charger GPIOs and the FSM states as described in Section 3.5.3 (omitted for the sake of simplicity).

3.6 Local display

3.6.1 Display data file

To display data on a local webpage [1.4] [1.5], it was decided that it would be the most convenient to write data to a text file, which would then be read by a script that controls the webpage. Because this read script would most likely be a JavaScript file, it was thought that the text file should be of the .json format. The code that is used to print this .json file can be found in Script 13.

Script 13: Writing a .json file for the local webpage

```
1  FILE *f = fopen("localDisplayData.json", "w");
2  if (f == NULL){
3      printf("Error opening file!\n");
4  } else {
5      fprintf(f, "{");
6      fprintf(f, "\"temperature\": %f,\n", weatherFinalValues[5]);
7      fprintf(f, "\"wind-chill\": %f,\n", weatherFinalValues[6]);
8      fprintf(f, "\"humidity\": %f,\n", weatherFinalValues[7]);
9      fprintf(f, "\"wind speed\": %f,\n", weatherFinalValues[8]);
10     fprintf(f, "\"radiation\": %f,\n", weatherFinalValues[9]);
11     fprintf(f, "\"air pressure\": %f,\n", weatherFinalValues[10]);
12     fprintf(f, "\"wind direction\": %f\n", weatherFinalValues[11]);
13     fprintf(f, "}");
14     fclose(f);
15 }
```

3.6.2 Testing the local webpage

A simple webpage was made to ensure that running the local webpage by reading the text file is possible. A problem could arise, for example, because the two programs are both accessing the same file.

The JavaScript file can be found in Script 14. The script makes use of *Ajax* [20] to asynchronously refresh the webpage, so it will refresh the data without refreshing the entire webpage. It will read the `localDisplayData.json` file every second (line 11) and will write this information to an HTML file (see Script 15). This writing is done in line 4 to 6 of Script 14.

Script 14: Test code for reading the .json file

```
1  function updateWeather() {
2      $.ajax("../_odroid_server/bin/Debug/localDisplayData.json").done(function (
           response) {
3          console.log(response);
4          $("#localTest").html("");
5          for (var i in response)
6              $("#localTest").append(i + ": " + response[i]).append("<br>");
7      });
8  }
9
10 updateWeather();
11 setInterval(updateWeather, 1000);
```

Script 15: HTML file for displaying the data of the .json file

```
1  <html>
2  <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.8.3/jquery.min.js"></
       script>
3  <script src="LocalDisplay.js"></script>
4  <div id=localTest></div>
5  </html>
```

3.6.3 WebGL

The webpage, designed by another team within project SUNRISE, makes use of WebGL to display the information generated by the ODROID. A guide made by Hardkernel was used to install WebGL on the ODROID [21], after which a tip on their forum was used to enable WebGL in Chromium [22].

At first, these solutions, including many others, were unable to get WebGL running on the ODROID, eventhough the ODROID should be able to run WebGL. A lot of time was spent trying to fix this, after which it was decided that a good option might be to simply use a fresh install of the operating system. This was done by using a new ODROID C1+ (so the previous ODROID could still be used as a backup). This device was able to run WebGL after using the earlier mentioned sources. It is unclear why this was the case, but it is thought that the previous ODROID was either partly broken or had some setting disabled by a previous developer.

During the configuration of the new ODROID, it was decided that it would be convenient for future development to make an instruction file containing all elements that need to be installed and configured to be able to run all the systems. This instruction can be found in Appendix A.

3.6.4 Remote desktop

The ODROID should be accessable remotely because it is not easily reachable when it is integrated in the charging station [2.1]. There are two elements required for this to work: a VPN connection to the router in the station and a remote desktop client. For the VPN connection, the OpenVPN client [23] was used. This required VPN files that can be downloaded from the router.

During the first setup of the VPN this solution did not work. It took some time to figure out that the downloaded files were not configured correctly and needed to be edited before they could be used. A more detailed description can be found in the setup manual (see Appendix A).

Finally, “Remote Desktop Connection” (installed default on Windows) was used to connect to the ODROID. For the ODROID to facilitate this connection, it needs a couple of packages. This too is described in detail in the setup manual (see Appendix A).

3.7 Bash

For the system to correctly function, a couple of scripts had to be designed in Bash. These scripts take part in ensuring that the system remains functioning [2.3] or is able to recover after a power failure [2.2]. All scripts are booted on startup of the desktop using the *Upstart* [24] package. A `README` file was made on the desktop of the ODROID to describe the scripts and holds information about how to turn them off (See Appendix B.1). This was done to ensure easy future use.

3.7.1 Xvnc killer

One of the first observations that was done when the remote desktop functionality was working, was that the ODROID was significantly slower whenever someone connected. This is as expected, because the ODROID simply has more information to process. However, this issue remained even when all remote desktops were closed. It became an even greater issue when the ODROID could not handle all the processing anymore, after which it would completely freeze

for up to an hour.

It was later discovered that a program called `Xvnc` would boot whenever a remote desktop connected, and would not shut down when the user disconnected. A Bash script was therefore made that would count the number of open `Xvnc` processes, and that would kill the oldest if it exceeded a certain amount (See Script 16). This was done in a similar way for a processes called `ssh-agent`, which booted in conjuction with `Xvnc`. `Xvnc_killer.sh` is booted on startup using an `Upstart` file (See Appendix B.3).

Script 16: Xvnc_killer.sh

```
#!/bin/bash
# If there are more than 4 remote desktops open, eject the oldest

while true; do #Continue indefinitely

    if [ $(ps aux | grep 'ssh-agent' | grep -v '<defunct>' -c) -gt 4 ]; then #if more than 3 ssh-agents are active
        kill $(ps aux | grep 'ssh-agent' | grep -v '<defunct>' | sort -n -k 9 | awk '{print $2}' | head -1) #kill oldest ssh-agent process
    fi

    if [ $(ps aux | grep 'Xvnc' -c) -gt 4 ]; then #if more than 3 Xvnc processes are active
        kill $(ps aux | grep 'Xvnc' | sort -n -k 9 | awk '{print $2}' | head -1) #kill oldest Xvnc process
    fi

    sleep 60 #check every minute

done
```

The script uses `ps aux` to list all the processes, `grep` to search for a specific process, `grep -v '<defunct>'` to remove zombies (if a process did not shut down correctly it remains in the processes list without using any memory) and `-gt` to check if the search returns a value greater than 4. If there are more than 3 clients connected (the search function also finds itself as a process), it will kill the oldest running client. This is done by sorting by the time the process booted by using `sort -n -k 9` (this will sort by the 9th column: start time), picking the process ID column by using `awk '{print $2}'` (this prints column 2: process IDs) and then picking the oldest process using `head -1`. The `kill` command kills the process that has this ID. Finally, the script sleeps for a minute, after which it repeats the cycle.

3.7.2 Browser booter

The browser booter was designed to automatically boot a browser for the local display [1.4]. The display should only show the local webpage, nothing else. This implies that it has to boot full-screen and that it should not display a scroll bar nor a mouse. An *Upstart* file was made to boot the Chromium browser (See Appendix B.6). This boot uses two flags: `--use-gl=egl` to enable WebGL features used in the local webpage and `--kiosk` to boot full-screen. A bash script was made to start the *Unclutter* [25] package on startup, which was set to hide the mouse after 5 seconds of inactivity (see Appendix B.4 and B.5).

3.7.3 C-code booter

The C-code booter was made to boot the C-code on startup (see Appendix B.7 and B.8). This can be useful when the system has been troubled by a powercut [2.2]. The booter will also check if the program remains running, and it will reboot whenever it finds that it is not. Several flags

in the command will ensure that the terminal does open (using `-hold`), but not in front of the browser (using `-iconic`). The `su` command is used because the code requires a SuperUser (because of the control of GPIO pins).

4 Results

4.1 Weather station

To verify correct functionality of both the weather station and the communication with the ODROID, a simple test setup was conducted to test all the used sensors. Most sensors could be read at any time (temperature, humidity, air pressure). However, both the radiation sensor and wind sensors would register zero because there was not enough light and wind to trigger these sensors. The weather station was set up at the window on a bright day to test the radiation sensor, and a fan was placed to check if the wind was received correctly. During these tests, all sensor values were correctly read and sent to the server. This confirmed the functionality of both the weather station and the communication with the server.

4.2 Victron system

For reading the registers of the Victron system the Modbus libraries are used. By printing the read values to the screen it was easily verified that the ODROID was able to read the Victron system.

To determine what the registers with respect to the power represent, a load was connected to the system. An easy way to do this is by connecting a large power resistor to the system. The power dissipated in this resistor was recognized as DC power on the Victron system.

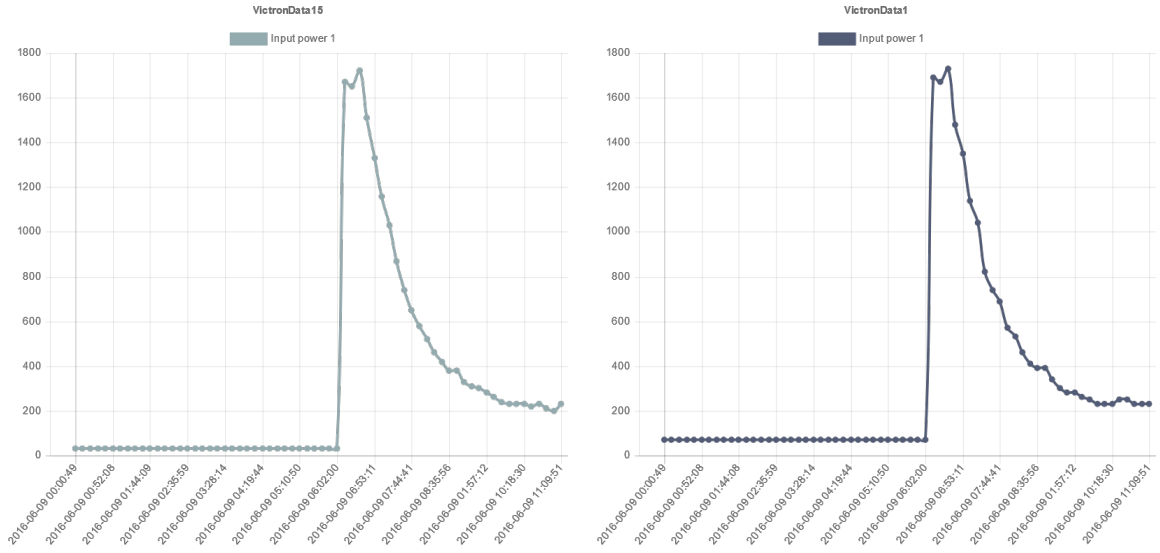
Some 230V sockets are also connected to the Victron system. The ODROID, router and display are connected to these sockets. By switching the display on, an increase in AC power was detected in one of the MultiPlus slaves (slave ID 0). The other MultiPlus slave, however, did not register it, so it was clear that although they are the same physical device, they do register different information.

To find out what they are measuring, the systems were both monitored for roughly a day. The results of input power can be found in Figure 15. The peak of roughly 1.8kW is the moment that the Victron system starts charging the batteries with energy from the grid.

At first, these graphs look identical. However, closer inspection reveals that the device at slave ID 0 has a standby power of 30W, and the other device has a standby power of 70W. There are also minor differences in the peak during charging. Similar results are found for the output powers.

Concluding, it is thought that MultiPlus devices both monitor the AC power regarding the batteries. External AC connections (e.g. router, local screen, etc.) are found in the MultiPlus device with slave ID 0, and all internal AC power (all the Victron devices) are monitored by the MultiPlus device with slave ID 246. The last guess is mainly based on the fact that the 70W standby does not change when anything is connected, and 70W is in the order of power usage of the Victron systems.

Another important result is that it is not possible to read all the registers that are mentioned in the register list. Most of these unreadable registers are a result of devices that are not connected due to the modularity of the Victron system. Moreover, it was not possible to read the state of health of the batteries on register address 304, despite the fact that there is a battery



(a) MultiPlus on slave ID 0 **(b)** MultiPlus on slave ID 246

Figure 15: Input power of the MultiPlus slaves

management system inside. It is unclear why this register cannot be read, but it is likely that it is simply not supported by the batteries that are used.

4.3 Chargers

The chargers are tested in a conceptual way by controlling the LEDs connected to the GPIO pins. Beforehand it is of course necessary to test the driver that is used for the LED as shown in Figure 12. During the test a 5V and 3.3V supply were used to deliver the energy.

First, the value of the highest possible Rb was determined. When using a 22kΩ resistance the transistor did not function as desired, hence a smaller resistor needed to be selected. A 10kΩ resistance Rb was enough to ensure that sufficient current could enter the base of the transistor. However, the current through the base is inversely proportional to the resistance Rb. This is a result of the constant voltage drop of about 0.7V over the base-emitter junction. A smaller Rb thus results in a higher base current and hence an increased current drawn from the GPIO pin.

When the transistor is in saturation, a voltage of about 0.69V was measured at the base terminal. The voltage at the collector terminal was about 0.073V. Now it can be concluded that the transistor is indeed in saturation. At the base, the voltage is higher than at the collector and the emitter. Using the fact that the P2N2222A is an NPN transistor it means that both the base-emitter junction and the base-collector junction are forward biased. Using these measurements and the value of Rb the base current is $\frac{3.3-0.69}{10k} = 0.26mA = 260\mu A$.

A voltage of 1.69V was measured over Rc; hence the current through the LED equals 5.1mA which is enough to make the LED clearly visible. By using the state of the chargers fetched from the server, the GPIO pins were set to a logic 0 or 1. A change in the state of a charger in

the database was indicated on the LEDs in less than 1.5 seconds.

The FSM controlling the chargers was tested by connecting both the LED array (described above) and the switch described in Figure 13. All the state changes were tested on their functionality (by printing all current states) as well as their timing (counting the number of cycles until a state change). This resulted in no significant problems, however it was evident that not all functionality was present. The main issue was that the charger would not respond to the administrator turning off a charger. This was solved by making an `if`-statement that sets the local charger state to OFF for all chargers that are set to 0 on the server. This force-statement resets all timers used in any of the states.

4.4 Solar panel temperature sensors

The temperature probes were easily tested by reading the registers from the ADAM module to which they are connected. Using the Modbus routines to read registers data was obtained. During the test, only one temperature sensor was connected (there are 5 more slots available to connect temperature sensors). From the registers that were not connected to a temperature sensor, the value $2^{16} - 1 = 65535$ was returned, which means it is unconnected but functioning. Register 4, to which a sensor was connected, returned a value between 0 and $2^{16} - 1$. Using the mapping as described in Section 3.4 this value was mapped to a temperature, which indeed corresponded to the current room temperature.

4.5 Local Display

The file that contains the data for the local display is correctly generated every 15 seconds by the display function. The webpage that is going to be displayed in the actual charging station is yet to be finished by another team in project SUNRISE, thus the actual implementation of reading this file is yet to be finalized. However, all other necessary precautions have been taken regarding this webpage: the ability to run WebGL and the ability to automatically boot all display programs (see Section 4.6). A small script was also written to test the reading of the file, and the data was displayed and refreshed without any issues (see Figure 16).

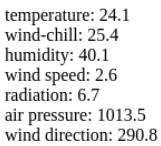


Figure 16: Output of the test webpage

4.6 Bash

The bash scripts all function as designed. The scripts can boot the browser on startup, boot and reboot the C-code, ensure there are not too many remote desktops connected and hide the mouse so it does not display on the local display.

4.7 General testing

4.7.1 Memory leaks

Although the C-code was written with the utmost care, chances are that not all memory allocations are freed. Because the code is running 24/7, any memory leaks would sooner or later become an issue for the system. It was therefore necessary to ensure no memory leaks were created by the code.

A Linux tool called "Valgrind" [26] was used to check the code of any memory leaks. Valgrind has a tool specifically designed for this: memcheck. Two memory leaks were found by this tool, both caused during casting of data. The problem was that a variable that would contain the result of the casts function had memory allocated. However, the same memory allocation was done within the cast function. The earlier mentioned variable would be overwritten by the data of the cast function, causing it to lose the location of the memory that was allocated to it. A memory leak was mentioned by Valgrind, and some manual searching of the code led to the core of the problem.

In later stages of the design, it was decided that memory allocations should be removed alltogether (if possible). This is further explained in Section 5.2.

4.7.2 Long term tests

The ability of the code to run for a longer time is a very important aspect to test. Multiple tests were conducted in which the code would run overnight by using Valgrind. During these test all available functionality of the code was used (reading the devices, controlling the GPIO pins and sending and receiving data from the server). Both tests resulted in a failure of the code. The database showed that data had been received up until 3 a.m. and then nothing more was received. The terminal window in which the program was running showed that wrong data was received on the ODROID about the state of the chargers. Only zeroes and ones should be received, however some random values were received which caused the program to crash. This is probably a result of a kind of reset that is done every night on the server of TU Delft. The server code did return a null pointer when no data was received, but this result was not checked and was therefore still written to the pins. This was probably random memory data, which finally crashed the program.

In the end, roughly 6 long-term tests were conducted, ranging between 24 and 96 hours. The duration of a full cycle (Figure 3) was not very constant, but it did not deviate more than about 15 seconds from the 5 or 10 minute cycle. This will not be a major issue since this deviation did not accumulate after several cycles.

4.7.3 Robustness tests

To guarantee robustness, it was important to unplug and replug all ODROID connections multiple times. This reveiled some minor bugs regarding server connectivity, but ultimately resulted in sturdy code that will simply reconnect whenever a device becomes available again.

A more interesting result occurred when unplugging both Modbus RTU connections (temperature probes and weather station). This sometimes resulted in the devices not being found

after reconnecting. It was soon noticed that the order in which the devices are connected is important. Due to limited development time, it was not considered important enough to dynamically allocate these USB connections to make the ODROID independent of connection order (if at all possible). Until that time, the USB of the temperature sensors should be plugged in before the weather station.

5 Discussion

5.1 Desynchronization

Although the loops described in Section 3.1 take only a couple of milliseconds to execute, it is technically possible that they take more than 1.5 seconds to complete. For example, when `j` equals 400, both `weatherDisplay()` and `functionVictronData()` have to be executed. This could potentially cause a small desynchronization.

To overcome this, functions could be seperated in smaller chunks. For example, splitting the three Victron functions, or even seperating fetching and sending of data. This will decrease the chance of a function taking more than 1.5 seconds to execute. Another solution could be the use of threads, so whenever a thread is unable to finish within 1.5 seconds, it will simply run parallel to the main function. With this implementation one should be careful not to have a weather station function and a temperature probe function fetching data at the same time, because the ADAM-4015 module cannot handle two parallel requests.

This potential issue is, however, not likely to occur in practice and will make the code unnecessary complex. Functions are simply executed too fast to cause any desynchronization (e.g. fetching and sending Victron data, weather data and charger data takes only 0.23 seconds at most), and even if it should happen, there is only a desynchronization in the order of a few seconds, which is not an issue. The sleep function (that is used to halt the program when 1.5 seconds have not yet passed) has also been secured by not executing whenever a negative amount of seconds is remaining (when it is taking more than 1.5 seconds), because it will run indefinately if a negative amount is entered.

Although the problem mentioned above has not occurred during testing, a cycle would still take longer than expected (10 seconds longer every 10 minutes). This desynchronization is caused by the code between calculating the remaining time until 1.5 seconds and sleeping (see Script 17). Line 3 and 6 therefore take roughly $\frac{5}{5*60} = 25$ms to execute. This issue can be solved by decreasing `loop_time` by 25ms. However, because this is not really an issue (and it is not really predictable), it was decided that this correction was unnecessary complex, thus it was not implemented.

Script 17: A cause of small desynchronization

```
1    diff = clock()-start; //calculate remaining time
2
3    printf("the time taken was %f \n",((float)diff/CLOCKS_PER_SEC));
4
5    //Make sure the cycles do not take less than 1.5 seconds
6    n = (int)((loop_time - ((double)diff)/CLOCKS_PER_SEC) * 1000000);
7    if (n>0) usleep(n); /*pause before next iteration*/
```

5.2 Memory allocation

Figure 3 shows that the program is built on an endless loop in which data is fetched and sent to the server. Since the data is stored in arrays it might be tempting to use dynamic memory allocation by using `malloc` or `calloc`. However, the program must run 24/7 [2.3] and therefore memory leaks are not permitted. As dynamic memory allocation is a major cause of memory leaks, it was chosen to use dynamic memory allocation only if it was absolutely necessary. Regarding the flexibility and optimal usage of memory resources, it may be an advantage to use

`malloc` or `calloc`. On the other hand, in larger programs a common mistake is to forget the use of `free()` to empty the used memory. Even more so when a variable is allocated and freed in different functions.

In the C-code it is therefore chosen not to use dynamic memory allocation. As a result, no `free()` is necessary and memory leaks are avoided. All arrays containing the station data are declared only once. To be able to use the arrays in multiple C-files, the arrays are defined `extern`. Unnecessarily passing the arrays as a function argument is prevented this way. Defining the length of the arrays is now done during compile time and therefore the length of the arrays cannot be a variable but must be a constant. This constant is chosen such that it can contain the maximum amount of registers that can be read in one request. For example, the maximum amount of registers that can be read from the Victron system using a single function call of `readVictron()` is 46. The length of the predefined array to store is registers is thus 46.

5.3 Temperature sensor polling failure

Although the ODROID can fetch data from the solar panel temperature probes through the ADAM-4015 module, it regularly fails in doing so. A failure in polling the temperature sensors happens in roughly $\frac{1}{3}$rd of the fetches. This problem is unlikely to be caused by the ADAM module, because this can handle a polling rate of 12 per second [27]. It is unclear if there is a maximum polling rate to the temperature probes, but these probes are most likely analogue and therefore not bound to a maximum polling rate.

It is therefore unclear what causes these frequent failures. The functionality is present, however, so data can be read, but it should be inspected more closely in future development if more data is desired.

6 Conclusions

6.1 General conclusion

The goal of the overall project was to design software packages and an intuitive user-interface for an e-bike charging station. Users can make a reservation in the user-interface after which they can charge their e-bike in the charging station. The main focus in the design process described in the sections above was the low-level design of data acquisition and control of the chargers using an ODROID minicomputer.

All targets with respect to the data acquisition were reached: the C-program on the ODROID is able to read all data generated by the power electronics, weather station and temperature sensors. Using the GPIO pins from the ODROID the relais can be controlled, which turn on/off the chargers. This concept was succesfully tested using LEDs since the chargers were not finished yet. Furthermore, a charger control Finite State Machine was designed and tested. This controller has buffers so the charger state does not immediately change when a cable is (dis)connected (for example, plugging the cable in badly could cause it to turn on and off immediately), as well as a timer between logging in on the server and plugging in. At this moment it is not yet possible to control the local display which will be mounted in the charging station, because there is no webpage yet to control. A conceptual design for the control mechanism is available though, including the working WebGL on the ODROID for the graphical elements of the webpage.

During tests of multiple days the C-code was succesful with respect to robustness and memory leaks. Moreover, additional Bash scripts were written to support the functionality of the code. For example, a script to solve memory leaks caused by the remote desktop was written, as well as a script to immediately start certain programs on the ODROID after a reboot (a full-screen browser and the C-code). Finally, an installation and configuration manual was made. Using this manual it is possible set up a new ODROID using some easy steps. After performing these steps the C-code should be able to run on the ODROID.

6.2 Future work and recommendations

As a low-level part of the charging station the ODROID must have solid software packages. To further improve the functionality and stability of this software some general ideas are given that can be elaborated in the future.

As an additional feature for end-users, the battery voltage of their e-bikes could be displayed on the webpage. A sensor and all the neccesary circuitry can be developed in the future for this purpose. The C-code running on the ODROID also has to be adapted to read the sensor values and transfer the data to the server.

Figure 1 shows that a local display will be mounted on the charging station. As a clearly visible device it is therefore important to have it running by the time the charging station is built. Currently, only the concept of the controlling mechanism is designed. The final implementation of the website together with the JavaScript and C-code still have to be designed and tested. Moreover, it is always possible to improve the information shown on the webpage and

the design of the webpage itself.

During robustness tests (see Section 4.7.3), it was discovered that the order in which the USB connections of the temperature probes and weather station were connected was of importance. This was due to the fact that the connection was established statically in the C-code. In future work, this could be allocated dynamically. By doing this, the connection order is no longer of importance, and the system will be more robust (for example during maintenance).

As described in Section 5.3, polling the temperature sensors is quite known to fail. The code was designed to never cause the C-code to crash when this happens, however it does mean that data points are missing about a third of the time. If a more regular set of data is desired, more research has to be done to extract the issue that causes this problem.

As an addition to the existing concept of the charging station, a LED strip could be added to the exterior of the station (behind matte glass, for example). This could be controlled by the ODROID based on the input of several proximity sensors. The LED strip should be fed by an external power supply (since these require a lot more power than the GPIO pins can deliver), probably a 230V connection of the Victron system. One should keep an eye on the number of available GPIO pins if this were to be developed.

References

[1] Multiple Authors, “Solar powered ebike charging station: Feasibility study and budget,” July 2014, Delft University of Technology.

[2] P. Purgat, “ebike charging station - communication software,” december 2015, general Description.

[3] Multiple Authors. (2013) libmodbus. [Online]. Available: http://libmodbus.org/

[4] ——. (2012) WiringPi. [Online]. Available: https://github.com/hardkernel/wiringPi

[5] ——, “EE3L11 bachelor graduation project,” 2016.

[6] ——, *Manual: Family of smart weather sensors*, september 2015, page 121. [Online]. Available: http://www.lufft.com/dateianzeige.php/?Dateiname=download/manual/WSx-UMB_V30_e.pdf

[7] R. V. Bugga and M. C. Smart, “Lithium plating behavior in lithium-ion cells,” *ECS Transactions*, vol. 25, 2010, Jet Propulsion Laboratory, California Institute of Technology. [Online]. Available: http://ecst.ecsdl.org/content/25/36/241.abstract

[8] Multiple Authors, *Manual: Family of smart weather sensors*, september 2015, page 119. [Online]. Available: http://www.lufft.com/dateianzeige.php/?Dateiname=download/manual/WSx-UMB_V30_e.pdf

[9] ——, *Manual Digital Multi Control GX Panel*, 2008. [Online]. Available: https://www.victronenergy.nl/upload/documents/Manual-Digital-Multi-Control-GX-Panel-EN-NL-FR-DE-ES.pdf

[10] “CCGX Modbus-TCP register list,” Victron Energy, excel file.

[11] *Temperature Probes Technical Datasheet*, page 33. [Online]. Available: http://www.kimo.fr/assets/docs/cat_tempe/cat_tempe_en.pdf

[12] *ADAM-4015 6-ch TRD Module with Modbus*, may 2015. [Online]. Available: http://downloadt.advantech.com/ProductFile/PIS/ADAM-4015/Product%20-%20Datasheet/ADAM-4015201507141450 37.pdf

[13] Multiple Authors. (2015) AdamApax .NET Utility. [Online]. Available: http://support.advantech.com.tw/support/DownloadSRDetail_New.aspx?SR_ID=1-2AKUDB&Doc_Source=Download

[14] ——. (2016) Block diagram and schematic. Hardkernel. [Online]. Available: http://odroid.com/dokuwiki/doku.php?id=en:c1_hardware#expansion_connectors

[15] Jimbo. Transistors. Sparkfun. [Online]. Available: https://learn.sparkfun.com/tutorials/transistors/applications-i-switches

[16] Multiple Authors. (2016) Fritzing. [Online]. Available: http://fritzing.org/home/

[17] *P2N2222A NPN General-Purpose Amplifier*, 2004. [Online]. Available: https://www.fairchildsemi.com/datasheets/PN/PN2222A.pdf

[18] (2016) Core functions. [Online]. Available: http://wiringpi.com/reference/core-functions/

[19] Multiple Authors. (2013) Technical detail. Hardkernel. [Online]. Available: http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143703355573&tab_idx=2

[20] ——. (2016) jQuery. [Online]. Available: http://jquery.com/

[21] ——. (2014) Open GLES for linux. [Online]. Available: http://odroid.com/dokuwiki/doku.php?id=en:opengleslinux

[22] benkong. (2015) How to enable chromium browser gpu hardware accelerate. [Online]. Available: http://forum.odroid.com/viewtopic.php?f=112&t=8267

[23] Multiple Authors. (2016) OpenVPN 2.3.11. [Online]. Available: https://openvpn.net/index.php/download/community-downloads.html

[24] ——. (2016) Upstart. Canonical Ltd. [Online]. Available: http://upstart.ubuntu.com/

[25] A. Beckert and I. Jackson. (2016) Unclutter. [Online]. Available: https://packages.debian.org/unstable/unclutter

[26] Multiple Authors. (2015) Valgrind. [Online]. Available: http://valgrind.org

[27] *ADAM 4000 Data Aquisition Modules User’s Manual*, sept. 2002, page 3-14. [Online]. Available: http://www.bb-elec.com/Products/Manuals/ADAM-4000.pdf

Appendix

A Setup manual

```
==== Setting up the ODROID ====

1. Update kernel and resize SDCard/eMMC through ODROID utilities

2. Update all files using the commands:
    sudo apt-get update
    sudo apt-get upgrade
    sudo apt-get dist-upgrade

3. Install some packages with the following command line:

sudo apt-get install libjpeg-dev libpng12-dev libx11-dev libglu1-mesa-dev subversion
    build-essential autoconf automake make libtool xorg-dev xutils-dev libdrm-dev
    libdri2-1 libdri2-dev git libglew-dev

4. Open Chromium and go to chrome://flags. Enable 'Override software rendering list'.

5. Go to /home/odroid/.local/share/applications and open properties of Chromium.
    Change command in Desktop Entry to 'chromium-browser --use-gl/egl' (this enables
    use of WebGL).

6. Download modbus package at libmodbus.org. Extract downloaded tarball and cd to the
    directory. Type the following commands within the map that was extracted:

    ./config
    make
    make install

libmodbus should now be installed.

7. Install WiringPi by cloning the wiringPi git. This can be done by using these
    commands in the directory you want it to be installed:

    git clone https://github.com/hardkernel/wiringpi
    cd wiringpi
    ./build

8.1. (optional) Install 'Code::Blocks IDE' by running the following command:

    sudo apt-get install codeblocks

8.2. (optional) In Code::Blocks IDE, go to Settings>Compiler... and change the
    following settings:

    Linker settings->Link libraries
        wiringPi; wiringPiDev; pthread; :libmodbus.so.5

    Search directories->Compiler
        /usr/include/; /usr/lib; /usr/local/lib;                /usr/local/include/
            modbus

    Search directories->Linker
        /usr/local/lib; /usr/lib; /usr/local/lib/pkgconfig;         /usr/local/include
            /modbus

9. Install Unclutter using the command:

    sudo apt-get install unclutter

This will be used to hide the mouse after 5 seconds of inactivity.

10.1. Install Upstart using the command:

    sudo apt-get install upstart
```

This will be used to start certain programs when booting.

10.2. Place the .conf files (browser_boot.conf, hide_mouse.conf, xvnc_killer.conf and ODROID_booter.conf) in the upstart map found in /home/odroid/.config. Make sure that hide_mouse.conf, xvnc_killer.conf and ODROID_booter.conf are looking in the correct directory (you can check by editing the file).

11.1. edit the sudoers file using the following command:

    sudo visudo

11.2. Add the default user as SuperUser by adding the following line to the end of this file:

    %odroid ALL=(ALL:ALL) ALL

This is needed to automatically boot the C-code on startup (after e.g. powercut) Note that you need to reboot for this change to become effective.

==== Setting up a VPN with the router ====

To setup a VPN connection with the router attached to the ODROID, follow these instructions (note: this should be done on the remote pc, not the ODROID!):

1. Install OpenVPN: https://openvpn.net/index.php/download/community-downloads.html

2. Login on the router and download the files found in Advanced > VPN-service.

3. Place the files in the config map of OpenVPN. By default, this can be found in C:\Program Files\OpenVPN\config.

4. Open 'client.ovpn' in notepad, and change the contents to the following:

```
client
dev tap
proto udp
dev-node NETGEAR-VPN
remote 145.94.62.199 12974
resolv-retry infinite
nobind
persist-key
persist-tun
ca ca.crt
cert client.crt
key client.key
cipher AES-128-CBC
comp-lzo
verb 5
```

NOTE: the IP address in the 5th line should correspond to the IP address of the router.

==== Setup remote desktop ====

To setup the ODROID to handle remote desktops, follow these instructions:

Step 1 - Install xRDP

Open Terminal (Crtl+Alt+T) and execute following commands :

```
sudo apt-get update
sudo apt-get install xrdp
sudo apt-get install vnc4server
```

Step 2 - Install XFCE4 (Unity doesnt seem to support xRDP in Ubuntu 14.04 although in Ubuntu 12.04 it was supported thats why we install XFCE4)

```
sudo apt-get install xfce4

Step 3 - Configure xRDP

In this step we modify 2 files to make sure xRDP uses xfce4. First we need to create
    or edit our .xsession file in our home directory. We can either use nano or simply
     redirect an echo statement (easier):

echo xfce4-session >~/.xsession

The second file we need to edit is the startup file for xRDP, so it will start xfce4.

sudo nano /etc/xrdp/startwm.sh

The content should look like this (pay attention to the last line and ignore . /etc/
    X11/Xsession):

#!/bin/sh

if [ -r /etc/default/locale ]; then
  . /etc/default/locale
  export LANG LANGUAGE
fi

startxfce4

Step 4 - Restart xRDP

To make all these changes effective, restart xRDP as such:

sudo service xrdp restart

Step 5 - Connecting to the ODROID

Use the "Remote Desktop Connection" built in Windows (or any other remote desktop
    client) and use the local IP of the ODROID when connected to the router (through
    VPN or wired). You can find this IP by typing the command 'ifconfig'. The IP
    adress will have a format looking like this: 192.168.1.xx. To login, simply use
    the login settings of the ODROID.
```

B Bash files

B.1 Startup Programs README

```
=== Xvnc_killer.sh ===
Exiting a remote desktop results in an incorrect termination of Xvnc on the ODROID
(it will remain in the memory), this causes a memory leak.
xvnc_killer.sh was developed to kill these remaining processes. It boots on startup
and ejects the oldest Xvnc client whenever there is more than 4 open (up to one every
minute). This script can be found in /home/odroid/Documents/SUNRISE-Odroid/Bash. To
    stop
this script from running on startup, remove /home/odroid/.config/upstart/xvnc_killer.
    conf

=== Unclutter ===
Unclutter is a package that is used to hide the mouse after 5 seconds of inactivity.
    This is
to ensure that the mouse does not remain on-screen on the local display of the
    charging
station. The script that runs this package is found in
/home/odroid/Documents/SUNRISE-Odroid/Bash. To undo this for the current session, end
the 'unclutter' task in the task manager. To never boot this on startup, remove
/home/odroid/.config/upstart/hide_mouse.conf.

=== Browser ===
Chromium is booted on startup to display information on the local screen of the
    charging
```

```
station. To Disable this on startup, remove /home/odroid/.config/upstart/browser_boot.
    conf.
To exit full screen use ALT+F4 or ALT-TAB.

=== C-code ===
The C-code is booted by the ODROID_booter.conf found in the same folder as the
    previous applications. It boots the C-code on startup and whenever the code has
    stopped running as a process. Remove the .conf file to stop it from booting on
    startup.
```

B.2 Xvnc_killer.sh

```
#!/bin/bash
# If there are more than 4 remote desktops open, eject the oldest

while true; do #Continue indefinitely

    if [ $(ps aux | grep 'ssh-agent' | grep -v '<defunct>' -c) -gt 4 ]; then #if more
        than 3 ssh-agents are active
        kill $(ps aux | grep 'ssh-agent' | grep -v '<defunct>' | sort -n -k 9 | awk '{
            print $2}' | head -1) #kill oldest ssh-agent process
    fi

    if [ $(ps aux | grep 'Xvnc' -c) -gt 4 ]; then #if more than 3 Xvnc processes are
        active
        kill $(ps aux | grep 'Xvnc' | sort -n -k 9 | awk '{print $2}' | head -1) #kill
            oldest Xvnc process
    fi

    sleep 60 #check every minute

done
```

B.3 Xvnc_killer.conf

```
start on startup
task
exec /home/odroid/Documents/SUNRISE-Odroid/Bash/xvnc_killer.sh
```

B.4 hide_mouse.sh

```
#!/bin/bash
#Hide mouse after 5 seconds of inactivity (if not opened already)

if [ $(ps aux | grep 'unclutter' | grep -v '<defunct>' -c) -le 1 ]; then #if no
    unclutter is active active
    unclutter -display :0.0 -idle 5
fi
```

B.5 hide_mouse.conf

```
start on startup
task
exec /home/odroid/Documents/SUNRISE-Odroid/Bash/hide_mouse.sh
```

B.6 browser_boot.conf

```
start on desktop-start
stop on desktop-end

exec chromium-browser --use-gl=egl --kiosk http://solarpoweredbikes.tudelft.nl/eud/
    app_status
```

B.7 ODROID_booter.sh

```
#!/bin/bash

while true; do #Continue indefinitely

    if [ $(ps aux | grep 'odroid_server' | grep -v '<defunct>' -c) -le 5 ]; then #if less than 4 odroid_servers are active (2 default, 2 if running)
        xterm -iconic -e su -c "xterm -iconic -hold /home/odroid/Documents/SUNRISE-Odroid/_odroid_server/bin/Debug/odroid_server"
    fi

    sleep 60 #check every minute

done
```

B.8 ODROID_booter.conf

```
start on startup
task
exec /home/odroid/Documents/SUNRISE-Odroid/Bash/ODROID_booter.sh
```