NTNU

Norwegian University of
Science and Technology

Data acquisition and analysis ofacquired data from geographically distributed sensors connected by 2G / 4G technology

Håkon Edøy Hanssen

Master of Science in Cybernetics and Robotics
Submission date: June 2018
Supervisor: Geir Mathisen, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

NTNU – Trondheim
Norwegian University of
Science and Technology

Master
Thesis

Data acquisition and analysis of acquired data from geographically distributed sensors connected by 2G / 4G technology

Author:
Håkon Edøy Hanssen
Supervisor:
Geir Mathisen

June 22, 2018

NTNU
Norges teknisk-naturvitenskapelige universitet

Faculty of Information Technology and Electrical Engineering
Department of Cybernetics

Master Thesis

Candidate: **Håkon Edøy Hanssen**
Subcject: **TTK4990**

Oppgavetittel (Norsk): **Innsamling og prosessering av data samlet fra geografisk spredte sensorer ved bruk av 2G/4G**

Thesis title (English): **Data acquisition and analysis of acquired data from geographically distributed sensors connected by 2G / 4G technology**

Thesis description:

The "smart"- solutions (smart grid, smart cities, smart water etc) require systems for massive acquisition of data from distributed sources, to be able to implement optimal control. Preferably, this acquisition should be cheap and take as little effort as possible to realize. As a contribution to this, we want to develop a (near) real-time IoT-inspired data acquisition system using 2G / 4G connectivity technology. The system should transfer the acquired data (e.g. PV and windmill energy production, temperature, humidity) to a cloud for further analysis. It should be possible to remotely adjusting sampling / transferring parameters. As an example, the analysis should consist of estimating next-day energy production from PV solar panels, based on historical data and weather forecasts.

.

The tasks will be:

- Conduct a literary study of systems used for acquisition of distributed data and methods for forecasting power production from PV solar panels.
- Propose a system for acquisition of data from distributed sources into a centralized data server, using 2G / 4G connection technology. The proposal shall include an architecture enabling continuously analyzing of the received data. To concretise the work, the proposal shall take as an example data acquisition from PV solar panels and forecasting next day power production.
- As far as time permits, implement the suggested data acquisition and analyzing system.

Thesis given: 11. January 2018
Thesis deadline: 02. July 2018

Faculty of Information Technology and Electrical Engineering IE
Department of Cybernetics ITK

Abstract

The usage of the 2G/4G network geographically in distributed embedded systems has multiple advantages. One is the coverage of the network, which in most advanced countries covers the vast majority of the land area. Another is that the maintenance of the network is performed by the service provider, reducing the work necessary to maintain and implement the service. Despite these advantages, the usage of embedded systems, and development kits with cellular connectivity is relatively limited. The motivation of this master thesis has therefore been to do some of the early work within this field of embedded systems, by doing a study of existing embedded systems with similar functionalities, a review of the factors affecting the accuracy of predicting solar PV power production, and trying to implement an embedded system using the 2G/4G network to communicate with a cloud service.

This thesis will present the different kinds of hardware and software used to implement the system. It will also present the system specifications, how it was implemented, the result of the implementation and its limitations.

Due to timing constraints, lack of libraries and some bugs experienced when using the Digi API, the author of this thesis had to use a ESP8266 Wi-Fi module, instead of the intended Digi 3G Global cellular modem. The embedded system runs FreeRTOS to make it (near) real-time. It communicates with Amazon Web Services (AWS), the chosen cloud service, by using the light-weight MQTT protocol. A Python script running on a virtual Ubuntu machine retrieves the data transmitted by the embedded system and places it in a DynamoDB table. Scripts for scraping weather forecasts from Yr, and for generating plots run once every 24 hours. All scripts on the Ubuntu machine are run in the background and produces log files for remote debugging.

The complete system is still in its early stages. The 2G/4G has yet to be successfully implemented, and the current cloud service requires, among other things, improved security and scalability.

Sammendrag

Det er flere fordeler ved å bruke 2G/4G nettverket sammen med distribuerte embedded systemer. Et ar at dekningsgraden til nettverket, hvilket ofte er nesten hele landmassen til de aller fleste land med høy levestandard. En annen er at alt vedlikehold av nettverket gjennomføres av telefonselskapene, noe som reduserer mengden nødvendig vedlikehold. På tross av disse fordelene er per i dag 2G/4G-nettet lite brukt i embedded systemer og sett for utviklere. Hovedmotivasjonen til denne masteroppgaven er å utføre noe av det tidlige arbeidet relatert til dette feltet innen embedded maskiner. Dette gjøres gjennom en litteraturstudie av eksisterende embedded systemer med liknende funksjonalitet og av hvilke faktorer som påvirker hvor nøyaktig estimeringen av strømproduksjonen til PV solceller. Tilslutt vil forfatteren prøve å implementere et embedded system som kommuniserer med en skytjeneste gjennom å bruke 2G/4G nettet.

Denne oppgaven vil vise maskin- og programvaren som ble benyttet for å implementere systemet. Den vil også presentere systemspesifikajsonene, hvordan det ble implementert og systemets nåværende begrensinger.

Som følge av tidsbegrensinger, mangel på bibliotek og noen programfeil når Digis API ble brukt, måtte forfatteren av denne oppgaven bruke en ESP8266 Wi-Fi modul, istedenfor det tiltenkte Digi 3G Global modemet. Embedded systemet kjører FreeRTOS for at det skal være et (tilnærmet) sanntidssystem. Den kommuniserer med Amazon Web Services (AWS) gjennom å benytte seg av MQTT protokollen. Et Python skript som kjører på en virtuell Ubuntu-maskin fanger opp meldingene sendt av systemet og plasserer dem i et DynamoDB tabell. Andre Python program benyttes for å skrape værmeldingsdata fra Yr og for å genrere grafer kjøres en gang i døgnet. All programmene som kjøres på Ubuntu-maskinen kjøres i bakgrunnen og lager loggfiler brukt til feilsøking.

Det komplette systemet er fortsatt på et tidlig stadie. 2G/4G-kommunikasjonen har enda ikke blitt implementert skikkelig, og det nåværende oppsettet i skytjenesten trenger blant annet bedre sikkerhet og skalerbarhet.

Preface

This master thesis project was written at the Department of Cybernetics(ITK), a subsidiary of the Faculty of Information Technology and Electrical Engineering(IE) at the Norwegian University of Science and Technology(NTNU). The purchase of the necessary hardware and software needed for this project has been conducted by financial means provided by ITK. This thesis builds on the work performed by the author of this thesis during the completion of the mandatory TTK4550 Specialization Project.

This document is intended to an independent documentation done to complete this master thesis assignment. This is accomplished by describing all the different modules, components and methods used through this project. They are all made available through proper citations and an extensive bibliography throughout this thesis report.

Prerequisites

This thesis project requires multiple software and hardware components. This mainly includes embedded hardware modules and associated software to implement an IoT-inspired data acquisition device, and an Amazon Web Services (AWS) account to utilize the various services provided by AWS.

List of necessary equipment and software

- LCPXpresso 4367 development kit
- ESP8266MOD NodeMCU
- Digi XBee cellular 3G modem
- An AWS Account
- MCUXpresso IDE (software)
- Python (version 3 or newer)
- Pip (for installing Python libraries)
- NXP LPC-Link2 JTAG debugger
- HO-P Hall Effect Sensor
- A lab bench power supply

Acknowledgements

This thesis project has been greatly helped by the work on cloud service technology done by Marit Tundal[80] and the collaboration with Erlend Grande M.Sc. His help with setting up and using FreeRTOS was of great value. I am also grateful for the help my girlfriend, Ingeborg Herjuaune, has done in removing grammatical errors that were present in this thesis report. I also wish to extend my gratitude to my supervisor, Geir Mathisen, for the initial thesis idea and his helpful advice during this project.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ADC Analog-digital Converter

ANN Artificial neural network

API Application Programming Interface

APN Access Point Name

AT Attention

AWS Amazon Web Services

CISC Complex instruction set computer

CLI Command Line Interface

CMOS Complementary metal–oxide–semiconductor

CP/SS Chip/Slave Select

CPU Central Processing Unit

DC Direct Current

EC2 Elastic Compute Cloud

GPIO General Purpose Input Output

GPR General purpose registers

GPRS General Packet Radio Service

GSM or 2G Global System for Mobile Communications

HTTP Hypertext Transfer Protocol

I2C Inter-Integrated Circuit

IP Internet Protocol

ISR Interrupt Service Routine

ISA Instruction Set Architecture

MISO Master In Slave Out

MOSI Master Out Slave In

MQTT Message Queuing Telemetry Transport

PKCS Public Key Cryptography Standards

PV Photovoltaic

RISC Reduced instruction set computer

RS Recommended Standard

RTC Real-Time Clock

RTOS Real-time Operating System

Rx Serial Receiver

S3 Simple Storage Service

SCL Serial Clock

SDA Serial Data

SDK Software Development Kit

SIM Subscriber Identity Module

SPI Serial Peripheral Interface

SoC System on Chip

TCP Transmission Control Protocol

TCP/IP Internet Protocol Suite

Tx Serial Transmitter

UART Universal Asynchronous Receiver-Transmitter

URL Uniform Resource Locator

Chapter 1

Introduction

With the falling cost of computing the usage of embedded systems in industry, business and private homes is increasing. These embedded systems are often connected to each other though the internet or by some other protocol. The hardware and software of these embedded systems are designed to perform a limited set of tasks efficiently. As opposed to a desktop or laptop personal computer, who are able to perform a wide set of tasks.

Increased integration between embedded systems are also one of the main features, which includes sharing data, collaboration in solving tasks, etc. This enables the implementation of practical features, such as remote environmental surveillance. By placing one or more embedded devices with the means of communicating with each other or a third party by using the 2G/4G network in a remote location, critical infrastructure or vulnerable parts of nature can be under remote surveillance. This reduces the labor cost of maintaining what the system monitors, gives more up-to-date information than inspections would and allows for quicker error responses.

With the urgent need to change the worlds energy infrastructure, both with respect to the exhaustion of natural resources and the devastating effects of climate change, from an overwhelming focus on fossil fuels, to a more sustainable and efficient one. Powered mainly by renewable power sources such as solar, wind and hydro. To help accomplish this task, distributed embedded system could be vital to ensure a continues surveillance of the new infrastructure. Energy sources such as solar and wind are not able to produce a set amount of power like fossil fuels are able to, but it is indeed possible to estimate what they *might* produce, with some uncertainty. For this kind of analysis and estimation, data collection is vital.

To ensure that the data collected for these ends are retrieved and handled in a timely manner a (near) real-time embedded system should be used. This allows for data to be collected and sent in appropriate time intervals and allows for a more robust implementation. Using a real-time system would also allow the system to handle more critical tasks with greater reliability and, in the case of partial or complete system failure, try to maintain the normal operation as long as possible.

Such distributed real-time embedded systems should be connected to a cloud service infrastructure capable of handling such distributed systems in an efficient and secure manner. It should also give access to resources for using the collected in a further analysis.

1.1 Intended Audience

The intended audience for this document is students and professionals within electrical, cybernetic and related engineering fields. The language and content of this document will reflect this choice. Individuals who do not have the mentioned qualifications will therefore have problems with some of the expressions and concepts in this document.

1.2 The Assignment

The main goal of this assignment was to implement an embedded system who communicate with a cloud service using the 2G/4G network, as far as time permits. The system should be able to have one or more sensors connected to it, for monitoring the power production of solar PV panels. The implemented system should be real-time, to improve the responsiveness and reliability of the system, since it might be placed in a remote location handling critical tasks. The embedded system should be implemented in a modular fashion, allowing parts of it to be exchanged requiring little modification to the remaining part of the system.

Software used to handle the collected information in the cloud service should also be implemented. This includes placing the collected data in a database, collecting information from the internet to be used for further analysis and for analyzing the collected data. The software should also be implemented in a modular fashion, to make it easy to add additional locations or pieces of information to the system.

1.3 Thesis Structure

The thesis is structured as follows:

Chapter 2 Theory presents the theoretical background of this thesis. It includes a presentation of the different technologies used in this project and how they work.

Chapter 3 Previous Work presents contains a presentation of two existing embedded systems with a similar functionality to the implemented system. It also includes a literary study of how and to what degree different factors influences the accuracy of predicting the power production of solar PVs.

Chapter 4 Specifications and Design presents the functionality and specifications of the embedded system. It also describe how the hardware used in the implementation was chosen and what kind of cloud computing platform was chosen hanlding the data collected by the implemented embedded system.

Chapter 5 Implementation resents how the embedded system and cloud computing software was implemented. It also describe the design decisions that were were made and why.

Chapter 6 Results presents the results of the different tests, the measured power consumption of the system and the stats generated by the different AWS services.

Chapter 7 Discussion presents an analysis of the results and discusses them. It will also present suggestions to improving the implemented system based on said discussion.

Chapter 8 Conclusion presents the conclusion drawn from the discussion in the previous chapter.

Chapter 9 Future Work presents a list of the possible system improvements discussed in chapter 7.

Chapter 2

Theory

This chapter of the thesis will contain the theoretical background on which this project builds upon. This chapter will assume that the reader has a basic comprehension of information technology, electronics and microcontrollers.

2.1 General Theory

2.1.1 Linear Regression

Linear regression is a linear approach within statistics for modelling the relationship between a scalar variable Y between one or more dependant (or independent) variables X[24]. Linear prediction functions are modeled using a set of data. These prediction functions try to estimate the linear relationship between one or more chosen input variables and the chosen output. The practical use of linear regression includes, among other things, estimation and error reduction.

2.1.2 Solar Photovoltaic - PV

Photovoltaic solar cells, or Solar PV, are a type of solar panel that converts light to electricity[26]. The advantages of using solar PVs include no pollution after installation, simple scalability and an abundance of silicone, the main component of the cell. Disadvantages to solar PVs mainly come from their need for direct sunlight. Dust, clouds and indirect sunlight are among the factors that may negatively affect the amount of electricity converted from light by the solar PVs.

2.1.3 Measuring Current and Voltage

Measuring both current and voltage is accomplished by using the well-known equation derived from Ohm's Law. The law states that the current through a conductor is directly proportional to the voltage across two points. The law can be expressed using equation 6.1. Where I represents the current through the wire, U the voltage across the two points and R the resistance of the conductor.

$$I = \frac{U}{R} \tag{2.1}$$

Using Ohm's Law, one can measure the current running through a wire by placing a small resistor on the wire and measure the voltage loss across the resistor.

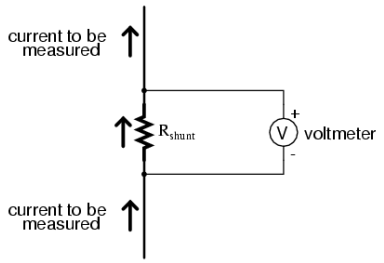


Figure 2.1: A simple graphical representation of measuring the current using a resistor.

An alternative, less invasive method of measuring the current through a wire is by measuring the magnetic field produced by the current. This is accomplished by using a hall effect sensor[85]. The magnetic field is measured by having the power cable run through a ring formed sensor. The sensor often produces an output voltage, linearly dependant on the current. The output voltage can be measured like in figure 2.2, by measuring the difference between the voltage and ground.

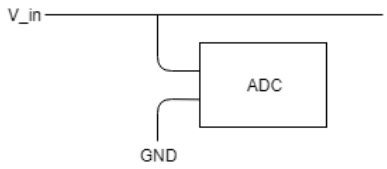


Figure 2.2: A simple graphical representation of measuring the voltage using an ADC to translate the difference in voltage between the two inputs to find V_{in}.

2.2 Communications Standards

2.2.1 Universal asynchronous receiver-transmitter - UART

A universal asynchronous receiver-transmitter is a hardware based device for transmitting digital information in serial and asynchronous manner[19, p. 180]. It is a robust, moderate-speed and relatively simple way of communicating between ICs. Due to being asynchronous the UART does not transmit a clock signal between the sender and receiver. The clocks of the separate micro-controllers are instead synced within reasonable margins to allow them to process the information they receive.

Figure 2.3 shows how two microcontroller units are connected using UART. The serial transmitter (Tx) is connected to the serial receiver (Rx). Both are connected to the same ground, to give them the same reference point. Figure 2.4 shows how the data is transmitted using the lines between the UART Tx and Rx. The start bit is the first bit in the transmission, signaling the end of the idle state. The stop bit is the last bit in the transmission, signaling a return to the idle state. Bit 7 or 8 is often used as a parity bit, a somewhat crude error-detection mechanism.

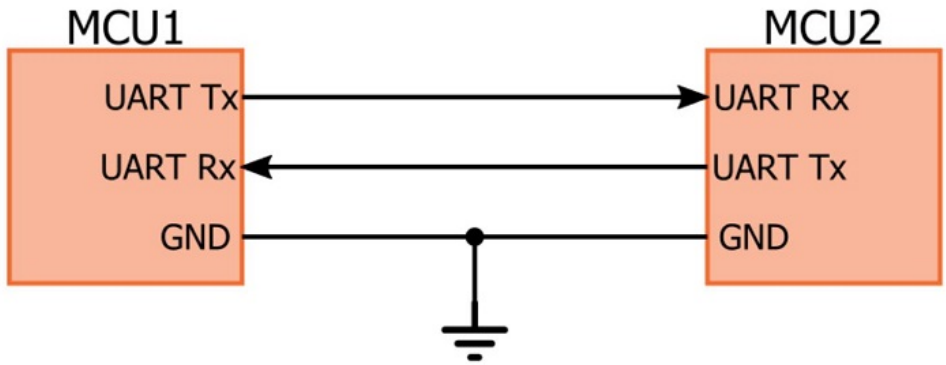


Figure 2.3: A simple graphical representation of two microcontrollers connected using UART.[1]

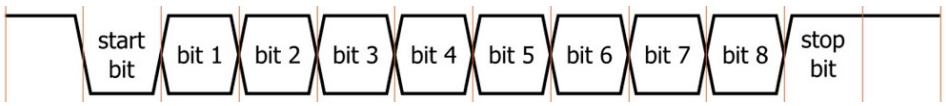


Figure 2.4: A graphical representation of the message format of UART[2]

[1] Picture originates from All About Circuits[1]

The UART operates within a certain voltage range, $0V - V_{CC}$, which is often $3.3V$ or $5V$. The bit value 0 is characterized by a zero voltage over the wire, while a bit value of 1 is characterized by a voltage of V_{CC}. The number of bits that can be transmitted over the UART connection is determined by the baud rate. A baud rate of 9600 means that 9600 bits can be transmitted over the connection in a second. This gives every bit a period of $104.2\mu s$.

2.2.2 Universal Serial Bus - USB

The Universal Serial Bus was developed to standardize the communication between computers and connected devices [19, p. 203]. The standard defines the cables, communication protocols and connectors used in the USB. The standard voltage of the USB is $5VDC$, with the USB 2.0 standard having a maximum current of $500mA$. The maximum number of devices connected to one USB network is defined as 127, with a current maximum data rate of $480Mbps$ for the 2.0 standard [19, p. 204].

The most basic structure of the USB network is a tiered star. With one *host* computer controlling the network. *Peripherals*, such as keyboard, data mouse or printers are connected to the host. *Hubs* can be used to extend the network and number of available USB ports [19, p.204-205]. The USB cable consist of four pins:

1. The USB device power pin, $V_{BUS}(+5VDC)$
2. Differential data line, $D+$
3. Differential data line, $D-$
4. Power and signal ground, GND

The USB standard defines four types of transfers over the bus[19, p. 206]. The *control transfer*, which configures the devices and returns status information. The *bulk tranfer* is used for asynchronous movement of data. The *isochronous transfer* is used to move time-critical information, such as audio or video data. The *interrupt transfer* is used for regular interval moving of data. Data is transferred using packets, with the packets consisting of a synchronization ($SYNC$) byte, a Packet ID (PID), the content and a cyclic-redundancy check (CRC).

2.2.3 Global System for Mobile Communications - GSM/2G

Global System for Mobile Communications (GSM or 2G) is a communication standard developed by the European Telecommunications Standards Institute to define the protocols for digital cellular networks used by mobile devices. It was mainly developed as a response to the over nine competing analog standards within

[2]Picture originates from All About Circuits[1]

Europe[74, p. 11]. The GSM standard was released for the first time in Finland in 1991 and had in 2014 a market share of over 90% [17].

Since GSM is a cellular network, the mobile equipment connects to the rest of the network wirelessly by connecting to the nearest cell generated by one of many Base Transceiver Stations. The frequencies of which the wireless signals operate are mostly within $900MHz$ and $1800MHz$. From the base station the signal is then sent through the Base Station Controller to the more fixed part of the network. In order to separate the different mobile nodes in the network a Subscriber Identity Module, or SIM, is used to identify an individual mobile equipment.

The Network Switching System is the part of the network which carries out the switching between the different mobile units within the cellular network[35]. The General Packet Radio Service (GPRS) core network facilitate the transmissions of Internet Protocol (IP) packets to and from mobile devices within the network[36].

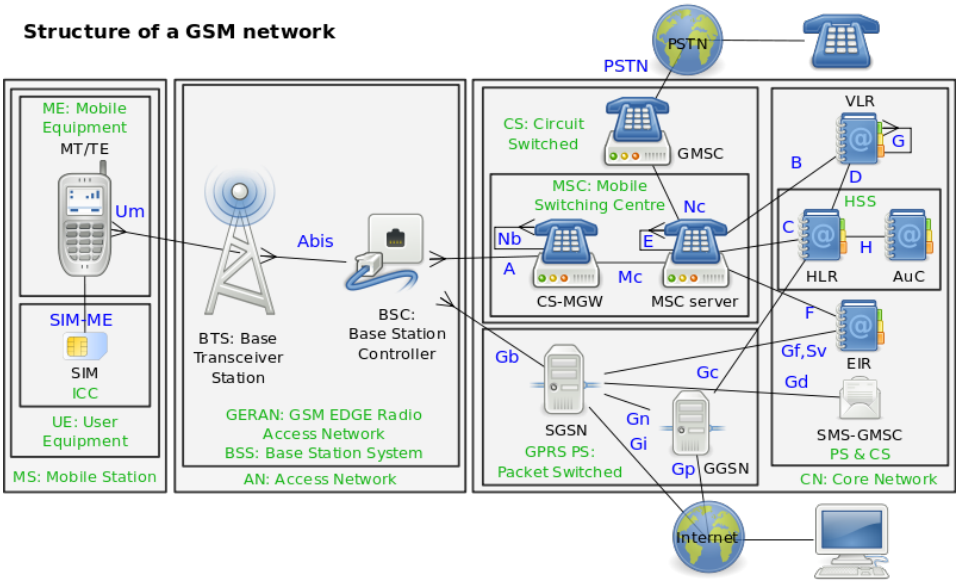


Figure 2.5: A graphical representation of the structure of a GSM network[3]

2.2.4 3G

The first commercial 3G cellular network was launched in October 2001 in Japan[21]. 3G uses the same network structure as GSM, but operates within the frequencies

[3]The picture originates from Wikipedia[82]

of $400MHz$ and $3GHz$[54].

The main difference between a 3G and a GSM network is the upload and download speeds the different networks provide users. GSM has a maximum uploading and downloading speed of $256Kbps$, 3G has a maximum upload and download speed of $5.7Mbit/s$ and $21Mbit/s$ respectively. These numbers vary somewhat depending on the version of 3G that is used. 3G also allows for greater network security by using the Kasumi block cipher, instead of the A5/1 stream cipher used in GSM networks. 3G also allows for a number of services that GSM does not. This includes GPS, streaming and sending video.

2.2.5 Long Term Evolution - 4G LTE/4G

The first commercial deployment of Long Term Evolution was in 2009 in Oslo, Norway[22]. Similarly to 3G, 4G uses a similar physical infrastructure as the GSM network. The main difference visible to users is the increase in speed which 4G is capable of. The original LTE has an upload and download speed of $50Mbit/s$ and $100Mbit/s$, respectively. Additionally, 4G, unlike 3G and GSM, does not support transfer of voice calls, only data transfers[23]. 4G also, unlike 3G and GSM, only uses packet switching when transferring data across the network.

Long Term Evolution for Machines - LTE-M

LTE-M is developed for IoT devices. The standard provides the same coverage as the orginal LTE, but with a smaller bandwidth, around $1Mbps$[86]. This is done to reduce both the cost and the energy usage of the devices using this technology. The LTE-M standard is intended for smart meters, and similar devices, that does not need send large amounts of data. Devices using this standard should mainly transmit the information in the form of simple UDP or TCP messages[55], communication standards such as MQTT might be too much to handle for the system.

Long Term Evolution for Internet of Things - LTE-IoT

LTE-IoT is similar to the LTE-M standard. The coverage is the same, however, the bandwidth of LTE-IoT is somewhat smaller compared to LTE-M, $250Kbps$[86]. The latency is also a bit larger, $1.6-10s$, as opposed to LTE-Ms $10-15ms$. Though neither LTE-M nor LTE-IoT is commercially available yet, LTE-IoT is thought of possibly being cheaper[48], as it does not require a gateway to function, reducing the required hardware.

2.3 Communication Protocols

2.3.1 Internet Protocol Suite

The Internet protocol suite, often refereed to as TCP/IP because the TCP (Transmission Control Protocol) and IP, are the foundational protocols of the suite[68]. The protocol suite specifies the end-to-end communication, including how data is packetized, addressed, transmitted, routed and received on the Internet. The suite consists of four layers, that together make the communication over the internet possible. The four layers are, from lowest to highest:

- **The link layer**, which contains the different communication protocols that operate on the links that the different hosts are connected to.
- **The internet layer**, which contains the different methods and protocols which transports packets of data between the different hosts.
- **The transport layer**, which is a set of methods that facilitate the host-to-host communication over the internet.
- **The application layer**, is an abstraction layer which specifies the methods and protocols used by the hosts of the network. For example servers and clients.

Transmission Control Protocol

TCP is one of the main protocols of the internet. It is a connection-oriented protocol, meaning that a connection between hosts is established and maintained until the exchange is completed on both sides [73]. TCP maintains order in the packets and error-check their transmission. TCP is a part of the transport layer of the internet protocol suite[73].

Internet Protocol

The Internet Protocol is the main communications protocol of the Internet Protocol Suite. The protocol is responsible for giving all the hosts the different addresses, encapsulating the data transmitted over the network into datagrams and routing them between hosts[68]. Unique IP addresses are used to route the datagrams to their correct location. There are currently two main versions of the IP address, IPv4 and IPv6. IPv4 is the oldest version and consists of 32 bits. It is mostly represented in a decimal fashion, for example 172.16.254.1. IPv6, on the other hand, consists of 128 bits and is represented in a hexidecimal fashion, for example $2607 : F8B0 : 4005 : 804 : 200E$.

Port

Within the Internet Protocol Suite is the end point of a communication between hosts[90]. A port is always associated with an IP address. Meaning that a given port, for example number 80 will always be associated with its IP address 1.2.3.4. The port is therefore often represented in the form of $1.2.3.4 : 80$. On most computers there are thousands of enumerated ports, with the 1024 most well known ports having specific applications associated with them. For example port 8080 is used for URLs (Uniform Resource Locator) on servers hosting web sites.

2.3.2 Message Queuing Telemetry Transport - MQTT

Message Queuing Telemetry Transport (MQTT) is a publish-subscribe-based messaging ISO standard protocol[25]. The protocol is designed to be used in remote locations with a small code footprint and works on top of the TCP/IP protocol. The protocol waits for a connection to be established with the server, and disconnects after the MQTT client has finished its necessary work and for the TCP/IP session to end.

2.4 Hardware Platform and modules

2.4.1 Cortex - M4

The Cortex-M4 is a 32-bit ARM-based microcontroller with $1 Mbyte$ of Flash memory, upto $192 + 4 Kbytes$ of SRAM, with up to 17 timers (twelve 16-bit and two 32-bit timers) with a maximum frequency of $168 MHz$. The controller has 140 pins with interrupt capability and 15 communication interfaces, including:

- 3x I2C
- 4x USART / 2x UART
- 3x SPI
- 2x CAN interfaces
- SDIO interface

The Cortex-M4 controller can be debugged using a serial wire interface (SWD) or JTAG.

2.4.2 Digi XBee

Digi XBee is a series of radio modules produced by Digi International[83]. These radio modules operate within the IEEE 802.15.4-2003 for point-to-point communication. The series includes both cellular modules, using both 3G and LTE, and the ZigBee standard. Fort shorter range communication.

The Digi XBee Hardware

To increase modularity all the Digi XBee modules have the same 20 pin mapping, see figure 2.6.

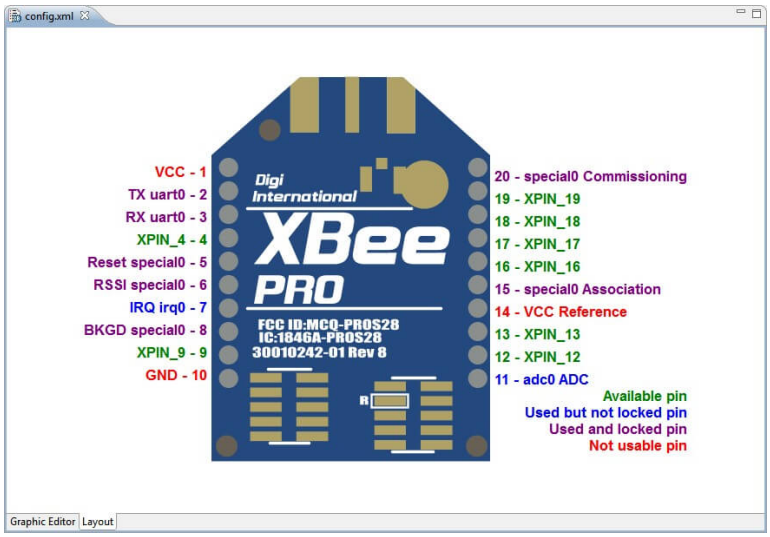


Figure 2.6: The XBee pin layout

This standardization allows for upgrading existing or deployed systems with little to no hardware modification and can extend product life cycles, according its producer[52].

Digi API

The Digi XBee modules can communicate using a serial protocol by utilizing the Digi API (application programming interface). This allows for greater software modularity, as a driver utilizing the API can be utilized with multiple modules with little to no modification. The Digi API consist of frames that are sent back and forth over a serial interface. The API frames have the format shown in table 2.1[30, p. 111].

Frame Fields	Byte	Description
Start delimiter	1	Always 0x7E
Length	2-3	Most Significant Byte, Least Significant Byte
Frame data	4 - number(n)	API-specific structure
Checksum	n + 1	1 byte

Table 2.1: The Digi API frame format

The API checksum is found by adding all the bytes in the frame data, then, subtracting the two least significant bytes from $0xFF$. To confirm that the checksum is valid, the two least significant bytes of the summation of all bytes in the frame data and the checksum should be $0xFF$.

The type of API frame is defined by the content of the frame data. The format of the frame data is shown in table 2.2 [28, p. 112].

Frame Fields	Byte	Description
API frame type	4	Defines what type of message is sent.
Data	5 - n	The data being sent.

Table 2.2: The Digi API frame format

The available frames that can be used by the different modules vary, depending of the module hardware. ZigBee modules cannot, for example, process cellular API frames.

2.5 Software Platforms

2.5.1 OS - Operating System

An operation system is a form of software that controls different forms of application programs and facilitate the interaction between these applications and the hardware[77, p. 77]. The OS does this by handling the scheduling of loading data in to the main memory for application execution and initializing files and I/O-devices. It also handles software and hardware error that occur during the execution of an application. This includes trying to correct the error or simply abort the running of the program, and report it to the user.

2.5.2 Real-time Operation System - RTOS

A real-time operating system is characterized by fulfilling five general areas of requirements[77, p. 477].

- Determinism
- Responsiveness
- User Control
- Reliability
- Fail-soft operation

An OS is deterministic when it is able to perform operations within a predefined time frame or intervals. If multiple processes competes with each other for resources and processor time, no system will be able to be fully deterministic. In RTOSs external events, such as interrupts, and timings decide which processes receive priority for processing time and other resources. Related to determinism is responsiveness, which is concerned as to how long, after an acknowledgement, an OS require to service an interrupt. This includes how long it takes for the system to handle and perform the interrupt service routine (ISR) and the effect of interrupt nesting. That is, if an ISR can be interrupted by another ISR.

In RTOSs the available user control is normally broader than in ordinary OSs[77, p. 478]. This includes allowing the user to define the priority of the different tasks the system performs, such as if the system is to use paging, disk transfer algorithms and so on.

Reliability is one of the most crucial features of RTOSs. If the function fulfilled by the real-time system is critical to, for example life support, degradation or loss of the system may have fatal results. Therefore, real-time systems need to be able to respond to various failures. This characteristic is called fail-soft operation, which defines the ability of a system to fail while preserving as much data and functionality as possible. A real-time system will therefore try to correct or minimize the effects of an error it detects, while it continues to run. One of the most important aspects of fail-soft operation is the stability of the system, which means that the real-time system is able to meet its most critical deadlines, even if it is unable to meet them all.

2.5.3 FreeRTOS

FreeRTOS is a widely used RTOS for embedded devices[40]. The FreeRTOS is licensed under the MIT license, which means that it can be used with few restrictions with regards to modifications and selling of products containing the software.

Due to the open nature of FreeRTOS a large amount of documentation is available, which eases the development of embedded systems. It also has a large development community, making it easier to find solutions to problems that might occur during implementation. The FreeRTOS kernel is also a very simple piece of software, containing only 3 C files[40]. This limits the size of the OS to around 6KB to 12KB.

FreeRTOS is capable of running in a tickless mode to reduce the power consumption of embedded devices[40]. This mode modifies the time interval of time ticks, to avoid unnecessarily interrupting tasks with time ticks. Doing so reduces the power consumption of the system, and is often used in mobile devices to increase battery life. This comes at a cost of a more complex implementation and may increase the time necessary for the CPU to switch to and from an idle mode[79]. It is also known for slowing user and kernel transactions somewhat. The tickless mode is therefore often avoided in systems with strict real-time requirements.

Tasks

FreeRTOS is a RTOS which is organized around independent tasks, with no dependencies between each other or the scheduler itself[44]. Each of the tasks are provided with their own stack, with the stack the execution context is saved to it, so that the stacks maintain their state when swapped. Each task is given a priority, represented by a number[43], where a low number represents a low priority. The FreeRTOS scheduler will always choose to run the ready task with the highest priority.

Semaphores

To prevent tasks from accessing the same resource at the same time, and possibly corrupting it, one can use binary semaphores[41]. When accessing a global resource, for example a buffer, the task needs to grab a semaphore in order to access it. FreeRTOS provides a function for this purpose, with the ability for the task to enter a blocked state if the semaphore is taken, allowing the task currently using the resource to finish, before grabbing it.

2.5.4 Amazon Web Services - AWS

Amazon Web Services (AWS) is a subsidiary of Amazon, providing on demand cloud computing platforms to individuals, companies and governments[75]. The AWS service is located on multiple server farms across the planet. The individual user pays a fee based on the actual use of the service. AWS provides a 12 month free trial period, with limited access to the different technologies available on the platform. AWS acquired FreeRTOS in 2017, and therefore supplies a number of

application programming interfaces (API) to the FreeRTOS kernel to ease the workload of connecting IoT devices to the cloud[76].

AWS DynamoDB

Amazon DynamoDB is a nonrelational database supplied by AWS[4, p. 1]. It provides storage methods for both document, such as JSON, and key-value items[4, p. 12]. DynamoDB is, according to Amazon, intended to be used by web scale applications, such as gaming, media sharing and IoT due to its low latency and high scalability[4, p. 21].

AWS IoT

AWS IoT is a service provided by AWS for collecting data from devices connected to the internet[12]. Users have the option of creating applications for monitoring sensors or other types of embedded systems. AWS IoT facilitates two-way communication between the devices and the cloud in a secure manner with low latency. Embedded systems are connected to AWS IoT by utilizing the MQTT protocol[11], either directly, or by using the Web socket protocol.

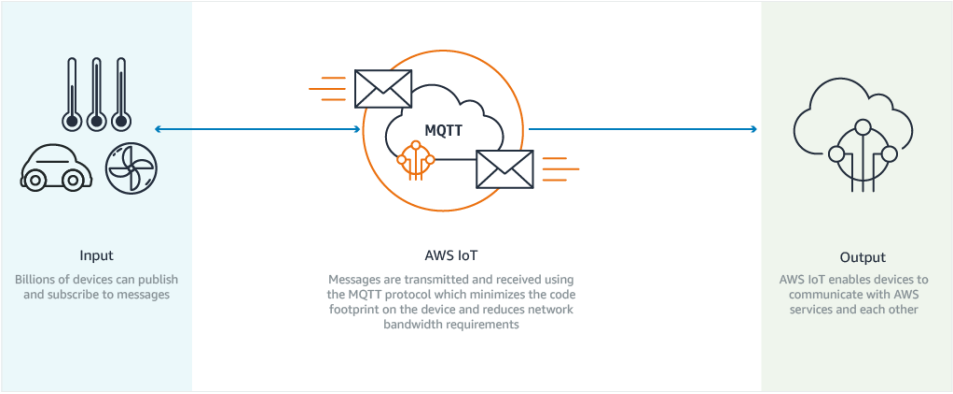


Figure 2.7: A graphical representation of the AWS IoT structure[4].

AWS IoT allows users to create certificates, to identify and authenticate the different devices connected to the service[11, p. 2]. It also allows user to define rules, which are macros that can trigger an action based on a message. The action can be defined by the device that transmitted the message and the message content, among other factors[11, p. 2]. AWS IoT also allows for remotely accessing and controlling connected devices. This is accomplished by using the Device Shadow service[11, p. 236]. It allows the user to define a JSON document, which contain chosen attributes of a given device. The Device Shadow can be used to monitor

the state of a given device, or to remotely change some attributes of the embedded system, when it is connected to AWS IoT.

AWS Elastic Beanstalk

AWS Elastic Beanstalk is a service for quickly deploying and managing applications in the AWS cloud[16]. It allows for easy and non-restrictive scaling of the storage and database services used, and one only pays for the usage of the underlying AWS services.

Amazon FreeRTOS

Amazon FreeRTOS is an extension of the FreeRTOS kernel discussed in section 2.5.3[6, p. 1]. This includes libraries for securely connection to AWS IoT cloud, connecting to AWS Greengrass cores[5]and managing Wi-Fi connections. There are some available development kits that can be used to quickly deploy Amazon FreeRTOS[6, p. 4].

In order to utilize Amazon FreeRTOS with embedded systems not included in this list it is necessary to port it. This is accomplished by implementing software for handling logging, connectivity and security[6, p. 63]. For connectivity this includes implementing TCP/IP communication in compliance with the Secure Sockets standard[6, p. 64]. Libraries for the cryptographic protocol, TLS, and the 11th version of the Public Key Cryptography Standards (PKCS#11) has also to be implemented to port the security functions[6, p. 66].

Amazon Elastic Compute Cloud - Amazon EC2

Amazon EC2 is a service providing cloud computing capacity[5, p. 1]. The capacity is available in the form of *instances*, which are virtual computing environments, or machines. The CPU, memory, storage and network capacity can be configured for each of these instances, depending on the requirements[5, p. 1]. The free tier system specifications allows for running a *t2.micro* instance for up to 750 hours per month. A t2.micro instance has access to 1GB of memory and 'Low to Moderate' network performance, according to the AWS management console.

To secure the communication between an EC2 instance and a remote user, public-key cryptography is used[5, p. 508]. Key pairs used for access can be created through the EC2 console, or a command line. The terminal of EC2 instances can be accessed remotely by using a SSH client, for example the remote terminal software PuTTY[5, p. 22]. This approach requires that the .pem key file that is

[4]The figure originates from the AWS IoT documentation[12]

[5]AWS Greengrass cores are a part of the AWS Greengrass service provided by Amazon. It is described as a software that extends capabilities provided by AWS to local devices[10, p. 1].

generated by AWS is converted to a .pkk file. This can be accomplished by using the PuTTYGen software[5, p. 512]. To transfer or remove files on an EC2 instance a command-line tool like PuTTY Secure Copy, or the GUI tool WinSCP can be used[5, p. 398]

Amazon Simple Storage Service - Amazon S3

Amazon S3 is an object storage service provided by AWS[7, p. 1]. This means that information stored on the S3 service is identified as objects. These objects contain the stored data, meta data and a form of unique identifier, or key. This form of storage allows for storing and retrieving large amounts of unstructured data[7, p. 3]. The individual objects are organized in different *buckets*, in Amazon S3; managed by users. In these buckets, distinct objects are assigned their unique *keys* to identify them. S3 allows for storage of vast amounts of data, users storing up to 50TB pays $0.023USD$ for each GB per month, while users storing over 500TB pays $0.021USD$[13]. It allows users to retrieve the objects stored on it by using associated URLs with the formats shown in listing 2.1.

```
1 http://s3.amazonaws.com/bucket/key
2 http://bucket.s3.amazonaws.com/key
3 http://bucket/key
```

Listing 2.1: The different URL formats that can be used to access an object stored on Amazon S3.

These formats allows S3 to be used to host web pages or information used by smart phone apps. It also allows users to generate time-bounded URLs. These gives a third party limited period access to a bucket, or some of the objects.

Chapter 3

Previous Work

This chapter will contain a study of two available commercial embedded systems with a similar functionality to the system implemented by the author of this thesis. It will also contain a review of a couple of scientific papers looking at how best to predict the energy production of PV solar panels. This is done by investigating what kind of weather attributes matter the most and in what ways.

3.1 A Study of Existing Embedded Systems

3.1.1 Evaluation Criteria

The evaluation of these embedded systems was conducted with these features in mind:

- Energy consumption
- Price
- Wireless and wired communication
- Accompanying Software
- Cloud Integration
- Scalability

3.1.2 Findings

Finding 1: ELITEpro XC

According to Dent Instruments the ELITEpro XC is a portable energy logger solution for pinpointing electric usage, recording and analyzing electrical consumption[49]. It has four channels which are able to measure $0-600VAC$ or DC for single or three phase systems and analog inputs that can measure $0-10VDC$ and $4-20mA$. For storage it has a $16MB$ non-volatile memory for multiple months of logging data. The ELITEpro can be configured using USB, Bluetooth or Wi-Fi with the ELOG software. It consumes a maximum of $500mA$ with a voltage of $6-10VDC$, which means it has a maximum power consumption of $3-5W$. The ELITEpro XC is powered by the bus line it monitors. The ELITEpro XC costs around $1,350USD$[2].

Figure 3.1: A picture of the ELITEpro XC

Finding 2: TCG120

The user manual for the TCG120 describes it as a remote monitoring controller for distributed surveillance and control [78]. The TCG120 has two digital inputs with "dry contact" and "logic level" modules, two analog inputs ($0-60VDC$) and 1-Wire support for up to four Teracom temperature or humidity sensors. It can use the GSM network to push its logged data using the HTTP post method on a remote server. The TCG120 can be configured using a USB or SMS and the firmware can be updated using a USB cable or GPRS. The TCG has a price tag

of $159EUR$ [3] and has a maximum power consumption of $0.76W$[1].

Figure 3.2: A picture of the TCG120

3.1.3 Comparing the Findings

Energy Consumption

There are some variation in the energy consumption of these embedded systems. The ELITEpro XC has the highest known energy consumption of $5W$, while the TCG120 consumes less than 20% of that, with a maximum of $0.76W$.

Price

The ELITEpro XC is by far the most expensive of the embedded systems. This might be the result of the intended usage of the system. It is, as opposed to the TCG120, a mobile and robust system used in an industrial setting, while the TCG120 is not intended for such harsh environments. It is used as stationary equipment in relatively friendly locations.

Wireless and Wired Communication

The ELITEpro XC is unable to communicate using the GSM network, but it is also the only one that can use Wi-Fi and Bluetooth for nearby wireless communication. The TCG120, on the other hand, uses an embedded GSM modem for remote wireless communication and control.

Both the ELITEpro XC and TCG120 uses analog and digital inputs for

[1] The figure was sent using email by one of the employees of Teracom as a response to a request written by the author of this thesis.

data collection and both can connect to a host computer using USB. Neither uses any industrial standards, such as RS-485.

Accompanying Software

Both embedded systems can be configured using a wireless network protocol and their own dedicated software. The ELITEpro XC uses ELOG and the TCG120 TC Monitor.

Cloud and Server Integration

The ELITEpro XC has a primitive internal web server[2]. This is intended to be used for remote access through PCs, smart phones etc[50, p. 126]. The TCG120 on the other hand is intended to send HTTP post messages to a remote server, and is accessed using commands.

Scalability

Neither of these two embedded systems are intended to be scaled up to a large scale systems. The ELITEpro XC is a mobile unit intended for short-term installation to pin point power usage. The TCG120 on the other hand, is intended for up to medium-size systems and permanent installations.

Conclusion

The ELITEpro XC is a portable and robust system intended for a rough environment; the factory floor. This functionality is reflected in its relatively high price, few connections and that it receives its power from the line it monitors. The TCG120 is intended for a permanent an distributed installation in a more friendly environment. Its server integration allows for scaling up to a medium sized system, something the ELITEpro XC is not designed to do. This difference in operation is reflected in the TCG120's relatively low price compared to the ELITEpro XC.

3.2 Studies on Estimating Electric Power Production By Solar PVs

This section will summarize the literature study conducted by the author of this thesis on the estimation of solar PV power production. The section will look at how different attributes of the weather affects the power production and what kind of mathematical and technical tools have been used to conduct these studies and what can be used for this thesis. It will not go into technical details regarding the different AI and statistical methods described here.

[2]No other mentions of servers were found in the user manual of the ELITEpro XC.

3.2.1 Weather Attributes

A study conducted at the Huazong University of Science and Technology in China found that the most useful weather attributes for estimation were solar irradiation, wind speed, air temperature and relative humidity[20, p. 2862-2863]. With the form of the graph of solar irradiation being almost identical to the one of power output. Both temperature and wind speed have a positive correlation with the power production. Where higher temperatures seem to correlate with solar irradiation and with the wind cooling down the solar panels, thereby increasing their efficiency. Relative humidity negatively correlates with the power output.

The results of this study is supported by others, where the inclusion of all available weather data gave the best solar power estimations[58, p. 94]. This study used the temperature of the solar cell, ambient temperature, solar irradiance (on both a 3° and 15° tilt).

The type of weather also has a significant effect on how accurate an estimation is. Chen, Duan, Cai and Liu found that during sunny days, their estimation have a correlation coefficient in the range of $98 - 99\%$ with a mean absolute percentage error of between $8.29 - 10.8\%$[20, p. 2868]. Cloudy days have a sightly lower accuracy, with a correlation coefficient of $96 - 99\%$ and a mean absolute percentage error of maximum 15.08%. Rainy days are far more difficult to estimate, with a correlation coefficient as low as 48.92% and a mean error of $24.16 - 54.44\%$. A note written by Geir Mathisen also underscore this difficulty in estimating the power of solar PV in locations with few sunny days[59]. Figure 5 in the note shows how snow that has fallen on the solar cells stops nearly all power production by preventing the solar rays hitting the PV cells.

3.2.2 Estimation Tools and Techniques

Every single paper that was studied for this thesis used some form of linear regression to estimate power output of solar PVs based on different environmental factors. What separates the different papers is the tools and techniques used for finding the expression used in the linear regression formulae.

The majority of the scientific papers on the estimation of solar power that was used for this section utilized Artificial Neural Networks (ANN) as the machine learning tool for improving their estimation[20, 58, 71]. Another, older study, used the Sigmund function together with a form of quality control of the data points[53]. This quality control included among other things limiting the maximum solar irradiance a given data point could have, to prevent exceptional situations from distorting the analysis and predictions[53, p. 884].

Looking at values of the mean squared errors (MSE) of the papers, the authors who uses ANNs often get the most accurate data. As described above, Chen, Duan, Cai and Liu had at times a MSE of less than 15.1%, with an average MSE of all days of 19.145%[3], while the Ruiz-Arias , Alsamamra, Tovar-Pescador and Pozo-Vázquez had a MSE of at least $20 - 25\%$, with a far larger data set from a larger number of locations, with additional quality control of the input data.

3.2.3 Conclusion

The number of sunny days seems to be the main environmental factor in how difficult it is to estimate the amount of power produced from solar PVs. Clouds, rain and particularly snow seem to make the estimation quite difficult, even when advanced statistical methods and powerful tools such as ANNs are used. The type of tools used to estimate the regression formulae is also of great importance. ANNs gives the best predictions given a particular amount of information. Other tools, such as the Sigmund function can give almost as accurate predictions, but need more data from multiple sources over a longer period of time, and a form of quality control over the collected data.

[3]The percentage was found by calculating the average value of the twelve MSEs in Table 1 in the paper by Chen, Duan, Cai and Liu[20, p. 2868]

Chapter 4

Specifications and Design

This chapter of this thesis will define the specifications of the embedded system and the cloud services that is to become the solution of this master thesis. It will also describe the process of choosing the hardware, software and other services used in the implementation of the system and which alternatives were considered.

4.1 An Overview of the Solution

The main function of the implemented embedded system is to monitor the power production of solar PVs. The collected data is then handled by the embedded system and then transmitted to a cloud service using the cellular network. The collected solar PV data is to be computed together with scraped forecast data to try to estimate the power production for the next 24 hours.

4.2 The Structure of the Systems

The system can be divided into three main modules, that can be split into multiple modules. The embedded system, which collects the power production data and transmits it to AWS, the AWS services utilized and the computing which is performed in the cloud service.

4.2.1 The Embedded System

The embedded system will consist of a hall effect sensor, used to measure the power that goes through a power cable. The sensor is connected to one of the analog inputs of the Cortex-M4 based MCU. The information collected by the sensor will

then by sent to the cloud service by using the cellular modem. The modem will be connected to the microcontroller by using a serial interface.

The Micro Controller Unit - MCU

The MCU to be used for the embedded system needs to have the an adequate amount of pins for communicating with its peripherals. This includes multiple serial interfaces available to used for communicating with the 2G/4G modem and GPIOs for sensors.

The Sensor(s)

One or more sensor units are to be connected to the main system. The sensor needs to be able to handle voltage and current produced by the solar PV system. It should also be possible to change the sensor with little to no modification to the PV installation.

The 2G/4G modem

One of the main objectives of the system is to send data over the 2G/4G network. Therefore, 2G/4G modem needs to be connected to the main embedded system. The modem should have one or more serial interfaces that can be used to connect it to the microcontroller.

4.2.2 The Cloud Services

The cloud service should be able to provide one or more databases to store the collected power and forecast information. It also needs to provide cloud computing, for handling the collected data.

4.2.3 The Cloud Computing

The cloud computing should be run using software that utilize the various tools of the platform. The chosen programming language and structure should be in a fashion that allows for a stable, secure and relatively fast implementation.

4.3 System Specifications

The following tables describe the system specifications in the form of product requirements and how their functionality is confirmed using the appropriate tests. The specifications were designed based on the findings from section 3.1.3 and the system structure presented in the previous section. They are meant to specify the functionality of a system ready for deployment in the field.

Req ref	Product req	Acceptance req	Test ref
PR-1	The system shall be able to communicate over the 2G/4G network with a cloud service or server for transmitting recorded sensor data.	Add elements to a database stored on a cloud service.	TR-1
PR-2	The system should be able to obtain data from one or more sensors	Communicate with one or more sensors.	TR-2
PR-3	The system should be able to store data received from its sensor modules on a non-volatile storage, with a time stamp.	See that the appropriate files exist on the non-volatile storage, with the correct content.	TR-3
PR-4	The data stored in the cloud service or server should be used for further computation and estimation.	The data stored at the cloud service is accessed using tools provided by the cloud platform.	TR-4
PR-5	The system is implemented in such a fashion that it is able to function under real-time constraints.	The system uses an oscillator for time synchronization.	TR-5
PR-6	The system is able to handle loss of power, both for short and longer periods of time.	The system is equipped with a rechargeable battery to be used if the system loses its main power supply.	TR-6
PR-1.1	Use a 2G/4G modem connected to the system using a serial protocol to communicate with the cloud service.	Send and receive confirmation that the messages reached the cloud service or server.	TR-1.1
PR-2.1	Use a means of communicating with the connected sensor(s).	Read the sensor output.	TR-2.1

Table 4.1: The first table containing the system specifications

Req ref	Specification	Acceptance req	Test ref
PR-3.1	Use an external module, connected to the control module for long-term storage.	Read and write to files stored on the storage unit.	TR-3.1
PR-4.1	Use the provided tools for data analysis and estimation provided by the cloud service or the programming language used to implement them.	The collected data is analyzed.	TR-4.1
PR-5.1	The system is sufficiently deterministic to meet its most critical deadlines to perform its intended function.	The system is able to meet its deadline below a certain maximal amount of work.	TR-5.1
PR-5.2	The system is able to handle unforeseen events, errors and accidents, by fail-soft-operations.	The system is able to function properly when exposed to some simulated errors and events.	TR-5.2
PR-6.1	Simulate a temporary loss of power from the supply.	The system is able to handle simulated loss of power	TR-6.1
PR-1.1.1	Send a TCP/IP message to a cloud service or server.	Confirm that the message is received.	TR-1.1.1
PR-1.1.2	Add standardized test data to the database located in the cloud service	Confirm that the test data is added to the database.	TR-1.1.2
PR-2.1.1	Read the measured sensor value(s).	Read the sensor content either in a log file, or through debugging.	TR-2.1.1
PR-2.1.2	Compute the sensor value(s) if needed for further use.	Confirm the computational result by either reading a log file, or through debugging.	TR-2.1.2
PR-3.1.1	Write a standard text string to a text file on the storage medium.	The file contains one or more instances of the text string with appropriate time tamps.	TR-3.1.1

Table 4.2: The second table containing the system specifications

PR-5.1.1	The system is able to handle the loss of a sensor.	The correct handling of loss is confirmed through debugging or logging.	TR-5.1.1
PR-6.1.1	The system behaves in an appropriate manner when it loses its power supply based on workload and remaining battery charge.	Confirm the correct behaviour either by reading a log file, or through debugging.	TR-6.1.1
PR-1.1.1.1	A modem capable of using the 2G/4G network.		TR-1.1.1.1
PR-1.1.1.2	A server accessible through the internet.		TR-1.1.1.2
PR-1.1.2.1	A database located in a cloud storage service.		TR-1.1.2.1
PR-2.1.1.1	A sensor capable of handling a medium amount of current (min. 10A) and that require little modification to the current line.		TR-2.1.1.1
PR-3.1.1.1	A hardware module containing a medium for non-volatile, long-term storage of data.		TR-3.1.1.1
PR-3.1.1.2	A hardware module for obtaining an accurate timestamp. For example a real-time clock.		TR-3.1.1.2
PR-4.1.1.1	A platform for running the analytical tools and/or programs, and a storage medium or database for handling the result.		TR-4.1.1.1

Table 4.3: The third table containing the system specifications

4.4 Choosing the Embedded Hardware

This section describes what kind of hardware were chosen to implement the embedded system and why.

4.4.1 The MCU

The MCU was not chosen by the author of this thesis, but by the supervisor in an effort to standardize the hardware across multiple thesis projects[1]. The NXP LCP4337 Cortex-M4 MCU was therefore chosen[64] for its low price and power consumption. The LCPXpresso 4367 development board was used to prototype the embedded system[65]. Choosing the LPC board platform gave access to the hardware specific functions provided by the LPCOpen library[63]. This eliminated the need to implement the low-level functions for UART communication to be used during the implementation phase of this thesis.

According to its product page, the LCPXpresso 4367 development board is a low cost platform for developing software for NXPs ARM based MCUs. The board has an attached JTAG-debugger, which uses the board's UART. The UART can be freed during debugging by using the LPC-Link 2 debugger[66], that uses the 10 pin segger port on the LCPXpresso 4367 to connect to the board.

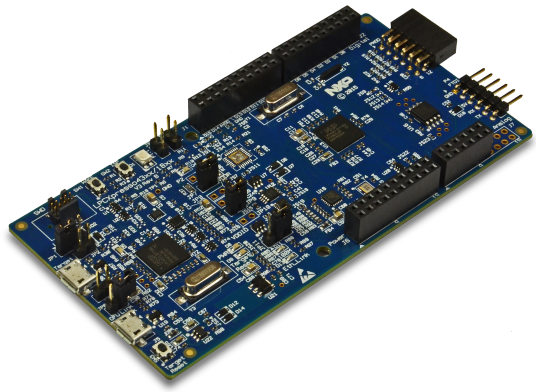


Figure 4.1: The LCPXpresso 4367 development board.

[1] This includes the master thesis written by Erlend Grande M.Sc.[45]

4.4.2 The Cellular Modem

The choice of cellular modem to connect to the rest of the embedded system was made with the following criteria in mind:

- Price
- Connectivity
- Power Consumption
- Amount of support, from the producer and the community around it
- Partnerships, or any other forms of guarantees with cloud services

Based on the criteria defined above, the Digi XBee Cellular 3G was chosen[28]. This was a result of its relatively low price, that it can be connected to by using UART or SPI, their costumer support was helpful and that their user forum had some activity. It also helped that Digi is an official partner of AWS[29]. The modem also uses the Digi specific API. This allows the multiple Digi modems to be controlled using the same software utilizing the API, allowing for a greater amount of modularity.

An alternative to the Digi 3G modem is the LTE-CAT version. The author of this thesis ended up choosing the 3G version for a couple of reasons. The 3G version is cheaper ($709NOK$[62] vs. $787NOK$[61]). Another is the higher power consumption of the LTE-CAT version[31], which compared to 3G modem is as follows[30].

Operation Mode	**3G**	**LTE-CAT**
Transmit	$702mA$ @ $3.3V$ $425mA$ @ $5V$	Avg. $860mA$ @ $3.3V$ Max. $1020mA$ @ $3.3V$
Receive	$224mA$ @ $3.3V$ $160mA$ @ $5V$	Avg. $530mA$ @ $3.3V$
Idle/Listening	$87mA$@$3.3V$ $73mA$ @ $5V$	$143mA$@$3.3V$
Deep sleep	$10\mu A$ @ $3.3V$	Approx. $10\mu A$@$3.3V$

Table 4.4: A comparison of the power consumption of the Digi 3G and the LTE-CAT modem

From table 4.4 we see that when transmitting data, the LTE-CAT modem uses approximately 30% more power, and more than double when receiving data. The LTE-CAT will use a shorter amount of time when transmitting and receiving information than the 3G version, reducing the gap. However, the author of this thesis

believe that this will not reduce the total power consumption sufficiently. This is because the modem is not intended to send or receive large amounts of information. Since the LTE-CAT also consumes almost the double when idle/listening the possible transmission offset will in all likelihood not make that much difference.

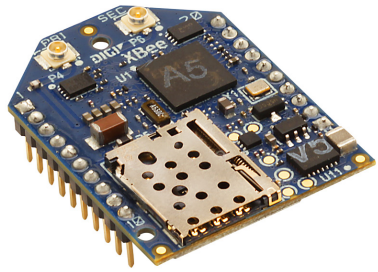


Figure 4.2: The Digi 3G Global Cellular modem.

4.4.3 The Sensor(s)

The choice of the sensor module to be connected to the embedded system was chosen with the following criteria in mind:

- Robustness
- Simplicity

One of the sensors that was considered for monitoring was the LTC4151 voltage and current sensor[57]. It can measure voltage in the range of $7 - 80V$ and a current with the current limited by the measuring resistor. This sensor was not chosen due to the fact that it uses the I2C standard and because it would require a physical modification to the power line to be measured. This would complicate the installation and require more work related to the implementation process.

The chosen sensor was the Current Transducer HO-P hall effect sensor[56]. It has the advantage of not needing to modify the power cable to measure the current running through a cable. The sensor only requires that the cable is put through its ring to measure the current. The sensor is able to measure up to $25A$, but can be changed if needed. The sensor is described as being 99% linear. Because it produces an output current, the system only use an ADC to read the value of the sensor. This can be accomplished using some of the functions provided by the LPCOpen library. Thus reducing the needed time for implementing the system.

Figure 4.3: The HO-Phall effect sensor.

The HO-P has an output voltage in the range of $2.5V - 5V$. While the ADC on the LCPXpresso 4367 is able to handle such voltages, it is only able to measure up to $3.3V$. The voltage therefore has to be divided, to two-thirds of its original value, so that the range is $1.7V - 3.3V$, to allow the LCPXpresso 4367 to measure the maximum output of the sensor.

4.4.4 The Final Embedded Hardware Setup

See figure 4.4 for a graphical representation of the final hardware setup used for the implementation of the system. The HO-P is connected to the LCPXpresso 4367 by a $5V$ and GND connection, giving in an input current and signal ground for the voltage output. The output voltage is divided as described in section 4.4.3. The LCPXpresso 4367 is powered by a USB cable and is connected to the Digi 3G modem by its UART. The modem is placed on its development board and powered by a standard power cord, giving it an input current of $5V$ and $1.5A$.

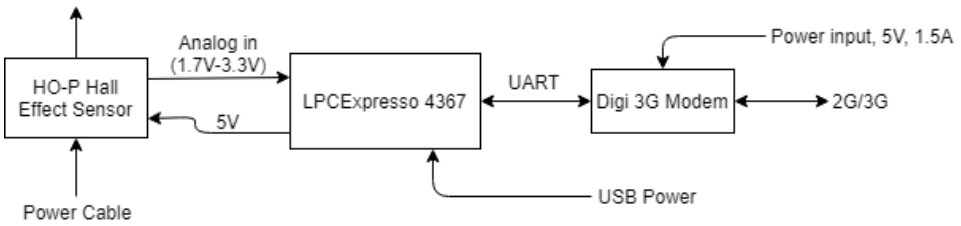


Figure 4.4: The hardware setup of the embedded system with the different communication standards and protocols used.

The cellular modem is to be encapsulated together with the development board

when the final system is deployed. The embedded system is designed for permanent installation, unlike the ELITEpro XC.

4.5 The Embedded Software Setup

Since the embedded system is to be a real-time system, the software to be used for the embedded needs to enable real-time capabilities for it. For this reason, the most widely used OS for MCUs, FreeRTOS, was chosen as the software platform for the embedded system. It was chosen for its broad use and support. Reducing the necessary time for implementing the embedded system, and potential errors. It was also chosen in order to standardize the software implementation over multiple thesis projects.

4.6 The Cloud Service Setup

The evaluation of the cloud service to be used was mainly based on the work done by Marit Tundal[80]. AWS was therefore chosen for its thorough and easy to navigate documentation[80, p. 15]. The fact that FreeRTOS is currently owned by Amazon, and that they have released their own version of the OS, called Amazon FreeRTOS[76] makes an even stronger case for using their platform instead of for example Microsoft Azure, or Google Cloud Platform, since the majority of their services with regards to IoT is quite similar.

4.6.1 Database

To store the information collected by the embedded system, the NoSQL database DynamoDB will be used. This gives the possibility of integrating the system with AWS IoT, allowing for increased scaling, customization and security.

4.6.2 Cloud Computing

For performing the cloud computing, an EC2 instance will be used to simulate an Ubuntu machine. This will allow the system to use all the functionalities of Linux for the handling and computing of data.

The software used to handle the data located in the DynamoDB database will be written in Python. This is done to take advantage of the vast amount of available open-source libraries for a speedy and high quality implementation of the needed software. While Python may not be as resource efficient as for example C, this is not a concern, as the intended workload is not particularly intensive, and the decrease in the necessary work to implement the solution far outweighs any loss in efficiency.

The following Python programs will be implemented to run on the EC2 instance.

Yr Scraper

The Yr Scraper will run periodically on the EC2 instance and collect the weather forecast, and the actual weather data on one or more locations. The scraped weather data will come from Yr, the Norwegian weather service.

MQTT Handler

This program will handle the data transmitted by the embedded system to the MQTT broker. It will retrieve the messages it receives when subscribed to one or more topics. It will then place the collected messages into the appropriate DynamoDB table. The messages stored in this table can later be used by other parts of the system for further analysis and computation.

Graph Handler

This script will use the data collected by the Yr scraper, and the embedded system to generate plots for a graphical visualization to be used for further analysis. It will also analyze the collected data and try to make estimations for the power production for the next 24 hours based on historical data, and previous analysis.

4.6.3 Long Term Storage

To store any output from the different Python scripts described above for a longer amount of time, Amazon S3 will be used. This service was chosen for its ability to store vast amounts of data, and the possibility of making the information stored available for third parties at a later date. It also allows the system to use fewer APIs, as the tools used to access the DynamoDB tables can be used to access Amazon S3 buckets.

4.7 The Final System Setup

See figure 4.5 for the final system setup.

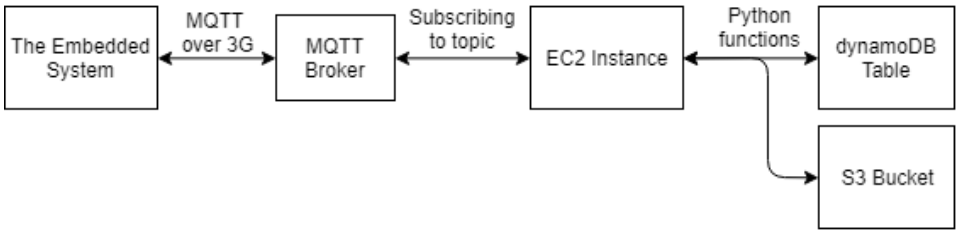


Figure 4.5: The setup of the system with the different communication standards and protocols used.

The embedded system will collect data using one or more sensors, and periodically send the information using its 3G modem using the MQTT protocol to an open MQTT broker. A Python program running on an EC2 instance on AWS will continuously subscribe to the appropriate topic on the broker, and when it receives a valid message from the broker, the program will handle the message and place it in the appropriate DynamoDB table.

The other Python programs running periodically on the EC2 instance will also place information in their respective DynamoDB tables. One for the scraped weather forecast data. Amazon S3 buckets will be used for long term storage of any files and plots from the analysis of the data stored on the different tables.

Chapter 5

Implementation

This chapter will go through the implementation of the embedded system and its modules, the software that was implemented for running in the cloud service provided by AWS and how these services were set up. The chapter will also describe which tools were used for testing and debugging the different modules of the overall solution and end with a description of the tests used to confirm both the functionality of the sub systems, and the system as a whole.

5.1 The Embedded System

This section explains how the embedded system was implemented using the software and hardware specified by the supervisor and the author of this thesis. See figure 5.1 for how the embedded system looked like at the end of the implementation phase of this thesis.

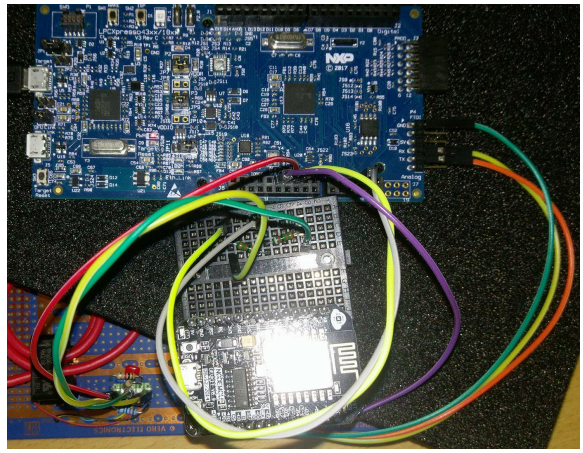


Figure 5.1: What the embedded system looked like at the end of its development.

5.1.1 Overall Implementation and Functionality

The main functionality of the embedded system is to measure the current running through a cable, temporarily store it locally before sending the data periodically to a cloud service using a 3G modem and the MQTT protocol. To accomplish this functionality, both the hardware and software of the embedded system were organized in a modular and intuitive fashion. Both the sensor and the modem have their respective source and header files.

When the system boots up it runs through the normal operations that characterize a FreeRTOS system. System-wide hardware is initialized, the different tasks are defined with their respective handler, priorities and stack sizes before being started by the *vTaskStartScheduler()* function. Afterwards, the task with the highest priority is given processor time, until it uses a *vTaskDelay()* function, which puts the task to sleep for a specific number of processor ticks. Then the next task is run, until it calls next delay function. If there are implemented tasks that needs to be run, the FreeRTOS idle task is run, ensuring that there are always at least one task occupying the processor. See figure 5.2 for a flow diagram of the overall functionality.

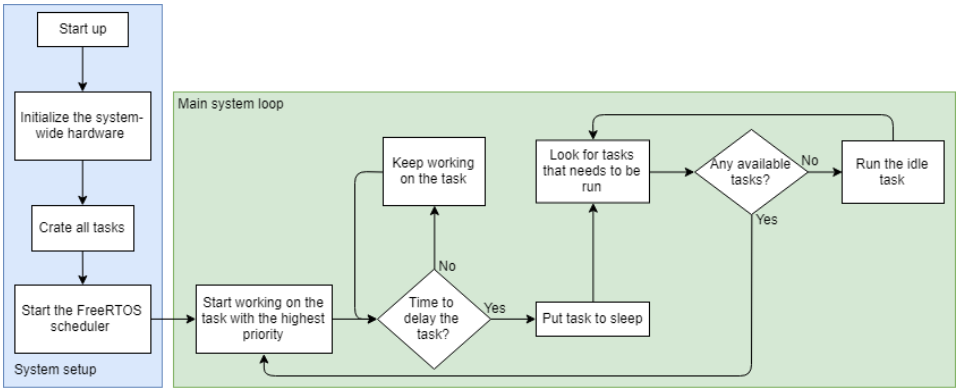


Figure 5.2: A flow diagram showing the overall functionality of the embedded system.

The system is intended to consist of three main tasks. One is used to handle the sensor, reading the ADC value and perform some minor computing using these value. The second is the task used to connect to the MQTT broker, and publish messages to a topic when the system is connected to the broker. The third task is used for receiving messages from the 3G modem. Since it may send status messages depending on the state of modem, being able to receive and handle the information is essential. A fourth and smaller task was also implemented. It turns one of LEDs of the LCPXpresso 4367 board on and off. This was done to give a visual confirmation that the system was running.

5.1.2 Changing the Implementation

As a result of time constraints, bugs and other issues, the implementation of the embedded system was changed towards the end of the implementation period. The Digi 3G cellular modem was replaced with the ESP8266 NodeMCU module. This changed both the hardware and software implementation. Due to the less sophisticated nature of the communication between the LCPXpresso 4367 and the ESP8266, versus the communication with the 3G modem, the number of FreeRTOS tasks was reduced by one. The task used to receive messages was was no longer necessary. The technical details of this implementation will be discussed below, and a more thorough discussion of this decision will be conducted in section 7.2 of this thesis.

5.1.3 FreeRTOS Tasks

This subsection elaborates on how the different FreeRTOS tasks described in section 5.1.1 were implemented for this thesis.

The Sensor Reading Task

The sensor reading task starts by setting up the ADC, using the *prvADCSetup()* function. This function sets it up by using the board specific function provided by the *LPCOpen* library. Then it sets up the ring buffers used to hold the read sensor values.

After this, the task starts its while loop. First it calls on the *handleHallSensor()* function. This function reads the ADC value and then places it into the ring buffer intended for the sensor. It then checks if the ring buffer is full. If yes, it pops every single value and calculates the average value using the *prvGetAverageHallSensorValue()* function. It then calculates the ADC bit value into the corresponding ampere value, using the *prvHallValueToAmp()* function. Finally the ampere value is added to the ring buffer used for all sensors, to be used by other tasks. See figure 5.3 for a flow diagram of the sensor reading task.

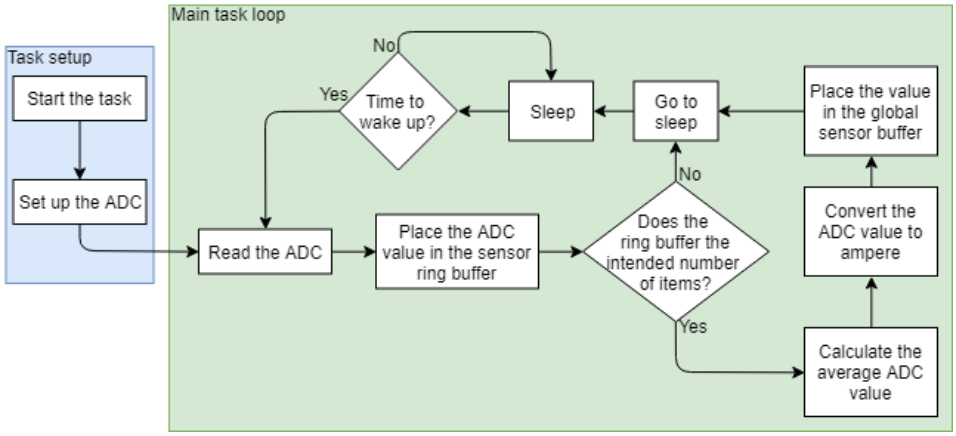


Figure 5.3: A flow diagram showing the functionality of the sensor reading task.

Modem Receiving Task

The modem receive task starts by setting up the modem, which includes setting up the UART hardware. Then it starts on the main loop. It checks if there are any bytes to be read in the RX ring buffer. If no, the task is put to sleep, before trying again. If there are bytes in the RX buffer, the modem checks if it is currently reading a message. If not, it reads the first four bytes stored in the ring buffer. If the bytes are from a valid API message, these bytes will include the start delimiter, length of the message type. The task first checks if these bytes are valid. If they are, it uses the length bytes to read the remaining of the message in the RX buffer. Then, depending on the message type, the task handles the message. If it has received a status message, it is placed in the *socket0_status_buff* ring buffer, which holds the status messages related to a given socket. If the received message is a RX IPv4 message, it is placed in the *socket0_buff* ring buffer. So that the content can be retrieved by a *modem_recv* function later(see the section on the modem driver in section 5.1.4). Figure 5.4 shows a flow diagram of the modem receiving task.

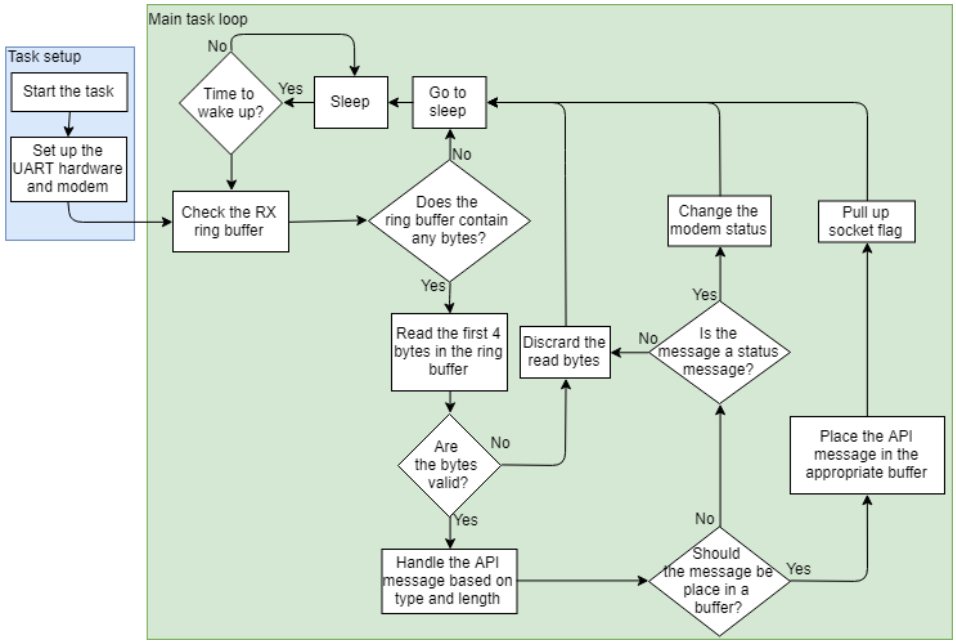


Figure 5.4: A flow diagram showing the functionality of the modem receiving task.

The MQTT Client Task

The MQTT client task starts declaring its local variables. Then it waits until the *modemReadyFlag* is pulled high, before starting to set up the network and MQTTClient structures, provided by the paho library. After this, the client uses the *NetworkConnect()* function to set up the modem socket. Next, it starts the paho MQTT thread, used to handle the connection. Next, the *MQTTConnect* was called to connect to the chosen MQTT broker.

Unfortunately this is a long as the author of this thesis came in implementing this FreeRTOS task. The plan was to model the client task after the echo example provided by the paho library[33]. After the system had connected to the chosen topic, it would publish messages when sensor values became available. See figure 5.5 for a flow diagram showing how far the implementation got, and what was intended.

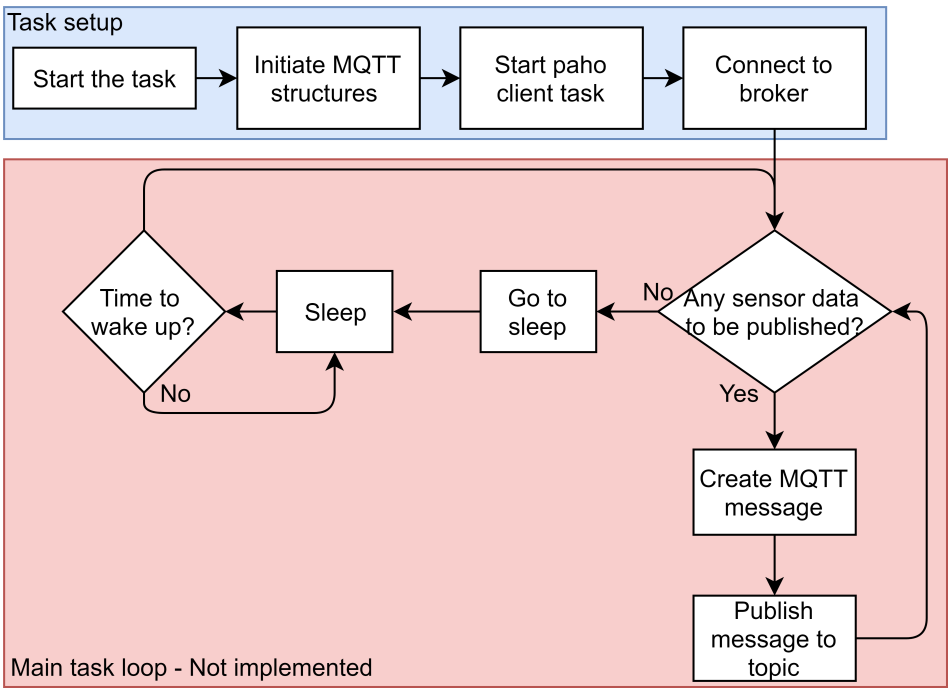


Figure 5.5: A flow diagram showing the functionality of the MQTT client task. It includes both the functionality that was implemented (part of the task setup) and what was planned (the end of the task setup, and the main task loop).

5.1.4 Software, Driver and Libraries

UART Driver

The UART driver for the embedded system was developed by using the board specific UART functions implemented by the producers of the development board. The functions were accessed by using the hardware specific *board.h* file. The functions were found by reading and testing different example projects.

The driver uses two 256 byte ring buffers to both transmit and receive data on the UART pins. The data is sent and received using interrupt that call on the board specific *Chip_UART_IRQRBHandler()* function, that handles the sending and receiving of bytes on the UART. An unsigned character array is transmitted by using the implemented *uartSendString()* function. For receiving of bytes, the *uartReceiveString()* function was implemented. It runs until the board has received a predefined number of bytes, or until a timer has run out. This was implemented by using the *xTaskCheckForTimeOut()* function provided by the *task.h*. The ring buffer used for receiving bytes on the UART is only called when the UART interrupt flag is high. Two semaphores are used to control the access to the TX and RX ring buffers respectively, to avoid race conditions.

The API Parser

In order to control the 3G modem, the Digi API was used. This was done to avoid defining a new protocol for the modem. To utilize the API a set of functions was implemented to generate and parse the different frames. A number of macros were defined in the header file *api_frame_lib.h* to ease the work. This header file included the integer values of the different message types, some of the modem status responses and the byte offset values of some of the frame types. A function for generating the message length and check sum was defined to increase modularity of the modem driver.

The parser implemented to handle the sending the API frames shown in table 5.1.

API message	**Usage**
TX IPv4	Used to send IP messages over the internet. Is able to communicate over UDP, TCP or SSL.
AT Command	Used to perform cellular commands, such as if it is registered to the cellular network, or the signal strength. Does currently not support AT commands with parameters.

Table 5.1: The API frame types that were implemented for sending.

The parser was able to handle the messages shown in table 5.2 from the modem

API message	**Usage**
TX Status	Indicates the success or failure of a TX frame.
AT Command Response	The response of an AT command. Indicates the status of the operation. And may contain additional data.
Modem Status	Cellular status messages sent by the modem in response to certain conditions.

Table 5.2: The API frame types that were implemented for receiving.

Typecasting was used to convert any provided data that was represented as integer values, such as IP addresses, port numbers and the defined macros. To get the appropriate byte values to represent the length using two bytes, the *getAPIMsgLenght()* function was implemented, and the check sum is calculated using the *getAPIMsgChecksum()* function.

Modem Driver

The driver communication functionality of the driver was modeled on the functions used in the FreeRTOS+TCP stack[42]. This includes the functionality to create a socket, set its target IP address and port number and sending and receiving data on this socket. The *modem_socket()* function creates a socket, similarly to the *FreeRTOS_socket()* function. *modem_connect()* sets the target IP address and port number and sends a TCP ping message to establish a socket connection. If it is successful, the *modem_send()* function can be used to send an IPv4 message to the socket destination. The *modem_recv()* is used to receive data on the socket. The modem was implemented to handle only one TCP socket. This choice was mainly a result of time constraints and simplicity. The access of the global socket buffers were controlled using semaphores.

MQTT Communication

The paho-MQTT library for embedded C[34] was used to implement the MQTT communication with the broker. The files located in *MQTTClient-C* and *MQTTPacket* directories were added to the source and include folders of the project. This was done after some time had been used to try to use it as a Dynamic Link Library (DLL), but was abandoned after talking to NXP support[1].

Some modification was needed to to be able to use the library. This mainly involved the *MQTTFreeRTOS.c* and *MQTTFreeRTOS.h* for the library to use

the modem specific functions and to change the name of some macros to avoid duplicates.

Problems with the Modem Solution

The problem originated from that the RX ring buffer on the LCPXpresso 4367 did not receive the last two bytes of the message received from the modem as a response to the request to connect to the MQTT broker. See figure 5.7 for the last five bytes in the RX buffer.

[14]	[15]	[16]	[17]	[18]
$0x20$	$0x02$	$0x00$	$0x00$	$0x00$

Figure 5.6: This figure show the last five bytes in the RX ring buffer. See figure A.2 in appendix A for a screen shot of the buffer.

[14]	[15]	[16]	[17]	[18]
$0x20$	$0x02$	$0x00$	$0x00$	$0xCC$

Figure 5.7: This figure shows the actual last five bytes in the message. See figure A.3 in appendix A for a screen shot of the message presented in the XCTU software.

The LPCOpen function *RingBuffer_GetCount()* was used to check the number of bytes placed in the ring buffer as a part of the debugging. It found that the buffer contained two bytes less than it should have. This bug only occurred when the paho libary was used. When trying to send the same connect message without the library, the UART was able to place all bytes into the ring buffer. This error occurred shortly before the choice of changing the network solution was made. It did not occur during testing, which was conducted afterwards.

As a result of the time constraints, and the delayed response from NXP support, the Digi modem had do be exchanged with a simple Wi-Fi module in order to complete the implementation and to conduct the system-wide tests.

5.1.5 The Hardware Implementation

This subsection describes how the hardware part of the embedded system was implemented. This mainly involves how the different hardware modules was connected.

[1]The author of this thesis was told by NXP support that they had little experience with using DLLs with the MCUXpresso IDE.

The Sensor

The hall effect sensor is connected to the LCPXpresso 4367 using the following pins:

Sensor Pin	LCPXpresso 4367 Pin
1 (V_{in})	4 (5V)
3 (V_{out})	5 (ADC3)
5 (GND)	1 (GND)

Table 5.3: The pins used connecting the hall effect sensor and the LCPXpresso 4367

In addition to the connection of a couple of pins between the HO-P and the LCPXpresso 4367 board, the pins of the sensor had to be connected using one resistor, and three capacitors to operate normally, as shown in figure 5.8.

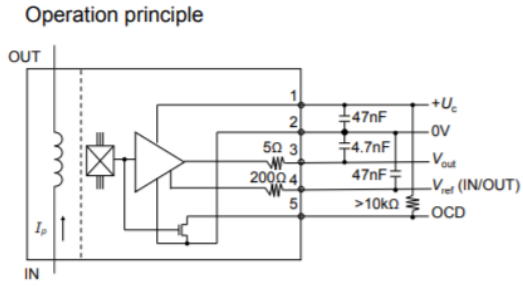
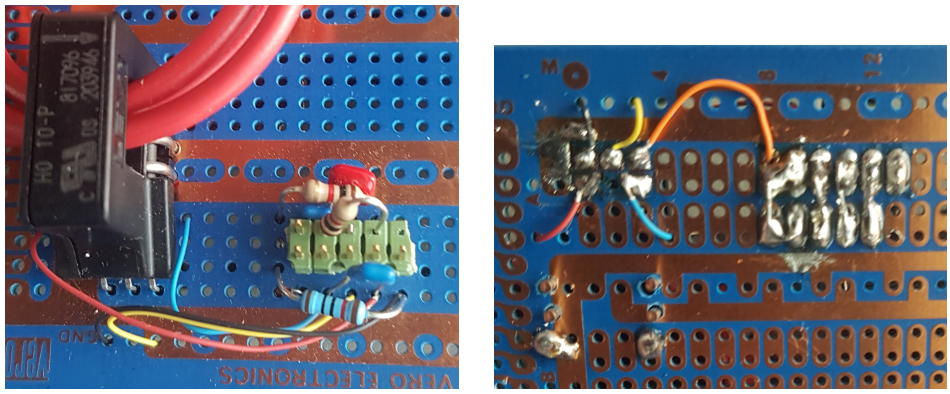


Figure 5.8: A graphical representation of the operation principle of the sensor set up using passive components.

The connection was implemented by shouldering the components together on the prototype board that the HO-P sensor was mounted to. The shouldering is shown in figure 5.9.



(a) Front view of the shouldering. **(b)** Back view of the shouldering.

Figure 5.9: How the operational principle shown in figure 5.8 was shouldered to the test board.

The Digi Modem

The modem is connected to the LCPXpresso 4367 board using the following pins.

Modem Pin	LCPXpresso 4367 Pin
3 (RX)	4 (TX)
2 (TX)	5 (RX)
10 (GND)	1 (GND)

Table 5.4: The pins used connecting the 3G modem and the LCPXpresso 4367

All of the pins of the LCPXpresso 4367 pins are located in header P4.

5.1.6 The Wi-Fi Implementation

This subsection describes both the hardware and software implementation that were done to facilitate the Wi-Fi communication used in the absence of functional cellular modem. To implement the Wi-Fi communication, the ESP8266 module was chosen. This was done due to problems with integrating the paho MQTT library with the implemented modem driver.

The ESP8266

The ESP8266 is a widely used, low cost Wi-Fi chip[84]. It has a 1MB flash memory, allowing for single chip Wi-Fi devices. The NodeMCU V0.9 was used in the implementation. The board supports the Arduino SDK and therefore gives access to a wide range of open-source libraries and previous projects[87].

Software Implementation

The software implemented for the ESP8266 consists of five files. First is the *ESP8266.ino* file, that contains the *setup()* and *loop()* functions present in all Arduino programs. Next is the *mqttHandler* .cpp and .h files. These implements the Wi-Fi and MQTT functionalities. This includes the Wi-Fi connection, the name of the MQTT broker and the topic to publish to. The *mqttSendData()* function is used to publish messages to the chosen topic bu using the *publish()* function provided by the *PubSubClient* Arduino library[67].

For communication with the LCPXpresso 4367 board the *uartHandler* .cpp and .h files were implemented. These files handles the receiving and sending of UART messages and debugging of the software. The function *uartGetMsg()* is used to check if the ESP8266 has received any commands from the LCPXpresso 4367. If yes, it returns the received message payload in the form of a string. Otherwise, it returns an empty string. The *uartDebug()* and *uartDebugByte()* functions were used to print messages to the UART during debugging. These functions only prints when the macro *UART_DEBUG* is set to one, and not when it is zero. This reduced the work needed to change the software when debugging was no longer needed.

UART Communication API

The API used to facilitate the communication between the ESP8266 and the LCPXpresso 4367 was designed to be similar to that of the Digi 3G modem. The structure of the messages used to transmit data to be published to the MQTT topic is shown in table 5.5.

Frame Fields	Byte	Description
Start delimiter	1	Always 0x7E
Length	2	$0x00$ to $0xFF$
Frame data	3 - number(n)	The message content.

Table 5.5: The frame format used in the communication between the ESP8266 and the LCPXpresso 4367.

The ESP8266 sends a confirmation message to the LCPXpresso 4367 when it has published the message. The confirmation message is always $0x7F$ $0xFF$.

The designed API is minimal, with no ways of handling errors, such as loss of Wi-Fi connection, or any other status updates. This limitation was the result of the time constraints with regards to the completion of this implementation.

Hardware Implementation

To connect the ESP8266 to the LCPXpresso 4367 board the following pins were connected:

ESP8266 Pin	LCPXpresso 4367 Pin
3.3V (V_{in})	3V3 (CN6)
GND	GND (CN6)
GPIO1 (TX)	4 (RX - P4)
GPIO3 (RX)	5 (TX - P4)

Table 5.6: The pins used connecting the NodeMCU ESP8266 module and the LCPXpresso 4367 using UART.

5.2 Cloud Computing

This section explains how the data collected by the embedded system described in section 5.1 is handled by the different services provided by AWS and software implemented by the author of this thesis.

5.2.1 MQTT Broker

The *iot.eclicpse.org* MQTT broker was initially chosen to be used for handling the messages sent by the embedded system. This broker was chosen because it is maintained by the same individuals that implemented the MQTT library used by the embedded system. It was later replaced by the *m23.cloudmqtt.com* broker, due downtime during the system-wide test, see section 7.1.4.

5.2.2 AWS Platforms

This subsection describes how the different AWS services were set up an configured.

EC2 Instance

In order to set up the EC2 instance a number of steps had to be completed. First, an IAM user had to be created by following the steps outlined in the EC2 documentation[5, p. 20]. This is because services like EC2 requires credentials when accessing them, something IAM provides[5, p. 19]. Next, a key pair was generated to handle the public-key cryptography used by AWS[5, p. 21-23]. After this key was generated through the AWS management console, it was converted using PuTTYgen in order for it to be used by PuTTY and WinSCP, as described in section 2.5.4. Finally, a security group was created to act as a firewall for the EC2 instance[5, p. 23-24]. To access the instances on the computers used during this thesis their IPv4 addresses were added to the exceptions in this security group, as defined by the documentation.

After these prerequisites were taken care of the EC2 was launched. By using the EC2 console, choosing *Launch Instance*, choosing the Ubuntu Server and the free tier hardware version[5, p. 27]. Finally, the instance was connected to the cryptography key pair described above. After being launched, the instance was added to the security group discussed earlier.

DynamoDB Tables

The DynamoDB tables were created by following the steps outlined by the DynamoDB web page in the AWS Management Console. By pressing the *Create Table* button, and then defining the table name, partition key and (optionally) a

sorting key. Then, the *Create* button was pressed, and within a few minutes, a new table was created.

S3 Bucket

The Amazon S3 bucket used to store the files generated by the software running on the EC2 instance was created using the S3 service in the AWS management console. This was done by following the steps outlined in the AWS documentation[8, p. 3-4]. By pressing the *Create Bucket* button, giving it a unique name, *haakonehmaster-bucket* and specifying the region it will be stored, *eu-central-1*.

5.3 Scripts Running on the EC2 Instance

This section will describe how the different scripts that were run on the EC2 instance provided by AWS. It will also discuss the tools used to access it and the Ubuntu services used to set up these scripts and how they were debugged remotely.

5.3.1 Accessing the EC2 Instance

To run the different Python programs in the cloud an EC2 instance was used. This instance simulated an Ubuntu 64-bit machine and only used the free-tier functionalities, as discussed in section 2.5.4. The instance was set up by following the steps outlined in the AWS documentation[15]. A key pair for accessing the EC2 instance remotely was also created. To access the instance , the terminal emulator, Putty[88], was used. WinSCP[89] was used to transfer, edit and remove files stored on the instance.

5.3.2 Overall Python Implementation

All of the Python scripts were designed with modularity in mind. Therefore, they were organized into different classes according to their main functions. The Yr scraper discussed in section 5.3.3 consist of the *YrHandler* class, which facilitates the scraping of the Yr forecasts, converts it to a Python dictionary and places them in the appropriate DynamoDB table. It then uses a simple *main.py* file to access the class, and set the target Yr location, and DynamoDB tables.

This architecture is common in all of the scripts, and can be graphically illustrated using figure 5.10.

This modularity allows for the different scripts to be extended or modified with relatively little effort. Additional locations can be scraped by running functions used in the main file multiple times, with different variables.

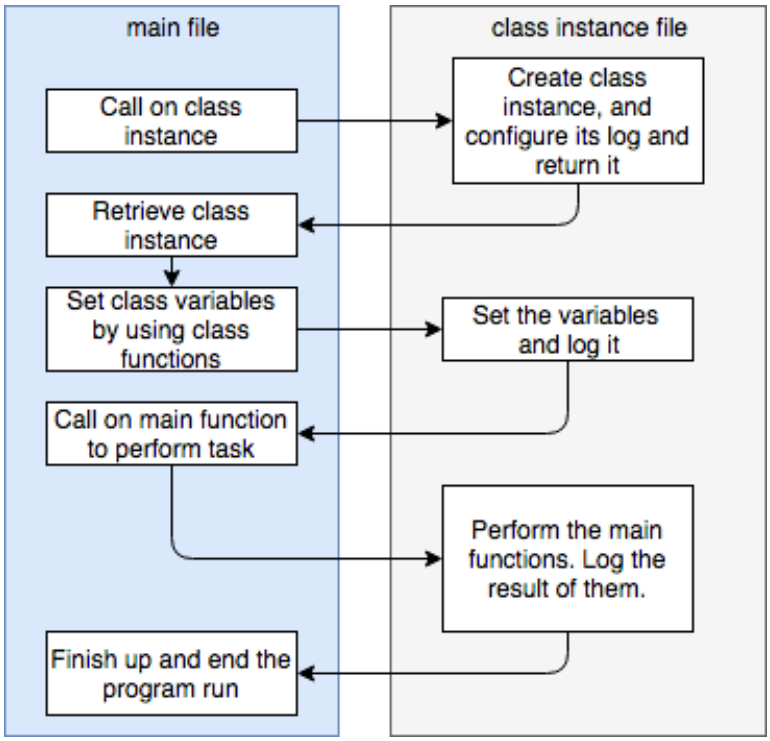


Figure 5.10: A flow diagram showing the overall functionality of all the python programs.

Accessing AWS

Another similar feature of the Python scripts is that they access one or more AWS services. To accomplish this the Python library and AWS software development kit (SDK) boto3[14] was used. To access the different DynamoDB tables by creating a DynamoDB class instance using the *resource()* function, then creating a table instance using the *Table()* function of the DynamoDB class.

Accessing the S3 bucket was done in a similar manner. A bucket instance was created using the *Bucket()* function of a S3 class instance.

To gain access to the different AWS services using the boto3 library a certification key had to be installed using the AWS Command Line Interface (CLI). The AWS CLI was configured by creating a key pair using the key file described in section 5.2.2[9, p.30].

5.3.3 The Yr Scraper

This sub section describes how the scraping of Yr forecast data, and their placement in the chosen DynamoDB table was implemented.

Software Architecture

As discussed in section 5.3.2 the Yr scraper is split into two main components. The *main.py* file, that is the authority of the scraping program, and *YrHandler.py*, that contains the class that handles the actual scraping of Yr data and storing the collected data in a DynamoDB table. See figure 5.11 for a simple flow diagram of the Yr scraper.

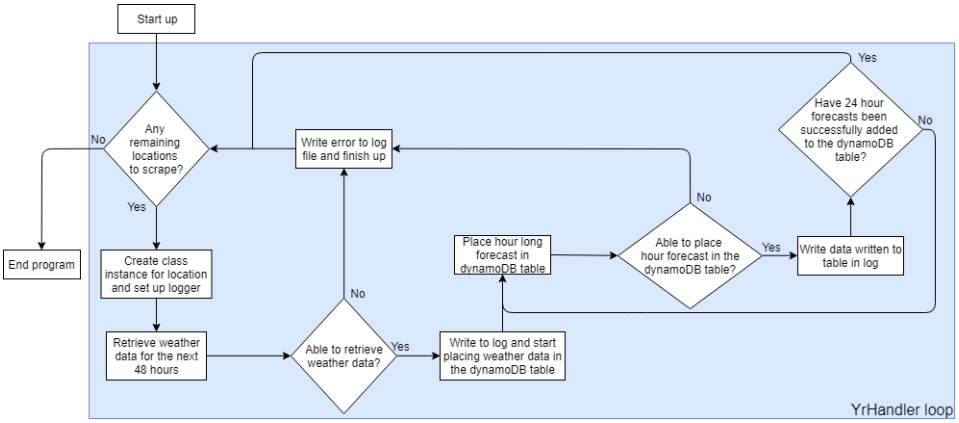


Figure 5.11: A flow diagram showing the overall functionality of the Yr scraper.

The program designed with modularity in mind. Additional locations can be added to the program by adding the full Yr path of the location, and the correct municipality ID. A two dimensional list could be used to traverse through a large number of locations and creating an YrHandler class instance and running it. There should be some form of delay between each iteration, since Yr limits how much data can be accessed during a short period of time by using their API[91].

The Yr Handler Class

The Yr handler class uses the functions provided by the open-source python library python-yr[46]. The data was collected by creating an Yr class instance by using the *Yr()* function, the hour long forecasts for the next 48 hours was collected by using the *forecast()* method. *Try* and *except* was used to handle any errors that may occur when these functions run.

Adding the Collected Data to the Database

By using the functions described in section 5.3.2 forecasts were added to a DynamoDB table. The individual hour forecast was added to the table by using the *put_item()* function of the table instance.

Forecast Content

The hour forecasts added to the DynamoDB table contain the majority of the information scraped by the program. Every hour long forecast contains the following weather data, in addition to the start and stop times of the forecast:

- Type of weather (sunny, cloudy etc.)
- The precipitation (in milimeters)
- The wind speed (in mps)
- The temperature (in celcius)

This series of data was chosen based on the study conducted in section 3.2. The forecast pressure was not added to the database as it was found to have no effect on the projected power production of solar PV.

The Weather Scraper

In addition to the main.py file, that runs every 24 hours, to scrape forecasted data. Another file was implemented, called *now.py*. It runs once every hour. This scraper is used to add the newest forecast data to another DynamoDB table, intended to be compared with the forecasted Yr data, and the power production. Instead of adding the next 24 hours to the database, it adds the next six, and updates existing values if they already exist in the DynamoDB table.

5.3.4 The MQTT Handler

The MQTT handler program has two main functions. The first is to subscribe to one or more topics on the provided MQTT broker and retrieve any valid messages. The second is to take the retrieved MQTT messages and place them into the appropriate DynamoDB table. To achieve this, threads were used from the Python library *threading*[38]. One thread for handling the incoming messages, another for placing them in the DynamoDB table. To handle the exchange of information between the threads safely, the python structure *queue* was used[39].

Software Architecture

See figure 5.12 for a flow diagram showing the overall functionality of the MQTT handler script.

Similarly to the Yr Scraper, the program can be expanded if needed. Additional brokers, or topics can be handled by creating multiple MQTT handler instances by using a for loop.

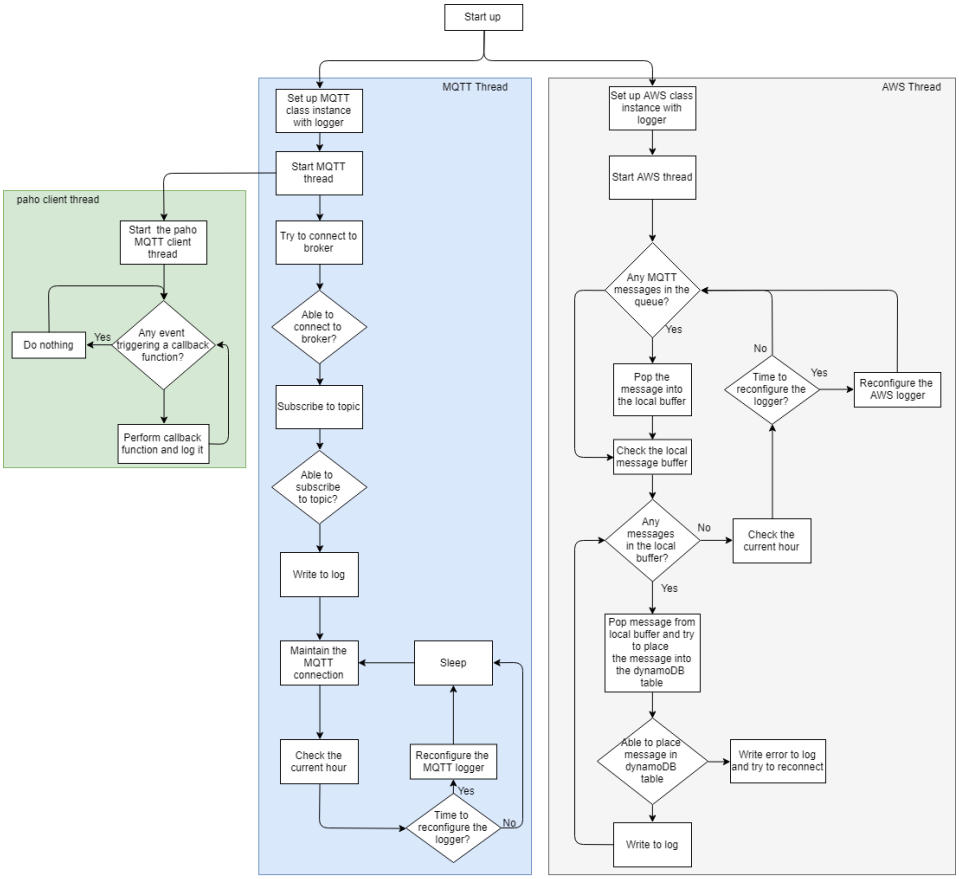


Figure 5.12: A flow diagram showing the overall functionality of the MQTT handler script.

The MQTT Handler Class

The MQTT uses the paho MQTT Python library[32] to handle the connection, subscription and handling of incoming MQTT messages. A series of callback functions was implemented in the *MQTT_Helper.py* file. This includes functions handling when the program successfully connects to the MQTT broker, or is disconnected. When the broker receives a message from the broker, the content of the message is checked. Since the access to the MQTT broker is open, there is a risk of the topic being used being spammed by outsiders. Due to the low data footprint of MQTT, it is not possible to screen out messages based on the client who sent it.

In the *MQTT_Handler.py* the *MQTT_Handler* class instance is implemented. This class is used to hold the client instance, that use the callback functions implemented in *MQTT_Helper.py*. The *MQTT_Handler* contains functions used for configuring the broker to be connected to, and for connecting to the particular broker. It also includes the main loop used when the MQTT thread is running. It mainly checks if the hour has changed. This is done to split the logging output in one file for each hour, in order to ease the debugging of the MQTT handler.

Similarly to the Yr Scraper, most of the functions use the *try* and *except* handling to prevent the program from crashing, and to capture any errors to the log.

The AWS Handler Class

The AWS thread starts after the AWS class instance is set up by the main loop. When running, the thread first checks if any messages has been placed in the queue structure. If yes, they are moved to a local buffer. Then the messages in the buffer are placed, one by one, into the appropriate DynamoDB table using the *table.put_item()*. When placing a message into a table, a time stamp is added, to identify the time the message was sent by the embedded system, and for further analysis. The result is then logged.

When the local buffer is empty, the thread checks the current time. If the hour has changed since the previous loop, the logger is reconfigured in the same manner as the logger for the MQTT thread.

5.3.5 The Graph Handler

The graph handler was implemented to generate plots from the collected by the embedded system and the Yr Scrapers, described in section 5.1 and 5.3.3 respectively.

Software Architecture

The Graph handler is split into four main modules. The main file, which performs the function calls to another module, the *GraphHandler* class, in the *Graph_Handler.py* file. The GraphHandler performs the retrieving of data from the DynamoDB tables, and computing. It also performs the function calls of the other two modules, the *PlotHandler* and the *SymbolHandler* class. The PlotHandler manages the plotting of the data computed by the GraphHandler, and stores them as images in an appropriate directory. The SymbolHandler manages some of the data not used by the PlotHandler and stores it in a text file located in the same directory as the plots. It also handles the zipping and uploading of the plot directory to the AWS service S3. The final module is the *RootClass*. It functions as a parent class of all the other functions.

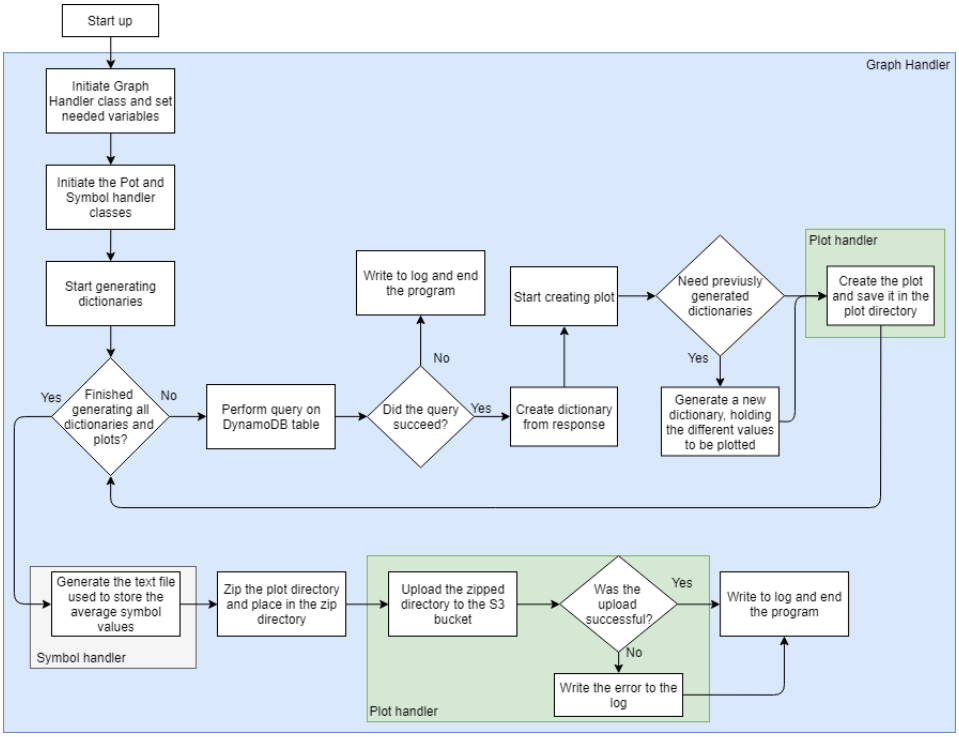


Figure 5.13: A flow diagram showing the overall functionality of the MQTT handler script.

The Graph Handler Class

The graph handler class is used as the main module of this script. It makes the function calls to the other two implemented classes and contain the majority of the code in this script. The *Graph_Handler* class handles the retrieving of data stored in three different DynamoDB tables and computation. It also performs the function calls necessary to generate multiple plots used to graphically represent the collected data, store it in multiple files and directories and places them in the appropriate Amazon S3 bucket.

It uses the boto3 library differently than the other scripts. Instead of placing data in the DynamoDB tables, it uses the *query()* function of the table instance to retrieve the intended table content. After it has received the DynamoDB object returned from the function, a function proved by the *DecimalEncoder* class was used to convert it to a JSON text object. The JSON was then converted to a dictionary using the *dumps()* function made available by the *json* Python library[69].

The graph handler class creates multiple dictionaries this way. This includes a dictionary containing the Yr forecasts for the previous day, the actual weather and a dictionary containing the average hourly power production measured for the previous day. It also generates 24 dictionaries, each containing all of the power measurements obtained during one hour. All of these dictionaries are used for at least one plot.

The Plot Handler Class

The Plot Handler class was implemented to handle the plotting of figures and the storage of them. It uses the open-source *mathplotlib.pyplot* library[60] to generate plots from the data retrieved from the DynamoDB tables.

The plots are generated by two functions. The first, *plotSimpleLine()* function generated plots containing only one line. It uses two arrays, one representing the values intended for the X axis and the other represents the Y axis. When plotting multiple lines in the same figure, the *plotMultipleLines()* function is used. It takes as parameter an array containing the values intended for the X axis, usually time stamps. For the Y axis, a dictionary is used. Where the keys are the names of the lines to be plotted, and the values are a list of the Y axis values for that particular line. Both functions then saves the plot as a PNG file by using the *savePlot()* function.

The plot handler class also contains the functions used to handle the different files created by this script. This includes uploading and downloading files from a chosen Amazon S3 bucket and zipping the content of a chosen directory.

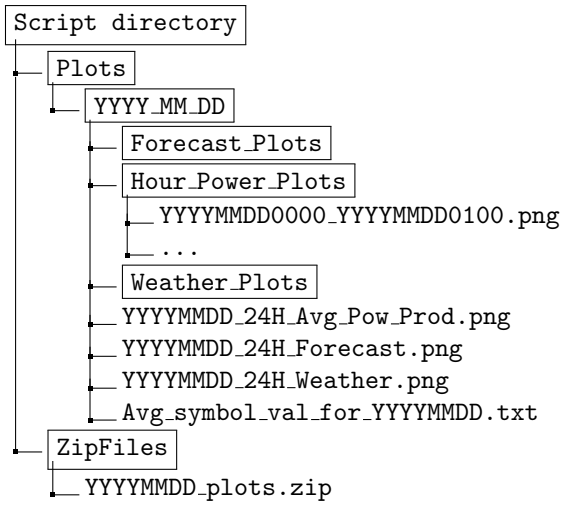


Figure 5.14: This figure illustrates how the different plot and zip directories are set up. YYYY denotes the year, MM the month $(01 - 12)$, DD the day $(01 - 31)$ and HH the hour $(01 - 23)$.

The Symbol Handler Class

Since the plot handler only handles the weather phenomena that can be represented numerically it does not handle the type of weather described by the *symbol* variable in each of the forecasts. A smaller class, called *SymbolHandler* was therefore implemented to calculate the average power production for the different types of weather during a day. It does this by using a dictionary where the keys are the different types of weather occurring during the day, and a list of the average power production for each hour during the different weather types. It returns a dictionary where the keys are the different symbols and the values are the average power production. The content of this dictionary is then written to a .txt file and placed in the same directory as all the plots.

It also assists the plot handler in generating plots for visualizing the change in weather type during the course of the day. It does this by using a simple dictionary was implemented as a global variable in the *Symbol_Handler.py* file. The keys of the dictionary were the name of the different types of weather in during the test period, while the values denoted an integer value. Where the highest value were given to the best kind of weather, 'Clear Sky'. This was done to show the correlation between the type of weather and the power production. The content of the dictionary is shown in table 5.7

Weather Type	Integer value
Clear sky	5
Fair	4
Partly cloudy	3
Cloudy	2
Light rain showers	1

Table 5.7: The dictionary used to convert the different weather types to integer values. Only the weather types present during the testing period were added to this dictionary.

The Root Class

The RootHandler class contains functions used by all the other classes in this script. This includes the setup and configuring of the class logger. It also contain a function for logging the content for a dictionary, this was primarily used for debugging during implementation.

5.3.6 Running the Scripts

This subsection will outline how the Python scripts described above were run on the EC2 instance.

The Cron Table

In Unix-based systems, including Ubuntu, programs and commands can be periodically run using the cron software. The YrScraper script, for example, runs every day at 00:30. The command to be run by the cronjob is written in a cron table file. The content of the cron table used for this thesis is shown in listing 5.1.

```
1  # Edit this file to introduce tasks to be run by cron.
2  #
3  # For example, you can run a backup of all your user accounts
4  # at 5 a.m every week with:
5  # 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
6  #
7  # m h dom mon dow command
8  30 0 * * * python3 /home/ubuntu/YrHandler/main.py
9  10 * * * * python3 /home/ubuntu/YrHandler/now.py
10 5 0 * * * python3 /home/ubuntu/Graph_Handler/main.py
```

Listing 5.1: The content of the cron table of the EC2 instance.

The Systemd Service

Unlike the other Python scripts, the MQTT handler runs continuously in the background on the EC2 instance. To accomplish this, the *Systemd* service provided by Ubuntu was used[81]. The Systemd service allows users to write .service files. These files allows certain scripts to run in the background when the system boots up. Since the MQTT handler runs continuously, the script will also run continuously in the background as long as the computer is on. See listing 5.2 to see the script used to run the MQTT handler.

```
1  [Unit]
2  Description=My Script Service
3  After=multi-user.target
4
5  [Service]
6  User=ubuntu
7  Type=idle
8  ExecStart=/usr/bin/python3 /home/ubuntu/MQTT_Handler/main.py
9
10 [Install]
11 WantedBy=multi-user.target
```

Listing 5.2: The *mqtt_handler.service* file content

5.3.7 Debugging the Python Scrips

Since the software is to be run periodically, or continuously, on a virtual Linux machine, being able to easily confirm that the programs worked successfully is essential. For this purpose, the standard logging library was used. The log files are placed in a directory with the following format shown in figure 5.15.

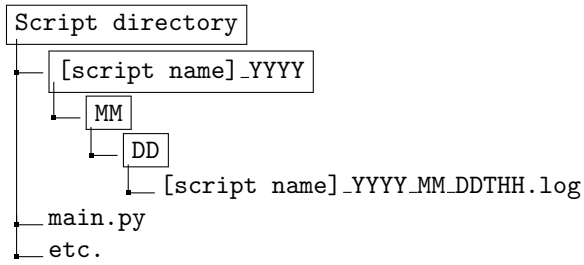


Figure 5.15: This figure illustrate the different logging directories were set up.

The logging directories were organized in this format to make it easier to locate logs for different periods of time. By identifying the different log files by the date

and hour, their individual size was reduced, thereby making debugging easier. This helped significantly when debugging the MQTT Handler. Since it handles and logs messages approximately once a minute, having 24 files with 60 log entries is far easier to handle than one file with 1440 log entries.

The Python programs uses *try* and *except* to handle both exceptions in the different operations of the program. If an operation succeeds, it is added to the log, possibly with additional information. For example if something is added successfully to a DynamoDB table, everything that was added to the table is written to the log file to clarify and make debugging easier. In case of failure, a message is written to the log file. Additional information about the specific failure is also added by using the *traceback* Python library[70]. This library gives access to the error message that normally would be displayed in the terminal, so that it can be placed in the log file.

```
2018-06-03 14:48:21,856:ERROR:Graph_Handler:Unable to generate the forecast graph.
Error code: Traceback (most recent call last):
  File "C:\Users\haakoneh\Desktop\AWS_VM_Script\Graph_Handler\Graph_Handler.py", line
      ↪ 341, in generateForeCastGraph
    self.plotHandler.plotMultipleLines(self.forecastDateTimeList,self.foreCastGraphDict,x_label,
        ↪ y_label, file_name, fig_name,True)
  File "C:\Users\haakoneh\Desktop\AWS_VM_Script\Graph_Handler \Plot_Handler.py", line
      ↪ 196, in plotMultipleLines
    self.savePlot(fig, file_name)
  File "C:\Users\haakoneh\Desktop\AWS_VM_Script\Graph_Handler\Plot_Handler.py", line
      ↪ 89, in savePlot
    self.createDirectory('{}/{}'.format(self.plotDir))
IndexError: tuple index out of range
```

Listing 5.3: An example of an error written to a log file using the logging and traceback library.

5.4 Tests

This section describes the tests used to confirm the different functionalities of the system.

5.4.1 The Embedded System

The functionality of the different parts of the embedded systems was verified by performing tests varying in the number of steps and size, comparable to the functionality being tested.

The Sensor Test

The HO-P sensor was tested by using a lab current source to check the read ADC value produced by the sensor. The sensor was tested using multiple different currents and voltages. Then a linear regression was conducted to check the linearity of the sensor output. As a result of the relatively low current output of the current source of $1.5A$, compared to the sensor's maximum reading of $25A$, the output cable was run through the sensor opening three times. Effectively tripling the strength of the magnetic field and the sensor reading. This increased the accuracy of the sensor test by widening the range of the current used for the test.

The UART Test

The UART functionality of the system was confirmed by connecting the LCPXpresso 4367 to a computer over UART using a TTL-USB converter and using the RealTerm serial terminal software[72] to send and view received data on the connection. The LPC-Link 2 debugger was used to check the content of the RX and TX ring buffers on the LCPXpresso 4367 to confirm the functionality on the embedded side of the connection.

The Modem Test

The functionality of the 3G modem was confirmed by sending multiple TCP messages to an echo server recommended by the modem's user guide[30, p. 18-19]. The sending and receiving of the TCP messages were confirmed by checking the socket buffers, and the content of an array used to receive the echoed data.

Another test of trying to connect the modem to an open MQTT broker was also performed. The broker URL and port number is shown in table 5.8.

Broker name (URL)	IPv4 Addr	Port number
eclipse.iot	198.41.30.241	1883

Table 5.8: The URL, IPv4 address and the port number of the MQTT broker used for testing the MQTT handler. The IPv4 address was needed because the modem requires it to send TCP messages.

The ESP8266 Test

To confirm that the ESP8266 module was able to connect to the correct MQTT broker and publish to the chosen topic, the MQTT client software MQTT.fx was used[27]. It displays messages sent to a broker on a specific topic, thereby confirming if the messages sent to the ESP8266 using the API over UART discussed in section 5.1.6 were transmitted to the correct topic. MQTT.fx was used to confirm that the messages were sent correctly, and the LPC-Link 2 was used to confirm the functionality of the LCPXpresso 4367. The ESP8266 test used the same broker as the modem test.

The Embedded System Test

The finished system was tested by combining the sensor and ESP8266 tests. The HO-P measured the current produced by the source, and the average minute value was transmitted to the ESP8266using the implemented API. Finally, the ESP8266 transmitted the measured value to the MQTT broker.

It was also tested by connecting the HO-P sensor to the cable output of a solar PV array mounted on the wall outside of the office used by the author of this thesis during the semester. This was used to get data on power produced by a solar array.

5.4.2 The Cloud Scripts

The functionality of the Python scripts run on the EC2 instance was confirmed by checking the log files after they were run manually on both a local computer and on the EC2 instance by using the PuTTY terminal. The logs were also checked periodically when they were deployed using the services discussed in section 5.3.6.

The Yr Scraper

The Yr scraper was tested by letting the EC2 instance run for a couple of days and then checking the log files to see the program functioned properly. The log messages and their time stamp were checked to confirm its functionality.

The MQTT Handler

The functionality of the MQTT Handler was confirmed by connecting it to an open broker, subscribing to a test subject, sending a test message from another client to the subject. Retrieving it, and placing the message in the appropriate DynamoDB table. The broker shown in table 5.8 was also used for this test.

The Graph Handler

The testing of the graph handler was done by using data collected by the embedded system and the Yr scraper after they had been running for at least the previous 24 hours. The functionality was confirmed by checking the various log files generated by the different classes, looking at the individual plots and checking the target Amazon S3 bucket to confirm that the correct zip file had been uploaded.

5.4.3 The System Test

The entire system implementation was tested over a period of three days. During this time interval all Python scripts were run using the services described in section 5.3.6 and the embedded system was connected to the output cable discussed above.

Initially the MQTT broker used for this test was the same as the one shown in table 5.8. However, due to the server being unavailable, the broker was changed during the second day of the test to the one shown in table 5.9. See section 7.1.4 for a discussion about this change and its implications for the system as a whole.

Broker name (URL)	Port number
m23.cloudmqtt.com	1883

Table 5.9: The URL and the port number of the second MQTT broker used for the system test.

Chapter 6

Results

This chapter of the thesis features the results of the tests discussed in section 5.4 of the previous chapter.

6.1 The Embedded System Test Results

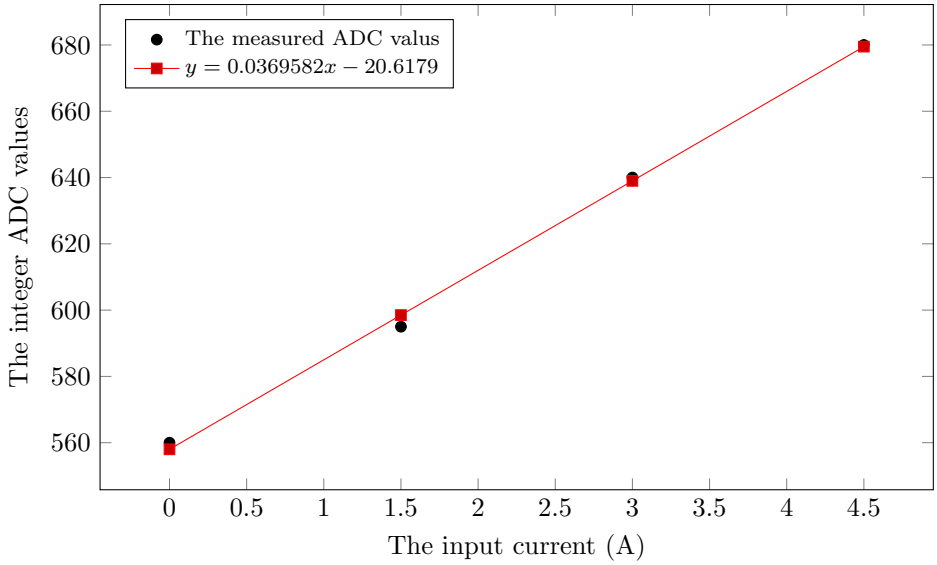
6.1.1 The Sensor Test

$I_{in}(A)$	$U_{in}(V)$	$U_{out}(V)$	**ADC(Int)**
0	0	1.72	560
1.5	2	1.76	595
3	3	1.79	640
4.5	5	1.82	680

Table 6.1: The measured output voltages and integer ADC values during the sensor test.

Using the Wolfram Alpha web service, the following equation was calculated. See figure A.4 in appendix A for a screenshot of the calculations.

$$y = 0.0369582x - 20.6179 \tag{6.1}$$



6.1.2 The UART Test

The UART test was performed by connecting the USART0 of the LCPXpresso 4367 board to a TTL-USB converter. A small test task was then run, where it sends a UART message, then waits 5 seconds before trying to read 10 bytes from the RX buffer. See figure 6.3 to see message sent by the LCPXpresso 4367, and figure 6.2 to see the received message stored in the array.

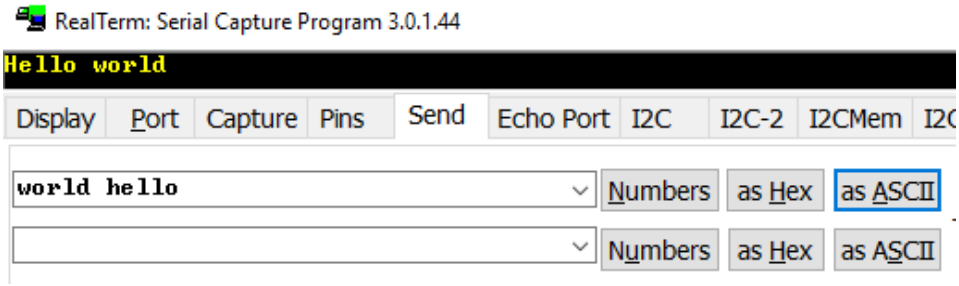


Figure 6.1: A screenshot of the received UART message and the message being sent using the RealTerm software.

ucTestSendArray	uint8_t [11]	0x10000f80
ucTestRecvArray	uint8_t [11]	0x10000f74
ucTestRecvArray[0]	uint8_t	119 'w'
ucTestRecvArray[1]	uint8_t	111 'o'
ucTestRecvArray[2]	uint8_t	114 'r'
ucTestRecvArray[3]	uint8_t	108 'l'
ucTestRecvArray[4]	uint8_t	100 'd'
ucTestRecvArray[5]	uint8_t	32 ' '
ucTestRecvArray[6]	uint8_t	104 'h'
ucTestRecvArray[7]	uint8_t	101 'e'
ucTestRecvArray[8]	uint8_t	108 'l'
ucTestRecvArray[9]	uint8_t	108 'l'
ucTestRecvArray[10]	uint8_t	111 'o'

Figure 6.2: A screenshot of the local variables in the UART test. The *ucTestRecvArray* contains the received UART message.

6.1.3 The Modem Test

This section contains the three different types of tests used to confirm the functionality of the 3G Digi Modem.

Echo Server Test

To test the modem's capability for sending and receiving simple TCP messages a test where a server that echoes the TCP messages it receives was used. A TCP message with the text *echotest* was sent to the server. See figure 6.3 to see the echoed TCP message.

Variable	Type	Value
ucRecvPayload	uint8_t [8]	0x10000000 <ucRecvPayload>
ucRecvPayload[0]	uint8_t	101 'e'
ucRecvPayload[1]	uint8_t	99 'c'
ucRecvPayload[2]	uint8_t	104 'h'
ucRecvPayload[3]	uint8_t	111 'o'
ucRecvPayload[4]	uint8_t	116 't'
ucRecvPayload[5]	uint8_t	101 'e'
ucRecvPayload[6]	uint8_t	115 's'
ucRecvPayload[7]	uint8_t	116 't'

Figure 6.3: A screenshot of the array used to hold the received echo message during the echo server test.

MQTT Test

To see if the modem was able to connect to a MQTT server without using the paho-MQTT library, a MQTT test was conducted. As figure 6.4 shows, it was not successful. The modem sent several status messages to indicate a loss, and the reestablishment of a connection to the cellular network. At another point in the test, the modem responded to TX IPv4 messages, with a resource error (see figure 6.5).

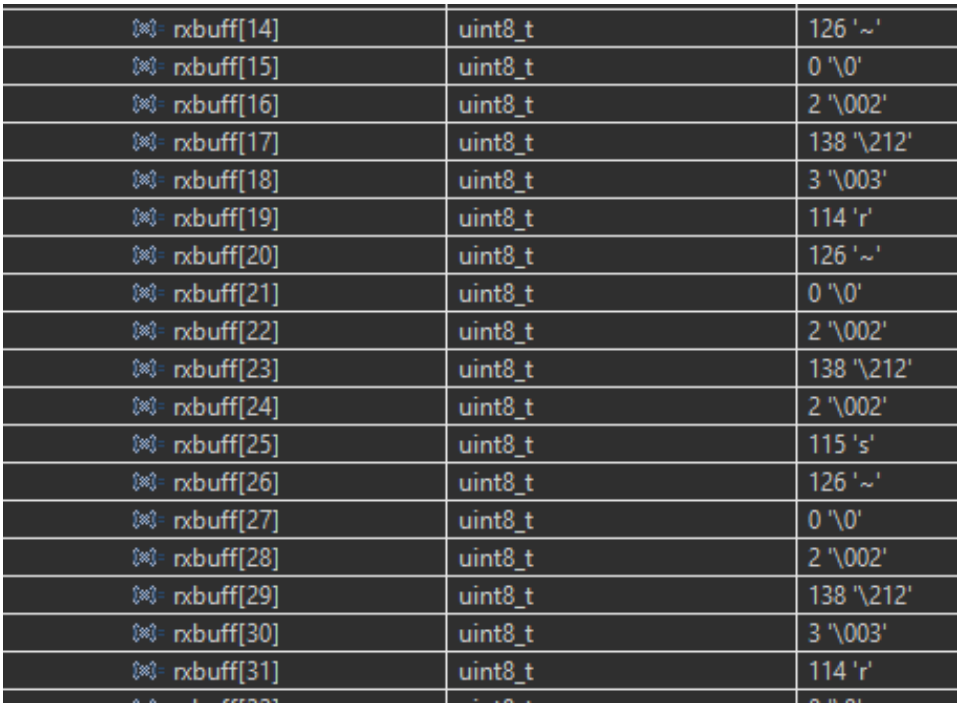


Figure 6.4: A screenshot of a part of the RX ring buffer during the MQTT test. The screenshot only includes modem status messages.

As seen in figure 6.4, the same API messages are received repeatedly. Since all these messages are modem statuses, see table 6.2 and 6.3, this indicates that certain conditions regarding the modem changes rapidly during the MQTT test.

Array index	**Int value**	**Hex value**	**API meaning**
14	126	$0x7E$	API start deliminator
15	0	$0x00$	First length byte
16	2	$0x02$	Second length byte
17	138	$0x8A$	Frame type: modem status[30, p. 119]
18	3	$0x03$	Status: Unregistered with cellular network
19	115	$0x73$	Message checksum

Table 6.2: The first parsed API message shown in figure 6.4

Array index	Int value	Hex value	API meaning
20	126	$0x7E$	API start deliminator
21	0	$0x00$	First length byte
22	2	$0x02$	Second length byte
23	138	$0x8A$	Frame type: modem status
24	2	$0x02$	Status: Registered with cellular network
25	114	$0x72$	Message checksum

Table 6.3: The second parsed API message shown in figure 6.4

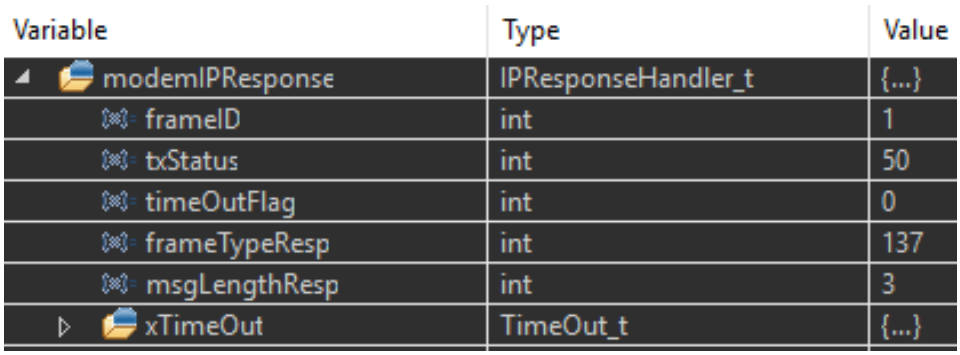


Figure 6.5: A screenshot of the modemIPResponse structure. The TX Status variable indicate a resource error[30, p. 118].

Paho-MQTT Test

A test to check if the modem would connect while using the paho-MQTT library was also conducted. As figure 6.6 shows, this test also experienced a resource error when trying to send TCP messages.

socket0_status_buff[0]	uint8_t	126 '~'
socket0_status_buff[1]	uint8_t	0 '\0'
socket0_status_buff[2]	uint8_t	3 '\003'
socket0_status_buff[3]	uint8_t	137 '\211'
socket0_status_buff[4]	uint8_t	1 '\001'
socket0_status_buff[5]	uint8_t	50 '2'
socket0_status_buff[6]	uint8_t	67 'C'
socket0_status_buff[7]	uint8_t	126 '~'
socket0_status_buff[8]	uint8_t	0 '\0'
socket0_status_buff[9]	uint8_t	3 '\003'
socket0_status_buff[10]	uint8_t	137 '\211'
socket0_status_buff[11]	uint8_t	1 '\001'
socket0_status_buff[12]	uint8_t	50 '2'
socket0_status_buff[13]	uint8_t	67 'C'

Figure 6.6: A screenshot of the status ring buffer for the TCP socket during the paho-MQTT test. All status messages indicate a resource error.

6.1.4 The ESP8266 Test

To test the functionality of the ESP8266 NoceMCU after it was used to replace the Digi 3G modem, a MQTT test was conducted to confirm its functionality. Figure 6.7 shows a MQTT message being successfully published to a topic on an open broker. Figure 6.8 shows the debug output from the ESP8266 when using the RealTerm software to transmit an API message during the test.

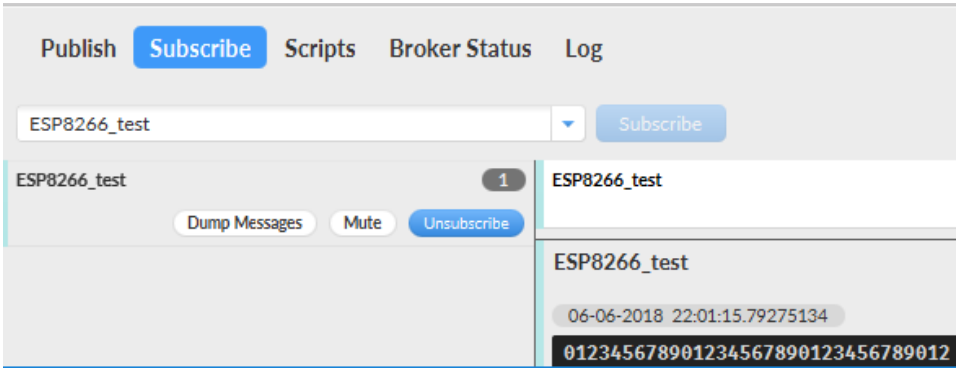


Figure 6.7: A screenshot of the MQTT.fx software used to subscribe to a open MQTT broker.

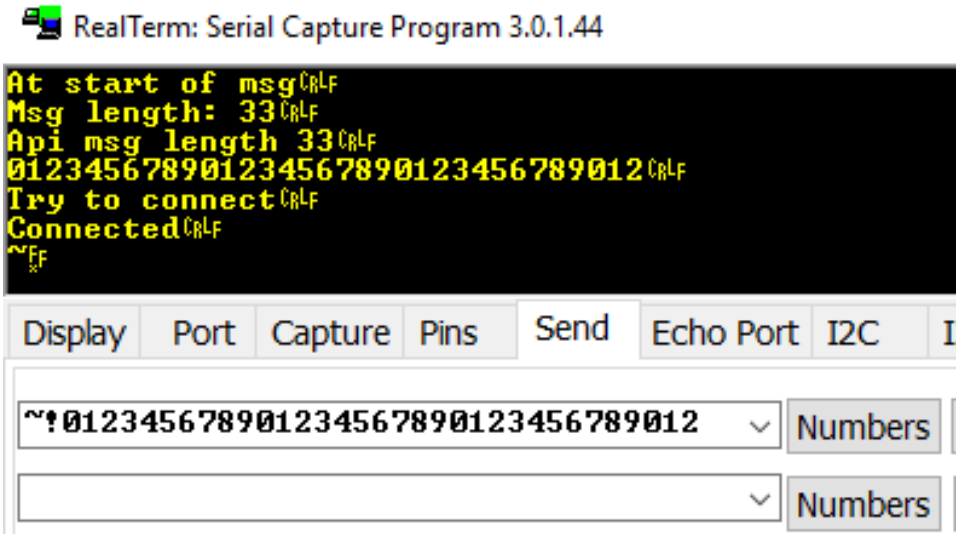


Figure 6.8: A screenshot of the RealTerm software used to send messages to the ESP8266 during testing. The ESP8266 was set to print out debug messages for clarity. The last line of is the confirmation message ($0x7E$ $0xFF$).

6.1.5 The Embedded System Test

To confirm the functionality of the embedded system using the ESP8266, an system test was conducted. Figure 6.9 shows a part of the buffer used to hold the sensor values measured using the ADC and the ampere value calculated using the equation found using linear regression. Figure 6.10 shows the ampere value being published

to a topic on an open MQTT broker.

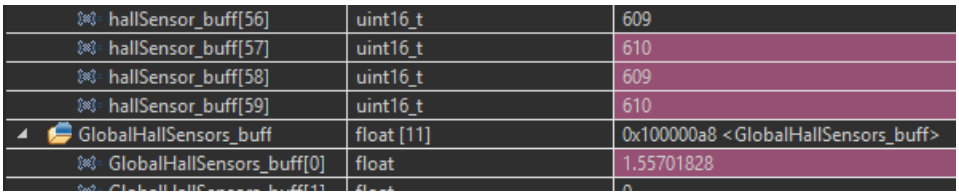


Figure 6.9: A screenshot of the content of the buffer

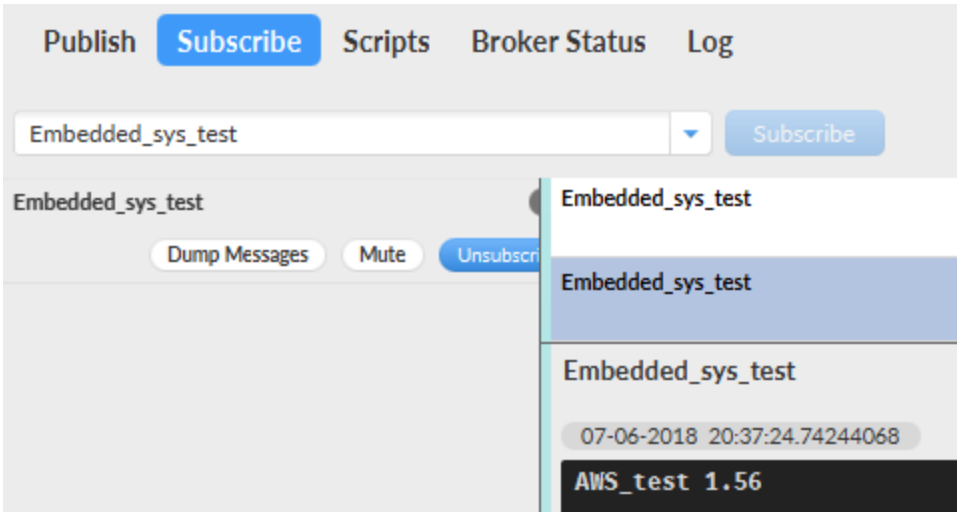


Figure 6.10: A screenshot of the MQTT.fx software used to confirm that the MQTT messages were sent correctly to the broker.

6.2 The Cloud Scripts Test Results

To confirm the functionality of the different Python scripts run on the EC2 instance, their logging output was read. See the the listings shown in section 6.2.1, 6.2.2 and 6.2.3 for the logging output.

6.2.1 The Yr Scraper Test

```
1  2018-06-03 00:30:01,837:INFO:yrhandler:Successfully accessed Norge/Trndelag/Trondheim/
       ↪ Trondheim
2  2018-06-03 00:30:01,837:INFO:yrhandler:Successfully retrieved the next 48 hrs
3  2018-06-03 00:30:01,909:INFO:yrhandler:Successfully added to db:
4  Norge/Trndelag/Trondheim/Trondheim
5  20180603010000
6  20180603020000
7  Clear sky
8  1.7
9  0
10 13
```

Listing 6.1: A small snippet of the log output from the Yr scraper during testing. Line 5 denotes the start time of the forecast, 6 the stop time, 7 the type of weather, 8 the wind speed, 9 the precipitation and 10 the temperature.

6.2.2 The MQTT Handler Test

```
1  2018-05-27 16:01:31,830:INFO:MQTT_helper:Connected to broker
2  2018-05-27 16:04:39,993:INFO:MQTT_helper:Added to queue from user: None
3  On topic: outFreeTopic
4  Received message:
5  test 123425
```

Listing 6.2: A small snippet of the log output from the MQTT callback functions during testing.

```
2018-05-27 16:04:43,456:INFO:aws_Handler:Added message from queue to local buffer:
test 123425
2018-05-27 16:04:44,048:INFO:aws_Handler:Successfully added to db
```

Listing 6.3: A small snippet of the log output from the AWS handler during testing.

6.2.3 The Graph Handler Test

```
2018-06-03 21:06:21,499:INFO:Graph_Handler:Key: Clear sky, value: 3.75375
2018-06-03 21:06:21,502:INFO:Graph_Handler:Successfully generates symbol files
2018-06-03 21:06:21,739:INFO:Graph_Handler:Successfully generated the forecast graph
2018-06-03 21:06:21,993:INFO:Graph_Handler:Successfully generated the weather graph
2018-06-03 21:06:28,141:INFO:Graph_Handler:Successfully generated a pow graph
2018-06-03 21:06:29,306:INFO:Graph_Handler:Successfully generated the power and forecast
    ↪ graph
2018-06-03 21:06:30,491:INFO:Graph_Handler:Successfully generated the power and forecast
    ↪ graph
2018-06-03 21:06:32,016:INFO:Graph_Handler:Zipped and uploaded plots for 20180603
2018-06-03 21:06:32,017:INFO:Graph_Handler:Successfully generated all graphs and uploaded
```

Listing 6.4: A small snippet of the log output from the Graph_Handler

```
2018-06-03 21:06:30,025:INFO:Plot_Handler:Generated multiple plots for Weather_Plots
    ↪ /20180603_24H_Pow_Weather_Curr_prod
2018-06-03 21:06:30,250:INFO:Plot_Handler:Generated multiple plots for Weather_Plots
    ↪ /20180603_24H_Pow_Weather_precipitation
2018-06-03 21:06:30,491:INFO:Plot_Handler:Generated multiple plots for Weather_Plots
    ↪ /20180603_24H_Pow_Weather_wind_speed
2018-06-03 21:06:30,570:INFO:Plot_Handler:Zipped folder: ZipFiles/20180603_plots from
    ↪ directory: ZipFiles
2018-06-03 21:06:32,016:INFO:Plot_Handler:Uploaded file: ZipFiles/20180603_plots.zip with
    ↪ key 20180606_plots from ZipFiles/20180606_plots.zip to bucket: haakonehmasterbucket
```

Listing 6.5: A small snippet of the log output from the Plot_Handler class

```
2018−06−03 21:06:18,353:INFO:Symbol_Handler:Symbol_Handler successfully set up
2018−06−03 21:06:21,499:INFO:Symbol_Handler:Successfully calculated the symbol average.
2018−06−03 21:06:21,502:INFO:Symbol_Handler:Successfully wrote dict to file: Plots/2018
    ↪ _06_03/Avg_symbol_val_for_20180603.
```

Listing 6.6: The log output from the symbol handler during testing.

6.3 The System Test Results

To confirm the functionality of the embedded system, cloud service set up and the Python scripts a system wide test was conducted of the span over three days. Listing 6.7 and 6.8 shows a snippet of the logging output of the MQTT and AWS thread, placing one of the many MQTT message being sent from the embedded system during the three days. Figure 6.11 show the plot of one hour of the second day, with a one minute resolution.

```
2018−06−05 10:00:58,510:INFO:aws_Handler:Added message from queue to local buffer:
AWS_test 6.21
2018−06−05 10:00:58,520:INFO:aws_Handler:Successfully added to db:
msg_content: 6.21
```

Listing 6.7: A small snippet of the log output from one of the log files generated by the AWS thread in the MQTT handler during the system-wide test.

```
2018−06−05 10:00:58,510:INFO:aws_Handler:Added message from queue to local buffer:
AWS_test 6.21
2018−06−05 10:00:58,520:INFO:aws_Handler:Successfully added to db:
msg_content: 6.21
```

Listing 6.8: A small snippet of the log output from one of the log files generated by the MQTT thread in the MQTT handler during the system-wide test.

Figure 6.14 shows the average power production for every hour and the forecasted temperature, wind speed and precipitation. Figure 6.15 shows the same, except with the actual weather, not the forecast. Figure 6.16 shows the content of the target S3 bucket after the system test was completed.

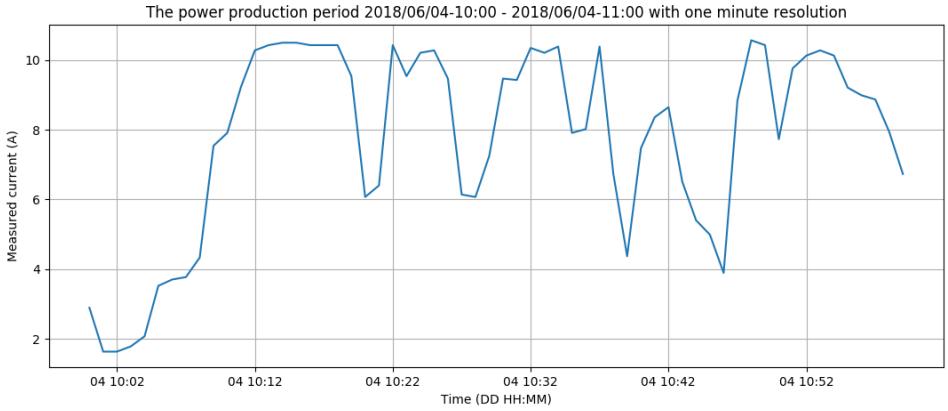


Figure 6.11: One of the 24 plots, from 10:00 to 11:00, generated during the second day of the system-wide test. The plot corresponds to the log snippets in listing 6.7 and 6.8.

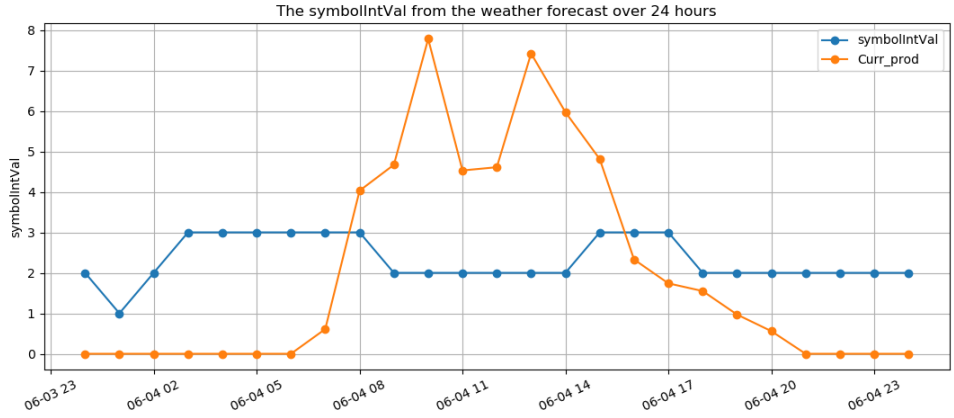


Figure 6.12: The plot shows both the power production and the forecasted weather symbol values. The power production is the average value per hour, with a one hour resolution. Period 06-04 10 to 06-04 11 corresponds to the average value of the plots in figure 6.11.

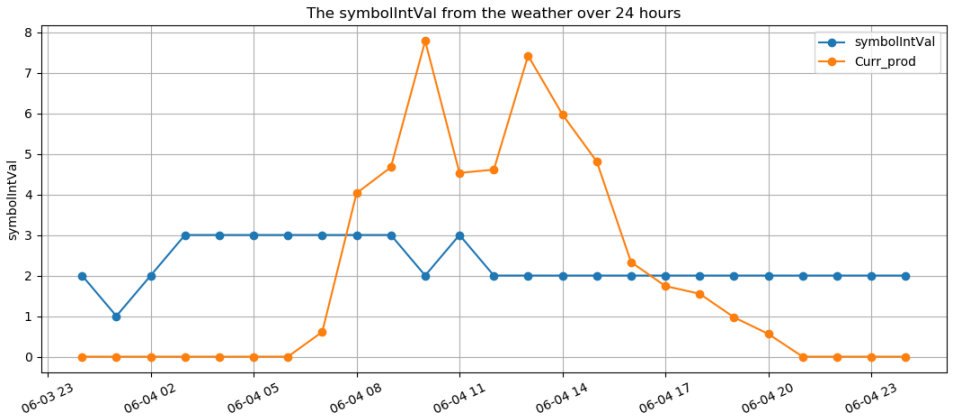


Figure 6.13: The plot shows both the power production and the actual weather symbol values. The power production is the average value per hour, with a one hour resolution. Period 06-04 10 to 06-04 11 corresponds to the average value of the plots in figure 6.11.

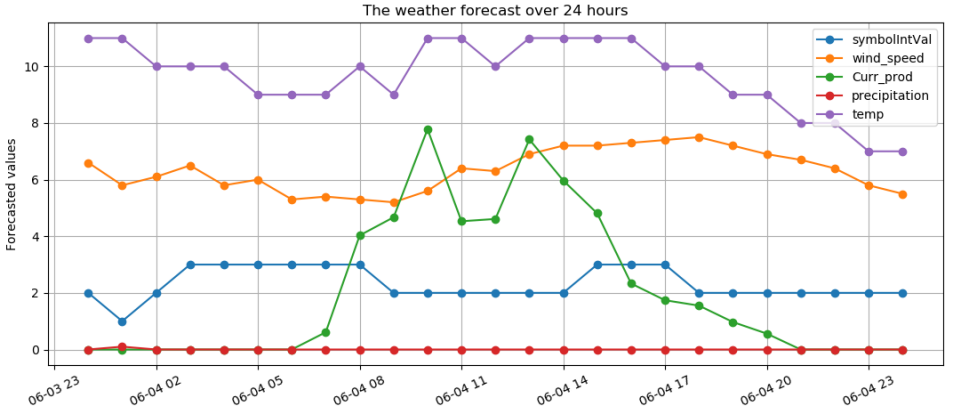


Figure 6.14: The plot shows the power production and all of the forecasted weather values. The power production is the average value per hour, with a one hour resolution. Period 06-04 10 to 06-04 11 corresponds to the average value of the plots in figure 6.11.

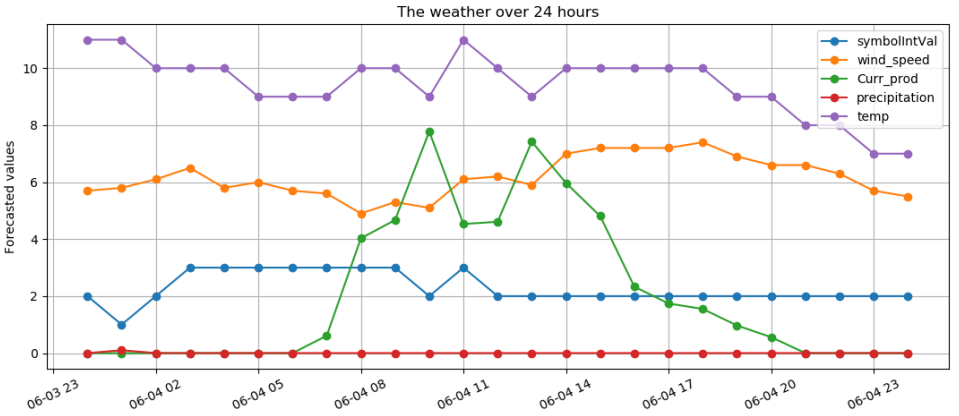


Figure 6.15: The plot shows the power production and the actual weather values. The power production is the average value per hour, with a one hour resolution. Period 06-04 10 to 06-04 11 corresponds to the average value of the plots in figure 6.11.

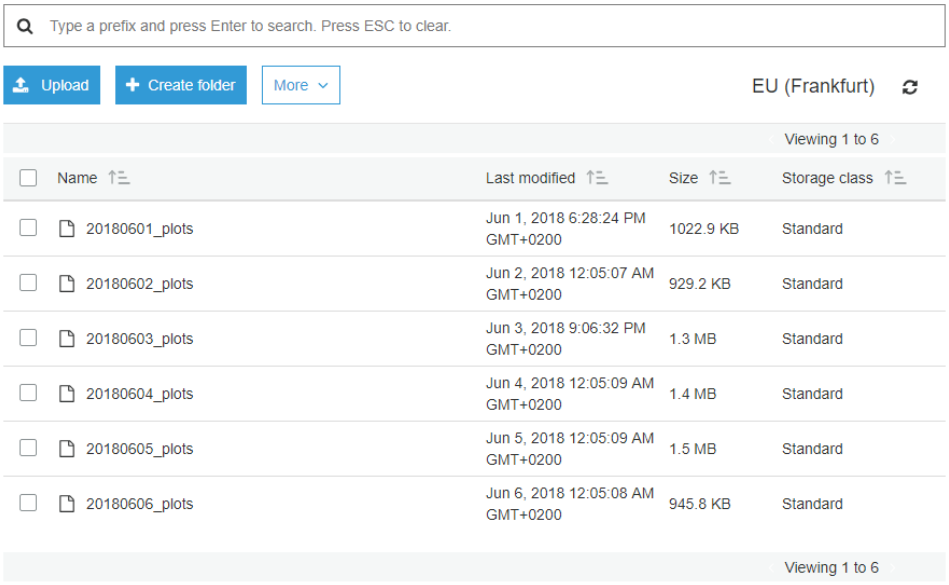


Figure 6.16: The content of the Amazon S3 bucket after the three day long test.

Chapter 7

Discussion

This chapter will discuss the results presented in the previous chapter, compare them to the specifications presented in chapter 4, as well as the limitations of the tests conducted for this thesis. Furthermore, the chapter will discuss the transition from a cellular-based, to a Wi-Fi-based solution, and the background for this choice. The chapter will also discuss some challenges that were encountered during the implementation process and a longer discussion of some of the challenges and limitations of the current cloud computing implementation. Finally, further improvements for both the embedded system and cloud computing setup will be discussed.

7.1 The Tests

7.1.1 The Embedded System Tests

The Sensor Test

The sensor test indicates that the HO-P functions properly and is indeed linear with an accuracy of over 99%, as specified by the sensor's datasheet. Thereby confirming PR-2.1 by using an analog input, and PR-1.1.2.

The UART Test

Similarly to the sensor test, the UART test seems to confirm that the system is able to send and receive messages while using the LPC-Link 2 debugger to free up the hardware UART.

The Modem Test

As section 6.1.3 shows, the Digi cellular modem is able to send simple TCP messages to an echo server, and handle the response, but not more complex tasks. It was therefore able to confirm PR-1.1.1, but only for the echo server. When trying to connect to a MQTT broker, the modem seems to experience multiple status changes. It loses the connection and then reconnects to the cellular network repeatedly. Similarly, when using the paho library, the modem experiences a resource error. This normally occurs when the modem is not able to open more sockets. Which seems strange, since the modem was restarted before conducting the test, and therefore should not have any sockets open, except for the one created during the test.

The ESP8266 Test

In contrast to the modem test, the ESP8266 test confirmed the network functionality of the embedded system. In accordance with PR-1.1.1 and PR-1.1.2, but not PR-1.1, as it uses Wi-Fi, not 2G/4G. The test was not run for more than a couple of minutes, and does therefore did not confirm that the system is able to stay connected to a Wi-Fi network for long periods of time.

The Embedded System Test

The embedded system-wide test indicates that it is able to function as intended. It is able to read the ADC value from the HO-P sensor, translate the average value to a corresponding ampere value and transmit it to a remote server. Similarly to the ESP8266 test, it did not run for more than ten minutes, and did therefore not guarantee long term operational integrity.

7.1.2 The Cloud Script Tests

The multiple logging snippets used during debugging the different Python scripts, do seem to all confirm the functionality of the embedded system. The author of this thesis has not read through all of the different log files produced during these tests, but snippets of them, as well as using the AWS management console. This is one of the weaknesses of this approach, when a significant amount of logging output is produced, and if not automated, the process of checking the log files can be quite time-consuming.

7.1.3 The System-Wide Test

In contrast with the previous tests discussed in this chapter, this test was run for circa three days, a significantly longer time. As shown by the logging snippets, plots and screenshot, the system seems to be working as intended. Confirming

the medium-term operational integrity of the current system. As discussed in section 7.1.2, this test produces a significant amount of output, resulting in only some parts of it being analyzed.

7.1.4 Limitations of the Tests

The test results currently looks promising for the Wi-Fi solution. While the tests conducted during this thesis are more robust than those conducted for the TTK4550 thesis[47, p. 55], they still do not guarantee that the system is able to function in harsher environments than an office. The encapsulation of the hardware would have to provide a similar internal environment for the performed tests to confirm the system functionality.

There is also the problem of test reliability. During the second day of the system-wide test, the MQTT broker used became unavailable. Due to this error, the CloudMQTT broker replaced the Eclipse broker. Luckily, this error was discovered within 30 minutes after it occurred, and was fixed in less than five. This error did not reoccur, but still illustrates the limitations of this system-wide test, and the solution as a whole. Had the error not been discovered, almost two days of testing would have been wasted in trying to confirm the system's functionality. They would instead have illustrated critical vulnerabilities in the system far more potently than what occurred. Finding an alternative solution to the usage of an open MQTT broker is therefore necessary.

7.2 The Implementation Change

As section 6.1.3 and 7.1.1 shows, the final modem implementation did not function as intended and was therefore replaced with a simple Wi-Fi solution to complete the project. This is in contrast to the solution implemented for the TTK4550 Specialization Project[47], implemented during the fall of 2017. For that particular thesis, a functioning GSM solution using the Arduino SDK and multiple open-source libraries were used for the implementation. Particularly the *TinyGSM* library, facilitating the controlling of the GSM modem, streamlined the implementation process.

The main difference between these two implementations is the community around the different tools. The Arduino community has a vast amount of previous projects, libraries and support for developers. This leads to the fact that even modules that are not staples in Arduino projects, such as GSM modules, have libraries available. In contrast, the Dig 3G modem has no such open-source libraries. It neither has a significant amount of previous open-source projects available. The majority of the support community is located on the official Digi support forum[51]. This difference in the amount of previous work available has

obvious consequences for the individual developer using the Digi modem. A significant amount of time has to be spent implementing, debugging and testing a driver or library to use the modem for more sophisticated operations. It also requires a significant amount of time to read the documentation of the product, that might not always be as clear as it could be. Something that required a significant amount of time during the implementation process was to read up on and obtain an intuitive understanding of the Digi API framework. This process was made more difficult due to the lack of API message examples in the documentation or any form of graphical representation of the message structure. When implementing the parser functions for the TX IPv4 API messages, a significant amount of the time used reading the documentation was spent switching between IPv4 message structure (p. 116) and the standard frame structure (p. 111). An example of a IPv4 API message in the documentation, with some additional information would have improved the readability of the documentation.

As a result of the limited amount of available open-source projects and libraries a lot of the period used for implementing the project was spent trying to implement a modem driver capable of connecting, publishing and maintaining a connection to an open MQTT broker. The planned driver was intended to have little, to no error handling in the case of a loss of cellular connection, mainly due to timing constraints. This would have reduced the probability of the embedded system being able to pass the system-wide test. Particularly since a cellular network connection would most likely have been less reliable than the Wi-Fi network of NTNU.

7.3 Implementation Challenges and Difficulties

The author of this thesis had never used the FreeRTOS stack before this thesis project. As a result of this there were a number of challenges, difficulties and bugs that may not have been as prevalent as they were for the author of this thesis. One of these bugs was a hardware fault that plagued the project for a couple of weeks, see figure A.1 in appendix A for a screenshot of the hard fault handler used to debug this project. The problem bug occurred when the embedded system started to send bytes over the UART using its interrupt. At first, the problem was solved by switching variables that had been local to global. This solved the problem temporarily. The problem was finally solved by increasing the stack size of the individual task.

Based on the solution to this bug, the problem stemmed from the chosen stack size of an individual task being too small, and when the scheduler switched between one task to another, the program counter was located in an invalid memory location based on the defined size. As the appendix makes clear, the fault

was a memory fault, that escalated to a hard fault.

In retrospect, the solution to this bug could have been found earlier. One of the main features of the FreeRTOS is that every task has its individual stack, used to store variables, and the state of the task when switching to another. Had the author been more experienced in using and debugging FreeRTOS systems, this problem could have been solved earlier.

7.4 Cloud Computing

This section contains a discussion on some of the shortcomings and challenges related to the cloud computing solution implemented for this thesis.

7.4.1 Analysis and Estimation

The analytical aspect of this system is currently rather simple. It is able to plot multiple plots, to be later studied by humans, but does not use the collected data in any analytical manner to either draw conclusions or estimations. While this may be sufficient in the systems current state and usage, this approach will be inadequate if the number of embedded systems collecting data increases dramatically. Checking the plots of hundreds or thousands of embedded systems every day would be a daunting task for any individual, and trying to make estimations practically impossible. To solve this problem, more sophisticated techniques and tools should be utilized.

Data Collection

The current data input for the analysis and estimation is currently inadequate. The weather factors currently used to estimate the power production would not make the estimation tool as accurate as those discussed in section 3.2. To obtain more accurate weather data, a local weather station should be used. This would give data for a smaller area, and thereby giving the estimation tool more relevant parameters.

The estimation tools described in chapter 3 also used additional information regarding the solar PV installation and the global positioning of it. This included the angle of the solar PV cells to the ground and the normal levels of solar irradiation in the area. The angle of the PV cells to the incoming solar radiation can greatly affect the amount of power produced, as mentioned earlier, and is therefore rather relevant to the estimation tool and its accuracy. It would also allow for collecting weather data with a one minute resolution, to further improve the analysis. This would help identify the factors responsible for the rapid changes in power production, as shown in figure 6.11.

Information regarding the installation would also improve the estimation and analysis. By taking into account the angle of the solar panels, and the direction they are placed, would give greater insight into how weather attributes affect the power production. As seen in figure 6.14 the correlation between the type of weather and power production changes depending on the time during the day. The panels used for this thesis were placed on the south side of the D-block of the Elektro Building at Gløshaugen. Meaning that even if the weather is at its best during the early morning, the solar rays will hit the panels on such an angle that the production will be relatively low. This can be seen in figure 6.13, where the power production reaches its peak at around ten in the morning, despite the weather becoming poorer. This indicates that the production increases because of the solar radiation hitting the panels at a better angle, than what they did previously. Therefore, adding functionality that takes this factor into account would improve both the analysis and the estimation accuracy.

7.4.2 Security

Another factor is the security of the data supplied to the cloud computing platform. Currently, the data is supplied by using an open MQTT broker. This comes with a number of security risk, in addition to the problem of reliability, as discussed in section 7.1.4. One such risk is that a third party would spam the topic used to retrieve the measures sensor data with messages that the MQTT handler would interpret as authentic messages from the embedded system. Thereby flooding the database with junk data, and possibly overloading the Python application.

Another security issue is the fact that all the collected data is not encrypted. Meaning that any third party who know of the system, and the topic used could retrieve all of the data collected by the embedded system. While this is not a pressing issue for the system in its current state, as a project in early development and mainly used in an educational and experimental manner, it is a major concern if it is ever deployed. Particularly if it is used for any application that is essential for society. Some form of encryption should therefore be added to the communication between the embedded system and the cloud platform.

7.4.3 Scalability

Scalability is also an issue for the current cloud computing platform. While there are few limits to how much computing power can be utilized by applications and services hosted on the platform, it is a question of cost and efficiency. In its current version, the MQTT handler confirms the validity of an embedded system by checking the ID field of the message published to the chosen topic, and then checks it against a list if it is valid. This approach was easy and

took little time to implement, but it has some major limitations. If the number of embedded systems would increase dramatically, to for example 10.000, the application would demand vast amount of resources. Keeping a list of 10.000 items continuously in the EC2 instance memory and to check its content millions of times a day would require a large amount of computing power, and corresponding high costs. Both in money and in material resources, such as energy.

A possible solution is to use a DynamoDB table to hold the items and instead perform queries to check if the ID is valid. This comes with its own drawbacks. While an individual DynamoDB query does not require a lot of time, it is not instantaneous. It usually takes close to 0.05 seconds[1]. Meaning that if 10.000 queries were to be performed every minute. It would take approximately 8.28 minutes to check every ID, every minute. See the equation below for the calculations.

$$\frac{10.000 \cdot 0.05s}{60s/min} = \frac{500s}{60s/min} = 8.28min \tag{7.1}$$

To reduce the amount of time used to check all of the different IDs two methods can be used, both incurring financial costs. One is to use a more powerful EC2 instance with a faster internet connection than that of the t2.micro used for this thesis, which was defined as 'Low to Moderate'. Another is to use extensive multithreading, something that also might require a more powerful virtual machine. Additionally, there is the cost of performing 10.000 queries to a DynamoDBtable every minute. This is because, beyond a certain threshold, it costs extra to perform queries to DynamoDB tables. There is also the problem of checking for errors. The script would have to be modified to allow for better debugging. Checking the logging output of 10.000 devices would be practically impossible, if it is not automated and improved. This might include further splitting up the logging files, based on the different devices, and reducing the logging to only log if an error or bug occurs. This would reduce the output, but it might make it more difficult to detect edge cases which do not trigger the error logging.

Another problem is the case of spamming of the MQTT broker if the system consist of such a large number of devices. The system would be more vulnerable to flooding of the MQTT topics, and because of the large amount of valid messages, the handler could be overwhelmed. Particularly if the attacker uses valid device IDs in these spam messages. Such an attack would be difficult to detect, and maybe impossible to stop.

[1] The time interval was found by looking at a log snipped from the Graph Handler, see listing A.1 and A.2 in appendix A

7.4.4 User Interface

Currently, the plots produced by the graph handler can be accessed using a Python script on a machine that has an approved AWS certification key installed. This makes the produced data potentially available anywhere with an internet connection. However, it is still quite cumbersome to set up such a connection. It also requires a certain level of programming knowledge, a significant drawback. Therefore, a more user friendly method of interfacing with this data should be implemented.

7.5 Further Improvements

This section contains a detailed discussion on changes to, or additions to the current system to improve its functionality. Chapter 9 presents this section in short-form as a list, for a faster read through.

7.5.1 An API Library

For the embedded system to fully utilize the Digi 3G modem, a more robust API library should be implemented. The library could be a series of C files, similarly to the paho MQTT library. It needs to be able to handle the many possible errors that may occur when using the modem. This includes the loss of signal or the unexpected closing of a socket on the server side. Similarly, the library could be modularized similarly to the paho library, by letting a user specify the send and receive functions to be utilized by the library. This would allow the library to be ported to multiple embedded system, including those that may not use FreeRTOS. It would also reduce the workload of any engineer who might try to incorporate the 3G modem, or any other Digi product utilizing the same API.

7.5.2 Embedded Hardware Setup

The current hardware setup is mainly intended for implementation and testing. As such, the current embedded system is not fit for deployment. A possible solution is to design a circuit board, using among other components, a NXP LPC43xx CPU and a 20 pin connector for Digi XBee devices. Allowing the 3G modem to be connected to the circuit board, and possibly upgraded at a later date.

7.5.3 Remote User Interface

Some form of interface for remote controlling should also be implemented. This is for a user to be able to change how often the system reads its sensor values, how often it transmit those data to the cloud service etc. This interface could be implemented by using the device shadow functionality of AWS IoT.

7.5.4 Additional Embedded Features

To make the embedded system fit for field deployment more features should be added. This includes, among other things, a form of long term non-volatile storage medium, in order to fulfill PR-3.1. Both for the possibility of storing measured data for longer periods of time, and system logging. This could be accomplished by using a SD card module and a Real-Time clock (RTC) for retrieving timestamps. The LCPXpresso 4367 does allow for RTC functions, but it does not have a constant power supply, thereby resetting it whenever the system loses power. A module connected to a battery should therefore be used, similarly to what was implemented for the TTK4550 thesis for this author[47, p. 27].

Another feature is increased robustness. The current embedded system has few features regarding the handling of unforeseen errors, other than what is embedded in the FreeRTOS stack. Functionality to handle power loss of parts, or the entire system, for example, should be implemented. To meet PR-7.

7.5.5 Improved Cloud Computing

This subsection discusses some of the improvements that can be implemented to improve the cloud computing of the system.

Analysis and Estimation

If the system is to be able to estimate the power production for the next 24 hours, an application with artificial intelligence capabilities such as machine learning, similarly to the techniques discussed in section 3.2.2, like ANNs, should be implemented. Such an application also needs extra and more reliable data than what is currently available in the system. Currently, the weather data used in the system comes from what is available through the Yr API, which covers the whole Trondheim Area. A local weather station, capable of collecting the necessary information would be better. Some form of data quality control should also be added, as one study showed, it greatly improved the estimation quality[53].

Additional information regarding the solar PV location should also be used in the analysis and estimations. This includes, among other things, the angle of the solar PV cells. Which is of great importance of the solar radiation hitting the cells. And the positioning of the panels, which would help pinpoint when the solar rays hit the PV panels at an angle which produces the most power. Another is the normal solar radiation of the geographical area, which can help remove extraordinary production levels that might reduce the accuracy of the estimations. Third is a form of historical weather analysis. As the Mathisen note in section 3.2.1 mentions, snow on the solar PV cells prevents nearly all power production, for several days after it has fallen on the cells. Adding this factor would be crucial,

particularly for a country like Norway, where in certain locations, it might snow in May.

Security and Scalability

A possible solution to the issues and limitations discussed in section 7.4 is to utilize the Amazon FreeRTOS SDK. This would give access to such cloud services as AWS IoT, which is intended to be used for a larger amount of different embedded systems. It would additionally give access to a more secure cloud connection by utilizing the mbedtls cryptographic standard and handle the task of identifying the different systems by using the certificate functionality provided by Amazon FreeRTOS. While the necessary porting discussed in section 2.5.4 does require a significant amount of work, the non-hardware specific functionality can mostly be directly copied from existing code. This includes among others the *pkcs11* header and source files. Where the only functions that needs to be modified are those handling the saving and retrieving of files on the non-volatile memory. Similarly, it is mainly private send and receive functions in the secure sockets files that needs to be modified to the particular hardware setup. This is by no means an insignificant task, as the author of this thesis experienced during the time spent trying to understand the Amazon FreeRTOS structure, but a significant amount of existing code can be reused.

There are of course downsides to this approach. The most important one is possibly how difficult it makes migrating the system to another cloud service in the future. Amazon has, through its acquisition of FreeRTOS, strengthened its position in the marked for embedded systems solutions. Making it an attractive choice for those who wants to build systems needing embedded security and vast cloud computing resources. While a large team of developers may build their own SDK to avoid this issue, smaller teams may not have the time, or resources, available to do this. Amazon FreeRTOS gives small teams a lower entry to developing secure applications, but it will also lock them to AWS.

Web Interface

To make the plots, and any other data produced by the graph handler available for individuals with little to no programming experience, a web interface could be implemented. Both for visualizing the data on the interface and make the plots available for download. This can be implemented by utilizing the Python web framework Django[37], similarly to the interface Marit Tundal implemented for her TTK4550 thesis.

Chapter 8

Conclusion

Through a systematic approach the author of this thesis has designed and implemented an embedded system, capable of communicating with a cloud service using a Wi-Fi module and the cloud computing used to handle the collected data. The stated goal of using a 2G/4G modem was not accomplished due to a lack of time and no existing libraries or previous projects available to reduce the workload during the implementation period. The system is able to measure the current running through a cable by using a HO-P hall effect sensor. The system calculates the average power production over the course of a minute and then publishes it to an open MQTT broker. A continuously running Python script retrieves the published data and adds it to a DynamoDB table. The collected is then plotted with scraped Yr weather and forecast data, using periodically run Python scripts. The plots are then placed in a zipped directory and uploaded to a Amazon S3 bucket, for long term storage and easy retrieval.

The functionality of the system has been confirmed by multiple tests, but it is still not fit for field deployment. Better error handling on both the embedded and cloud service side of the system needs to be added. It also needs better security features in the form of cryptography, and scalability.

Chapter 9

Future Work

This chapter contains a list of the future improvements discussed in chapter 7. This list is intended for a fast read through. Therefore, the justifications and the approach is discussed more thoroughly in section 7.5. Every list item also includes a reference to the specific section they are discussed in long-form, to make reading this thesis easier.

9.1 Propositions for Future Work

- An API library to better control the Digi modem, see section 7.5.1.
- A printed circuit board for making the hardware ready for field deployment, see section 7.5.2
- A remote interface for controlling the embedded systems. Possibly implemented by using the Device Shadow service of AWS IoT, see section 7.5.3
- Additional features for increased robustness should be added to the embedded system. This include both a SD and RTC module for logging, and additional error handling, particularly with regards to power loss, see section 7.5.4
- Cloud computing improvements, see section 7.5.5
  - Improved analysis and estimation by using AI tools such as Artificial Neural Networks. These would be further improved by utilizing more data regarding the solar PV installation, the environment and weather measurement, accompanied by information quality control.
  - Better security and scalability by using the Amazon FreeRTOS SDK.

– A web user interface for easier data retrieval for third parties using the Django web framework.

Appendices

Appendix A

Additional Figures and Listings

A.1 Additional Figures

The Hard Fault

By using the Keil MDK tutorial document to debug the hard fault registers[18], and looking at the value of the CSFR register, that handles usage, bus and memory faults[18, p. 8] one can debug the hard fault described in section 7.3. The value 1024 in CFSR denotes an imprecise data bus error (IMPRECISERR, bit 10)[18, p. 10]. The HFSR denotes that the bus fault was escalated to a hard fault because it cannot be handled (FORCED, bit 30)[18, p. 7]. The bus fault (BFAR) status register shows that an instructtion bus error has occurred (IBUSERR, bit 8)[18, p. 8-9], and that the error is not related to the instruction that caused the error (IMPRECISERR, bit 10), and that is a floating point error (LSPERR, bit 13)

Status register	Int value	Binary value
CFSR	1024	00000000 00000000 00000100 00000000
HFSR	1073741824	01000000 00000000 00000000 00000000
BFAR	3758157112	11100000 00000000 11101101 00111000

Table A.1: The three most relevant status registers shown in figure A.1 for additional clarity.

The Modem Bugs

Name	Type	Value
hardfault_args	unsigned long *	0x10000b28
stacked_r0	volatile unsigned long	268438500
stacked_r1	volatile unsigned long	0
stacked_r2	volatile unsigned long	268440220
stacked_r3	volatile unsigned long	268440220
stacked_r12	volatile unsigned long	126
stacked_lr	volatile unsigned long	436225089
stacked_pc	volatile unsigned long	436219634
stacked_psr	volatile unsigned long	16777216
_CFSR	volatile unsigned long	1024
_HFSR	volatile unsigned long	1073741824
_DFSR	volatile unsigned long	1
_AFSR	volatile unsigned long	0
_BFAR	volatile unsigned long	3758157112
_MMAR	volatile unsigned long	3758157108

Figure A.1: A screen shot of the output of the hard fault handler implemented for debugging.

rxbuff[14]	uint8_t	126 '~'
rxbuff[15]	uint8_t	0 '\0'
rxbuff[16]	uint8_t	15 '\017'
rxbuff[17]	uint8_t	176 '°'
rxbuff[18]	uint8_t	198 'Æ'
rxbuff[19]	uint8_t	41 ')'
rxbuff[20]	uint8_t	30 '\036'
rxbuff[21]	uint8_t	241 'ñ'
rxbuff[22]	uint8_t	0 '\0'
rxbuff[23]	uint8_t	0 '\0'
rxbuff[24]	uint8_t	7 '\a'
rxbuff[25]	uint8_t	91 '['
rxbuff[26]	uint8_t	1 '\001'
rxbuff[27]	uint8_t	0 '\0'
rxbuff[28]	uint8_t	32 ' '
rxbuff[29]	uint8_t	2 '\002'
rxbuff[30]	uint8_t	0 '\0'

Figure A.2: A screen shot of parts of the RX ring buffer .

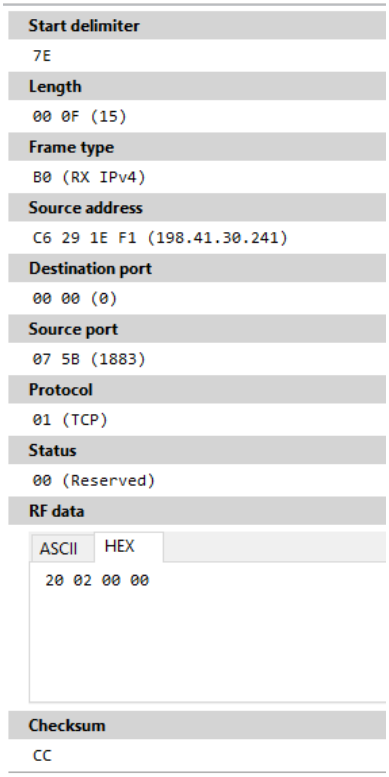


Figure A.3: A screen shot of the proper RX IPv4 frame generated by the XCTU software.

A.2 Sensor Figures

Figure A.4: A screen shot from the linear regression calculations performed using Wolfram Alpha

A.3 Additional Listings

Looking at line 1 and 2 in listing A.1 and A.2 we see the time difference needed to generate a Python dictionary from the query result from a DynamoDB query using the boto3 library. The choice of defining the period needed to perform a query as 0.05 seconds is a pessimistic one, but was chosen for simplicity and to give some leeway with respect to the calculations.

```
1 2018-06-05 00:05:02,207:INFO:Graph_Handler:Successfully generated the forecastDict
2 2018-06-05 00:05:02,241:INFO:Graph_Handler:Successfully generated the forecastDict
3 2018-06-05 00:05:02,484:INFO:Graph_Handler:Successfully generated the entire powerAvgDict
4 2018-06-05 00:05:02,655:INFO:Graph_Handler:Successfully generated an avg pow graph
```

Listing A.1: A small snippet of the log output from the Graph Handler to illustrate a DynamoDB query needs to complete a query.

```
1 2018-06-04 00:05:01,718:INFO:Graph_Handler:Successfully generated the forecastDict
2 2018-06-04 00:05:01,750:INFO:Graph_Handler:Successfully generated the forecastDict
3 2018-06-04 00:05:02,023:INFO:Graph_Handler:Successfully generated the entire powerAvgDict
4 2018-06-04 00:05:02,188:INFO:Graph_Handler:Successfully generated an avg pow graph
```

Listing A.2: A small snippet of the log output from the Graph Handler to illustrate a DynamoDB query needs to complete a query.

Bibliography

[1] Back to basics: The universal asynchronous receiver/transmitter (uart). https://www.allaboutcircuits.com/technical-articles/back-to-basics-the-universal-asynchronous-receiver-transmitter-uart/. Accessed: 2017-10-15.

[2] Dent elitepro xc series power meters. https://www.powermeterstore.com/P16777/dent-elitepro-esp-xc-power-meters. Accessed: 2017-12-02.

[3] Tcg120 - gsm/gprs controller. https://shop.marcomweb.it/en/shop-online/telecontrollo/tutti-i-prodotti/remote-i-o/tcg120-gsm-gprs-controller-dettagli.html. Accessed: 2017-12-02.

[4] Amazon. *Amazon DynamoDB - Developer Guide*. Amazon.

[5] Amazon. *Amazon Elastic Compute Cloud - User Guide for Linux Instances*.

[6] Amazon. *Amazon FreeRTOS - User Guide*.

[7] Amazon. *Amazon Simple Storage Service - Developer Guide*. Amazon.

[8] Amazon. *Amazon Simple Storage Service - Getting Started Guide*.

[9] Amazon. *AWS Command Line Interface - Getting Started Guide*.

[10] Amazon. *AWS Greengrass - Developer Guide*.

[11] Amazon. *AWS IoT - Developer Guide*.

[12] Amazon. Aws iot core. https://aws.amazon.com/iot-core/, 2018. [Online; accessed 13-February-2018].

[13] Amazon. Aws pricing. https://aws.amazon.com/pricing/?nc2=h_ql_pr&awsm=ql-3, 2018. [Online; accessed 2-June-2018].

[14] Amazon. Boto 3 documentation. https://boto3.readthedocs.io/en/latest/, 2018. [Online; accessed 28-March-2018].

[15] Amazon. Setting up with amazon ec2. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html, 2018. [Online; accessed 21-May-2018].

[16] Amazon. What is aws elastic beanstalk? https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html, 2018. [Online; accessed 13-February-2018].

[17] 4G Americas. Gsm global system for mobile communications. https://web.archive.org/web/20140208025938/http://www.4gamericas.org/index.cfm?fuseaction=page&sectionid=242, 2017. [Online; accessed 15-October-2017].

[18] arm Keil. *Using Cortex-M3/M4/M7 Fault Exceptions*, 2017. MDK Tutorial.

[19] John Catsoulis. *Designing Embedded Hardware*. Morgan Kaufmann, second edition, May 2005.

[20] Tao Cai Bangyin Liu Changsong Chen, Shanxu Duan. Online 24-h solar power forecasting based on weather type classification using artificial neural network. *Solar Energy*, 85(11):2856–2870, 2011.

[21] Wikipedia contributors. 3g — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=3G&oldid=823833557, 2018. [Online; accessed 4-February-2018].

[22] Wikipedia contributors. 4g — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=4G&oldid=823766065, 2018. [Online; accessed 5-February-2018].

[23] Wikipedia contributors. Comparison of mobile phone standards — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Comparison_of_mobile_phone_standards&oldid=821136749, 2018. [Online; accessed 5-February-2018].

[24] Wikipedia contributors. Linear regression — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Linear_regression&oldid=823846035, 2018. [Online; accessed 18-February-2018].

[25] Wikipedia contributors. Mqtt — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=MQTT&oldid=825426299, 2018. [Online; accessed 13-February-2018].

[26] Wikipedia contributors. Photovoltaics — wikipedia, the free encyclopedia, 2018. [Online; accessed 10-February-2018].

[27] Jens Deters. Welcome to the home of mqtt.fx. http://mqttfx.jensd.de/, 2018. [Online; accessed 3-June-2018].

[28] Digi. Digi xbee® cellular 3g. https://www.digi.com/products/xbee-rf-solutions/embedded-cellular-modems/digi-xbee-cellular-3g, 2018. [Online; accessed 5-April-2018].

[29] Digi. Rapid prototyping and connectivety to aws. https://www.digi.com/resources/standards-and-technologies/aws, 2018. [Online; accessed 5-April-2018].

[30] Digi International. *DIGI XBEE CELLULAR 3G User Guide*, 2018. User Guide.

[31] Digi International. *DIGI XBEE CELLULAR LTE CAT 1*, 2018. Datasheet.

[32] Eclipse. paho-mqtt 1.3.1. https://pypi.org/project/paho-mqtt/, 2018. [Online; accessed 21-May-2018].

[33] eclipse. paho.mqtt.embedded-c. https://github.com/eclipse/paho.mqtt.embedded-c/blob/master/MQTTClient-C/samples/FreeRTOS/MQTTEcho.c, 2018. [Online; accessed 9-June-2018].

[34] eclipse. paho.mqtt.embedded-c. https://github.com/eclipse/paho.mqtt.embedded-c, 2018. [Online; accessed 19-May-2018].

[35] Radio Electronics. Gsm network architecture. http://www.radio-electronics.com/info/cellulartelecomms/gsm_technical/gsm_architecture.php, 2017. [Online; accessed 15-October-2017].

[36] GSM Favorites. Introduction to general packet radio service (gprs). https://en.wikipedia.org/w/index.php?title=GPRS_core_network&oldid=804499437, 2017. [Online; accessed 15-October-2017].

[37] Django Software Foundation. Django: The web framework for perfectionists with deadlines. https://www.digi.com/support/forum/, 2018. [Online; accessed 10-June-2018].

[38] Python Software Foundation. 17.1. threading — thread-based parallelism. https://docs.python.org/3/library/threading.html, 2018. [Online; accessed 8-May-2018].

[39] Python Software Foundation. 17.7. queue — a synchronized queue class. https://docs.python.org/3/library/queue.html, 2018. [Online; accessed 8-May-2018].

[40] Amazon Web Services. The freertos™ kernel. https://www.freertos.org/. [Online; accessed 3-February-2018].

[41] Amazon Web Services. Freertos binary semaphores. https://www.freertos.org/Embedded-RTOS-Binary-Semaphores.html, 2018. [Online; accessed 9-May-2018].

[42] Amazon Web Services. Freertos+tcp. https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_TCP/index.html, 2018. [Online; accessed 31-May-2018].

[43] Amazon Web Services. Task priorities. https://www.freertos.org/RTOS-task-priority.html, 2018. [Online; accessed 9-May-2018].

[44] Amazon Web Services. Tasks and co-routines. https://www.freertos.org/taskandcr.html, 2018. [Online; accessed 9-May-2018].

[45] Erlend Grande. Data gathering and -assembling from several smart meter han ports. Ttk4990, Norwegian University of Science and Technology, Trondheim, June 2018. TTK4990 Master Thesis.

[46] Alexander Hansen. python-yr. https://github.com/wckd/python-yr, 2018. [Online; accessed 28-March-2018].

[47] Håkon Edøy Hanssen. Embedded system sensor monitoring with 2g/4g communication. Ttk4550, Norwegian University of Science and Technology, Trondheim, December 2017. TTK4550 Specialixation Project Thesis.

[48] Yitaek Hwang. Cellular iot explained – nb-iot vs. lte-m vs. 5g and more. https://www.iotforall.com/cellular-iot-explained-nb-iot-vs-lte-m/, 2018. [Online; accessed 8-May-2018].

[49] Dent Instruments. Elitepro energy logger. https://shop.dentinstruments.com/collections/elitepro-energy-logger, 2017. [Online; accessed 17-October-2017].

[50] Dent Instruments. *Operator's Guide Eitepro XC and ELOG 15*. Dent Instruments, 2017.

[51] Digi International. Digi forum. https://www.digi.com/support/forum/, 2018. [Online; accessed 10-June-2018].

[52] Digi International. The new dig xbee3 series. https://www.digi.com/xbee, 2018. [Online; accessed 31-May-2018].

[53] J. Tovar-Pescador D. Pozo-Vázquez J.A. Ruiz-Arias *, H. Alsamamra. Proposal of a regressive model for the hourly diffuse solar radiation under all sky conditions. *Energy Conversion and Management*, 51(5):881–893, 2009.

[54] James F. Kurose and Keith W. Ross. *Computer Networking - A Top-Down Approach*. Pearson, sixth edition, 2013.

[55] Link Labs. What is lte-m? https://https://www.link-labs.com/blog/what-is-lte-m, 2018. [Online; accessed 8-May-2018].

[56] LEM. *Current Tranducer HO-P Series*, 2014. Datasheet.

[57] Linear Technology. *LTC4145 - High Voltage I2C Current and Voltage Monitor*, 2008. Rev F.

[58] Maria Malvoni Maria De Giorgi, Paolo Congedo. Photovoltaic power forecasting using statistical methods: impact of weather data. *IET Science, Measurement Technology*, 8(3):90–97, 2014.

[59] Geir Mathisen. Notat flexnet - case 3: Last- og fleksibilitetspotensialet for en plusskunde. *Project number: 1020190782*, 2017.

[60] mathplotlib. The pyplot api. https://matplotlib.org/api/pyplot_summary.html, 2018. [Online; accessed 2-June-2018].

[61] Mouser. Bc-v1-ut-001. https://no.mouser.com/ProductDetail/Digi-International/XBC-V1-UT-001?qs=sGAEpiMZZMuCv89HBVkAk%2fJATD96F4I5YkJaOQjY3Po%3d, 2018. [Online; accessed 19-May-2018].

[62] Mouser. Xbc-m5-ut-001. https://no.mouser.com/ProductDetail/Digi-International/XBC-M5-UT-001?qs=pfd5qewlna6hjdV4MSQYsQ%3d%3d, 2018. [Online; accessed 19-May-2018].

[63] NXP. Lpcopen platform v1.03 lpcopen platform for nxp lpc microcontrollers. http://docs.lpcware.com/lpcopen/v1.03/index.html. [Online; accessed 9-May-2018].

[64] NXP. Lpc4337fet256: 32-bit arm cortex-m4/m0 mcu. https://www.nxp.com/products/processors-and-microcontrollers/arm-based-processors-and-mcus/lpc-cortex-m-mcus/lpc4300-cortex-m4-m0/32-bit-arm-cortex-m4-m0-mcu-up-to-1-mb-flash-and-136-kb-sram-ethernet-two-hi LPC4337FET256, 2018. [Online; accessed 5-April-2018].

[65] NXP. Lpcxpresso4337 development board. https://www.nxp.com/support/developer-resources/hardware-development-tools/lpcxpresso-boards/lpcxpresso4337-development-board:OM13070, 2018. [Online; accessed 5-April-2018].

[66] NXP. Om13054: Lpc-link2. https://www.nxp.com/products/processors-and-microcontrollers/arm-based-processors-and-mcus/

lpc-cortex-m-mcus/lpc1100-cortex-m0-plus-m0/lpc-link2:OM13054, 2018. [Online; accessed 16-May-2018].

[67] Nick O'Leary. Arduino client for mqtt. https://github.com/knolleary/pubsubclient, 2018. [Online; accessed 31-May-2018].

[68] Oracle. Introducing the internet protocol suite. https://docs.oracle.com/cd/E19455-01/806-0916/6ja85398m/index.html, 2017. [Online; accessed 15-November-2017].

[69] Python. 19.2. json — json encoder and decoder. https://docs.python.org/3/library/json.html, 2018. [Online; accessed 2-June-2018].

[70] Python. 28.10. traceback — print or retrieve a stack traceback. https://docs.python.org/2/library/traceback.html, 2018. [Online; accessed 19-May-2018].

[71] Nurettin Çetinkaya Qudsia Memon. Short-term power production forecasting in smart grid based on solar power plants. *International Journal of Engineering and Applied Sciences (IJEAS)*, 4(12):48–52, 2018.

[72] RealTerm. Realterm: Serial terminal. https://realterm.sourceforge.io/, 2018. [Online; accessed 3-June-2018].

[73] Margaret Rouse. Tcp (transmission control protocol). http://searchnetworking.techtarget.com/definition/TCP, 2017. [Online; accessed 15-November-2017].

[74] Audrey Selian. 3g mobile licensing policy: From gsm to imt-2000 - a comparative analysis. 2002.

[75] Amazon Web Services. About aws. https://aws.amazon.com/about-aws/, 2018. [Online; accessed 9-February-2018].

[76] Amazon Web Services. Amazon freertos. https://aws.amazon.com/freertos/, 2018. [Online; accessed 9-February-2018].

[77] William Stallings. *Operating Systems - Internals and Design Principles*. Eighth edition, 2015.

[78] Teracom. *GSM-GPRS Remote monitoring controller TCG120 - User Manual*. Teracom Systems, April 2017.

[79] torvalds. Reducing scheduling-clock ticks. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/timers/NO_HZ.txt. [Online; accessed 3-February-2018].

[80] Marit Schei Tundal. Using cloud services for data exchange with iot like devices. Ttk4550, Norwegian University of Science and Technology, Trondheim, December 2017. TTK4550 Specialixation Project Thesis.

[81] ubuntu wiki contributors. Systemdforupstartusers. https://wiki.ubuntu.com/SystemdForUpstartUsers, 2018. [Online; accessed 27-May-2018].

[82] Wikipedia. Gsm — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=GSMoldid=804880242, 2017. [Online; accessed 15-October-2017].

[83] Wikipedia contributors. Xbee — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=XBee&oldid=763025486, 2017. [Online; accessed 31-May-2018].

[84] Wikipedia contributors. Esp8266 — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=ESP8266&oldid=843195845, 2018. [Online; accessed 31-May-2018].

[85] Wikipedia contributors. Hall effect sensor — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Hall_effect_sensor&oldid=840083072, 2018. [Online; accessed 9-May-2018].

[86] Wikipedia contributors. Narrowband iot — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Narrowband_IoT&oldid=837360224, 2018. [Online; accessed 8-May-2018].

[87] Wikipedia contributors. Nodemcu — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=NodeMCU&oldid=835549174, 2018. [Online; accessed 31-May-2018].

[88] Wikipedia contributors. Putty — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=PuTTY&oldid=839195267, 2018. [Online; accessed 21-May-2018].

[89] Wikipedia contributors. Winscp — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=WinSCP&oldid=827766657, 2018. [Online; accessed 21-May-2018].

[90] Sean Wilkins. Tcp/ip ports and protocols. http://www.pearsonitcertification.com/articles/article.aspx?p=1868080, 2012. [Online; accessed 8-December-2017].

[91] Yr. Vilkår for bruk av gratis data frå yr. http://om.yr.no/info/verdata/vilkar/, 2018. [Online; accessed 29-March-2018].