Bachelor's thesis

Information Technology

Digital Media

2019

Sami Kankaristo

DEVELOPMENT OF A SECURE SENSOR SYSTEM FOR THE INTERNET OF THINGS

– Gathering and visualizing data from IoT sensors

TURKU AMK

TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information Technology | Digital Media

2019 | 93 pages

Instructor: Principal Lecturer Mika Luimula, Adj.Prof.

Sami Kankaristo

DEVELOPING A SECURE SENSOR SYSTEM FOR THE INTERNET OF THINGS

– Gathering and visualizing data from IoT sensors

This Bachelor's thesis is about the development of an affordable, easy to use and secure IoT system for monitoring and visualizing living conditions in homes, offices, and schools. The system uses sensors to monitor temperature, humidity, ambient air pressure, ambient light intensity, air quality, and carbon dioxide concentration in air. The purpose of the system is to allow users to explore different ways of improving their living conditions.

The theoretical section explores various technologies used in the system, such as wireless networks, HTTP, and the architecture of web APIs. All communication in the system must be secure, since the data could, for example, be used to determine when a home is not occupied. Due to this, the thesis also studies security in IoT systems.

The finished system uses the ESP32 microcontroller, which has an integrated Wi-Fi radio for connecting to the Internet. The sensor data is sent to a server application, which stores the data for later use. A client application can then retrieve and visualize the data for users.

This thesis covers the early development of the system, and focuses on the hardware, software and security of the IoT device itself, although an early version of the visualization software is also developed. Some small datasets are gathered, visualized, and analyzed for both home and office environments. The completed prototype only gathers and visualizes sensor data, but can be further developed to automatically control various things as well.

KEYWORDS:

Internet of Things, sensors, data visualization, wireless local area networks, data security

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietotekniikka | Mediatekniikka

2019 | 93 sivua

Ohjaaja: yliopettaja Mika Luimula, dos.

Sami Kankaristo

TIETOTURVALLISEN SENSORIJÄRJESTELMÄN KEHITTÄMINEN ESINEIDEN INTERNETIIN

– Datan keräys ja visualisointi IoT-sensoreilta

Opinnäytetyön tavoitteena oli kehittää edullinen, helppokäyttöinen ja tietoturvallinen sensorijärjestelmä esineiden Internetiin (englanniksi Internet of Things, IoT). Järjestelmää käytetään elinolosuhteiden mittaamiseen ja visualisointiin kodeissa, toimistoissa, ja kouluissa. Järjestelmä mittaa sensoreilla lämpötilaa, ilmankosteutta, ilmanpainetta, valaistusvoimakkuutta, ilmanlaatua ja hiilidioksidipitoisuutta ilmassa. Tarkoituksena on antaa käyttäjien tutkia eri tapoja parantaa elinolosuhteitaan.

Teoriaosuudessa tutustutaan järjestelmän käyttämiin keskeisiin teknologioihin, kuten langattomiin lähiverkkoihin, HTTP-protokollaan, sekä verkkopalveluiden ohjelmointirajapintojen arkkitehtuuriin. Kaikki järjestelmän kommunikaatio tulee olla tietoturvallista, koska dataa voitaisiin käyttää esimerkiksi päättelemään, milloin käyttäjän koti on tyhjillään. Tämän vuoksi opinnäytetyössä keskitytään myös IoT-järjestelmien tietoturvaan.

Valmistunut järjestelmä käyttää ESP32-mikro-ohjainta, jossa on integroitu Wi-Fi-radio Internet-yhteyttä varten. Sensoreilta saatu data lähetetään palvelinsovellukselle, joka tallentaa sen myöhempää käyttöä varten. Asiakassovellus hakee datan ja visualisoi sen käyttäjille.

Opinnäytetyö kattaa järjestelmän alkukehityksen ja keskittyy palvelinsovellukseen, laiteohjelmistoon, laitteistoon ja tietoturvaan. Myös visualisointiohjelmistosta kehitettiin varhainen versio. Joitakin pieniä tietoaineistoja kerättiin ja visualisoitiin sekä koti- että toimistoympäristöissä. Järjestelmä mittaa, tallentaa ja visualisoi dataa, mutta sitä voidaan kehittää myös automaattiohjaukseen.

ASIASANAT:

esineiden internet, sensorit, datan visualisointi, langattomat lähiverkot, tietoturva

CONTENTS

APPENDICES

FIGURES

PICTURES

LIST OF ABBREVIATIONS

AP	Access point, a device which serves as a central wireless connection point for a Wi-Fi network (WireShark.com 2019)
API	Application Programming Interface
BME280	The Bosch BME280 is a sensor that measures temperature, humidity, and ambient air pressure (Bosch 2018)
CA	Certificate Authority: an entity that issues certificates (Ristić 2017, 63)
ESP32	The ESP32 is a microcontroller manufactured by Espressif Systems. The ESP32 has integrated Wi-Fi and Bluetooth radios (Espressif Systems 2019a)
HTTP	Hypertext Transfer Protocol: the most common protocol for communication on the Internet (Gourley & Totty 2002, 3)
HTTPS	A secure version of HTTP, essentially HTTP over SSL (deprecated) or HTTP over TLS (Ristić 2017, 1-4)
IC	Integrated circuit
IoT	Internet of Things: a term used to describe the extension of the internet onto everyday devices (Burgess 2019)
JSON	JavaScript Object Notation: a common way store and transmit data (Bray 2014)
JWT	JSON Web Token: a way to encode JSON so it can be used as part of a URL, and so it can be signed and encrypted (Jones, et al. 2015)
MH-Z19B	The Winsen MH-Z19B is a sensor for measuring carbon dioxide concentration in air (Winsen 2016)
MQ-135	The Winsen MQ-135 is an air quality / gas sensor (Winsen 2015)
OSI model	The OSI model splits computer networking into 7 individual layers (Ristić 2017, 2-3)
PCB	Printed Circuit Board
RESTful API	Representational State Transfer API: the most popular method of designing web APIs (Richardson & Amundsen 2013, xvii)
SPIFFS	Serial Peripheral Interface Flash File System (RandomNerdTutorials.com), a flash filesystem for the ESP32 (Espressif Systems 2019k)

SSID	Service set identifier, the name of a wireless network (Goggi 2014)
SSL	Secure Sockets Layer: a now-deprecated predecessor to TLS, but is often used interchangeably as a term (Ristić 2017, 1-4)
TLS	Transport Layer Security, a cryptographic protocol used to provide security for communication in computer networks (Ristić 2017, 1-4)
TSL2561	The AMS TSL2561 is a sensor that measures light intensity and reports readings in the unit lux (AMS 2018)
UART	Univeral asynchronous receiver-transmitter, a serial port (Horowitz & Hill 2015, 873)
URL	Uniform Resource Locator, the address of a resource on the Internet (Berners-Lee et al. 1994)
Wi-Fi	A family of radio technologies used for wireless local area networks (Pogue 2012)
WLAN	Wireless local area network
WS2812B	The Worldsemi WS2812B is an intelligent RGB LED, which has data in and data out ports, and has the ability to chain multiple WS2812B RGB LEDs together (Worldsemi 2018)

1 INTRODUCTION

The Internet of Things (IoT) is one of the most talked about concepts in modern computing, alongside artificial intelligence (AI), cloud computing and big data. These emerging technologies are sometimes referred to as “the fourth industrial revolution”. These things have the potential to transform people’s lives into something that used to be subjects in science fiction, like the Internet or smart phones before they became an everyday part of life. (Forbes 2018)

Microcontrollers and wireless technology are constantly evolving, and these days you can quite easily create an Internet connected device for just a few euros. This has caused an explosion in the amount of Internet connected devices, and many companies have started developing their own IoT solutions.

One example of an ingenious IoT solution is that multiple companies have developed devices, which are embedded inside concrete, where they monitor the hardening and long-term strength of the concrete. This optimizes construction, and improves the safety of the concrete structure. (Giatec Scientific Inc. 2018)

One very important aspect of IoT is data security, which has been notoriously bad in the past, and is still in the news quite often. In a worst case scenario, just a single compromised device in a wireless network can compromise the entire network. Not only are most IoT devices wireless - which makes them inherently easier to hack than wired devices - but they are also connected to the Internet, so they can become compromised by a hacker across the globe. Because of this, data security has to be a part of the discussion when talking about the Internet of Things. (Wallace 2019)

The objective of this thesis is to develop the first prototype of a secure, affordable, and easy to use system for gathering and visualizing data from sensors, which are connected to an IoT device. This first prototype will likely not be very easy to use or perfectly secure, but these will be the main goals of the final system.

The sensors used in the prototype measure a wide variety of ambient variables, like temperature and carbon dioxide concentration in air, with the aim of monitoring and visualizing living conditions in homes, offices, or schools. The device will also use an RGB LED as visual feedback for the user.

The overall aim of the system goes further than just sensor measurements, and after the current sensor system is working well, other sensors can be added, and the devices can begin to control and automate things as well, in addition to making measurements. The first prototype will be a rather typical IoT sensor network, but the overall aim of the project beyond this thesis will be to provide the backbone of the company’s IoT system, which can be used for various IoT products.

The IoT system is being developed by Indium Technology Ltd under the working title “Taio”. Indium was established in 2014, originally as a game development company, but has since focused more on software development subcontracting. Taio is currently in early internal development, but if it proves to be viable, it may become the company’s first commercial product. Subcontracting in the software industry pays well, but is limited by the number of employees and their work hours, but a company with its own products doesn’t have a similar limitations on earning potential. The project is also a way to learn new technologies, even if it doesn’t ultimately result in a product.

Developing an IoT system has become relatively easy ever since the Espressif ESP8266 was introduced. The ESP8266 is a 32-bit, 160 MHz microcontroller with an integrated Wi-Fi radio and antenna (Espressif Systems 2018b). A ready-to-use ESP8266 module can be bought for under 1.5 euros, shipping included (price checked on 2.5.2019 from AliExpress). This thesis was the perfect opportunity to get started with development on Espressif microcontrollers.

The data visualization part of the project was suggested by the thesis’ instructor Mika Luimula, during the initial discussions about the thesis in January 2019. He also suggested testing the product as part of “The Smart Learning Environments for the Future”, which is a collaborative project between 6 Finnish cities, and is funded by the European Regional Development Fund. The project provides companies with the opportunity to develop their products in collaboration with teachers and students. (Tulevaisuuden älykkäät oppimisympäristöt 2019)

However, due to the thesis’ time constraints, the system will initially be tested at the author’s home, and at 2 offices and 1 conference room at the Hive: Turku Game Hub. This is done for easier access while the system is in the prototyping phase. Once the system has an enclosure, has been internally tested, and has a better user experience, it can be tested with its first actual users as part of “The Smart Learning Environments of the Future”.

2 THE INTERNET OF THINGS

The Internet of Things is a concept, which means the extension of the Internet from what are usually thought of as computers, into everyday “things”. In an exaggerated form, the Internet of Things means that just about anything can be connected to the Internet, and it can send data, make decisions, and be controlled via the Internet. (Burgess 2018)

Security is a big concern on the Internet of Things (Wallace 2019), and IoT security is discussed in **Chapter 3**. Before moving on to security, some basic Internet technologies used in the system should be introduced.

2.1 The OSI model

The Open Systems Interconnection (OSI) model splits computer networking into 7 layers, each of which are, in theory, independent from each other (Fall & Stevens 2012, 8-9; Ristić 2017, 2-3). The lowest layer (layer 1, the physical layer) is the closest to the hardware, and the higher layers have higher levels of abstraction (Ristić 2017, 2-3).

The OSI model was developed by the International Organization for Standardization (ISO). During the 1970s, there was much debate about the benefits and deficiencies of the OSI model and the ARPANET reference model that preceded it, which was ultimately adopted by the TCP/IP suite. (Fall & Stevens 2012, 8-14)

OSI and TCP/IP were developed at roughly the same time, and were competing protocols (Russell 2013). TCP/IP eventually “won”, but despite this, the OSI layering model and numbering is still widely used, although the actual layering used on the Internet (the TCP/IP suite) is simpler, and is normally divided into 5 layers. (Fall & Stevens 2012, 8-14)

Russell (2013) divides TCP/IP into only 4 layers, but both he and Fall & Stevens (2012) combine layers 5-7, which Ristić (2017, 2-3) also describes as “fuzzy”. The OSI model and numbering is still in wide use today (Fall & Stevens 2012, 9), and was also taught to the author at the Turku University of Applied Sciences. For the purposes of this thesis, the OSI model will be used to describe how HTTP (layer 7) can be used with or without SSL/TLS (layer 6), since the layers are independent of each other.

Because the layers are independent of each other, the higher layers don't need to worry about which lower layer is being used. Layers can switch to a different protocol, without needing to change any of the other layers, and some layers can be omitted. For example, when encryption is not required, the layer that implements encryption can be omitted, which also removes the overhead introduced by encryption. The OSI model layers are shown in **Table 1**. (Ristić 2017, 2-3)

Table 1. OSI model layers (Ristić 2017, 3; Fall & Stevens 2012, 9).

#	OSI layer	Description	Example protocols
7	Application	Application data	HTTP, SMTP, IMAP
6	Presentation	Data representation, conversion, encryption	SSL/TLS
5	Session	Management of multiple connections	-
4	Transport	Reliable delivery of packets and streams	TCP, UDP
3	Network	Routing and delivery of datagrams	IP, IPSec
2	Data link	Reliable local data connection (LAN)	Ethernet
1	Physical	Direct physical data connection (cables)	Cat. 5

HTTP is one of the protocols used in the application layer (layer 7). Securing the HTTP protocol (using HTTPS) doesn't change anything in layer 7, but instead adds encryption in layer 6 (the presentation layer). Because of this, HTTPS is sometimes called HTTP over TLS, because the higher-level HTTP protocol is used on top of the lower-level TLS protocol. (Ristić 2017, 3)

2.2 Wi-Fi

Most IoT devices are wirelessly connected to the Internet. While some have a "direct" connection via a cellular network, many IoT devices use Wi-Fi, and connect to the Internet through a wireless access point (AP). (Wallace 2019)

Wi-Fi is the name for a family of wireless technologies ratified in a set of IEEE standards that all begin with "802.11". Wi-Fi is not an abbreviation, although it is a pun on "hi-fi" (high fidelity). Essentially, Wi-Fi is a "friendly" name for the IEEE 802.11 family of standards. Some of these standards are for different versions of Wi-Fi (like 802.11n and 802.11ac), while others are standards about Wi-Fi security (like 802.11i) or other aspects of how Wi-Fi devices should work (like 802.11e and 802.11u). (Pogue 2012)

To help with the ”alphabet soup” that these standards have created on the market, the Wi-Fi Alliance – the group responsible for Wi-Fi standards – have retroactively introduced Wi-Fi version numbers. For example, instead of ”802.11ac” the name ”Wi-Fi 5” can be used. (Kastrenakes 2018)

Different versions of Wi-Fi have different maximum speeds, and operate in either the 2.4 GHz band or the 5 GHz band of radio frequencies. Some versions are ”dual-band”, meaning they operate in both frequency bands. The 5 GHz band has less interference from other devices, but the shorter wavelengths don’t penetrate walls as well, so the 5 GHz band usually has a shorter range. (Pogue 2012)

2.3 URI and URL

A URL or Uniform Resource Locator is the address of a resource on the Internet. URLs are Uniform Resource Identifiers (URIs) on the World Wide Web, so they can be thought of as a subset of URIs. Most people are familiar with URLs due to how ubiquitous they are. (Berners-Lee et al. 1994)

A URL begins with a scheme, which is usually a protocol like HTTP, and usually written in lowercase. After the scheme comes a colon, and the ”scheme specific part”. On the Internet, there is very little variance on the syntax, and a common syntax of <scheme>://<user>:<password>@<host>:<port>/<url-path> is used for most URLs. Most parts of this common syntax can be excluded, often leaving something like <scheme>://<host>/<url-path>. (Berners-Lee et al. 1994)

2.4 HTTP and HTTPS

The most common schemes in a URL are the protocols HTTP and HTTPS. HTTP is short for Hypertext Transfer Protocol, and HTTPS is a secure version of HTTP, and is essentially HTTP over a secure connection. HTTPS is studied more in **Chapter 3**, but HTTPS doesn’t really change how HTTP works, since it is essentially just encrypted HTTP. (Ristić 2017)

Understanding how HTTP works at a lower level is especially important when working with a embedded device like an IoT device, although it is also useful when developing applications on a higher level of abstraction. Being able understand the “bare” HTTP messages being sent greatly helps when debugging problems in client-server communication. In order to limit the scope of this thesis, things like HTTP methods and HTTP status codes won’t be covered in great detail, but “HTTP - The Definitive Guide” by David Gourley and Brian Totty is a great resource for learning more about the HTTP protocol.

In order for a client to get a resource from a server, an HTTP transaction is performed. The transaction consists of a request sent to the server, and a response sent back from the server. (Gourley & Totty 2002, 8)

The request and response are HTTP messages, and each message has 3 parts: a start line, headers, and a body. The start line and headers are always ASCII text, but the body can be either text (usually UTF-8) or binary, and can be omitted entirely, depending on the message. The start line is the first line of the message, followed by the headers, one on each line. The headers are key-value pairs, separated by a colon. The body is separated from the headers by an empty line. The exact syntax of the message depends on whether the message is a request or a response, and on the HTTP method being used. (Gourley & Totty 2002, 43-46)

The syntax of the request message is the following (Gourley & Totty 2002, 45):

```
<method> <request-URL> <version>
<headers>

<entity-body>
```

The method is one of the available HTTP methods, like GET, HEAD, or POST. The request URL is the path to the resource being requested (domain name resolution, et cetera, have already been done on the lower levels of the OSI model, so this URL does not contain an IP address or domain name). The version is the version of the HTTP protocol being used, e.g. “HTTP/1.1”. The headers can, for example, describe in which formats the requested resource will be accepted by the client, and in case the body is not omitted, the headers can also describe the format of the body. (Gourley & Totty 2002, 46-47)

The syntax of the response message is the following (Gourley & Totty 2002, 46):

```
<version> <status> <reason-phrase>
<headers>

<entity-body>
```

The version is the version of the HTTP protocol being used, like in the request message. The status is an HTTP status code, which is a 3-digit number describing the result of the request. The first digit describes the general “class” of the status code (e.g. success or error). The reason phrase is meant only for human consumption, and only the status code should be used by applications. (Gourley & Totty 2002, 46-47)

Like in the request, the headers can describe the format of the body, among other things. The body contains the requested resource, or it can contain more information about why a request failed (if the status code reports an error). Like in the request, the body can also be omitted. For example, the HEAD method only gets the headers for a resource, so the body is omitted in both the request and the response. (Gourley & Totty 2002, 46-48)

An exhaustive list of the status codes and their uses can be found online in MDN (the Mozilla Developer Network) or in “HTTP - The Definitive Guide”, which is cited as a source in this thesis. However, **Table 2** describes the status code classes and gives some example status codes.

Table 2. HTTP status code classes (Gourley & Totty 2002, 49; 2002, 59-67).

Overall range	**Defined range**	**Category**	**Examples (code and reason phrase)**
100-199	100-101	Informational	100 Continue
200-299	200-206	Successful	200 OK 201 Created
300-399	300-305	Redirection	301 Moved Permanently
400-499	400-415	Client error	400 Bad Request 401 Unauthorized 404 Not Found 405 Method Not Allowed
500-599	500-505	Server error	500 Internal Server Error 502 Bad Gateway

HTTP is also extensible, because a developer can define “extension headers”, which are custom headers that are not explicitly defined in the HTTP specification. This allows developers to define their own headers, and as long as both the server and the client understand the headers, they can be useful. Extension methods can also be used, although they are rarer. (Gourley & Totty 2002, 50-51)

2.5 JSON

JavaScript Object Notation (JSON) is a data syntax commonly used on the Internet. JSON syntax is essentially the same as the syntax for defining objects in JavaScript, with some exceptions, like not supporting JavaScript functions and undefined not being an allowed value (although null is). (Bray 2014)

Despite this, JSON is widely supported in many other programming languages as well. JSON is hierarchical, extensible, easy for programs to parse, and easy for humans to read. Because of this, it is very commonly used in modern web APIs (application programming interfaces). (Richardson & Amundsen 2013, 20)

2.6 JWT

JSON Web Tokens (JWTs) are a way to encode JSON into base64. One use of this is that JSON can be included as part of URLs. JWT also supports encryption (JSON Web Encryption), which is useful for ensuring that the data can be securely transmitted. JWT also supports signing (JSON Web Signature), so even without encryption, the data can be verified to not have been changed or sent by someone not intended to be sending it. JWT is commonly used for authentication tokens. (Jones, et al. 2015)

2.7 RESTful web APIs

REST (Representational State Transfer) is “an architectural style for distributed hypermedia systems” (Fielding 2000), which has a set of principles that are said to be RESTful (RESTfulAPI.net 2017).

There used to be another popular method of designing web APIs called SOAP (Simple Object Access Protocol), which used XML (Extensible Markup Language). But once

RESTful APIs “won the war” over SOAP APIs, JSON has become the de facto data format to use with RESTful APIs (Richardson & Amundsen 2013, xvii). JSON has many advantages over XML, which is considered to be too verbose by many developers (Atwood 2008).

In essence, a RESTful web API manipulates resources (identified by URLs) with common HTTP methods like GET, PUT, POST, and DELETE in a uniform way, so that the same method always performs the same type of operation. The ”official definition” of what REST means is Roy Fielding’s PhD dissertation from 2000, which had a chapter about REST. However, a stricter set of rules and best practices has formed around what most people mean by a RESTful API, and these rules should generally be followed. (RESTfulAPI.net 2017)

As an example of these best practices, PUT should always be used to create a new resource, while POST should only be used to modify or update a resource (Bondy 2010). Not all developers follow this practice and it the original description of REST by Fielding does not go into these kinds of specifics.

3 CRYPTOGRAPHY AND INTERNET SECURITY

Internet security is a vital part of modern society. Without proper security, many parts of everyday life, like online banking, bank card payments, or secure messaging could not function. In the early days of computer networking, security was not thought about very much. The earliest computer networks were internal networks in places like universities, where everyone using the network was “trusted”. (Ristić 2017, 1)

Because of this, many of the networking protocols that are still in use today are not secure, and security has been added afterwards. For example, HTTP is inherently insecure, and HTTPS is essentially HTTP with an additional secure layer called TLS (Transport Layer Security). (Ristić 2017, 1-3)

Internet security is also a vital part of the Internet of Things, but many IoT devices have bad security (Wallace 2019). This chapter studies some of the important concepts in secure Internet communications, in order to avoid the common security pitfalls found in IoT devices.

A note about the sources used in this chapter

This thesis uses 2 main sources in this chapter. The first is “Understanding Cryptography” by Christof Paar & Jan Pelzl, which was released in 2010, and is a respected book when it comes to teaching cryptography. However, it is almost 10 years old as of writing. This is a long time in the IT world, but this source is mostly used to explain the basics of cryptography itself, which changes slowly as a field. Some of the most used cryptographic algorithms and protocols, that are still considered secure, have been around since the late 1970s.

The second main source, “Bulletproof SSL and TLS” by Ivan Ristić, was published in 2017, and is kept updated online as new security exploits are introduced. This source is used for more up-to-date security information.

Instead of focusing on specific algorithms and protocols, which may or may not become insecure in the next decade, this chapter will focus on symmetric and asymmetric cryptography, and the basics of Internet security, like the public-key infrastructure (PKI). The focus of the chapter is less on what is considered secure as of writing, and more on the basic principles that need to be grasped, in order to understand cryptography and Internet security.

3.1 Cryptology

Most people associate cryptology with modern computing, but cryptology has a long history, with early examples dating back to about 4,000 years ago, when “secret” hieroglyphics were used in ancient Egypt. Cryptology has played an important part in history, some well-known examples being the Caesar cipher used by Roman leaders, and the Enigma machine from World War II. (Paar & Pelzl 2010, 2)

Cryptography and cryptanalysis

Paar & Pelzl (2010, 3) split cryptology into two main branches:

- **Cryptography**, or the “science of secret writing”, meaning the creation and use of algorithms to write secret messages.

- **Cryptanalysis**, or the science of breaking cryptographic systems.

The cracking of the aforementioned Enigma machine by British mathematicians (most famously by Alan Turing) played an important part in World War II (Paar & Pelzl 2010, 3-9). Cryptanalysis is a fascinating subject in itself, but it is a much too broad and advanced topic for this thesis. Instead, we will trust expert consensus that the cryptographic methods in use on the Internet today are secure, and this thesis will simply focus on cryptography in computer systems (and only scratching the surface there, too).

Cryptography

According to Paar & Pelzl (2010, 3), cryptography splits into three main parts (also shown in **Figure 1**):

- **Symmetric algorithms**, where all parties have a shared (symmetric) secret key.

- **Asymmetric (or public-key) algorithms**, where one party has a private key (similar to symmetric cryptography), but all other parties are given a different public key.

- **Cryptographic protocols**, which is, roughly speaking, the application of the above mentioned algorithms.

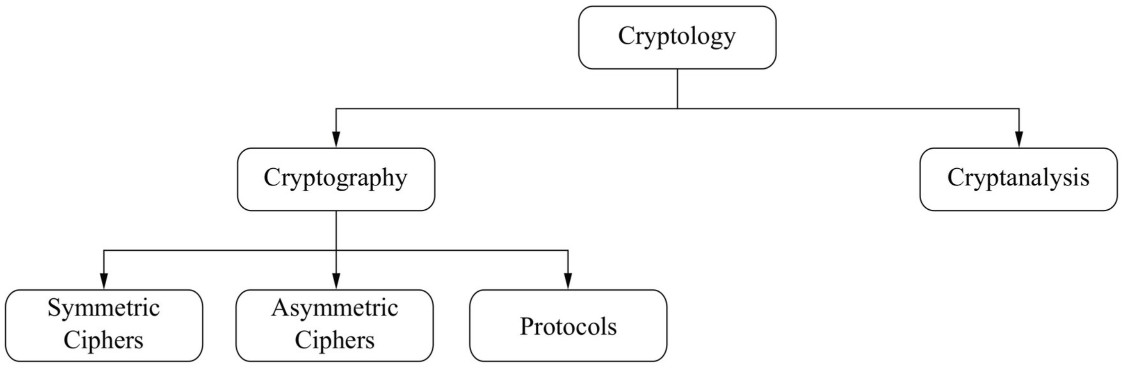


Figure 1. Overview of the field of cryptology (Paar & Pelzl 2010, 3).

Paar & Pelzl (2010, 4) also mention that hash functions would form a third class of algorithms, but they share enough similarities with symmetric algorithms, that the authors split cryptography in this way.

3.2 Confidentiality, authenticity & integrity

According to Ristić (2017, 4), the basic goal of cryptography is to achieve the three core requirements of security:

- **confidentiality** (keeping secrets),

- **authenticity** (verifying identities), and

- **integrity** (ensuring that data does not change during transport).

Many people may only associate cryptography with confidentiality, or in other words, encrypting data, so that outsiders cannot read it. Authenticity and integrity are equally important. Authenticity means that other parties can be identified to be who they purport to be, which is especially important in public-key cryptography. Integrity is important, so that a man-in-the-middle attack cannot compromise an otherwise secure communication channel.

Before moving on to all of the above, the following chapters explain some cryptographic “primitives”, such as hash functions, random number generators, and ciphers. These work as the basic building blocks for cryptography. (Ristić 2017, 5-14)

3.3 Hash functions

A hash function is an algorithm, which converts an input of arbitrary length (possibly very long) into a small, fixed-size output, which is called a digest (or “hash value”, or informally simply “hash”). Like was mentioned in **Chapter 3.1**, hash functions share many similarities with symmetric algorithms, but unlike those, hash functions do not use a key, and are not used for encryption. Instead, the most important use of hash functions is to provide a “fingerprint” of the input message. (Paar & Pelzl 2010, 293)

Checksum functions vs. hash functions

A digest is sometimes called a “checksum”. This is not incorrect, but can be confusing, because hash functions can be thought of as a subset of checksum functions. The difference between what is normally meant by “checksum” and by “digest” (or “hash”, or “cryptographic checksum”) should therefore be clarified.

In general usage, the term “checksum” refers to a checksum which is used to protect from accidental changes to data, but it can be extremely easy to intentionally modify the data so that it produces the same checksum. These checksums are used to protect from the corruption of data on filesystems (e.g. CRCs or cyclic redundancy checks), or to verify that a file has been downloaded without transmission errors (e.g. MD5). (Atwood 2005)

The term “cryptographic checksum” is the only unambiguous way to refer to checksums, which are cryptographically secure. In other words, they are meant to protect from intentional changes to the data (although they also protect from accidental changes). One example of a cryptographically secure hash function is SHA (Secure Hash Algorithm). Cryptographic checksums are usually more costly to calculate, so using them for something like a filesystem integrity could degrade performance. (Atwood 2005)

In regards to performance, the ESP32 microcontroller used in this thesis has hardware acceleration for the SHA-2 hash function (Espressif Systems 2019, 3-4).

The line between these is sometimes blurred, because some cryptographic hash functions are no longer considered cryptographically secure, but are still useful as general checksums. For example, MD5 is very widely used as a file checksum, but was originally proposed in 1991 by Ronald Rivest (known as one of the original authors of the RSA algorithm) to be a cryptographically secure hash function (Paar & Pelzl 2010, 304). Despite no longer being considered cryptographically secure in all aspects, these hash functions can still be sufficient for many applications, even storage of password hashes (Paar & Pelzl 2010, 304-305). This is because a hash function is not reversible, so the password cannot be recovered from the hash; a hash is not encryption (Friedl 2005).

Perhaps to avoid this confusion, of the 2 main cryptography sources used in this thesis, one does not contain the word “checksum” at all (“Bulletproof SSL and TLS” by Ivan Ristić) and the other only has the word 10 times, despite being about 350 pages long (“Understanding Cryptography” by Christof Paar & Jan Pelzl). In this thesis, the term “checksum (function)” will be used to refer to any checksum (function), and all other terms are to be understood as cryptographically secure.

Cryptographically secure hash functions

In order for a hash function to be useful for cryptographic purposes, it needs to have the below additional properties when compared to an insecure checksum function. (Ristić 2017, 9-10)

Preimage resistance: Given a hash, it is computationally infeasible to find the original message that produces the hash. (Ristić 2017, 9-10; Friedl 2005)

Second preimage resistance: Given a message and its hash, it is computationally infeasible to find another message with the same hash (e.g. to maliciously change the message without changing the hash). (Ristić 2017, 9-10; Friedl 2005)

Collision resistance: It is computationally infeasible to find any 2 messages that have the same hash (Ristić 2017, 9-10; Friedl 2005). This is sometimes incorrectly referred to as the hash function producing “unique” hashes, although this is actually impossible, since the digest is usually shorter than the message, so it is impossible to *not* have collisions (Friedl 2005).

3.4 Ciphers

A cipher is a pair of cryptographic algorithms used for encryption or decryption. The way a particular cipher works depends on the algorithm. (Lyons 2012)

An unencrypted message is called a plaintext (or cleartext). Once encrypted with a cipher, the message is called a ciphertext. All ciphers use a key, which is a secret that is used to run the encryption algorithm. (Paar & Pelzl 2010, 5; Lyons 2012)

As mentioned in **Chapter 3.1**, ciphers are also split into symmetric ciphers and asymmetric ciphers, both of which are discussed later. Symmetric ciphers can be further split into stream ciphers and block ciphers as shown in **Figure 2**. (Paar & Pelzl 2010, 29)

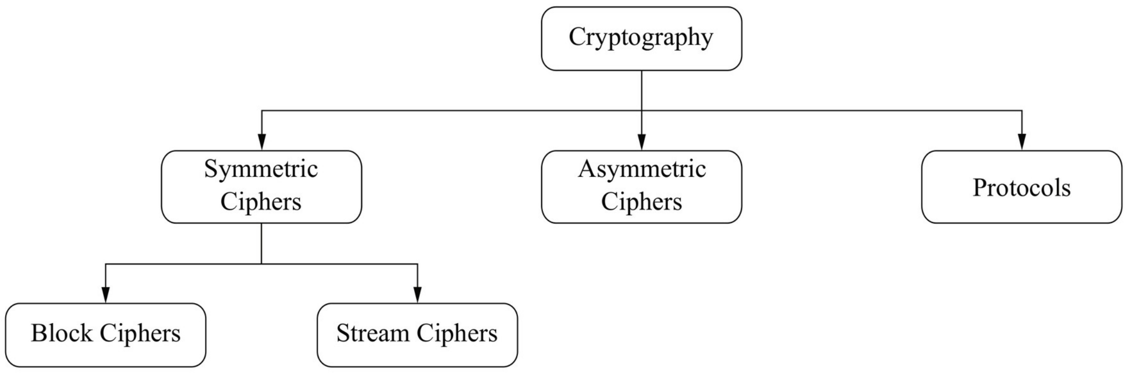


Figure 2. Main areas within cryptography (Paar & Pelzl 2010, 29).

Stream ciphers

Stream ciphers encrypt one bit at a time, which is achieved by adding a bit from a key stream to a plaintext bit. Stream ciphers can be further split into synchronous stream ciphers, where the key stream depends only on the key, and asynchronous stream ciphers, where the key stream is also affected by the ciphertext. (Paar & Pelzl 2010, 30)

In addition to the key mentioned earlier, almost all modern stream ciphers also require an IV (initialization vector), which should take a new value for every encryption session. Unlike the key, the IV does not need to be kept secret, it just needs to change for every session. (Paar & Pelzl 2010, 48)

Block ciphers

Block ciphers encrypt one block at a time. Each block is a certain number of plaintext bits. This is called the block length, and is usually either 128 bits (16 bytes) or 64 bits (8 bytes). With block encryption, the encryption of any plaintext bit affects the encryption of every other bit in the same block. (Paar & Pelzl 2010, 30-31)

In practice, block ciphers are naturally used to encrypt more than a single block. In order to choose how to encrypt subsequent blocks, block ciphers have “modes of operation” (Paar & Pelzl 2010, 124). This thesis will not go into the specifics of any particular cipher, or the modes of operation for block ciphers.

Still, some noteworthy things about the modes of operation should be mentioned:

- Some modes of operation require that the length of the plaintext to be encrypted needs to be a multiple of the block size. If it is not, padding needs to be added to the plaintext (Paar & Pelzl 2010, 124).

- Some modes of operation, like CBC (Cipher Block Chaining mode) utilize an IV, similar to stream ciphers (Paar & Pelzl 2010, 128).

Stream ciphers vs. block ciphers

On the Internet, block ciphers are used more often than stream ciphers. Stream ciphers tend to be small and fast, which may be relevant for this thesis, since this has the larger impact in embedded devices, like IoT devices. However, modern block ciphers, such as AES, tend to have good performance as well. (Paar & Pelzl 2010, 31)

The ESP32 microcontroller used in this thesis has hardware acceleration for RSA and AES (Espressif Systems 2019a, 3-4).

3.5 Random number generators (RNG)

In **Chapter 3.3**, it was mentioned that most stream ciphers and some modes of operation for block ciphers use an IV, which is an important part of the encryption. The IV is used as a randomizer, and it should be different for each session of encryption. The IV's purpose is to ensure that the key streams that are derived from the key are always different, even though the key has not changed. Without this, if an attacker has both the plaintext and the ciphertext from one encryption session, they can compute the key stream. If the same key stream is used for another session, the attacker can immediately decrypt the new encrypted message. Values like IVs are referred to as nonces, short for "number used once". (Paar & Pelzl 2010, 48)

For values like keys and nonces (including IVs) used in cryptographic functions, randomness is of central importance (Paar & Pelzl 2010, 50). In cryptography, there are 2 types of random number generators (RNG): true random number generators (TRNG) and pseudorandom number generators (PRNG). True RNGs are characterized by the fact that their output cannot be reproduced. A TRNG should be used to generate all of the initial values used in a cipher (like keys and nonces). A PRNG can then use these as seed values, which are used to generate pseudorandom values. (Paar & Pelzl 2010, 35; Ristić 2017, 14)

A pseudorandom generator generates a sequence of pseudorandom numbers from a given seed value (Ristić 2017, 14). If given the same seed value (like an IV), the pseudorandom numbers are always the same. The parties with access to all initial values can generate the same pseudorandom values, which is necessary in order to decrypt an encrypted message. (Paar & Pelzl 2010, 35)

A good PRNG has good statistical properties, meaning that the values it generates approximate true random numbers. However, all good PRNGs are not acceptable for cryptographic purposes (Paar & Pelzl 2010, 35-36; Ristić 2017, 14). For this, a CSPRNG (cryptographically secure PRNG) is needed, which means a PRNG which is unpredictable (Paar & Pelzl 2010, 35-36; Ristić 2017, 14). This means that even when given some values from the PRNG, the next value is infeasible to compute.

Unpredictability is of very little importance outside of cryptography, but in cryptography it is essential. (Paar & Pelzl 2010, 35-36)

Any software RNG is a PRNG (Paar & Pelzl 2010, 35). For true randomness, a physical process is required to collect entropy (Ristić 2017, 14). Flipping a coin or throwing a die is truly random, but instead of a coin flipping robot, most computers use something like semiconductor noise or the jitter of digital clock signals (Paar & Pelzl 2010, 35). This is a true RNG (TRNG) (Paar & Pelzl 2010, 35; Ristić 2017, 14).

The ESP32 has a hardware RNG, which uses radio noise from the environment, from either the Wi-Fi or Bluetooth radio, depending on which of them is enabled. If either one is enabled, the hardware RNG is a TRNG, but If neither radio is enabled, the hardware RNG is a PRNG. (Espressif Systems 2018a, 544)

3.6 Symmetric-key cryptography

All early ciphers were symmetric ciphers. Symmetric ciphers require a symmetric key, also called a pre-shared key, since it cannot be transmitted at the beginning of the communication, but needs to be known by all parties in advance (hence, pre-shared). (Paar & Pelzl 2010, 150-151)

A simple example: substitution ciphers

One symmetric cipher was already mentioned by name, the famous Caesar cipher, which is said to have been used by Julius Caesar to communicate with his army. The Caesar cipher (also known as shift cipher) is a substitution cipher, where each letter of the alphabet is simply shifted a constant number of steps, to another letter. When the end of the alphabet is reached, the shifting starts from the beginning of the alphabet. (Paar & Pelzl 2010, 4-18)

All symmetric ciphers require both parties to have a pre-shared key. In the Caesar cipher’s case, the key is the number of steps to shift the letters. In the case of more complex substitution ciphers, it would be a “codebook”, which would tell which letter should be substituted for which. Without knowing the key, the only option is to attempt to break the encryption. (Paar & Pelzl 2010, 4-9)

Delving lightly onto the cryptanalysis side of things, it can be observed for substitution ciphers that since every character of the ciphertext directly corresponds with a character of the original message, the encryption can be quite easily broken. For example, by looking for common short words like “is”, “and” or “the”, a simple substitution cipher can be broken piece by piece, revealing the key. (Paar & Pelzl 2010, 4-9)

The problem with symmetric ciphers

Even the Enigma machine, once a marvel of cryptography, has very weak encryption by today’s standards (Paar & Pelzl 2010, 57). All cryptography from ancient times until 1976 was symmetric (Paar & Pelzl 2010, 43). But by no means are symmetric ciphers “outdated”, despite asymmetric ciphers being the “new kid on the block” after about 4000 years (Paar & Pelzl 2010, 2) of symmetric ciphers.

Most readers will recognize by name the AES (Advanced Encryption Standard) algorithm. AES is a symmetric block cipher, and the most widely used symmetric cipher in use today. AES is used in TLS, Wi-Fi encryption, SSH (Secure Shell), and in many other security standards and products across the world. (Paar & Pelzl 2010, 87) Symmetric ciphers, like AES, are also widely used in data encryption and the integrity check of messages (Paar & Pelzl 2010, 3).

However, the fundamental problem with symmetric key algorithms is that the key needs to be pre-shared. In many use cases, like all of the ones mentioned above, this is not an issue. For example, it is expected for you to know the Wi-Fi passphrase in order to connect, or the passphrase to access an encrypted storage medium. On the Internet, however, a pre-shared key is often impossible, so asymmetric cryptography is needed.

3.7 Asymmetric-key cryptography

Asymmetric cryptography (synonymous with public-key cryptography) arrived in 1976, when an entirely new type of cipher was introduced by Whitfield Diffie, Martin Hellman, and Ralph Merkle. In addition to a secret key like the one which is used in symmetric cryptography, the user of an asymmetric cipher also possesses a public key. In asymmetric cryptography, the secret key is also called a private key. (Paar & Pelzl 2010, 3)

According to Paar & Pelzl (2010, 150), symmetric cryptography is symmetric in 2 respects:

1. The same secret key is used for encryption and decryption.

2. The encryption and decryption functions are very similar.

An analogy for this is a safe, and two parties have a copy of the same (secret) key. One puts something in the safe and locks it (encryption). The other party uses a copy of the same key to unlock the safe to retrieve the item (decryption). Locking and unlocking are very similar operations. (Paar & Pelzl 2010, 150)

Modern symmetric algorithms such as AES are very secure, fast, and widely used (Paar & Pelzl 2010, 150). However, symmetric-key algorithms have some serious shortcomings, as listed by Paar & Pelzl (2010, 150-151), which become very apparent on the Internet:

- **Key distribution problem:** The key must be agreed upon over a secure channel. Without using asymmetric cryptography, this means the need for a pre-shared key.

- **Number of keys:** Even if the distribution problem is solved, each pair of users needs a unique shared key. For 2000 users, this would mean securely storing over 4 million keys (Paar & Pelzl 2010, 151).

- **No protection against cheating:** If 2 or more people possess the same secret key, it cannot be determined who did the encryption (who locked the safe), and anyone with the key can decrypt the data (unlock the safe).

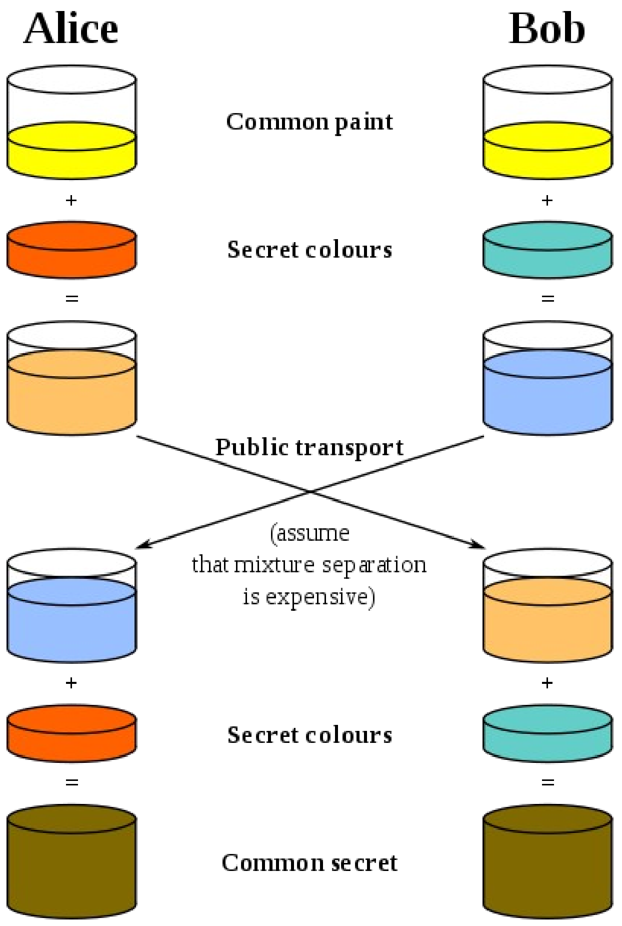
Asymmetric cryptography solves these problems, because asymmetric ciphers do not require that the encryption key (public key) and the decryption key (private key) are the same. With asymmetric cryptography, the receiving party keeps the private key as a secret, but shares the public key with any other party. The public key can only be used to encrypt data, and the only way to decrypt the data is by using the private key. (Paar & Pelzl 2010, 152)

The safe analogy still somewhat works with asymmetric cryptography, but becomes a bit convoluted. Now everyone has an unlocked safe, which automatically locks when the door is closed, without the needing a key (the safe is the public key). Only the person with the private key can unlock the safes that have been locked.

The Diffie-Hellman key exchange

The original 1976 paper which introduced asymmetric cryptography was called “New Directions in Cryptography” and was written by Whitfield Diffie and Martin Hellman. Ralph Merkle is also often cited as one of the inventors of asymmetric cryptography, because he invented it independently, but proposed an entirely different public-key algorithm. (Paar & Pelzl 2010, 168) The Diffie-Hellman key exchange was the very first asymmetric scheme, but it was also influenced by the work of Ralph Merkle (Paar & Pelzl 2010, 206).

The Diffie-Hellman key exchange (DHKE) is still widely used today, among other things in TLS and SSH (Paar & Pelzl 2010, 206). Without going into the specifics of the algorithm, DHKE is often explained with an analogy of mixing colors as illustrated in **Picture 1**.



Picture 1. Illustration of the idea of the Diffie-Hellman key exchange (Han Vinck 2011).

In the color mixing analogy, the algorithm is the following:

- The 2 parties agree on a common public color (one party randomly selects a color and sends it to the other party over a public channel).

- Each party mixes the common color with their own secret color, and sends the resulting color mixture to the other party over a public channel (the “unmixing” of the common and the secret color in the analogy is very expensive in the actual algorithm).

- Each party mixes their own secret color with the mixture they received from the other party.

- The result is a shared secret, which has mixed the same 3 colors (the 2 secret colors and the 1 public color).

In the actual algorithm, the common “color” is a large prime p and an integer α in the range between 2 and p-2. Each party then chooses another “secret” integer, calculates their own “mixed color”, sends it to the other party, and after another calculation, both parties arrive at the shared secret. (Paar & Pelzl 2010, 206)

The DHKE algorithm has some problems, because an attacker could hijack the communication after the original parties have agreed on a shared color, insert their own secret color, and the attacker could pretend to be the other party. Because of this, DHKE is commonly used with authentication provided by some other method. (Ristić 2017, 38-39) The 1976 paper by Diffie and Hellman opened a new branch of cryptography, and a year later in 1977, Ronald Rivest, Adi Shamir and Leonard Adleman proposed a scheme called RSA, which is the most widely used asymmetric crypto scheme (Paar & Pelzl 2010, 173).

Symmetric cryptography vs. asymmetric cryptography

So, which is better, symmetric or asymmetric cryptography, RSA or AES? In the majority of practical cryptographic systems, symmetric and asymmetric algorithms (and often also hash functions) are all used to together. This is because each has their own strengths and weaknesses. (Paar & Pelzl 2010, 4)

The weaknesses of symmetric cryptography have already been discussed, but it has a distinct benefit over asymmetric key cryptography, because “an 80-bit symmetric key provides roughly the same security as a 1024-bit RSA key”, RSA being a popular asymmetric algorithm (Paar & Pelzl 2010, 12). Considering the “strength” of a particular encryption algorithm usually involves considering the key size, so “bit-for-bit” symmetric algorithms have stronger security, but there is probably little sense in comparing such distinctly different algorithm families. This is why the combination of both symmetric and asymmetric methods, also known as “hybrid schemes”, are often used (Paar & Pelzl 2010, 4).

Public key cryptography (asymmetric cryptography) is also slower than symmetric cryptography. This means that it is usually used where it is necessary, like authentication and the negotiation of shared secrets (pre-shared keys). After this, symmetric cryptography can be used. (Ristić 2017, 13)

3.8 Protocols

Let’s review the 3 core requirements of security from **Chapter 3.2**: confidentiality, authenticity, and integrity. After all of these chapters, only confidentiality has been properly explained. This chapter will explain authenticity and integrity. In order to do this, we need to look into the protocols: the third and last part of cryptography as shown in Figure 1.

The cryptographic algorithms that have been introduced in earlier chapters are seldom used by themselves. In order to apply all of them, and to fulfill all 3 core requirements of security, cryptographic protocols are needed. One example of such a protocol is TLS, Transport Level Security. (Paar & Pelzl 2010, 3-4)

In order to maintain a reasonable scope, this thesis will not explain any particular protocol (like TLS), so only the basic principles of secure communication will be explained.

Message authentication codes (MACs)

In order to provide integrity, a hash function could be used on the data. However, an attacker could modify both the data and the hash of the data, easily avoiding detection. A message authentication code (MAC), also called a keyed-hash, can be used to provide integrity. Only those who posses the hashing key can produce a valid MAC. (Ristić 2017, 10)

MACs are commonly used alongside encryption, because encryption cannot provide integrity by itself. Even if an attacker cannot decrypt the ciphertext, they can still modify it. If they’re smart about how they do this, they can fool the receiver into accepting a forged message as authentic. (Ristić 2017, 10)

However, if a MAC is used, an attacker cannot do this without knowing the hashing key, and the parties in possession of the hashing key can be certain the message has not been tampered with. Any hash function can be used to create a MAC. This is called a hash-based MAC (HMAC), which works by interleaving the hashing key with the message in a secure way. (Ristić 2017, 10)

Digital signatures

In asymmetric encryption, the private and public keys have a special mathematical relationship between them. Anyone can use the public key to encrypt data, but only the private key can be used to decrypt the data. This provides confidentiality (encryption). (Ristić 2017, 12)

However, the same also works in reverse. Data can be encrypted with the private key, and this encrypted data can then be decrypted with the public key. This may seem useless at first glance, since the public key is meant to be public, so this cannot be used to provide confidentiality. However, anyone with the public key can confirm that the party who encrypted the data is the real owner of the private key. (Ristić 2017, 12)

This can be used for digital signatures, which provide authenticity, the third of our 3 core requirements of security. A digital signature allows the verification of the authenticity of a digital message or document. (Ristić 2017, 12-13; 2017, 4)

MACs are a type of digital signature, but they require that the hashing key is shared beforehand in a secure manner. This is still useful in certain situations, but has inherent limitations, so other types of digital signatures are also needed. (Ristić 2017, 13)

The exact method used for digital signatures varies on the public key cryptosystem that is used. Below is an example of how it can be done using RSA and SHA256 (Ristić 2017, 13-14):

1. Calculate the SHA256 hash of the document that should be signed. The output of SHA256 is always 256 bits in length.
2. Encode the hash of the document, along with some additional metadata. For example, the receiver will need to know the hash function that was used (SHA256 in this case), in order to check the validity of the signature.
3. Encrypt the encoded hash using the private key; the result is the digital signature, which is appended to the document as proof of the document’s authenticity.

In order to verify the signature, the hash of the document is calculated by the receiver, using the same algorithm that was used during signing. Then, the public key is used to decrypt the signature. The decrypted signature should show the same hash and hashing algorithm, if the signature is valid and the document has not been altered after signing. The strength of the signature depends on the individual strengths of the encryption, hashing, and encoding components that were used. (Ristić 2017, 13-14)

A communication example, and some common attacks

With everything that has been introduced so far, secure communication should finally be possible. Let’s go through a short example of secure communication, which fulfills all 3 core requirements. It is also assumed that there’s someone trying to attack the communication.

For the bulk of the data to be transmitted, strong symmetric encryption should be used (AES, for example). This will mean that the attacker cannot read the data. (Ristić 2017, 14-15)

However, they can still do other things, like modify the messages without being detected. To prevent this, MACs will be used, and the hashing key will only be known by the sender and receiver, not by the attacker. (Ristić 2017, 14-15)

Even while using MACs, the attacker could still be able to drop or repeat messages. To prevent this, a sequence number should be added to the messages; ideally, to the MACs. If a sequence number is repeated, the receiver will know that the message has been repeated, which is also known as a replay attack. If an expected sequence number is missing between messages, the receiver will know that a message has been dropped. Now all the attacker can effectively do is to prevent communication, which can’t really be helped. (Ristić 2017, 15)

The last thing that’s still needed, is to securely share the keys that are necessary for the above: the encryption key and the hashing key. To do this, the first thing that’s necessary is authentication, so that the people communicating can be sure that the keys are really sent to the correct person, and not an attacker. One way to do this would be to generate a (large) random number and ask the other party to sign it, and then verify the signature. Authentication is achieved when this is done in both directions. (Ristić 2017, 15)

With authentication done, the keys can be exchanged using a key-exchange scheme. For example, RSA key exchange works by one party generating all of the keys, encrypting them with the public key of the other party, and the other party decrypting the keys with their private key. The Diffie-Hellman key exchange (DHKE) introduced earlier in this chapter could also be used. DHKE is slower, but has better security. (Ristić 2017, 15)

At a high level, the above example is similar to TLS (Ristić 2017, 15). For the purposes of this thesis, this example is enough to explain secure communication, so TLS will not be discussed in detail. However, public-key infrastructure (PKI), including how SSL/TLS certificates are used, will be described next.

3.9 Public-key infrastructure (PKI)

Public-key cryptography allows secure communication with parties, whose public keys are known. There are still a number of problems related to this, like how to communicate with people whose public keys are not known beforehand, how to store and revoke public keys, and how to do this for the millions of servers and billions of devices online today. This chapter will explain how this is done, using public-key infrastructure (PKI). (Ristić 2017, 63)

Technically, PKI is a much wider term, so either Internet PKI or Web PKI would be a more accurate term (Ristić 2017, 63). In this thesis, the term PKI will be used to mean Internet PKI, and to describe how PKI is used on the Internet.

3.9.1 Certificates and certificate authorities (CAs)

The goal of PKI is to allow communication between parties who do not know each other beforehand. The model used to achieve this relies on trusted third parties, called certificate authorities (CAs), sometimes also known as certification authorities. CAs are unconditionally trusted by everyone, and are the root of PKI. (Ristić 2017, 63)

A certificate is a digital document, which contains a public key, some information about the entity associated with the certificate (the certificate's "owner"), and a digital signature from the issuer of the certificate. A certificate allows the exchange, storing, and use of public keys, and are the basic building block of PKI. (Ristić 2017, 66)

A root certificate is a certificate owned by a certificate authority themselves. Root certificates are stored in a root trust store, which are maintained by the operating system, a web browser, or some other program. This store is essentially a list of all trusted CAs, and is included (and updated) along with the operating system or program. (Ristić 2017, 64)

3.9.2 Identity validation

A server or other entity gets its certificate from a CA, which is trusted by everyone. The certificate is signed by the CA, so the certificate can be verified without having to store every single certificate, but just the root certificates. (Ristić 2017, 64)

Before issuing a certificate, the certificate authority will validate the identity of the entity requesting the certificate. There are many methods for a CA to do this, which creates 3 different classes of certificates. These classes, in order of increasing complexity of the validation are: domain validated (DV) certificates, organization validated (OV) certificates, and extended validation (EV) certificates. (Ristić 2017, 74-75)

Domain validated certificates are the simplest of these, and are issues based on proof of control over a domain name. In some cases, this means sending a confirmation email to one of the approved email addresses for the domain. If the recipient follows a link in the email, the certificate is issued. (Ristić 2017, 75)

Other methods of providing proof of control over a domain are by setting special DNS records as requested by the CA, or by using a special protocol (e.g. the ACME protocol) to access a server on the domain. These 2 methods are used by Let's Encrypt, the certificate authority used in the project. (Let's Encrypt 2019e)

3.9.3 Certificate chains

For most cases, a single certificate is not sufficient for authenticity. In practice, a server must provide a certificate chain, which leads to a trusted root certificate. Most certificates are issued by an intermediate CA, whose certificate is issued (and signed) by a root CA. A certificate chain can also be called a trust chain. (Ristić 2017, 71-72)

Each certificate in the chain is signed by the certificate that precedes it. Certificate chains provide improved security, and are also used to protect root certificates, by not commonly issuing certificates that are signed directly with the root certificate. (Ristić 2017, 71-72)

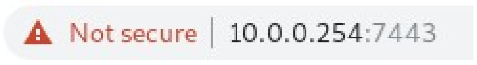
A server can also provide multiple certificate chains, which lead to multiple root CAs. This is called cross-certification or cross-signing, and provides improved security. Multiple certificate chains also protect PKI from situations where an intermediate or even root certificate is compromised, and can no longer be trusted. Having multiple certificate chains means that a certificate can still be trusted, while at least one of the chains can be followed to be a trusted root certificate, although such certificates should naturally be replaced. (Ristić 2017, 71-72)

This project uses certificates issues by a CA called Let's Encrypt. They use intermediate CAs, whose certificates are signed by both their own root CA, as well as being cross-signed by another root CA called IdenTrust. (Let's Encrypt 2019d)

The problem with certificates on embedded devices

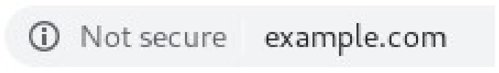
In order for PKI to properly function, certificates are issued with an expiration date. This is good for security, but is a problem on embedded devices, since they may act as HTTPS servers, but may not have an Internet connection in order to renew certificates.

One example of this issue are routers, which are commonly configured using a web browser, which connects to the router's HTTPS server (Ristić 2017, 137-138). Picture 2 shows how Google Chrome shows this connection as "not secure", because the certificate is self-signed, in order to not expire.



Picture 2. Example of a self-signed certificate on the Ubiquiti EdgeRouter X, when connecting with Google Chrome.

In the author's opinion, this is a bad decision from Chrome's developers, since the connection is still encrypted, but the certificate authority is unknown, so it cannot be trusted. This could be an attack, but since this is also a common situation on routers (and other embedded devices), it could be shown in a better way for the user. An encrypted connection with an unknown CA is still better than an unencrypted HTTP connection, but Chrome shows those in a much less alerting manner, as shown in **Picture 3**.



Picture 3. Example of an unencrypted HTTP connection in Google Chrome.

As far as I’m aware, there’s no existing solution for this. The only way to have a certificate on an embedded device is to self-sign the certificate, which means that they won’t be trusted by default.

Ivan Ristić (2017, 137-138), the author of “Bulletproof SSL and TLS”, acknowledges the problem. He argues that Firefox’s method of requiring the user to add a per-certificate exception is the best way to deal with this. Certificate exceptions allow the use of self-signed certificates, on something like a router, while still protecting from attackers who are trying to insert their own certificate.

4 HARDWARE AND SOFTWARE RESEARCH

The purpose of the system under development is to gather sensor readings, and to send them to a server on the Internet. In order to achieve this, the system naturally needs sensors, and a microcontroller with an Internet connection. The microcontroller reads the sensors, and sends the readings to the server.

There are many options out there for both the sensors and the microcontroller, and this chapter will explain the reasoning behind the specific hardware that was chosen for this project. The hardware choices were also heavily influenced by the libraries and software tools available for the hardware; software and hardware choices were made, and are now discussed, in tandem.

Datasheets

When trying to find parts for an electronics project, the definitive source of information for a part is the *datasheet*. A datasheet is a document provided by the manufacturer, which lists essential specifications about the part. These can include recommended, minimum and maximum operating conditions (voltages, temperatures, and so on), the part's physical dimensions and pinout, functional block diagrams, example circuits, etc. (Grusin 2010)

Sometimes the language in a datasheet can be fairly technical and rough for beginners (Grusin 2010). This also makes them succinct and to the point, once you're familiar with the central terms and concepts. Datasheets are an essential resource when designing a project like this.

Based on personal experience, the content of a datasheet varies wildly from manufacturer to manufacturer. Some datasheets are a hundred pages long (e.g. microcontroller datasheets), while others are only a few pages. Some datasheets are very low quality, containing typos and other errors, and some are only partially translated to English (if the manufacturer is Chinese, for example). This can make it difficult to use the part, and lowers trust in the manufacturer.

Datasheets give measurable, objective values, which makes it easy to directly compare alternative electronics parts. However, datasheets and the physical characteristics of a part are only one factor for choosing a part. There are also practical things like price and availability, the libraries or software development kits (SDKs) available, and so on, so there’s not always a clear-cut “winner”.

Software libraries and software licenses

One factor in choosing the correct part for the project is the software available for the part. It would be far too much work to write all of the software from scratch, since there are usually high-quality libraries and SDKs available for free. These greatly reduce the development time required for a project like this.

One issue to consider when researching the available software is the license that’s used to distribute the software. For a personal project, this is of little consequence, but the licenses can have implications for a project that might be commercialized at some point.

Broadly speaking, open source licenses can be divided into 2 categories: permissive licenses and copyleft licenses. Permissive licenses essentially mean “do whatever you want, use at your own risk, acknowledge the authors”. Copyleft licenses, however, often require that the derivative work must be distributed under the same license. In practice, using such a license might lead to the authors seeking to recover damages for copyright infringement, so it wouldn’t force you to disclose the project’s source code, but should be avoided nonetheless. (Meeker 2017)

Most copyleft licenses do in fact allow use in proprietary/closed-source software, but require dynamic linking of the licensed software (Meeker 2017), which can be practically impossible to do for embedded software. Because of these factors, only permissive licenses were considered for this project.

Counterfeit electronics and clones

Another factor to consider when researching parts for an electronic project is the thriving counterfeit market, an estimated 100 billion dollar loss of global revenue for electronics companies. It can be very difficult to tell the difference between the original part, and a counterfeit part. (Pecht 2013, 12)

It can also be difficult to find out if a part is “generic” or a patented/trademarked part. For example, if the original manufacturer is Chinese, there can be very little material available in English. Buying the cheapest available part on an online marketplace may mean buying a counterfeit. However, since this project is currently at the prototyping phase, some counterfeit parts will be acceptable (although not intentional). If the project is commercialized, the parts will be bought from a reputable dealer, and not an online marketplace, which is a good way to avoid buying counterfeit parts (Pecht 2013, 12).

One important distinction is that a “clone” is not the same thing as a counterfeit. For example, in the next chapter we’ll talk about Arduino, which is based on open-source hardware (Arduino 2019a). An “Arduino-compatible” clone board can be perfectly legal, as long as it doesn’t claim to be an Arduino, which is a trademarked name (Arduino 2019b).

4.1 Arduino

Arduino is an open-source electronics platform, consisting of both hardware and software. Arduino is aimed at students and makers, and has gained a large community. It’s widely used among electronics hobbyists and even professionals. Arduino is both a collection of hardware devices (“boards”) and software, including a large collection of libraries developed by the community and even its own IDE (integrated development environment). (Arduino 2019a)

4.1.1 Arduino hardware

Arduino boards are an easy way to get started in electronics (Arduino 2019a), and an Arduino board was the author’s first foray into embedded development. The official Arduino boards are high-quality, but also relatively expensive, since they’re funding Arduino development. Plenty of cheap clones also exist, since the Arduino hardware is open-source (Arduino 2019a).

However, this project won’t be using an Arduino, since one of the most common ways to connect an Arduino to the Internet is to use a module based on the ESP8266 (Sachleen 2014), which itself is a powerful microcontroller. It and its successor, the

ESP32, will be introduced in a later chapter. The community and the build tools on the Arduino platform are still very helpful, so the Arduino platform as a whole is used by the project, even though no actual Arduino hardware is used.

4.1.2 The Arduino language

Arduino has its own language, based on Wiring (Arduino 2019a). I would argue that it's not really a language, but an SDK, since the actual language is C++. In fact, even the Arduino website's Frequently Asked Questions say that "the Arduino language is merely a set of C/C++ functions that can be called from your code" (Arduino 2019c). In addition to this, there are also some preprocessor steps, after which the C++ code compiled by avr-g++, a compiler for AVR microcontrollers (Arduino 2019c).

4.1.3 Arduino IDE

The Arduino IDE is the default development environment for the Arduino platform (Arduino 2019a). The IDE is simple to use, and makes it very easy to configure and flash the compiled software onto a device. But, it has been criticized fairly widely for being a terrible IDE (Williams 2019).

Arduino also has an online IDE called Arduino Web Editor (Arduino 2019d), as well as recently introducing an early version of the Arduino Pro IDE, which is a more advanced version of their IDE, with many new features (Cipriani 2019). However, it's possible to use any other IDE by running some of the build process steps yourself (Arduino 2019e). This project was developed using VSCode and using the Arduino IDE only as a build tool.

4.1.4 Arduino libraries, tools, and community

The community and all of the existing libraries are the real reason why this project uses Arduino. The ESP32 has its own official development framework called ESP-IDF (Espressif Systems 2019b), which might even be a better choice for development on the ESP32.

But the vast selection of libraries available on the Arduino platform means that most sensors and other devices you may come across will probably already have a library available. This saves a lot of development time. Luckily, Espressif also provides tools for developing ESP32 software on the Arduino platform, called the “Arduino core for the ESP32” (Espressif Systems 2019c), which is a perfect middle-ground.

4.2 Espressif Systems

Espressif Systems is a semiconductor company based in Shanghai. They are focused on IoT solutions and are best known for their ESP8266 and ESP32 chips and modules. (Espressif Systems 2019d)

4.2.1 The inexpensive ESP8266

The ESP8266 module is fairly popular in the maker community, because it offers an easy way to add wireless Internet connectivity to an electronics project (Sachleen 2014). It has a microcontroller, an integrated Wi-Fi radio and antenna, flash memory, and other circuitry, which makes it a complete IoT solution, needing a minimum of only 7 external components (Espressif Systems 2019e).

The ESP8266 is available in multiple official modules from Espressif Systems (Espressif Systems 2019f), as well as modules offered by other companies. One of the most popular ones, the ESP-12F, can be bought for a little over 1 euro, or as a ready-to-use USB-powered development board for a little under 2 euros, including shipping (prices checked on 17.11.2019 from AliExpress).

4.2.2 Improvements introduced with the ESP32

The ESP8266 has a much more powerful processor than the microprocessors used in most Arduino devices. This, coupled with the how inexpensive it is, would make it an ideal choice for the project. However, the ESP8266 lacks flash encryption and secure boot (Espressif Systems 2019e), which means that one of the IoT devices could be modified, in order to perform attacks on the server. The ESP8266’s successor, the ESP32, adds these features. (Espressif Systems 2019a, 3)

The ESP32 has a more powerful, dual-core processor, which means that one of the cores can be dedicated to maintaining an encrypted Wi-Fi connection, without slowing down other functionality. The more powerful processor also allows for better security, since it can run more complex cryptographic algorithms. The ESP32 also adds Bluetooth connectivity, although it won’t be used for this thesis. (Espressif Systems 2019g)

The ESP32 is a bit more expensive than the ESP8266, but not significantly. The ESP8266 may still be more popular, but this is largely due to lower price. All of the extra features available in the ESP32 made it the obvious choice when thinking about the future plans for this project.

The ESP32 is available in many different modules and development boards, with either a single-core or dual-core processor, as well as different flash memory capacities and antenna configurations. The module that was chosen for this project is the ESP32-WROOM-32, with 4 megabytes of flash storage, and an integrated PCB antenna. (Espressif Systems 2019f)

More specifically, the prototype system will use a development board based on that module. The development board has all of the necessary circuitry to program the module, and to power it with a USB power supply. It also has a power LED, and an “activity” LED connected to one the digital pins.

Espressif IoT Development Framework (ESP-IDF)

The official development framework for the ESP32 is called ESP-IDF, short for Espressif IoT Development Framework (Espressif Systems 2019b). This project will be using the “Arduino core for the ESP32”, which allows using the ESP32 libraries straight from the Arduino IDE (Espressif Systems 2019c), along with other Arduino libraries (the benefits of which were discussed earlier).

Still, the ESP-IDF documentation will be referenced a lot during development, since the features and functions are essentially the same. Among these features are over-the-air (OTA) updates, running a file system, Wi-Fi connectivity, making HTTP requests, running an HTTP server, and much more. (Espressif Systems 2019b)

Comparison of hardware platforms

Table 3 shows a comparison of some of the hardware platforms that were considered for the system.

Table 3. Comparison of hardware platforms (Arduino 2019h; Raspberry Pi Foundation 2019a; 2019b; 2019c; Espressif Systems 2019a; 2019f).

Development board	**Arduino MKR1000**	**Raspberry Pi Zero W**	**Espressif ESP32-DevKitC (clone)**
Module/SoC	Atmel ATSAMW25	N/A	ESP32-WROOM-32
Processor	SAMD21 32-bit, single-core 48 MHz	BCM2835 64-bit, single-core 1 GHz	ESP32-D0WD 32-bit, dual-core up to 240 MHz
Runtime memory	32 KB SRAM	512 MB RAM	520 KB SRAM
Persistent memory	256 KB (Flash)	SD card, up to 256 GB	4 MB (Flash)
Wi-Fi	802.11 b/g/n	802.11 b/g/n	802.11 b/g/n
Bluetooth	-	Bluetooth 4.1, BLE	Bluetooth 4.1, BLE
Logic voltage	3.3 V	3.3 V	3.3 V
Digital I/O	8 pins	28 pins	34 pins
Hardware PWM	12 pins	4 pins	16 pins
Analog input	7 pins 8/10/12-bit ADC	Not supported	18 pins 12-bit ADC
Analog output	1 pin 10-bit DAC	Not supported	2 pins 8-pit DAC
Hardware UART	1 channel	1 channel	3 channels
Hardware SPI	1 channel	2 channels	3 channels
Hardware I²C	1 channel	1 channel	2 channels
Approximate cost (dev board)	31 € (price on 13.12.2019, store.arduino.cc)	11,5 € (price on 13.12.2019, ThePiHut.com)	9,0 € (price on 13.12.2019, Mouser.fi)
Approximate cost (module/SoC)	10,5 € (price on 13.12.2019, Mouser.fi)	N/A	3,42 € (price on 13.12.2019, Mouser.fi)

The table compares development boards, but a final device would likely use the microcontrollers and SoCs (system on chip) that are used the development boards. In addition to the hardware specifications, there are many other things to consider. Some of the most important factors are the ease of interfacing with sensors and implementing

Internet connectivity, the available SDKs and libraries, and the cost of the hardware. Since Internet connectivity is a core requirement, only platforms that have ready-to-use Wi-Fi connectivity were considered for the comparison.

As of writing, over 20 different Arduino boards exist (Arduino 2019g). Out of these, the Arduino MKR1000 and the Arduino Yún offer Wi-Fi connectivity (Arduino 2019h; 2019i). However, the Arduino Yún has been retired (Arduino 2019i), so it was left out of the comparison. The MKR1000 is based on the Atmel ATSAMW25 SoC (system on chip), which has a SAMD21 Cortex-M0+ 32-bit ARM microcontroller, a WINC1500 Wi-Fi module, and an ECC508 cryptographic module (Arduino 2019h).

The MKR1000 and the ATSAMW25 SoC it uses are fine alternatives to the ESP32, but when comparing the hardware specifications visible in **Table 3**, the ESP32 is more powerful and has more features in most areas. When comparing online prices as of writing, both the SAMW25 and the MKR1000 development board are almost twice as expensive as the ESP32 equivalents. Atmel is a respected company, but the ESP8266 and ESP32 also seem to have a much larger online developer community than the Arduino MKR1000.

For a project like this, the Raspberry Pi Zero W is the best suited out of the various Raspberry Pi boards available, and is surprisingly low-cost as far as development boards are concerned. When looking at the specs, the Raspberry Pi Zero W is the most powerful out of all of the presented options. However, the Raspberry Pi runs a full Linux operating system instead of an embedded program, so a direct performance comparison is difficult to do. Unlike the other choices, the Raspberry Pi isn't based on a ready-to-use module, so it would require much more hardware development to create a custom PCB for production, unless the board were used as-is.

The Raspberry Pi is also difficult to interface with sensors, since it has the worst GPIO (general purpose input/output) features in the comparison, for example having the fewest PWM outputs and having no analog inputs or outputs at all. Because of this, perhaps the best way to interface with sensors would be a “hybrid” solution using an external microcontroller. However, this would increase costs and system complexity.

The Raspberry Pi might the most flexible option out of the ones presented, since it runs a full Linux operating system. However, it is also likely to be the most expensive and would probably require the most development effort, since an external microcontroller may also be necessary. For certain use cases, like computer vision, the Raspberry Pi may be the best option.

4.3 Connecting devices to a microcontroller

In order to be able to get readings from the sensors, they will naturally need to be connected to the microcontroller. This chapter will go through all of the things to consider when doing this, like power delivery, and the various interfaces and protocols to communicate with the sensors.

4.3.1 Power delivery and device voltages

One of the first things to consider when choosing a sensor, are its power delivery requirements (current consumption), as well as the voltage ranges it works with. For example, if the project is battery-powered, using a sensor with a high current consumption may not make sense.

Similarly, using a sensor which has 3.3 volt logic with a 5 volt microcontroller will require using a logic level converter. This doesn’t make using the sensor impossible by any means, but if another sensor uses a compatible voltage for logic, it may be a better fit for the project.

Some devices may use different voltages for power delivery and for logic. This is because the same amount of power at a higher voltage will require less current. Higher currents will require thicker wires and PCB traces. Then again, lower voltage microcontrollers will generally be more power-efficient, and using multiple different voltages will require extra circuitry in order to provide multiple well-regulated voltages.

Since the ESP32 has already been decided as the microcontroller, the logic level voltage will be 3.3 volts (Espressif Systems 2019a). However, since USB power supplies are a cheap and ubiquitous solution for power delivery, 5 volt power will also be easily available. The development board used in the project requires a 5 volt power supply, and has an integrated 3.3 volt voltage regulator, although this regulator cannot deliver a huge amount of current. A similar power solution would likely be used for a production PCB, which would no longer use the development board.

4.3.2 Reading analog signals

One of the simplest ways to connect a sensor to a microcontroller is by reading an analog voltage. This is simple, and won't require a library. However, reading an analog voltage requires an analog-to-digital converter (ADC), of which there will be a limited number on any microcontroller. An ADC will also have a specific voltage range, and if this doesn't by coincidence match that of the sensor's output voltages, some supporting circuitry will also be needed.

Also, a sensor which provides only an analog voltage is likely to be a pretty "dumb" sensor. For example, an NTC or PTC thermistor (short for negative and positive temperature coefficient, respectively) can be used as a simple temperature sensor (Omega Engineering Inc. 2018), but the values from such components will be rather inaccurate without calibration. More advanced temperature sensors (like the one that will be introduced later) will have supporting circuitry, in order to give more accurate and reliable readings.

4.3.3 Reading digital signals

Another common method to read values from sensors is by reading a digital signal. Some sensors will use a completely custom protocol, but there are a few common standardized protocols, some of which will be introduced in the following chapters.

In general, one thing to note about reading digital signals is that they can be very time sensitive. Especially if the sensor uses some custom protocol, this can make development rather time consuming, if a good quality library isn't available. So, when researching possible sensors, it's always a big bonus if the sensor uses a standard protocol and/or if there's a library available.

4.3.4 I²C

One standard protocol to communicate with sensors and other integrated circuits (IC) is I²C. Short for Inter-Integrated Circuit, I²C is a synchronous, multi-master, multi-slave protocol. It uses only 2 wires for communication, a clock line (SCL) and a data line (SDA). (Horowitz & Hill 2015, 1034-1035)

The clock line is controlled by the master device, which sends a 7-bit address for the slave, followed by a single bit specifying the direction of the data (to the slave or from the slave). If the slave is sending data, it will control the data line, but will follow the master's clock. The SMBus protocol is closely related to I²C, but enforces stricter standards. (Horowitz & Hill 2015, 1034-1035)

I²C has multiple benefits, like only requiring 2 wires for data, clock, and addressing, so multiple devices can share the same 2 wires. Many other interfaces, like the SPI interface introduced in the next chapter, may require more wires and may not have any form of addressing, meaning that multiple devices may not be able to share the same bus. (Horowitz & Hill 2015, 1034-1035)

However, I²C has a lower bandwidth than many other protocols, so some other protocol may be preferred if a higher bandwidth is required. I²C also uses predefined 7-bit addresses, and while some devices allow selecting an alternative address by dedicated pins, this somewhat defeats the advantage of only requiring 2 wires. (Horowitz & Hill 2015, 1034-1035)

4.3.5 SPI

Another common protocol is the Serial Peripheral Interface (SPI). SPI uses 4 wires for communication between ICs: SS (slave select), SCLK (serial clock), MOSI (master out, slave in), and MISO (master in, slave out). Compared to I²C, SPI is a simpler protocol, despite using more wires. SPI is simpler, but also less structured than I²C, so it can potentially be a bit more difficult to work with, depending on the IC. (Horowitz & Hill 2015, 1032-1033)

SPI has higher bandwidth than I²C, which makes it common for ICs that require this (like SD cards or LCD control chips). However, SPI doesn't have any intrinsic addressing, so the same bus can be shared by fewer ICs, and each additional slave IC will require its own slave select wire. (Horowitz & Hill 2015, 1032-1033)

SPI is a bit "loosely" standardized, and sometimes the names SDI (serial data in) and SDO (serial data out) are used instead of the more unambiguous MOSI and MISO. SPI is usually full-duplex, meaning that it can send data in both directions simultaneously, since it has dedicated data in and and data out pins. A half-duplex, 3-wire version of SPI also exists, which uses just one pin for both incoming and outgoing data. This saves on the number of pins used, but hurts bandwidth and is a bit more complex to use. (Horowitz & Hill 2015, 1032-1033; 2015, 1082)

SPI isn’t used for the first prototype, although some of the sensors used in the project do support it. SPI may be used for later additions, like an SD card or LCD screen.

4.3.6 Serial ports and UART

Another digital interface for communication between integrated circuits is called UART, universal asynchronous receiver-transmitter (Horowitz & Hill 2015, 1039). Sometimes this is referred to as a COM port or simply a “serial port” (Horowitz & Hill 2015, 873). UART can use many different protocols, which can make communication difficult at times. When using UART for communication, the device’s datasheet is the ultimate source of information.

One advantage with UART is that it’s fairly easy to connect to a computer by using a serial-to-USB converter, which allows you to plug in a USB host at one end, and essentially gives you an RS-232 interface at the other end. Most microcontrollers include one or more serial ports, and they are one of the most common ways of programming them, including the ESP32. (Horowitz & Hill 2015, 873)

4.4 Choosing the sensors and other devices

Earlier in this chapter, some general principles were introduced, which should be kept in mind when choosing parts for an electronics project. Among these have been power requirements, logic level voltages, and the libraries available for the part (if libraries are necessary).

Some other things to keep in mind are the price and availability of the part. An outdated part which is no longer being manufactured may be much cheaper, and while choosing such a part may make sense in the short term, it may add extra work if the part needs to be replaced later on. The remaining stock of a deprecated part may also become much more expensive, if demand is still high. How common a given part is can also be visible in the amount of information that’s available online, since the more users a part has, the more people will talk about it online.

Different parts can also have very different features. The most commonly used parts may have been in use for decades, and new technology may have come along, which offers better features, accuracy, reliability, or cheaper prices. Choosing a part will often involve compromises, and there is rarely a part which is objectively “best”.

Adafruit Industries

Adafruit Industries is a company based in New York City, and focuses on making products for the maker community. The company employs over 100 people, and has been listed as one of the top USA manufacturing companies. (Adafruit 2019a)

Among their products, Adafruit manufactures easy to use modules of popular ICs. Like Arduino, Adafruit’s products are open-source hardware, and all of the software included with the products is also open-source. (Adafruit 2019b)

This means that even though no Adafruit products were used, the project can still make use of Adafruit’s libraries. The libraries are generally high-quality and fairly well maintained, which greatly reduces this project’s workload.

4.4.1 Worldsemi WS2812B intelligent RGB LED

The WS2812B is an intelligent RGB LED, manufactured by Worldsemi. It is “intelligent”, because multiple of these LEDs can be chained together, and they will be individually addressable. This allows using multiple individually addressable RGB LEDs, while only using a single digital pin on the microcontroller, as opposed to a minimum of 3 PWM (pulse-width modulation) capable pins for a single regular RGB LED. (Worldsemi 2018)

Use in the project

Physically, the system will be pretty minimal, but some form of feedback for the user was wanted. If there’s some problem in the system, like a lost Internet connection, it’s important that the user has some way of noticing this, apart from noticing that the past week’s sensor readings are missing. The activity LED on the ESP32 development board used for the prototype could be used for this, but an RGB LED can provide a wider range of feedback.

The WS2812B uses a custom control protocol, which has very strict timing requirements in order to work properly (Burgess 2013). This project will be using one of the many libraries developed by Adafruit, the Adafruit NeoPixel library (Adafruit 2019c). The library was developed for use with the NeoPixel line of products from Adafruit, which are based on the WS2812B (Burgess 2013).

4.4.2 Bosch BME280 temperature, humidity, and ambient air pressure sensor

The BME280 is a combined digital humidity, barometric pressure and temperature sensor manufactured by Bosch. It can be used in a wide range of operating conditions, and the internal temperature sensor provides compensation for the humidity and pressure measurements. (Bosch 2018)

The sensor works with a range of supply and logic voltages (up to 3.6 volts). It has a very low current consumption (under 4 µA when measuring humidity, pressure and temperature at 1 Hz). The sensor package is also very small, measuring at just 2.5x2.5x0.93 millimeters, although an easy to solder module will naturally be larger. (Bosch 2018)

Use in the project

The barometric pressure sensor won't really be used in this project, but the accurate humidity and temperature measurements will be desirable. Bosch also manufactures other similar sensors, including the more advanced BME680, which also includes Volatile Organic Compound (VOC) measurement (Bosch 2019), but the BME280 is cheaper, more readily available, and accurate enough for the project's purposes.

Temperature and humidity will be important measurements for the project. A store-bought, cheap indoor temperature and humidity sensor will be used to compare readings. Holding both sensors close to a heat source can be used to test different temperature ranges, and simply breathing on the sensors should be enough to dramatically raise humidity levels for a short time.

The sensor will be useful when comparing room air temperature and humidity between different seasons, as well as measuring humidity in a room like a bathroom, where humidity will rise dramatically in a short period of time. The barometric pressure readings will also be recorded, although they don't have a particular use case at this point.

The sensor has a range of supply and logic voltages, both up to 3.6 volts (Bosch 2019). Since the ESP32 works at 3.3 volts, this is the voltage that will be used with the sensor as well.

For digital interfaces, the sensor supports I²C, and both 4-wire and 3-wire SPI (Bosch 2019). This project will be using I²C, since another sensor also uses it, and the higher bandwidth of SPI isn't necessary. This sensor also has a high-quality library available, made by Adafruit (Adafruit 2019d).

4.4.3 AMS TSL2561 light intensity sensor

The TSL2561 is light intensity sensor manufactured by AMS, although the manufacturer calls it a "light-to-digital converter". The TSL2561 is designed particularly for measuring the lighting conditions when using a display panel, so the display's brightness can be adjusted to increase battery life. Despite this, the sensor can also be used for general light sensing applications. (AMS 2018)

Unlike something like a photoresistor, the TSL2561 is automatically calibrated, and gives accurate readings in lumens. The TSL2561 has 2 separate sensors, an infrared light sensor and an infrared+visible light sensor, so it can be used to measure both of these light spectra independently. The sensor has a high dynamic range (1 million to 1), so it can be used in both a dark room or in bright sunlight, and it automatically rejects 50/60 Hz lighting ripple, which can be caused by some electric lights. (AMS 2018)

The sensor package measures only 2.6x3.8 millimeters, and consumes only 0.75 mW when active (and less in sleep mode). The sensor supports a range of supply and logic voltages (up to 3.6 volts), and has an I²C interface. (AMS 2018)

Use in the project

The project will use the sensor's I²C interface, and will be used with a 3.3 volt supply voltage, and the same logic voltage. Adafruit has a library for the sensor (Adafruit 2019e). In order to accurately measure light intensity, the sensor should have a direct line-of-sight to the outside world, which should be taken into account when designing a case. The first few prototypes won't have a case at all, so this won't be an issue during the thesis.

Light sensing can be used as a roundabout way of sensing human presence in a room, since the room lights will generally be on when a room is occupied. This isn't an accurate method, since the same light intensity may be caused by natural light, and the room lights being on isn't necessarily an indicator of room occupancy. However, light intensity coupled with room temperature, humidity, and carbon dioxide increases could be used to at least estimate occupancy, while crossing less privacy borders, like when using computer vision to measure occupancy.

Another interesting use case is to sense natural light across seasons, and during the course of a day, if the approximate layout of the rooms and the position of windows is known. At least theoretically, it should be possible to measure how the easternmost rooms see more natural light in the morning, the southernmost rooms during the day, and the westernmost rooms in the evening. Couple this with a nice visualization, and it might be quite eye-catching.

4.4.4 Winsen MQ-135 air quality sensor

The MQ-135 is an "air quality gas sensor" manufactured by Zhengzhou Winsen Electronics Co., Ltd. The sensor works by measuring the conductivity of its tin-oxide element, which has a higher resistance in clean air. The sensor has a high sensitivity to ammonia gas, sulfide, benzene series steam, and can also monitor smoke and other toxic gases. (Winsen 2015)

Use in the project

The sensor is sold as an air quality sensor, but since it mostly measures toxic gases which are only common in some industrial/manufacturing applications, it may not be a good fit for this project. Nevertheless, the sensor is cheap, and may produce interesting results, so it was included in the lineup of sensors.

The sensor requires a supply voltage of 5 volts, because it has an internal heating element, in order to provide a steady temperature to provide accurate measurements. The sensor does not have a digital interface, and measurements are read by measuring an analog output voltage, which is nominally between 2 and 4 volts. (Winsen 2015)

4.4.5 Winsen MH-Z19B carbon dioxide sensor

Another sensor produced by Winsen, the MH-Z19B is a carbon dioxide (CO_2) sensor. The sensor uses the NDIR (non-dispersive infrared) principle to measure CO_2 concentration. The sensor has an internal heating element to provide the steady temperature required for accurate measurements. (Winsen 2016)

NDIR sensors

Normal CO_2 concentrations in air are measured in parts per million (ppm), so a very high-accuracy detection method is required (Edaphic Scientific 2019). One of these methods is called NDIR, short for non-dispersive infrared (Edaphic Scientific 2019), and it is used in sensors to detect various gases (Palidwar 2014).

An NDIR sensor shines an infrared light through a gas chamber, to an infrared detector. Right before the detector, there is a light spectrum filter, which only lets a specific wavelength of light through to the detector. (Palidwar 2014)

All elements and compounds absorb light at numerous wavelengths. Depending on the wavelength selected for the light filter, the detector will only detect a specific wavelength of light. If the wavelength is absorbed by a specific compound, and is not absorbed very much by other compounds, the concentration of the compound can be accurately measured up to a high accuracy. (Edaphic Scientific 2019)

For CO_2, the wavelength that's commonly used is 4.26 µm (Palidwar 2014). The datasheet for the MH-Z19B doesn't specify the wavelength it uses, but it is likely to be this wavelength.

Use in the project

The sensor has 3 ways of reading measurements: measuring an analog output voltage, measuring the duty-cycle of a PWM signal, or by using the sensor's UART interface (Winsen 2016). The UART interface provides the most features and the best accuracy out of these (Winsen 2016), so that will be used in the project. According to the manufacturer's datasheet (Winsen 2016), the UART interface seems to use a custom protocol, and while a few libraries exist, none of them appear to be particularly high-quality, so this sensor may require more custom code than the other parts in the project.

Despite the MH-Z19B being by far the most expensive sensor used in the project, a CO_2 sensor was one of the first ones that was selected. Since the project will primarily be used indoors, CO_2 concentration was considered one of the most important measurements to make, since CO_2 concentration is an important factor for indoor air quality (Lazović et al. 2015; Borino 2016). CO_2 concentration is also unlikely to fluctuate if a room is not occupied, so it can be used along with light sensing to tell if a room has people in it.

In order to test the sensor, breathing on it should be enough to temporarily raise CO_2 levels. A mixture of vinegar and baking soda can also be used to make CO_2 (Helmenstine 2019), and is a common science experiment for kids, which can be used to “magically” extinguish a candle (Craig 2018).

The absolute accuracy of the sensor will be very difficult to determine, but the CO_2 concentration in outdoor air should be fairly stable. With default settings, the MH-Z19B performs a zero-point calibration to 400 ppm CO_2 concentration every 24 hours (Winsen 2016).

5 DESIGN AND IMPLEMENTATION

In **Chapter 4**, the hardware and software to be used for the system were chosen. Based on those initial design decisions, this chapter will cover the internal design and implementation of the system. Later, in **Chapter 6**, the data gathered by the completed system will be inspected.

The basic idea of how the system will work has not changed since the beginning of the project: the IoT device reads sensor data, and sends it to a server app. A client application can then read and visualize the data from the server. The server app and the client app can do all of the heavy operations, like storing the data, hosting the client application for the users, and so on.

The IoT device will do the bare minimum it needs to do, which is essentially reading sensor data. Since it is an embedded device, and doing something like hosting the client application, or storing all of the gathered data would degrade performance or increase costs.

Taio, the name chosen for the system

The system's original name was "Domo", from "majordomo", meaning the head steward of a household (Merriam-Webster dictionary). This was too widely used on the Internet, so a new, unique name was needed. The name "Taio" was selected, from the Finnish word "taikoa", meaning "to do magic".

In Finnish, the word "taio" is used as a third person command. For example, "taio olohuoneen valot päälle" would essentially mean "conjure the living room lights off". A short name was wanted in case voice commands would be added in the future, and in Finnish, this would make for natural feeling and short voice commands, as opposed to something like "OK Google, turn the living room lights off".

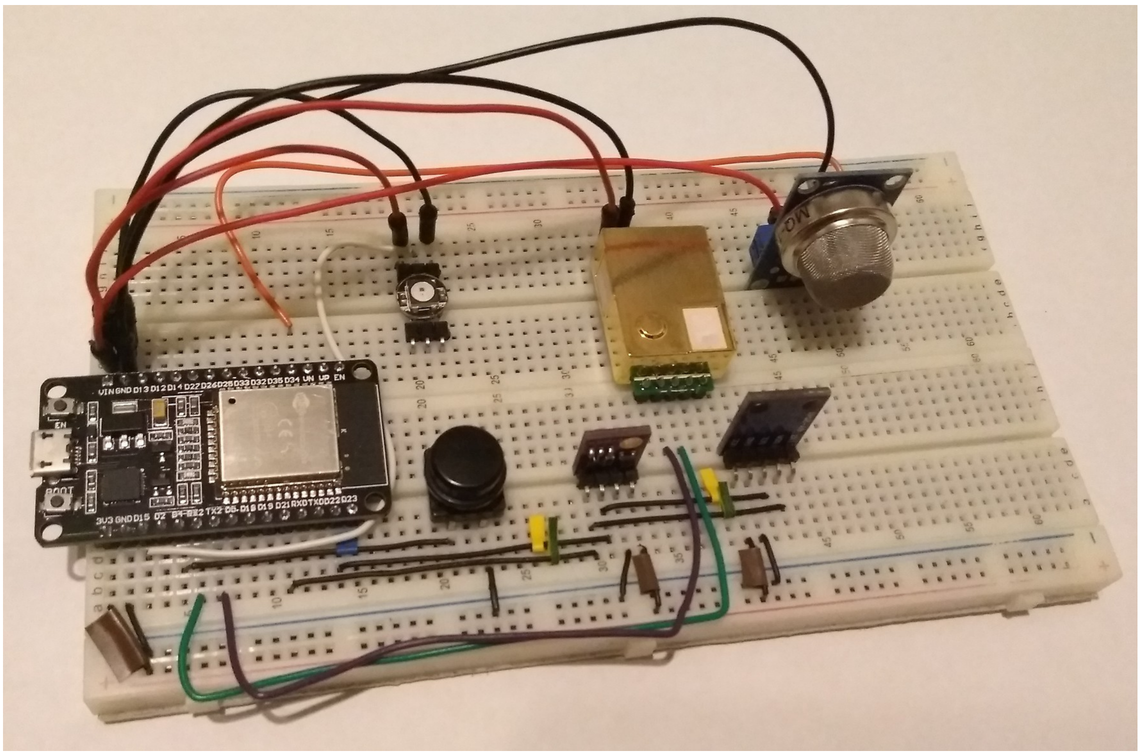
In English, "Taio" sounds like a name. Taio is pronounced almost the same in English and in Finnish (in English, there's an extra "u" sound at the end). So, the same name would work globally as well. Taio is just a working title, but it's helpful to have at least an internal name for the project, instead of referring to it as "the IoT system" or "the IoT project".

5.1 Hardware prototyping

Before writing the software can really be started, a hardware prototype must be built. It's impossible to write code for reading sensors, if the sensors are not attached to the microcontroller.

5.1.1 Breadboard prototype

For very early development, a "breadboard" prototype is extremely useful. A breadboard is a solderless prototyping board (Buckley 2018), which allows wiring up components by just pushing them into the rows of sockets on the breadboard. This way, the actual layout of the components doesn't matter much, as long as the wiring is correct. **Picture 4** shows a breadboard prototype of the device.



Picture 4. Breadboard prototype of the device.

For some use cases however, the solderless connections and long wires can add unwanted resistance, which can cause voltage drops and poor power delivery, which can cause components to behave erratically. This caused some problems when reading the analog voltages of the MQ-135 air quality sensor.

The breadboard also caused some power delivery problems for both the MQ-135 air quality sensor and the MHZ19 carbon dioxide sensor, which both have an internal heating element (Winsen 2015; Winsen 2016), and therefore draw much more power than the other sensors. These problems were fixed by running shorter wires directly from the power supply to the components, instead of using jump wires through the breadboard's power strips.

5.1.2 Perfboard prototype

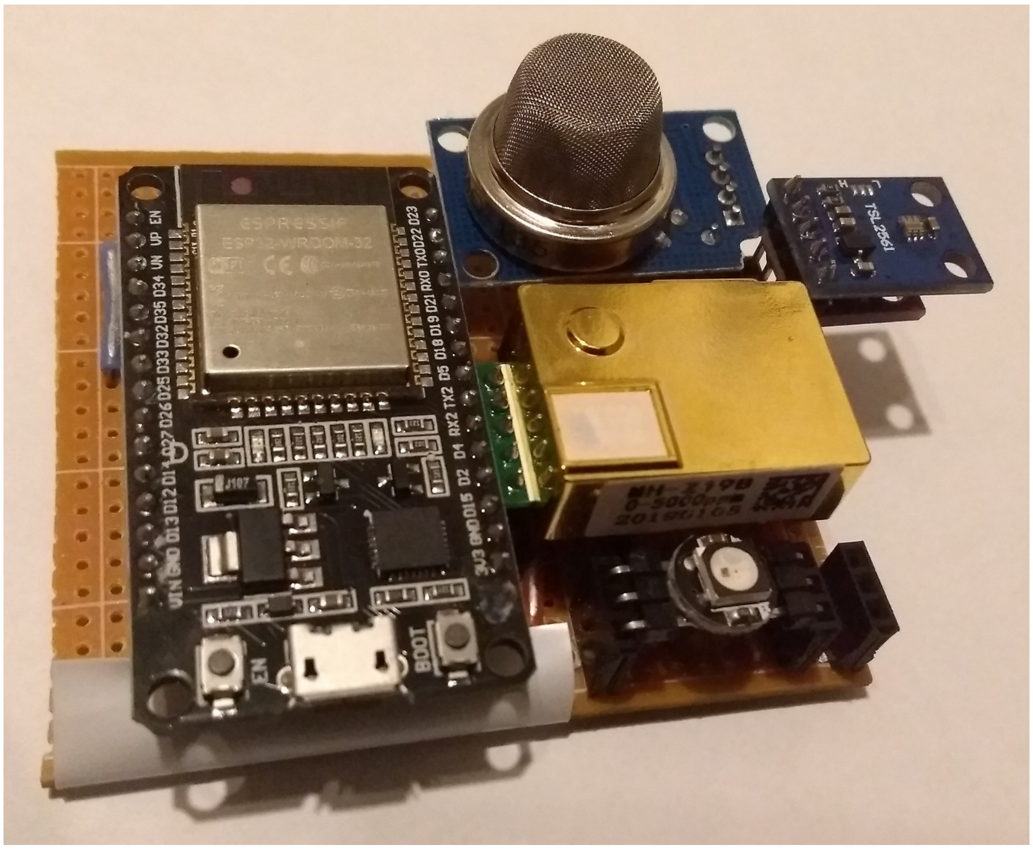
A breadboard is useful for initial prototyping, when figuring out the wiring of the components, but it can cause issues, like the inaccurate sensor readings and power delivery issues mentioned earlier. Breadboard circuits also take up much more room than a neatly organized circuit board, which is okay for one device, but becomes difficult to maintain for multiple devices.

However, designing a custom PCB (printed circuit board) takes a lot of time, and so does waiting for the PCBs to be manufactured (if you don't acid etch them yourself). It also may not make sense for prototyping, since a chosen sensor may still change, or more parts may be added.

Perfboards, also called *protoboards*, and also sold under the brand name *Veroboard* (Vero Technologies Ltd. 2019), are a perfect middle-ground. A perfboard (short for perforated board) has pre-drilled holes for component leads and wires, and either circular pads or rows of copper cladding for each row of holes. The kind with copper rows is often called stripboard. (Bunker 2015)

A perfboard is a fast and inexpensive way to create a prototype circuit board, without having to design and manufacture a custom PCB. Using a drill bit or sharp knife, it's easy to remove the copper cladding where needed, and you can make and solder a prototype in a couple of hours.

For the later stage prototype, designing and ordering 30 pieces of a custom PCB is much faster than cutting traces on 30 perfboards. But for now, for the first dozen devices, perfboard is a great prototyping tool. **Picture 5** shows a perfboard prototype of the device.



Picture 5. Perfboard prototype of the device.

The ESP32 and all sensors are slotted onto female headers on the perfboard. This makes it easy to swap faulty parts, and also to reuse the relatively expensive parts on later iterations of the device.

On the left side of the above picture, there’s the ESP32. On the bottom right, the WS2812B RGB LED module can be seen, soldered onto 90-degree male headers, which slot into female headers on the PCB. The golden colored sensor above that is the MH-Z19B CO_2 sensor. Above that, there’s the MQ-135 air quality sensor. Not visible in the picture, there’s a simple resistor voltage divider, in order to get the MQ-135’s analog output voltage to fall into the ESP32’s ADC voltage range.

On the far right of the above picture, hanging off the side of the circuit board, is a male header with the TSL2561 light sensor on top, and the BME280 on the same header. This could be done, because they both use the same supply voltage of 3.3 volts, and they both also use the I²C interface for communicating with the ESP32. The pin layout of the modules happened to allow putting them on the same header, which made for a neat and compact layout for the sensors. The light sensor is on top, since it would be shadowed by the other module, hindering light sensing.

Later, 4 more copies of the perfboard prototype were made. Later still, a different clone of the ESP32 development board was used, with a slightly different physical layout, and 3 more perfboard devices were made, now totalling 8 devices. If more are needed, a custom PCB should probably be designed, because cutting the traces on the perfboard is a bit laborious.

Power supply issues

Initially, the cheapest possible USB power supplies were used, since they seemed to meet the basic power requirements of the ESP32 and the sensors. However, it turned out that the ESP32 and some of the sensors need a fair amount of power, quickly.

The cheap power supplies stated that they could provide 1 amp of current at 5 volts, which is far above the current draw of the device. However, apparently these power supplies couldn’t respond to fast changes in current draw.

The voltage provided by the power supply likely dropped too low when the ESP32 or the sensors required more current. The power supply can probably provide the current and voltage that the manufacturer promises, but the power supplies couldn’t handle the fast fluctuations in current draw. This caused wireless connection issues, as well as issues getting readings from the devices. Changing to a higher quality power supply seemed to fix these issues.

5.2 Espressif IoT Development Framework (ESP-IDF)

After some closer design, the system will consist of 4 different codebases:

- **TaioDeviceFirmware:** the IoT device’s firmware. Its main job is to read the sensors, and to send the data to the server. It will also host TaioDeviceClient.

- **TaioDeviceClient:** a configuration app hosted by the firmware. This will be kept to the bare minimum, essentially only for configuring the Wi-Fi network the device connects to, and other necessary configuration.

- **TaioServer:** the server application, which will have an API for accepting data from the devices, and for allowing TaioClient to retrieve the data. The server app will also host TaioClient.

- **TaioClient:** the client application, which is used for all user interaction (after the initial device config, which is in TaioDeviceClient). The main feature is the visualization of the data retrieved from the server app.

In addition to the above, there will be some additional software on the server, for things like managing HTTPS certificates, and so on. But these are similar for any web server, and will not require any custom code, so they will not be discussed much.

5.3 TaioDeviceFirmware

The device's firmware will be reading data from the sensors, which have been selected for the system. As was described in **Chapter 4**, most of the sensors have existing libraries available, so the firmware can focus on the things that are unique for this system.

In addition to the sensor libraries, the firmware will be using many ESP32 libraries, which were also mentioned in **Chapter 4**. These libraries will do most of the difficult work when it comes to things like the Wi-Fi connection, checking HTTPS certificates, encryption, and so on.

WLAN configuration

In addition to reading the sensors, the device will also need to connect to a WLAN. In order to connect to the network, the device needs configuration for the network name (SSID) and passphrase. For initial testing, just hardcoding the values is fine, but eventually, the user will need some way to configure these, and since one of the values is a secret passphrase, they also need to be stored securely.

Thoughts about C++ on embedded devices

The firmware will be written in C++, although C++ on embedded devices is pretty close to C, or “C with Classes” as C++ was originally named (Stroustrup 1997, 10). This is because most embedded devices do not have enough memory or storage space to use the C++ standard libraries, which are one of the biggest advantages of using C++.

Without the standard libraries, you revert back to things like “regular” C strings (arrays of characters). I remember trying to learn C when I was first getting into programming (over 15 years ago), and getting stuck when I had to concatenate 2 strings. Concatenating strings is such a common thing, and really easy to do in most languages, but in C it requires creating a new array for the new string, and doing array copies. That’s the way all languages do it internally, but often it’s nice to not worry about things like this, and focus on the “domain problem”, and let the compiler optimize the low-level stuff. Luckily, among many other libraries, Arduino has its own string library (Arduino 2019f), although it’s not nearly as nice as the C++ standard string.

Still, many useful features of C++, like classes, streams, and templates will be available on embedded devices as well, as can be expected. For example, the project uses a custom-made templated logging function, which employs streams to easily convert sensor readings and other values into a printable format.

High-level description of the firmware’s functionality

The idea is to keep the device as “dumb” as possible, since an embedded device will always have rather limited processing power. All of the heavy processing will be done by TaioServer and TaioClient, so the device itself can just focus on reading the sensors.

The device polls the sensors once every 10 seconds. The readings are stored in runtime memory, and sent to the server once per minute. Originally, the sensors were polled once every second, but this simply produced so much data, that the poll rate had to be reduced. The data format, that the readings are stored in, was also made as minimal as possible, so that even months of data wouldn’t take too much storage space. The sensor readings are only sent once per minute, so that having many devices won’t produce too much load on the server.

In order to be able to connect to the server, the device needs to be able to connect to a Wi-Fi access point, and the configuration for said AP will also need to be stored. Once connected, the device will need to be able to make secure requests to the server, in order to report sensor readings. The implementation of these will be explained along with the libraries that were used.

5.3.1 The ESP32 libraries

ESP-IDF has a collection of classes/libraries which allow using its many features, like the Wi-Fi radio, running a web server, using a file system on the flash memory, and so on. The libraries are fairly high-level for embedded libraries, which makes them easy to use. The libraries in this chapter are not an exhaustive list of all libraries that were used, but introduces some of the central ESP32 libraries used in the project.

Non-volatile storage library (Preferences library)

The Preferences library is part of the Arduino core for the ESP32, and uses the ESP32’s non-volatile storage (NVS) library (Espressif Systems 2019c). The ESP32’s non-volatile storage can be used to store configuration, which will be stored across reboots and power outages. The library creates a partition for this purpose on the ESP32’s flash memory, and offers encryption as an optional feature. (Espressif Systems 2019h)

The NVS library works best for storing small values; string values are limited to a maximum of 4000 bytes, and the entire partition used by the NVS library is about 512 kilobytes in size (Espressif Systems 2019h). Larger values can use a filesystem library like SPIFFS, which is introduced later.

In the project, the NVS library will be used to store configuration, like SSIDs and passwords for access point configuration. The values are also kept stored when flashing a new program onto the ESP32, which was only discovered when testing the library. This is actually very useful, so that the SSID and password never need to be in the program’s code, despite not losing them when flashing. This avoids having to reconfigure the device after every flash.

WiFi library

The WiFi library is used to control the ESP32’s Wi-Fi radio. The Wi-Fi radio can be used in 1 of 3 different modes: STA (station), AP (soft access point), or AP-STA (simultaneous station and soft access point). (Espressif Systems 2019i)

The “station” mode means that the device acts as client device for another AP. The AP mode allows the ESP32 to act as a Wi-Fi access point, so it can accept connections from other wireless devices. The third mode allows acting both as a station and an AP at the same time. (Espressif Systems 2019i)

The term “soft” AP in the ESP-IDF documentation appears to refer to the fact that the ESP32 often acts as a sort of “intermediate” AP, since it has to connect to some other AP in order to offer any Internet connectivity to its own clients.

When running as an AP, the WiFi library is used to configure the SSID, password, and other configuration of the AP that the ESP32 runs. When running as a station, the WiFi library is used to set the similar configuration for the AP where the ESP32 connects to. The library is also used to set the hostname of the ESP32, and various other configuration relating to Wi-Fi connectivity, as well as scanning for available APs. (Espressif Systems 2019i)

The Wi-Fi configuration (SSIDs, passwords, etc.) are stored using the non-volatile storage library introduced earlier. The device’s user can set this configuration by connecting to the device’s AP and using TaioDeviceClient, which will be introduced in a later chapter.

WiFiClientSecure library

The WiFiClientSecure class implements support for secure connections using TLS. The class inherits from WiFiClient, which implements the actual connections, which are secured by WiFiClientSecure. (Espressif Systems 2019j)

The library allows a few different methods to establish a secure connection (Espressif Systems 2019j), but the method this project will use is to use the root CA’s certificate. The server will use a certificate issued by Let’s Encrypt, whose certificates are cross-signed by IdenTrust, which allows them to be trusted by all major browsers (Let’s

Encrypt 2019d). This doesn’t really matter for the ESP32, since the root certificate will have to be added on to the device anyway, but having the certificates be cross-signed by a widely trusted CA allows most clients to automatically trust the server’s certificate.

WiFiClientSecure helps greatly with the rather complex TLS stuff, but the HTTP(S) client itself is rather simple. The source code and examples from the Arduino core for the ESP32 (Espressif Systems 2019j) show that the client opens a socket, and HTTP requests are done as “raw” requests (printed line-by-line through the socket).

This is why **Chapter 2** went into relatively high detail about how HTTP requests work at a base level, because TaioDeviceFirmware does not use a high-level HTTP client library, but instead has to manually build HTTP requests. Naturally, utility functions were built for this purpose, so that there’s an extra level of abstraction to make programming easier.

SPIFFS library

The Serial Peripheral Interface Flash File System (RandomNerdTutorials.com 2019), or SPIFFS, is a filesystem for the flash memory on the ESP32. SPIFFS is a pretty simple filesystem, and doesn’t have real directory support, for example. (Espressif Systems 2019k)

Nevertheless, being able to store files on the ESP32 is very useful. For example, it’s much cleaner to use actual files rather than having very long strings in the code. One common use case for SPIFFS is to store files, which a web server on the ESP32 can host. This will be used to host TaioDeviceClient.

SPIFFS works by flashing a binary image onto a partition on the ESP32’s flash memory (Espressif Systems 2019k). Espressif provides a Python script for doing this (Espressif Systems 2019k), and a tool for the Arduino IDE also exists for creating the SPIFFS image and flashing it (RandomNerdTutorials.com 2019).

SPIFFS can also be used to store configuration files (RandomNerdTutorials.com 2019), but the NVS system is better suited for this, since it stores values across device flashes, whereas configuration stored in SPIFFS would be cleared if other files in the filesystem are flashed on to the device.

WebServer library

The WebServer library allows the device to act as an HTTP server. It can be used as a file server, or to manually handle HTTP requests, in order to build something like a REST API. (LastMinuteEngineers.com 2019)

The device will host the files for TaioDeviceClient. The web server will accept connections from clients connected to the ESP32’s access point.

This is currently the least secure part of the system, since the WebServer library acts as an HTTP server, and does not implement secure connections with TLS. Securing the connections would also require a certificate to be stored on the device, which can be problematic, as was discussed at the end of **Chapter 3**.

However, the web server will only be used to configure the device, and after configuration, the web server should not be a security risk. At a later stage, the web server may need to be re-implemented as an HTTPS server, or the configuration could use some other form of encryption.

5.3.2 Other libraries

The other libraries used in the project were already discussed in **Chapter 4**, as part of the hardware and software research for the project. This chapter will shortly discuss how the libraries were used.

The Adafruit NeoPixel library (WS2812B RGB LED)

Adafruit sells a family of products called NeoPixel, which uses the WS2812B intelligent RGB LED. The library works with any WS2812B LED, including modules used in this project. (Burgess 2013)

The WS2812B’s DIN (data in) pin was connected to the GPIO15 pin of the ESP32. When the device is operating normally, the RGB LED will “breath”, by choosing a random color and slowly transitioning to it, and then repeating the same process.

If there’s an issue, like a lost wireless connection, the RGB LED will be off, and the activity LED will blink in a pattern depending on the issue. Slow blinking was used for a lost connection, and fast blinking was used for other issues (like a sensor issue).

The Adafruit BME280 library

Readings from the Bosch BME280 sensor were read using Adafruit’s BME280 library. The sensor and library support both I²C and SPI (Bosch 2018; Adafruit 2019d), but this project will use I²C. This was done because I²C uses fewer wires, another sensor also uses I²C, and the higher bandwidth of SPI was not necessary for the sensor. SPI may be used later for other sensors, or for something like an SD card or LCD display.

Initially, the sensor seemed to report very high temperatures, which was thought to be a software issue. It turned out that this was due to the sensor being close to the MH-Z19B and MQ-135 sensors, which both have an internal heating element (Winsen 2015; Winsen 2016). So, the sensor was reporting correct temperature values, but was too close to heat sources on the device. This was solved by using an approximately 15 cm long cable to place the BME280 further away from the heat sources.

The Adafruit TSL2561 library

Another Adafruit library was used to get readings from the TSL2561 sensor. Adafruit’s modules seem to use a different I²C address, because the default address used in the library was not connecting to the sensor. The library allowed setting the address to the one that was used by the module used in the project, after which the sensor started working.

The sensor was tested in different lighting conditions. This was initially done by simply covering the sensor from light, and by shining a bright light onto it. The accuracy of the actual lumen readings from the sensor were not tested, due to lack of access to a suitable reference sensor. However, relative readings are enough for the project, so accurate absolute lumen readings were not needed.

Winsen MQ-135 sensor

As was discussed in **Chapter 4**, the MQ-135 works by outputting an analog voltage, so it doesn't require the use of a library. The output voltage range of the MQ-135 is 2-4 volts (Winsen 2015). A resistor voltage divider was used to divide the the sensor's output voltage into half, so that the maximum output voltage became 2 volts, which is under the maximum of the ADC on the ESP32 (Espressif 2019a).

Winsen MH-Z19B sensor

A high-quality library, with a suitable open-source license, was not found for the MH-Z19B. Because of this, a custom class was written to communicate with the MH-Z19B sensor. Getting readings from this sensor was a bit arduous, when compared to the other sensors, which worked almost immediately, thanks to the libraries that were available.

Writing the custom interfacing code took some time, but was relatively simple, due to the protocol being well explained in the datasheet (Winsen 2016). However, the datasheet had some typographical errors, and low-quality English translations (for example, using "pls" instead of "please"), making it a bit difficult to read (Winsen 2016).

Like was planned in **Chapter 4**, the sensor was tested by breathing on it, and even by making a vinegar and baking soda mixture to create CO_2. This seemed to work well; the CO_2 from the vinegar and baking soda mixture easily got the sensor to report 5000 ppm, which is the maximum according to the datasheet (Winsen 2016).

5.4 TaioDeviceClient

TaioDeviceClient is a simple web app, written in AngularDart. AngularDart works with HTML and CSS, and compiles the Dart code into JavaScript (Google 2019a). This allows hosting it similar to a regular website, meaning that it can hosted on the ESP32, without running a server-side language on the device.

As was explained earlier in this chapter, the ESP32's web server hosts TaioDeviceClient, and it can be accessed by connecting to the ESP32's access point.

The access point is secured with WPA2 and uses a password that's stored in the ESP32's non-volatile memory. The access point's default password is set by the code of TaioDeviceFirmware, if a password is not found in memory. The password can be changed inside TaioDeviceClient, since the default password is the same on all of the devices.

TaioDeviceClient also has a basic username and password login, which is required before allowing the user to change or see any configuration values. The login username and password can also be changed once logged in.

These measures provide passable security for the device, when it is not being configured (i.e. it's not possible to see or change the configuration values without having the correct credentials). However, if the device is being configured by an authorized party, the traffic is potentially visible to a man-in-the-middle attack, since the device runs an unsecure HTTP web server. This will need to be improved in the future, but works well enough for the time being.

In addition to the other configuration values mentioned earlier, a client token for sending data to TaioServer can also be set with TaioDeviceClient. In fact, setting the token is necessary before the device will be able to send data to the server. In a future version, this could be automated by connecting the device to the user's account on the server. This way, the user would be able to manage their devices from within TaioClient, instead of having to individually connect to each device's AP to configure it.

The actual application is rather simple. It starts with a login form, after which it presents a form for changing configuration values. The existing passwords are not displayed in the app, and are never sent across the network, but they can be changed by an authenticated user, which means that they could be eavesdropped when setting the passwords.

5.5 TaioServer

TaioServer, the server application, is responsible for receiving sensor readings from the devices (from TaioDeviceFirmware) and storing them on the server. In addition to receiving and storing the sensor data from the devices, the server app will also host TaioClient, and provide it with the sensor data it needs for visualization.

The server app was written in Dart, like TaioDeviceClient and TaioClient, but unlike those apps, the server app is not compiled into JavaScript. Instead, Dart server apps run on the Dart VM (virtual machine) (Google 2019b).

The devices authenticate with a client token, which the user sets by connecting to the device's AP and logging into TaioDeviceClient. Without the token, or with an invalid token, the server app will ignore the data that was sent to the server.

The tokens are JWT strings, which were introduced in **Chapter 2**. Since the tokens use JWT, the token itself can contain the permissions it has as JSON data, so the server app doesn't need to store a separate list of tokens and the permissions attached to them.

The JWT is signed with a private key only known by the server app, so the token's validity can be checked, in order to ensure that it has not been tampered with. In addition to checking the JWT signature, the server app has a list of all valid tokens. This list is used so that a token can be invalidated without having to change the private key, which would invalidate all of the tokens at once.

In the current version, the client tokens are manually generated, and added into the server app and into the device with TaioDeviceClient. A future version of the system would make this more automated by associating the devices and their tokens with the user's account, as was explained earlier.

Server hardware

TaioServer was run on a Raspberry Pi 2. This was chosen simply because there was a spare one available. The server software could have also been run on a cloud service like Amazon AWS or Microsoft Azure, and ESP-IDF even has libraries and examples available for these platforms (Espressif Systems 2019l).

A number of different cloud servers were tested, but all of the free trial instances were quite poor in performance. Where the backend runs has very little importance at this point, so a readily available solution (the Raspberry Pi 2) was chosen, instead of setting up and paying for a cloud server. After there are more than about a dozen or so devices connecting to it, TaioServer may be moved to a cloud service, so that the home network isn't taxed too heavily.

Nginx HTTPS proxy

The server app runs an HTTP server (not encrypted), but encryption can be added separately, since it works on a separate layer of the OSI model, as was explained in **Chapter 2**. This can be done with a proxy server, which listens for connections from the outside network (the Internet) and forwards them to the server application (Nginx Inc. 2019). Using a proxy server, we don't have to rely on the server app to handle encryption, so an industry-standard application like Apache or Nginx can be used instead. This allows for more configuration and arguably better security.

Let's Encrypt and Certbot

In order to have HTTPS connections, we need an SSL/TLS certificate, as was explained in **Chapter 3**. In the same chapter, it was explained that the certificates are issued by a certificate authority (CA), which is trusted by the clients connecting to the server.

One way to obtain a certificate is by creating a "self-signed" certificate, by creating your own internal CA. This is problematic, because an internal CA will not be trusted by a client, unless you manually add it to the list of trusted authorities. (Nginx Inc. 2019)

Another common way is to purchase a certificate from a trusted certificate authority (Nginx Inc. 2019), but these can be expensive. Fortunately, there are certificate authorities, who offer free certificates. Among these is Let's Encrypt, which is free and automated, and is provided by the Internet Security Research Group (ISRG). (Let's Encrypt 2019a)

Let's Encrypt recommends using Certbot, which uses the ACME protocol to automate certificate issuance (Let's Encrypt 2019b). Let's Encrypt issues certificates with a 90-day lifetime (Let's Encrypt 2019c), and Certbot automatically renews certificates every 60 days (EFF 2019), so the certificate is always kept valid and up-to-date.

The IoT device only stores the root certificate's signature, which it can use to validate our server's certificate. This way, the 90-day lifetime of the certificate doesn't cause the problems mentioned at the end of **Chapter 3**, without having to use a possibly insecure self-signed certificate, since the root certificate never expires.

5.6 TaioClient

TaioClient is the part of the system that the users will be interacting with the most. TaioClient is responsible for visualizing the sensor data, which it retrieves from the server app. Like TaioDeviceClient, this app is also written in AngularDart, which compiles Dart code into JavaScript (Google 2019a).

The current version of TaioClient does not even feature user login, only a very bare-bones visualization. The user interface and overall user experience will also require some more work before it can be used by actual end users. Likewise, configuring the devices must be made easier in a future version.

A future version of TaioClient will also be used for device configuration. This will allow the user to manage all of their devices in one place, instead of having to connect to each device individually. Naturally, Wi-Fi configuration will remain in TaioDeviceClient, since the device cannot fetch configuration from the server app without being able to connect to the Internet first.

Data visualization

TaioClient's currently only feature is data visualization, which is also rather simple. The app is written in AngularDart, and while Dart is growing in popularity, it is still relatively little used when compared to JavaScript (Ramel 2019). However, Dart apps are able to use JavaScript libraries (Google 2019c).

After researching a number of different JavaScript data visualization libraries, a library called Chart.js was chosen for the project. Chart.js advertises itself as a simple yet flexible JavaScript charting library. One of the main reasons it was chosen was because it has beautiful animations and allows a high level of interaction with the charts. It uses modern HTML5 technologies, including HTML5 canvas for rendering. (Chart.js 2019a)

Among the top contenders was another library called D3.js, as well as a library called C3.js, which is based on D3.js (Tanaka 2019). C3.js and D3.js seemed more flexible and customizable, but Chart.js seemed to be more beginner-friendly, with easy to follow examples in their documentation (Chart.js 2019b).

And since nothing forces the app to use just one of these libraries, a future version may even use different libraries for different visualizations. The charts in the next chapter were visualized using Chart.js.

6 ANALYZING THE GATHERED DATA

Once the device and software were both fully functional, data could finally be gathered. I wanted to make this it’s own chapter after the implementation, but I ran a bit short on time for gathering and analyzing the data for the thesis, due to this project not being a main focus in the company.

It would have been ideal to gather good quality data for 1-2 months, so changes in weather could have been observed in indoor temperature and air quality. Data was gathered for multiple months, but quality could be better. For now, the gathered data is enough for a proof of concept, but it cannot be seriously analyzed.

The devices will be kept running indefinitely in the author’s home and office, and seeing even seasonal changes will be interesting. Some outdoor sensors would have also been interesting to get readings from, but there wasn’t time to make a weatherproof case.

A weatherproof case would have needed ventilation holes for air circulation, and an external temperature sensor, since some of the sensors have heating elements, so temperature readings would have been wildly incorrect inside of the case. The light sensor would have also needed a window on the case, which should be kept clear of debris and snow.

There’s much more data to comb through, and this was only a small part of it. This chapter presents some interesting findings from the data, although it is difficult to do any analysis that goes more than skin-deep.

Missing data

Some of the devices have been running for as long as 7 months, some for a much shorter time. Most of the devices have not had CO_2 measurements, because it was the most expensive out of all of the sensors, and wasn’t purchased for every device. The CO_2 sensors also had issues communicating with the microcontroller. The issues were solved with a device reboot, but were left unnoticed, sometimes months at a time, because the data gathering wasn’t closely monitored, and there was no automated alert system. Because of this, a lot of the CO_2 measurements are missing, even on the devices where CO_2 sensors were connected.

For some of the devices, the data has large gaps, because some devices had connection issues, as described earlier. Likewise, the server was down from time to time (quite rarely), and couldn’t receive measurements from the devices.

The devices can’t store more than about 150 datapoints (about 25 minutes of measurements), so some of the data was simply lost, since the devices couldn’t store the data if it wasn’t sent to the server in a timely manner. A later version of the device could use an SD card or flash memory chip to store more data, to handle connection issues and server outages.

The main reason why external storage was not done for the prototype device in this thesis was the complexity of implementing such a system, since using external data storage would have required encryption, in order to maintain the wanted security level of the system. Using external storage would have also increased costs, and would have required new parts and a new circuit board layout, which would have extended the hardware development time, and pushed back the beginning of the data gathering.

Interesting and uninteresting data

The Winsen MQ-135 air quality sensor measurements were largely useless. Since the sensor is designed to measure specific toxic gases as listed in the datasheet (Winsen 2015), it could not really be used to measure regular indoor air quality, as the readings were too stable to do any interesting analysis. This was suspected in the beginning of the project, but as was stated in **Chapter 4**, the sensor was cheap, so it was included in the prototype in case it would have produced any interesting results.

Similarly, the barometric air pressure readings from the Bosch BME280 sensor were simply too stable for indoor air to produce any interesting results. The issues for these measurements were known since the beginning, but should be restated here, to explain why they are not presented.

Therefore, the only sensor readings that are interesting enough to be analyzed for the entire time the devices have been running, are temperature, humidity, and light intensity. CO_2 measurements, where they were available, also gave some results that are interesting enough to analyze.

Even the usefulness of the light intensity measurements could be better. Because the device didn't have a case, the light sensor wasn't pointed in any regular manner across different devices. This would have been better, since the sensor is fairly directional. Because of this, light intensity measurements across different devices probably cannot be directly compared, but relative measurements during different time periods for the same device can still be analyzed.

Most of the sensor choices made for the prototype were for cost reasons, and some of the sensors that were chosen proved to be inappropriate for typical home and office environments, where conditions are fairly stable. This thesis deals with these proof of concept measurements, but the particular sensors chosen for the prototype weren't of much importance, since the main purpose of the prototype was to develop the framework of the IoT system itself. With the system complete, it is easy to switch sensors and add features to the next version of the device.

Big data

Each device reads its sensors once every 10 seconds. During the time they have been running, the devices have gathered millions of data points, despite the data from some of the devices having large sections missing.

Each data point has at least temperature, humidity and light intensity measurements. Each data point is between 54-64 bytes (depending on which measurements were included). Despite a single data point being so small, due to 8 devices collecting millions of data points over the course of months, there is a total of about 400 megabytes of data.

Having such a large amount of data has its own challenges. Since the measurements were taken once every 10 seconds, trying to view just a week's worth of data from one device becomes pretty resource intensive. The data also appears to be very noisy, since there are natural outlier measurements in the physical world. For example, calculating averages, minimums, and maximums over the period of 1 hour was much more practical and useful, than viewing the "raw" data. This was done for all of the data presented in this chapter.

Understanding humidity measurements

Air humidity is most often reported as relative humidity, which is a percentage from 0 to 100 %. 0 % relative humidity means that the air that has no water vapour at all, which is mostly impossible in regular conditions. 100 % relative humidity means that the air has reached its saturation point, and water can no longer evaporate, causing any extra water to remain as liquid. (Chandler 2001)

Relative humidity is a useful measurement, since it is impossible to have a relative humidity below 0 % or above 100 %. According to Vaisala (2013), it is in fact impossible to even reach 100 % in an unpressurized system. High in the atmosphere, high humidity creates clouds and the possibility of rain. Humans are also sensitive to relative humidity, because the higher the relative humidity, the less sweat can evaporate from your skin into the air. This is why higher humidity will feel hotter and more uncomfortable, because your body can’t keep cool as efficiently. (Chandler 2001)

However, relative humidity can also be problematic, since it depends on the temperature of the gas, whose humidity is being measured. The warmer the air is, the more water can evaporate. This means that the same absolute amount of water vapour in air will translate to a lower relative humidity as temperature rises, since the saturation point rises. (Vaisala 2013)

This can be confusing, because relative humidity falls as temperature rises, despite the actual amount of water vapour in the air staying the same. In reality, the only thing changing is the air temperature, which causes the maximum amount of water vapour that the air can hold to also increase.

This is why morning dew exists; the colder air during the night causes water vapour to condense from the air, which mostly evaporates back into the warmer air during the day. This is also why indoor air heating “dries” the air; the absolute amount of water vapour doesn’t necessarily change, but relative humidity becomes lower due to the higher temperature. The cold air during the winter can hold less water vapour, so winter air is drier than summer air, but indoor heating doesn’t really dry the air (unless the heating system actually replaces indoor air with drier outdoor air).

Absolute humidity can be calculated from relative humidity and temperature. The formulas are a bit too complex to properly explain given the scope of this thesis, because they contain a lot of constants, like the “critical” temperature and pressure of the gas in question (air, in this case). The formulas that were used for the calculation were from a white paper by Vaisala, a Finnish sensor manufacturer. The unit for absolute humidity is usually g/m^3, since absolute humidity is measured as the amount of water vapour (in grams) in 1 cubic metre of air. (Vaisala 2013)

6.1 Data from a home environment

The data in this subchapter are from some selected time periods from the author’s home. The household has 2 adults, and 2 medium sized dogs (who also contribute to the readings). The house was built in the 1940s, and is in a small wooded area, which would have been interesting for an outdoor sensor, since there’s likely very little air pollution from motor vehicles.

There were a total of 4 devices, but 2 of these were of particular interest, at least to test the sensors:

- Monitoring the CO_2 level and temperature in the office, because these factors are believed to affect productivity. Too much CO_2 or a high temperature can make you feel uncomfortable or groggy.

- Monitoring moisture levels in the bathroom, mostly as a proof of concept for the moisture sensor.

Monitoring the CO_2 level and temperature was also planned for the bedroom, but this device was right at the edge of the Wi-Fi network, causing it to work most of the time, but sometimes losing the connection altogether.

Initially, this was thought to be another power supply problem, which was also seen with the other devices, and was thought to be fixed, since the device began to successfully send data to the server. The gaps in the data caused by the intermittent connection losses was only noticed when data visualization was started, and the gaps in the data made analysis too difficult. Still, this device provided interesting info for the project, since it determined that a future version should include external storage to cover intermittent connection losses.

The fourth and final device was in the living room, which is a larger room, causing the readings to be much more stable than the other rooms, and weren't as interesting to analyze.

Appendix 1 is a nice illustration of how relative humidity depends on temperature. The readings are from the author's home office from the first 4 days of August 2019. The windows were kept open during the night to let cool outdoor air in, and were closed during the day to keep cool indoor air in. This caused large fluctuations in temperature (almost 10 degrees Celsius). The "raw" values were calculated into the hourly averages, minimums, and maximums visible in the figure.

Appendix 1 shows how temperature falls during the nights, and rises during the days. The relative humidity moves in the opposite direction, being highest during the nights and lowest during the days. The purple lines at the bottom of the figure shows the absolute humidity, which remains very stable, despite the large changes in temperature and relative humidity.

Appendix 2 shows the same time period from the bathroom. Here, the absolute humidity has clear spikes as well, since the actual amount of water vapour in the air rises during showers. Relative humidity is also much higher. The temperature also rises a small amount during showers. **Appendix 3** shows all of August from the same bathroom. There's a conspicuous flat part in the readings, which is caused by the house being unoccupied during that week.

Appendix 4 has a similar flat section. These readings are from the home office. The difference is less noticeable, since there are no sharp humidity and temperature spikes, which are caused by taking showers in **Appendix 3**. The room simply being unoccupied shows a clear change in the temperature and humidity fluctuations. In **Appendix 4**, it seems to take a longer time for the fluctuations to return back to their previous levels.

Appendix 5 and **Appendix 6** show the same time period as **Appendix 3** and **Appendix 4**, from the home office. **Appendix 5** shows CO_2 readings, where too, the week that the house was unoccupied is clearly visible. The base CO_2 levels are very high, because the CO_2 sensors were not properly calibrated. Calibration should have been done before data collection, but the base CO_2 levels can be assumed to be around 400 ppm according the datasheet, and this is the default calibration level for the sensor (Winsen 2016).

Appendix 6 shows light readings, but somewhat surprisingly, this figure doesn’t show any clear difference between the house being occupied or unoccupied. It could be that there was enough natural light, that the additional light from electric lights isn’t very visible in the chart. **Appendix 7** is similar to **Appendix 6**, but only shows the minimum amount of light recorded every hour. This again, shows a clear difference during the week that the house was unoccupied.

Light readings from the living room during the same time period do not show a similar clear difference. This is most likely because the living room has more and larger windows, which offer more natural light. The differences would likely be much more visible during the winter, but unfortunately the house wasn’t empty for a week during the winter to get comparable readings.

As one last example from a home environment is **Appendix 8**, which shows the same time period as the previous figures, but from the living room. It is the largest room in the house, so occupancy causes much smaller temperature and humidity fluctuations. Still, the week that the house was unoccupied is visible in the figure, although much less clearly as for the other rooms.

6.2 Data from an office environment

This data was gathered from offices at the Turku Game Hub (Hive). Devices were placed into 3 rooms:

- **Indium Technology’s office**: a small, often over-occupied office, which can cause air quality issues. However, the door is mostly kept open, so there’s relatively good air circulation.

- **Mitale’s office**: a bigger office, but has more people. By average office standards, may still be slightly over-occupied.

- **Conference room**: a small conference room. Unlike an office, a conference room is mostly empty, and the air conditioning is usually set higher than in a regular office, since when a conference room is occupied, it has more people than an office.

Offices are a good control group, since they’re usually empty during the weekends. This allows for “base readings” without any people, so it is much easier to see how much the readings change based on occupancy.

Appendix 9 shows a week of light intensity and CO_2 measurements from the conference room. It can clearly be seen that every time the conference room is used, the light intensity and CO_2 readings display a sharp rise. Near the left side of the figure, a small gap can be seen. This is due to missing data, which was likely caused by a lost Internet connection.

The data starts on a Monday and ends on a Sunday, and the conference room doesn't appear to have been used during the weekend. Unfortunately, actual occupancy was not logged during the same time period. It would have been nice to know if the room's occupancy was higher during Wednesday, which shows a clearly higher CO_2 level than the other days. However, without knowing the occupancy, it's pure speculation to say if this is the case, or if it's due to poor sensor accuracy, or something else.

Appendix 10 shows the same week's light intensity and CO_2 levels in a small office. Here too, light and CO_2 measurements rise in tandem, and the weekend shows base level readings. Interestingly, Friday seems to have been a slightly shorter work day.

The base levels of CO_2 are clearly higher than in the conference room, but this is likely due to the CO_2 sensors not being calibrated. Despite this, there are clearly longer periods of higher CO_2 levels than in the conference room.

This office has south-facing windows, and seems to get a fair amount of natural light. This is visible in the chart, because the amount of light increases during the day, as the sun shines in more through the window. Eventually, there's a sharp drop in the amount of light, because the office's window is near a wall which blocks sunlight later in the day. If this is the correct explanation for the values seen here, it seems strange that the weekend shows such low values, but the window blinds may have been left closed for the weekend, and the electric lights don't provide a base light level for natural light to increase.

Appendix 11 shows the same week and same office, but shows temperature and humidity readings. The office has air conditioning, which clearly causes much lower humidity levels than in the home environment (without air conditioning) in the earlier figures. Despite air conditioning, temperatures rise by a couple of degrees when the room is occupied. During the weekend, the air conditioning appears to be turned off, because humidity levels rise at the end of the chart.

The larger office shows very similar measurements, but one thing of interest is that this office has north-facing windows, and therefore much less natural light. This shows as a clear difference in light levels in **Appendix 12**, although direct value comparisons shouldn’t be made, since the light sensors weren’t strategically placed or aimed.

The difference in CO_2 levels is interesting, when comparing Figure 14 and Figure 12. Although the absolute CO_2 level readings can’t be trusted, because the sensors weren’t calibrated, it’s worth noting that the difference between the base and peak CO_2 levels is higher in the larger office than in the smaller office. This could be because the smaller office’s door is usually kept open throughout the day, giving better air circulation in the smaller office, even though the larger office has more air volume per occupant.

7 CONCLUSION

Overall, I would consider the prototype developed in this thesis to be a success. This system has been a personal “passion project”, and was often shadowed by customer projects, which pay the bills in our company. This system may eventually become a commercial product, but for the time being, our company has had to focus on where the income comes from currently.

As was described in the introduction, this thesis only covers the initial prototype of the IoT system, and gathering and analyzing sensor data was only a proof of concept for the system. The ultimate goal is to develop a system which can be used to develop various different IoT product ideas, and this sensor system was never intended to be the final product, only a simple example of what such a system can be used for. Therefore, most of the issues that were faced, like some of the sensors being bad choices, don’t really matter, since this is only the first prototype. Issues were expected, and will be fixed in future versions of the system.

7.1 Issues faced during the project

As can be expected for a prototype, not everything went as planned. This being a side project instead of my (or our company’s) main focus meant that this project was often not worked on for long periods of time. The project took much longer than was estimated, despite the scope of the prototype staying roughly the same as it originally was (and even reducing in scope a little).

There were various other issues as well, like wireless connection issues and power delivery problems. I started out using the cheapest possible power sources, but had to upgrade to better ones, because the wireless radio requires a steady supply of power, in order to maintain a reliable connection. Even after upgrading the power supply, there were still some connection issues on some the devices, but these were caused by my home’s wireless network, which didn’t reach the whole house, causing connection problems on my smartphone as well.

The spotty connections caused some large gaps in the gathered data for a few of the devices, since the device has limited memory, and cannot store more than a few hours of data. In a future version, this could be solved by adding some extra flash memory onto the device, or maybe even an SD card, which are relatively simple to interface with a microcontroller. In order to maintain the wanted security level, this would require encryption of the external storage.

7.2 Choice of sensors

When it comes to the choice of sensors, most were a great fit for the project. The only real exception was the Winsen MQ-135, which, despite being a decent and inexpensive sensor, was a bad choice for this project, since it is better suited for specific industrial/manufacturing applications. This was known since the beginning, but the sensor was included with a “why not” mindframe.

A future version of the project may use the BME680, which measures VOC (volatile organic compound) concentrations in air (Bosch 2019), instead of the toxic gases detected by the MQ-135. The BME680 would also replace the BME280, and may give more accurate temperature readings. However, the BME680 is quite a bit more expensive and somewhat less common than the BME280, so it would raise the cost of the device.

7.3 Future plans

Before user testing, the device will still need over-the-air updates, since “manually” updating the devices would be too much additional work. This was initially planned to be done during the thesis, but was dropped to save time.

Also, the software will still need additional development, especially when it comes to the usability of the web application. The current app was used to make the diagrams used in this thesis, but is lacking many features, and is not very user-friendly.

The system has a lot of potential, and the next features that will be developed are going to relate to controlling things (lights, power outlets, etc.) in addition to just measuring the environment. This will naturally require new hardware, like relays and opto-isolators to control mains electricity. It will also require rather large changes on the software side of the system, like implementing WebSocket or similar real-time connections, so that the server app can send commands to the device.

These “control” use cases are actually the original motivation behind the project. I was not very interested in the actual measurements from the device, but more about developing a platform for IoT development, which was definitely achieved. This is why **Chapter 6** is not very long, because the focus of the project was about the development of the platform (**Chapter 4** and **Chapter 5**), and the actual measurement results were rather inconsequential.

Depending on how far the project will be developed, it may also require PCB design and manufacturing, designing and manufacturing enclosures for the device, getting electrical and safety certifications, and many other things. I’ve never done any of these, but I’m looking forward to learning all of it. The project taught me a lot, and will continue to do so in the future.

REFERENCES

Adafruit 2019a. About Us. Referenced 18.11.2019 https://www.adafruit.com/about

Adafruit 2019b. Frequently Asked Questions. Referenced 18.11.2019 https://www.adafruit.com/faq

Adafruit 2019c. Adafruit NeoPixel Library. Referenced 03.05.2019 https://github.com/adafruit/Adafruit_NeoPixel

Adafruit 2019d. Adafruit BME280 Library. Referenced 03.05.2019 https://github.com/adafruit/Adafruit_BME280_Library

Adafruit 2019e. Adafruit TSL2561 Library. Referenced 03.05.2019 https://github.com/adafruit/TSL2561-Arduino-Library

AMS 2018. TSL2561 datasheet. Version 1-01. Referenced 03.05.2019 https://ams.com/documents/20143/36005/TSL2561_DS000110_3-00.pdf/18a41097-2035-4333-c70e-bfa544c0a98b

Arduino 2019a. What is Arduino? Referenced 03.05.2019 https://www.arduino.cc/en/guide/introduction

Arduino 2019b. Trademarks and Copyright Notice. Referenced 03.05.2019 https://www.arduino.cc/en/Main/CopyrightNotice

Arduino 2019c. Frequently Asked Questions - Can I program the Arduino in C? Referenced 05.10.2019 https://www.arduino.cc/en/main/FAQ#toc13

Arduino 2019d. Getting Started with Arduino Web Editor on Various Platforms. Referenced 05.10.2019 https://create.arduino.cc/projecthub/Arduino_Genuino/getting-started-with-arduino-web-editor-on-various-platforms-4b3e4a

Arduino 2019e. Frequently Asked Questions - Can I use a different IDE to program the Arduino board? Referenced 05.10.2019 https://www.arduino.cc/en/main/FAQ#toc14

Arduino 2019f. String library. Refererenced 18.11.2019 https://www.arduino.cc/en/Reference/StringLibrary

Arduino 2019g. Compare board specs. Referenced 13.12.2019 https://www.arduino.cc/en/products/compare

Arduino 2019h. Arduino Yún. Referenced 13.12.2019 https://store.arduino.cc/arduino-yun

Arduino 2019i. Arduino MKR1000. Referenced 13.12.2019 https://store.arduino.cc/arduino-mkr1000-wifi

Atwood, J. 2005. Checksums and Hashes. Referenced 16.06.2019 https://blog.codinghorror.com/checksums-and-hashes/

Atwood, J. 2008. XML: The Angle Bracket Tax. Referenced 03.05.2019 https://blog.codinghorror.com/xml-the-angle-bracket-tax/

Berners-Lee, T.; Masinter, L. & McCahill, M. 1994. RFC 1738 – Uniform Resource Locator (URL). Referenced 02.05.2019 https://tools.ietf.org/html/rfc1738

Bondy, B.R. & community 2010. PUT vs. POST in REST. Stack Overflow. Referenced 13.06.2019 https://stackoverflow.com/questions/630453/put-vs-post-in-rest

Borino, S. 2016. Carbon Dioxide Detection and Indoor Air Quality Control. Referenced 18.11.2019 https://ohsonline.com/articles/2016/04/01/carbon-dioxide-detection-and-indoor-air-quality-control.aspx

Bosch 2018. BME280 – Data sheet. Document revision 1.6. Referenced 03.05.2019 https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST-BME280-DS002.pdf

Bosch 2019. BME680. Referenced 18.11.2019 https://www.bosch-sensortec.com/bst/products/all_products/bme680

Bray, T. 2014. RFC 7159 – The JavaScript Object Notation (JSON) Data Interchange Format. Referenced 02.05.2019 https://tools.ietf.org/html/rfc7159

Buckley, I. 2018. What Is a Breadboard and How Does It Work? A Quick Crash Course. Referenced 18.11.2019 https://www.makeuseof.com/tag/what-is-breadboard/

Bunker, J. 2015. How and When to Use Protoboard. Referenced 18.11.2019 https://makezine.com/2015/10/15/how-and-when-to-use-protoboard/

Burgess, M. 2018. What is the Internet of Things? WIRED explains. WIRED. Referenced 02.05.2019 https://www.wired.co.uk/article/internet-of-things-what-is-explained-iot

Burgess, P. 2013. The Magic of NeoPixels. Referenced 17.11.2019 https://learn.adafruit.com/adafruit-neopixel-uberguide

Chandler, N. 2001. What Is Relative Humidity and How Does it Affect How I Feel Outside? Referenced 27.11.2019 https://science.howstuffworks.com/nature/climate-weather/atmospheric/question651.htm

Chart.js 2019. Chart.js. Referenced 21.11.2019 https://www.chartjs.org/

Cipriani, L. 2019. Arduino Pro IDE (alpha preview) with advanced features. Referenced 31.10.2019 https://blog.arduino.cc/2019/10/18/arduino-pro-ide-alpha-preview-with-advanced-features/

Craig, A. 2018. How to Make Carbon Dioxide. Referenced 18.11.2019 https://sciencing.com/make-carbon-dioxide-6532065.html

Edaphic Scientific 2019. NDIR explained. Referenced 18.11.2019 https://www.edaphic.com.au/knowledge-base/articles/gas-articles/ndir-explained/

EFF 2019. About Certbot. Referenced 17.11.2019 https://certbot.eff.org/about/

Espressif Systems 2018a. ESP32 Technical Reference Manual. Version 4.0. Referenced 16.06.2019 https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf

Espressif Systems 2018b. ESP8266EX Datasheet. Version 6.0. Referenced 02.05.2019 https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf

Espressif Systems 2019a. ESP32 Series Datasheet. Version 3.0. Referenced 02.05.2019 https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

Espressif Systems 2019b. ESP-IDF Programming Guide. Referenced 17.11.2019 https://docs.espressif.com/projects/esp-idf/en/latest/

Espressif Systems 2019c. Arduino core for the ESP32. Referenced 03.05.2019 https://github.com/espressif/arduino-esp32

Espressif Systems 2019d. Who We Are. Referenced 16.11.2019 https://www.espressif.com/en/company/about-us/who-we-are

Espressif Systems 2019e. ESP8266 Overview. Referenced 17.11.2019 https://www.espressif.com/en/products/hardware/esp8266ex/overview

Espressif Systems 2019f. Modules. Referenced 17.11.2019 https://www.espressif.com/en/products/hardware/modules

Espressif Systems 2019g. Espressif Announces the Launch of ESP32 Cloud on Chip and Funding by Fosun Group. Referenced 17.11.2019 https://www.espressif.com/en/media_overview/news/espressif-announces-launch-esp32-cloud-chip-and-funding-fosun-group

Espressif Systems 2019h. Non-volatile storage library. ESP-IDF Programming Guide. Referenced 19.11.2019 https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/storage/nvs_flash.html

Espressif Systems 2019i. Wi-Fi. ESP-IDF Programming Guide. Referenced 19.11.2019 https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/network/esp_wifi.html

Espressif Systems 2019j. WiFiClientSecure. ESP-IDF Programming Guide. Referenced 19.11.2019 https://github.com/espressif/arduino-esp32/tree/master/libraries/WiFiClientSecure

Espressif Systems 2019k. SPIFFS Filesystem. ESP-IDF Programming Guide. Referenced 19.11.2019 https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/storage/spiffs.html

Espressif Systems 2019l. Cloud Frameworks. ESP-IDF Programming Guide. Referenced 19.11.2019 https://docs.espressif.com/projects/esp-idf/en/latest/libraries-and-frameworks/cloud-frameworks.html

Fall, K.R. & Stevens, W.R. 2012. TCP/IP Illustrated, Volume 1 - The Protocols. Upper Saddle River: Addison-Wesley.

Forbes 2018. The Next Industrial Revolution Is Rising In Japan. Referenced 22.5.2019 https://www.forbes.com/sites/japan/2018/05/21/the-next-industrial-revolution-is-rising-in-japan/

Friedl, S. 2005. An Illustrated Guide to Cryptographic Hashes. Referenced 15.12.2019 http://www.unixwiz.net/techtips/iguide-crypto-hashes.html

Giatec Scientific Inc. 2018. Monitor Your Concrete Slab with a Push of a Button. Referenced 22.5.2019 https://www.giatecscientific.com/education/want-to-check-on-that-concrete-slab-you-just-poured-now-theres-a-smartphone-app/

Goggi, C. 2014. Wi-Fi glossary -71 terms you need to know. Referenced 19.11.2019 https://techtalk.gfi.com/wi-fi-glossary-71-terms-you-need-to-know/

Google 2019a. Deplyoment. AngularDart. Referenced 19.11.2019 https://angulardart.dev/guide/deployment

Google 2019b. Write command-line apps. Dart. Referenced 21.11.2019 https://dart.dev/tutorials/server/cmdline

Google 2019c. JavaScript interoperability. Dart. Referenced 21.11.2019 https://dart.dev/web/js-interop

Gourley, D. & Totty, B. 2002. HTTP: The Definitive Guide. Sebastopol: O'Reilly.

Grusin, M., 2010. How to Read a Datasheet. Referenced 14.09.2019 https://www.sparkfun.com/tutorials/223

Han Vinck, A.J. 2011. Introduction to public key cryptography. Referenced 16.06.2019 https://en.wikipedia.org/wiki/File:Diffie-Hellman_Key_Exchange.svg

Hawkins, M. 2018. Raspberry Pi Power Consumption Data. Referenced 13.12.2019 https://www.raspberrypi-spy.co.uk/2018/11/raspberry-pi-power-consumption-data/

Helmenstine, A. M. 2019. Equation for the Reaction Between Baking Soda and Vinegar. Referenced 18.11.2019 https://www.thoughtco.com/equation-for-the-reaction-of-baking-soda-and-vinegar-604043

Horowitz, P. & Hill W. 2015. The Art of Electronics. 3rd Edition. Cambridge: Cambridge University Press.

Jones, M.; Bradley, J. & Sakimura, N. 2015. RFC 7519 – JSON Web Token (JWT). Referenced 02.05.2019 https://tools.ietf.org/html/rfc7519

Kastrenakes, J. 2018. Wi-Fi now has version numbers, and Wi-Fi 6 comes out next year. The Verge. Referenced 02.05.2019 https://www.theverge.com/2018/10/3/17926212/wifi-6-version-numbers-announced

Kennedy, J.; Jacobs, M. & Satran, M. 2018. Cipher Suites in TLS/SSL. Referenced 16.06.2019 https://docs.microsoft.com/en-au/windows/desktop/SecAuthN/cipher-suites-in-schannel

LastMinuteEngineers.com 2019. Create A Simple ESP32 Web Server In Arduino IDE. Referenced 19.11.2019 https://lastminuteengineers.com/creating-esp32-web-server-arduino-ide/

Lazović, I. M.; Stevanović, Ź. M.; Jovaśević-Stojanović, M. V.; Źivković, M. M. & Banjac M. J. 2015. Impact of CO2 concentration on indoor air quality and correlation with relative humidity and indoor air temperature in school buildings, Serbia. Referenced 18.11.2019 https://www.researchgate.net/publication/283526039_Impact_of_CO2_concentration_on_indoor_air_quality_and_correlation_with_relative_humidity_and_indoor_air_temperature_in_school_buildings_Serbia

Let's Encrypt 2019a. About Let's Encrypt. Referenced 17.11.2019 https://letsencrypt.org/about/

Let's Encrypt 2019b. Getting Started. Referenced 17.11.2019 https://letsencrypt.org/getting-started/

Let's Encrypt 2019c. Why ninety-day lifetimes for certificates?. Referenced 17.11.2019 https://letsencrypt.org/2015/11/09/why-90-days.html

Let's Encrypt 2019d. Chain of Trust. Referenced 24.11.2019 https://letsencrypt.org/certificates/

Let's Encrypt 2019e. How it Works. Referenced 24.11.2019 https://letsencrypt.org/how-it-works/

Lyons, J. 2012. Meet Cryptography. Referenced 15.11.2019 http://practicalcryptography.com/

Meeker, H. 2017. Open source licensing: What every technologist should know. Referenced 15.8.2019 https://opensource.com/article/17/9/open-source-licensing

Merriam-Webster dictionary. Definition of "majordomo". Referenced 16.06.2019 https://www.merriam-webster.com/dictionary/majordomo

Nginx Inc. 2019. Securing HTTP Traffic to Upstream Servers. Referenced 17.11.2019 https://docs.nginx.com/nginx/admin-guide/security-controls/securing-http-traffic-upstream/

Omega Engineering Inc. 2018. What is a Thermistor and how does it work?. Referenced 17.11.2019 https://www.omega.com/en-us/resources/thermistor

Paar, C. & Pelzl, J. 2010. Understanding Cryptography – A Textbook for Students and Practitioners. Berlin: Springer.

Palidwar, J. 2014. Optical Filters Open Up New Uses for MWIR, LWIR Systems. Referenced 18.11.2019 https://www.photonics.com/a56392

Pecht, M., 2013. The Counterfeit Electronics Problem. Referenced 14.09.2019 https://www.researchgate.net/publication/276494674_The_Counterfeit_Electronics_Problem

Pogue, D. 2012. What Wi-Fi Stands for – and Other Wireless Questions Answered. Scientific American. Referenced: 02.05.2019 https://www.scientificamerican.com/article/pogue-what-wifi-stands-for-other-wireless-questions-answered/

Ramel, D. 2019. Dart Cracks IEEE Spectrum Programming Language Popularity List. ADTmag. Referenced 21.11.2019 https://adtmag.com/articles/2019/09/18/ieee-spectrum-ranking.aspx

RandomNerdTutorials.com 2019. Install ESP32 Filesystem Uploader in Arduino IDE. Referenced 19.11.2019 https://randomnerdtutorials.com/install-esp32-filesystem-uploader-arduino-ide/

Raspberry Pi Foundation 2019a. Raspberry Pi Zero W. Referenced 13.12.2019 https://www.raspberrypi.org/products/raspberry-pi-zero-w/

Raspberry Pi Foundation 2019b. BCM2835. Referenced 13.12.2019 https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/README.md

Raspberry Pi Foundation 2019c. SD cards. Referenced 13.12.2019 https://www.raspberrypi.org/documentation/installation/sd-cards.md

RESTfulAPI.net 2017. What is REST. Referenced 03.05.2019 https://restfulapi.net/

Richardson, L. & Amundsen, M. 2013. RESTful Web APIs. Sebastopol: O'Reilly Media Inc.

Ristić, I. 2017. Bulletproof SSL and TLS. 2017 Revision. London: Feisty Duck Limited.

Russell, A.L. 2013. OSI: The Internet That Wasn't. Referenced 14.12.2019 https://spectrum.ieee.org/tech-history/cyberspace/osi-the-internet-that-wasnt

Sachleen, S. 2014. Answer to "How can I connect to an Arduino using WiFi?". Referenced 05.10.2019 https://arduino.stackexchange.com/a/436

Stroustrup, B. 1997. The C++ Programming Language. Third Edition. Reading: Addison-Wesley.

Tanaka, M. 2019. C3.js | D3-based reusable chart library. Referended 21.11.2019 https://c3js.org/

Tulevaisuuden älykkäät oppimisympäristöt 2019. The New Era of Learning. Referenced 02.05.2019 https://www.oppimisenuusiaika.fi/the-new-era-of-learning/

Vaisala 2013. Humidity Conversion Formulas - Calculation formulas for humidity. Referenced 27.11.2019 https://www.hatchability.com/Vaisala.pdf

Vero Technologies Ltd. 2019. Veroboards. Referenced 18.11.2019 https://www.verotl.com/circuitboards/veroboards

Wallace, B. 2018. The Dangerous State of IoT security. Hacker Noon. Referenced 02.05.2019 https://hackernoon.com/the-dangerous-state-of-iot-security-e12d27552145

Williams, A. 2019. The Arduino IDE Finally Grows Up. Referenced 31.10.2019 https://hackaday.com/2019/10/21/the-arduino-ide-finally-grows-up/

Winsen 2015. MQ135 datasheet. Version 1.4. Referenced 03.05.2019 https://www.winsen-sensor.com/d/files/PDF/Semiconductor%20Gas%20Sensor/MQ135%20(Ver1.4)%20-%20Manual.pdf

Winsen 2016. MH-Z19B datasheet. Version 1.0. Referenced 03.05.2018 https://www.winsen-sensor.com/d/files/infrared-gas-sensor/mh-z19b-co2-ver1_0.pdf

WireShark.com 2019. Online Network Glossary. Referenced 19.11.2019 https://wireshark.com/wifi-terminology.html

Worldsemi 2018. WS2812B-V4 datasheet. Version 1.0. Referenced 03.05.2018 http://www.world-semi.com/Certifications/WS2812B.html

Temperature and humidity readings from a home office in the beginning of August 2019

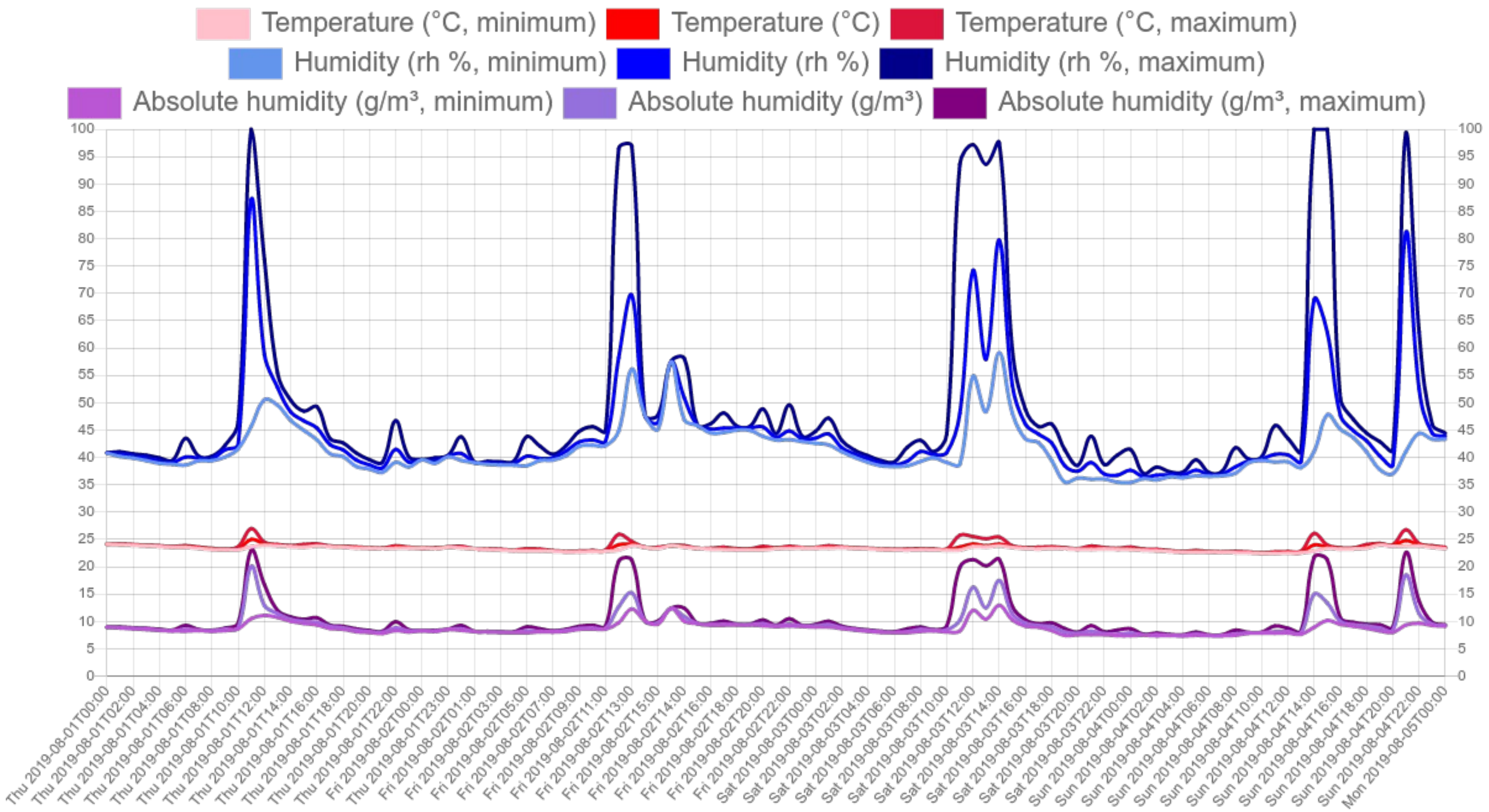


Figure 3. Temperature and humidity readings from a home office in the beginning of August 2019.

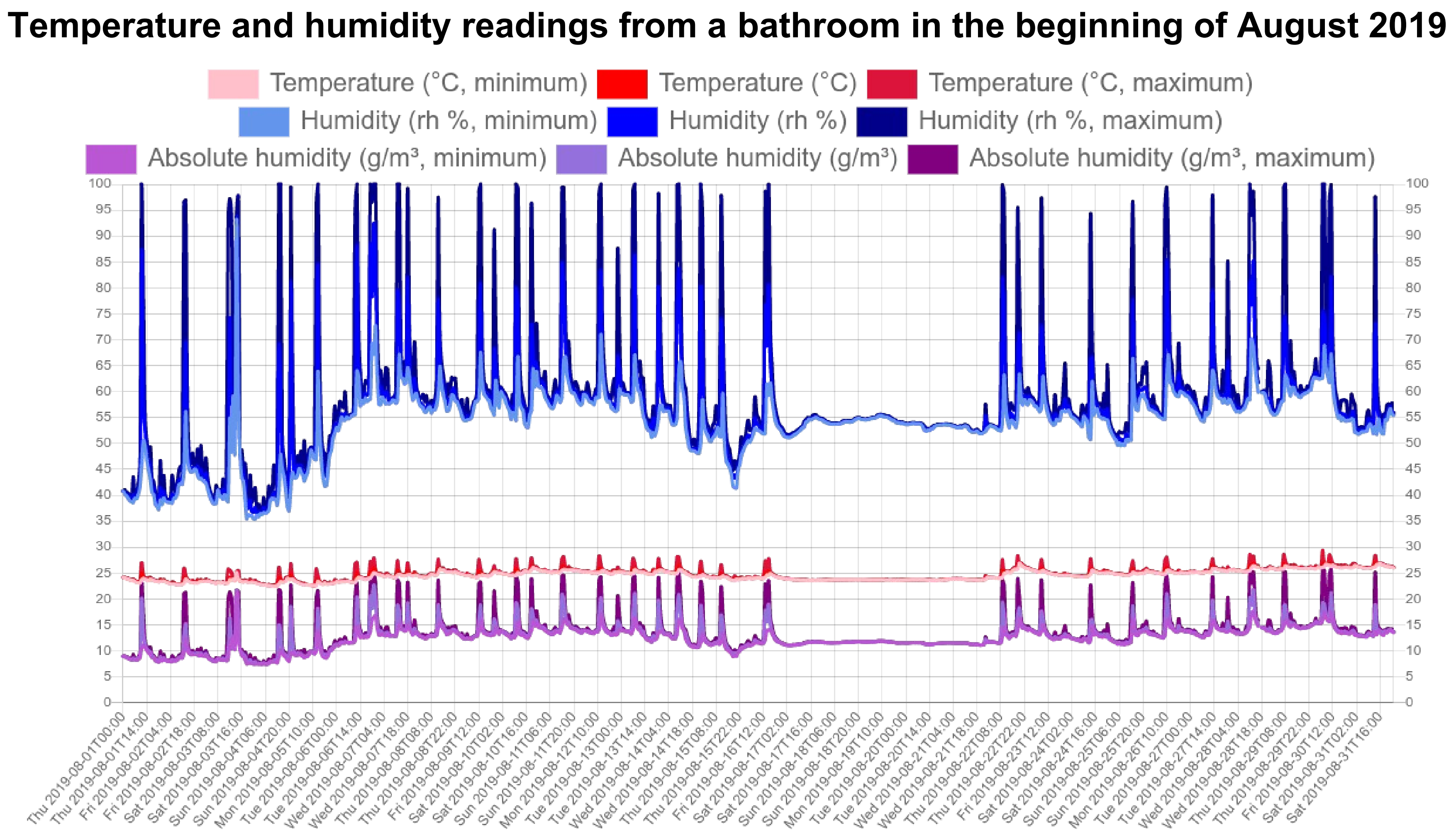


Figure 4. Temperature and humidity readings from a bathroom during the beginning of August 2019.

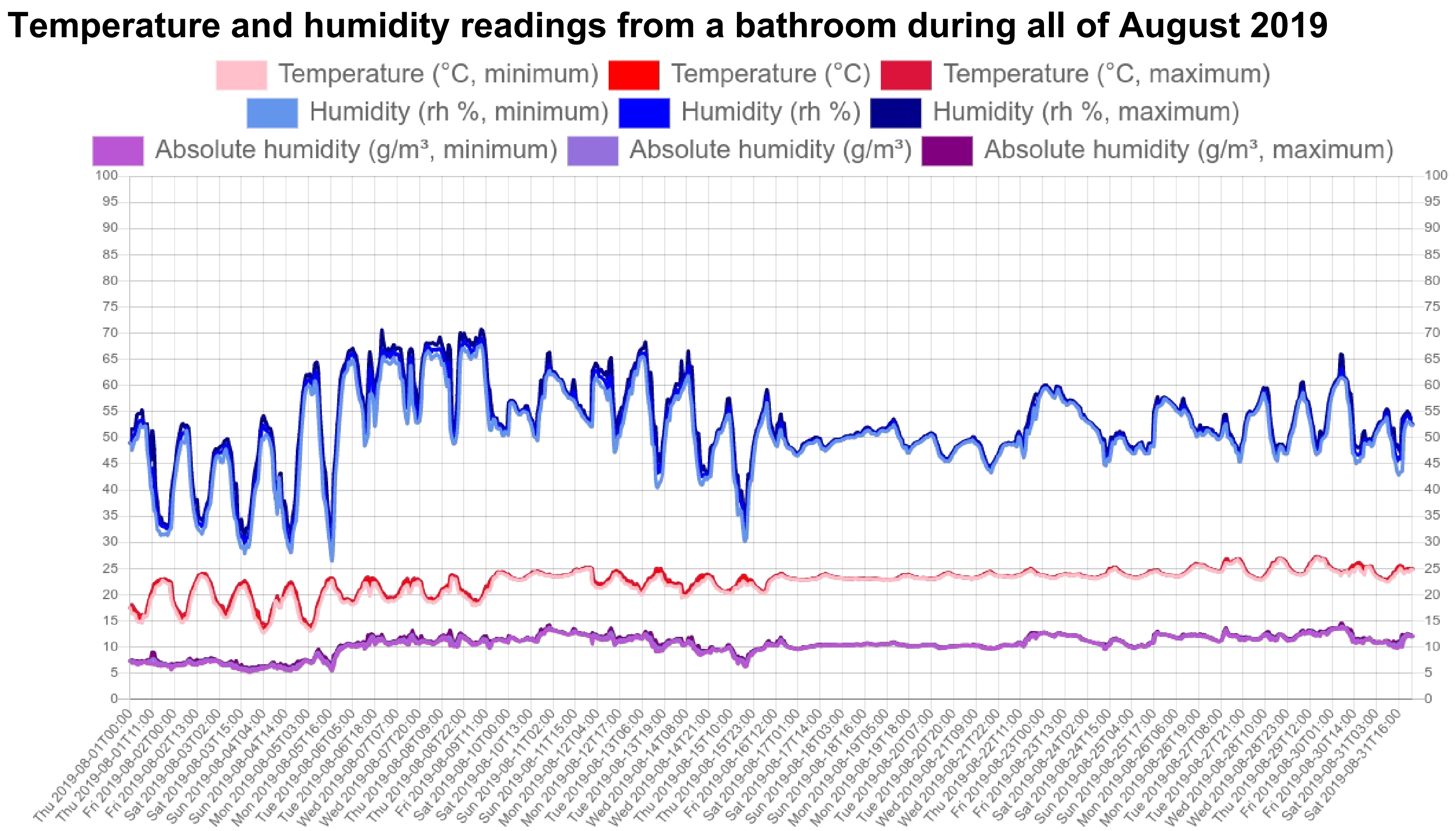


Figure 5. Temperature and humidity readings from a bathroom during all of August 2019.

Temperature and humidity readings from a home office during all of August 2019

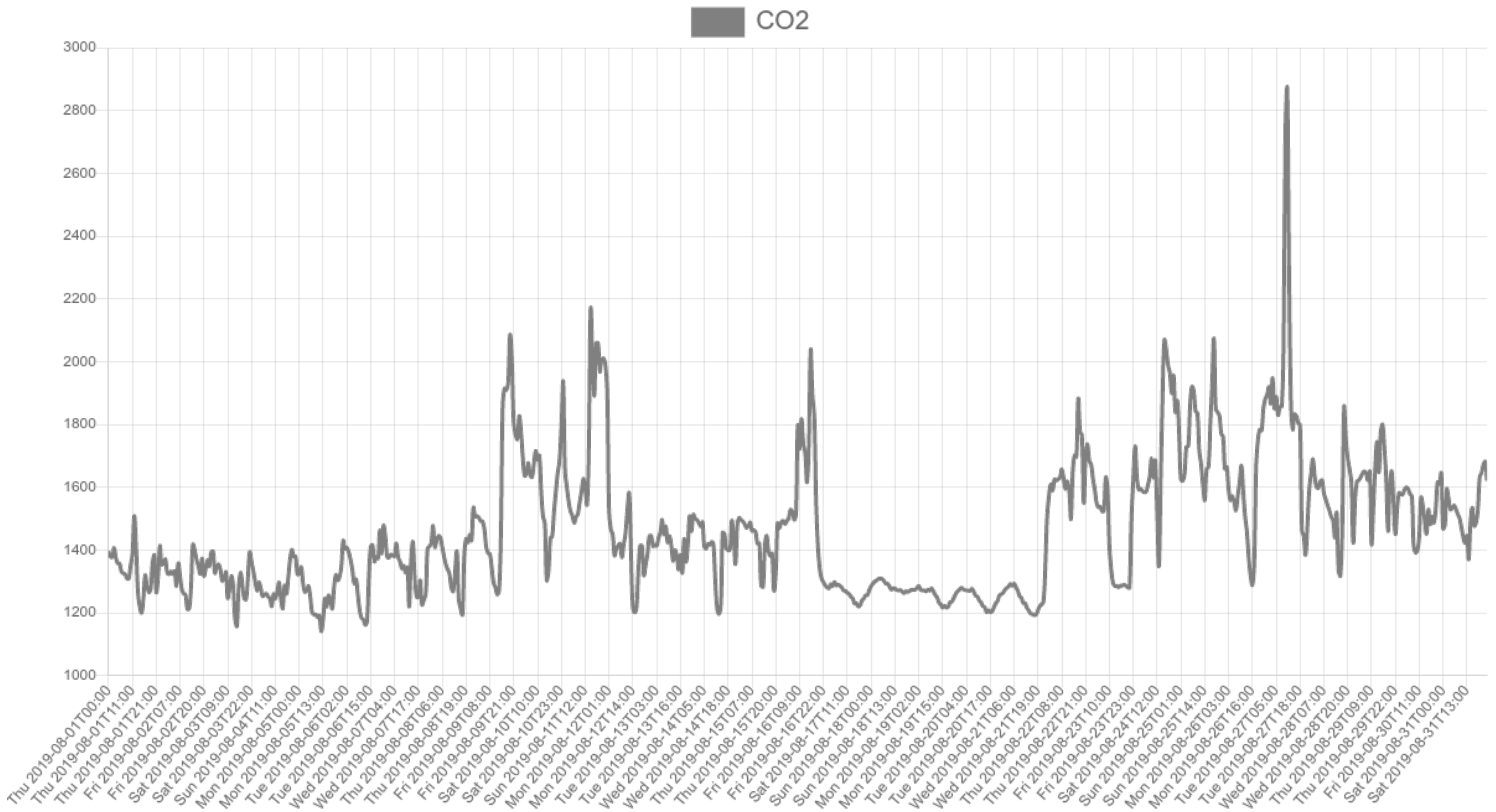


Figure 6. Temperature and humidity readings from a home office during all of August 2019.

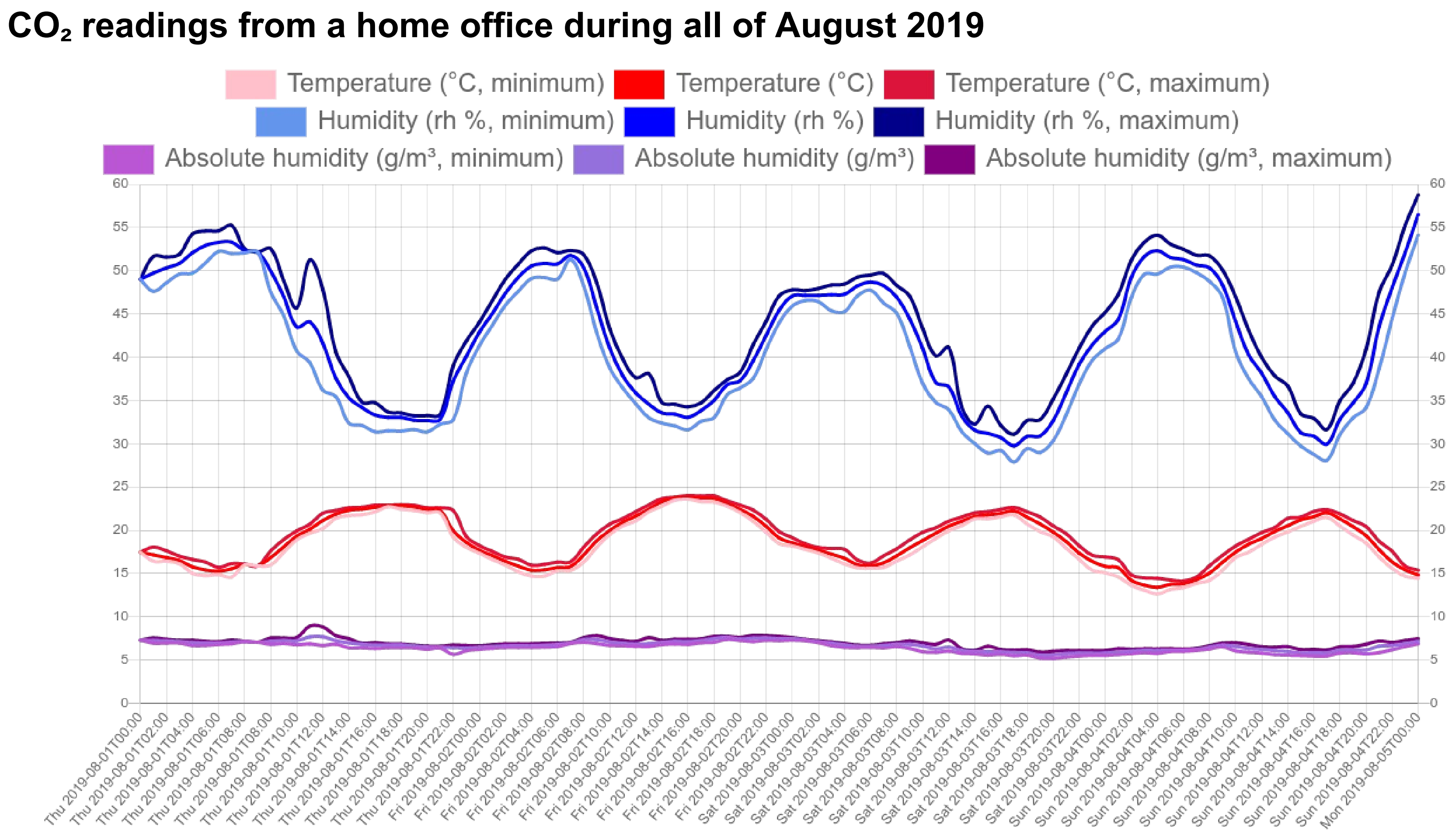


Figure 7. CO_2 readings from a home office during all of August 2019.

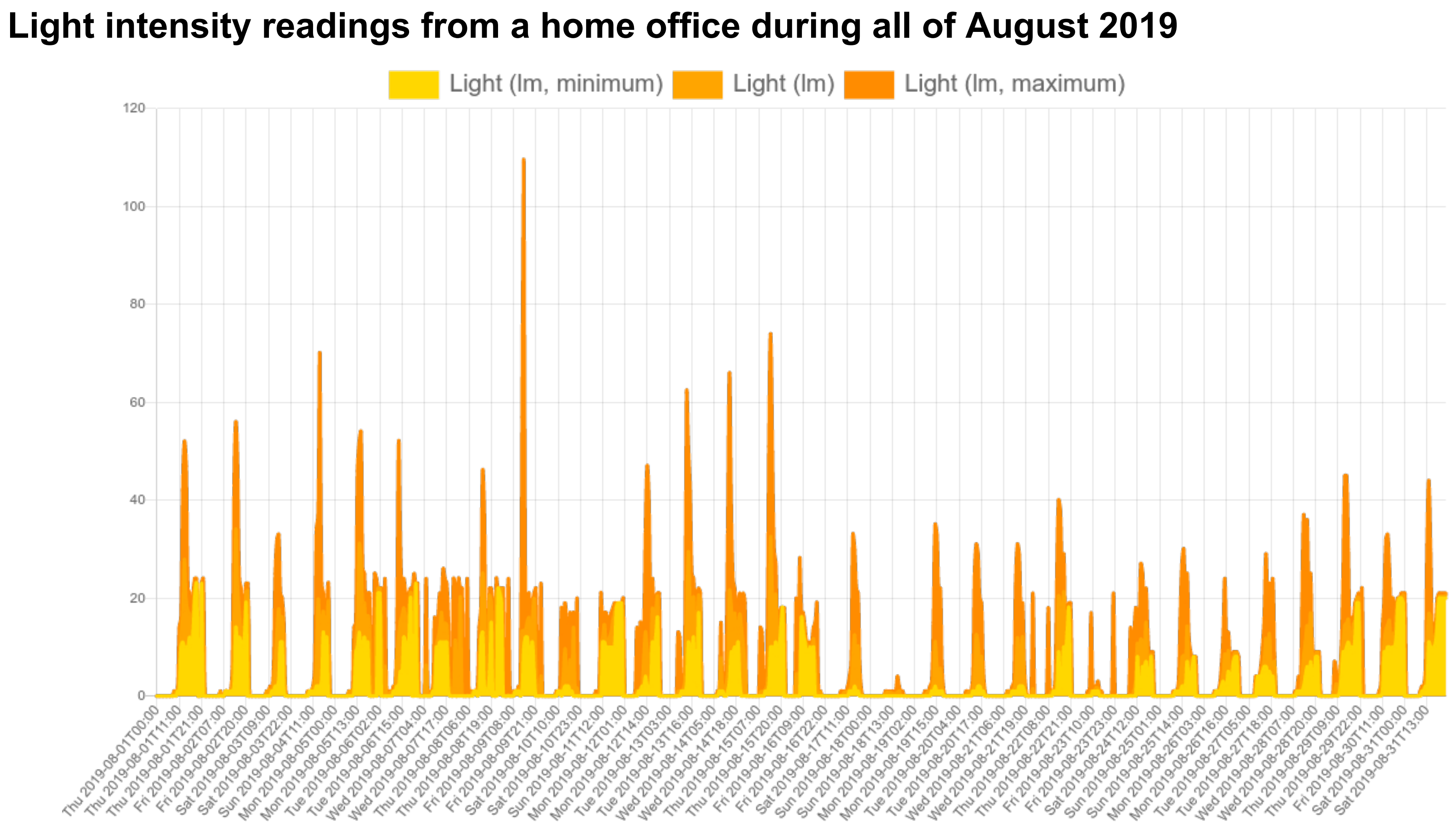


Figure 8. Light intensity readings from a home office during all of August 2019.

Light intensity minimum hourly readings from a home office during all of August 2019

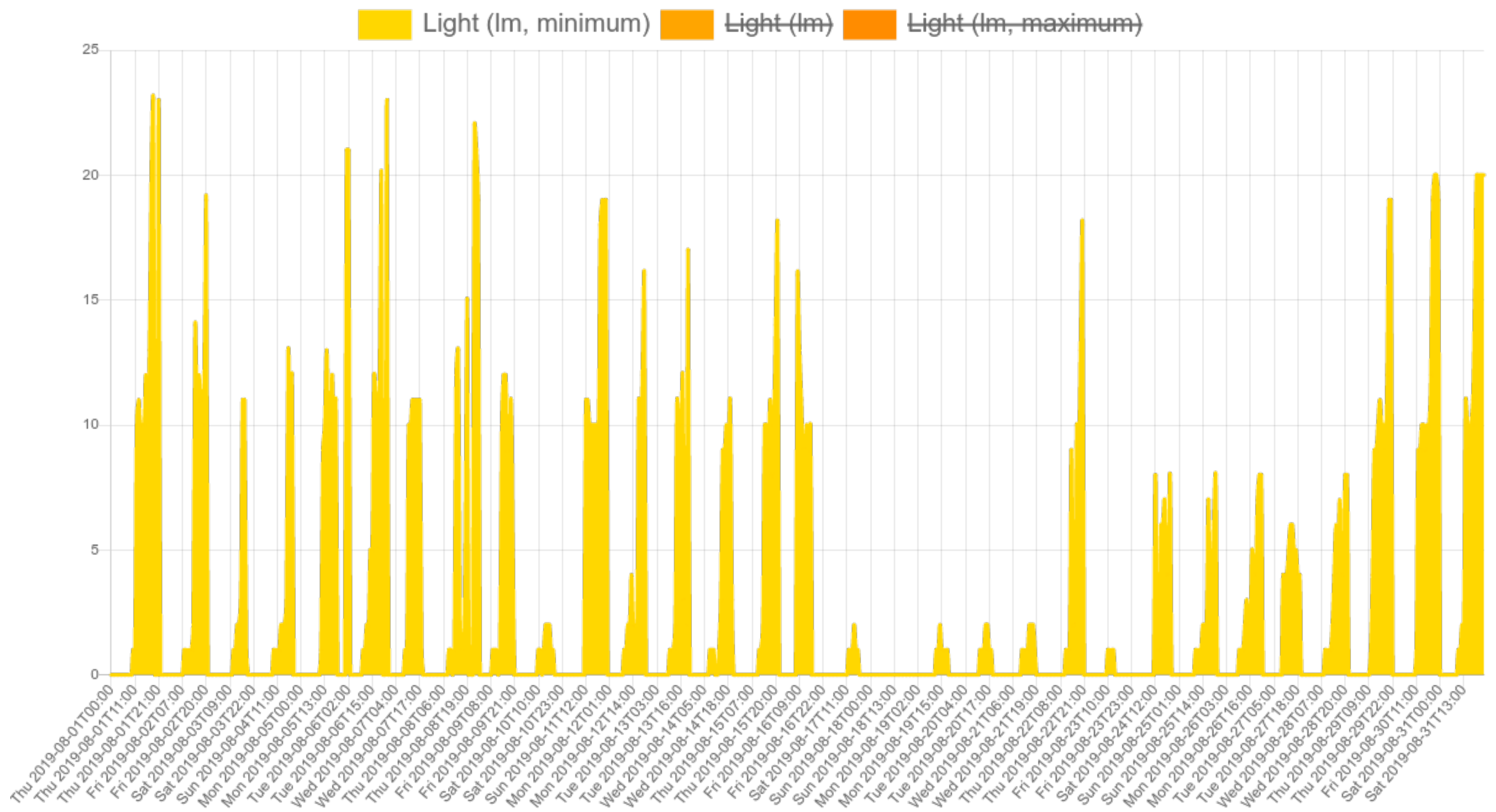


Figure 9. Light intensity minimum hourly readings from a home office during all of August 2019.

Temperature and humidity readings from a living room during all of August 2019

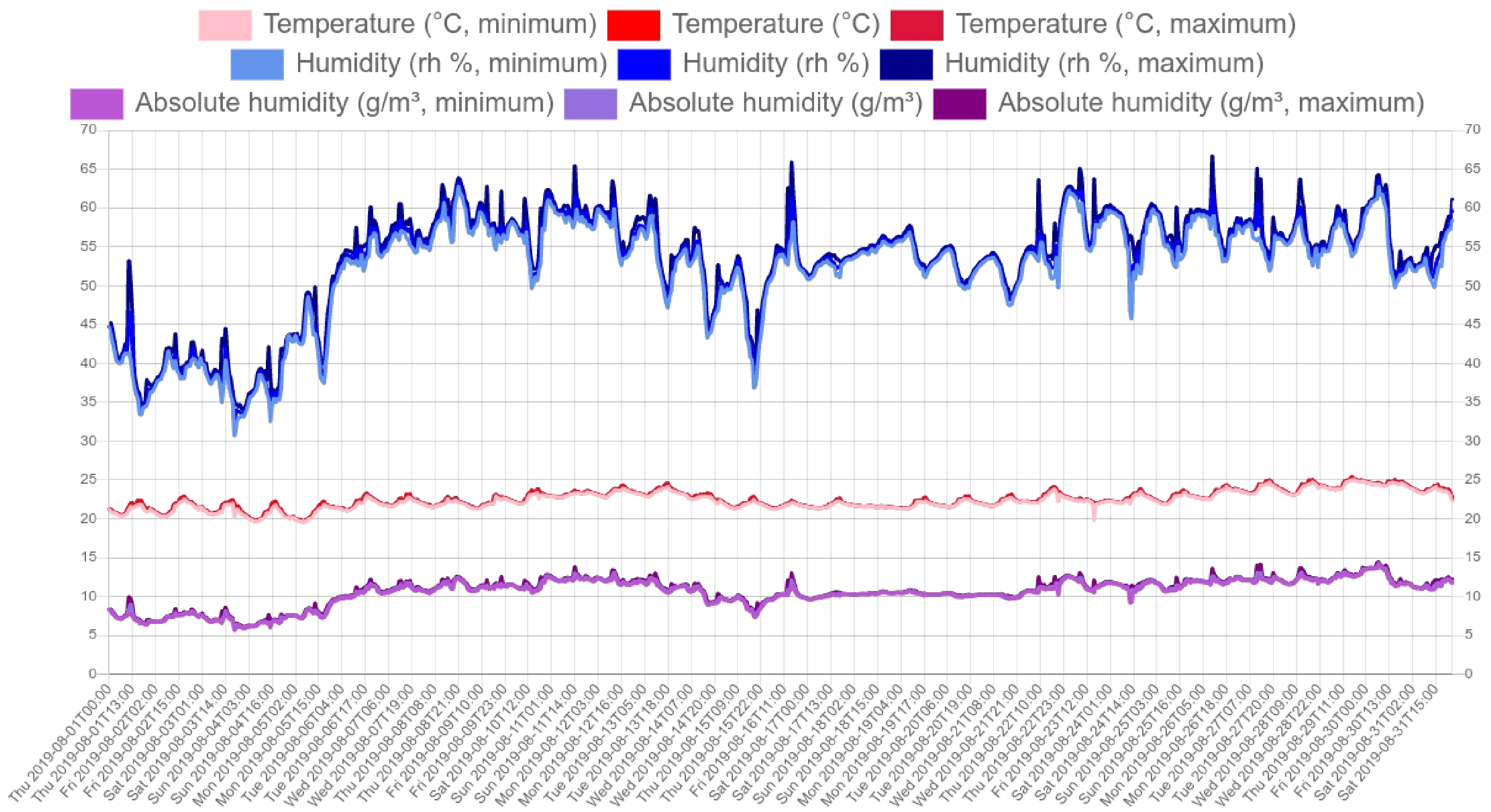


Figure 10. Temperature and humidity readings from a living room during all of August 2019.

Light and CO_2 measurements from a conference room

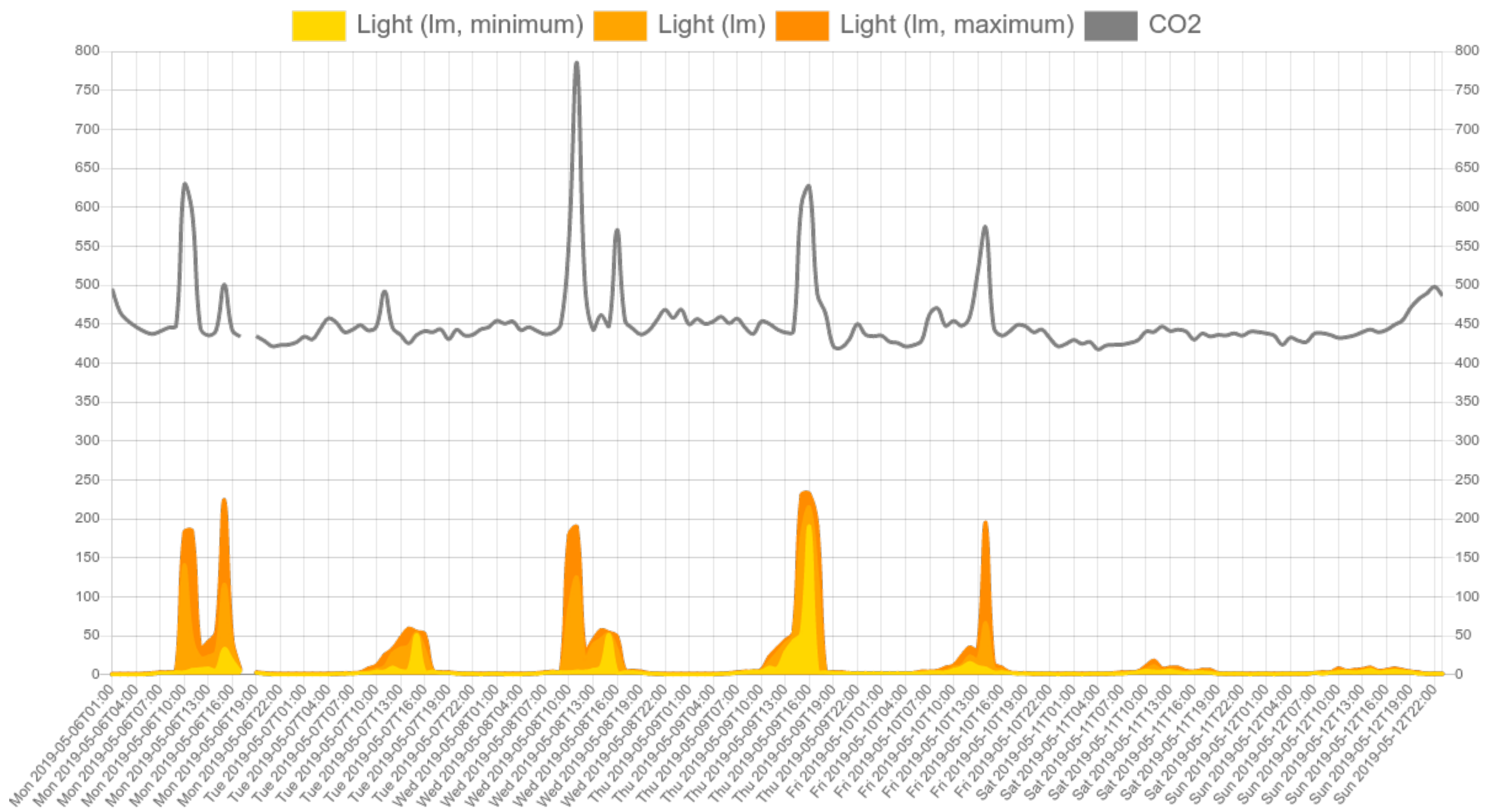


Figure 11. Light and CO_2 measurements from a conference room.

Light and CO_2 measurements from a small office

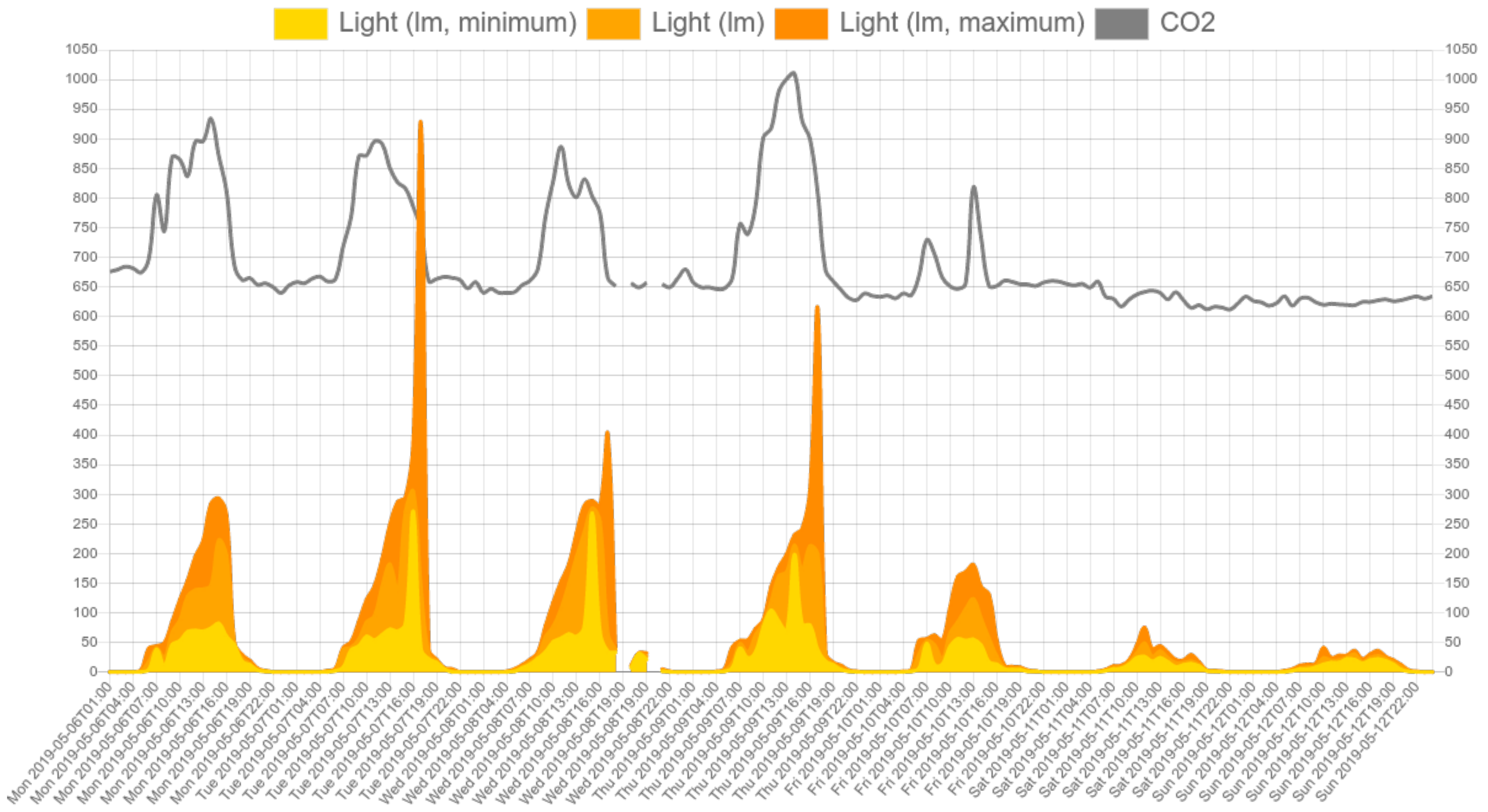


Figure 12. Light and CO_2 measurements from a small office.

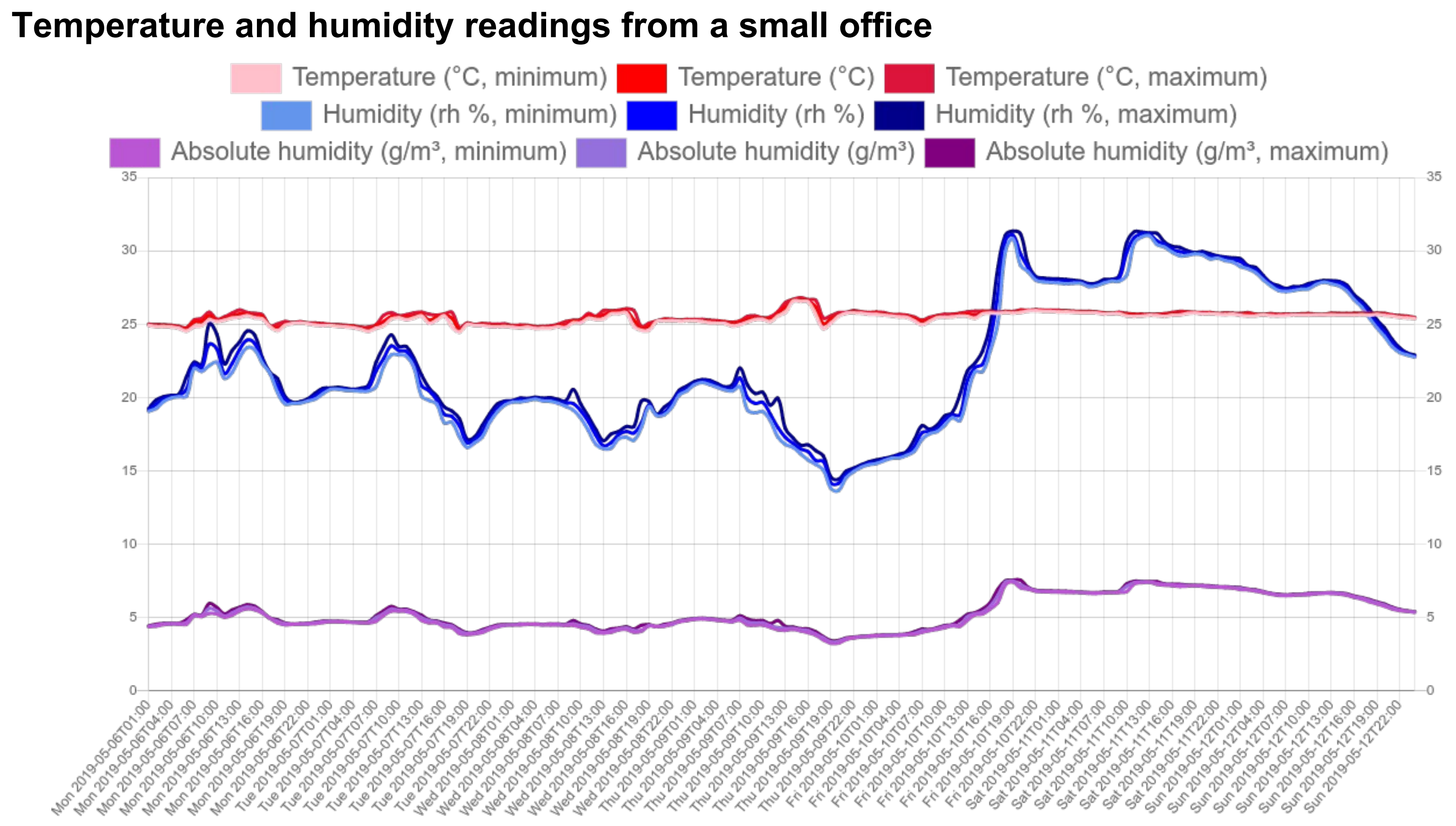


Figure 13. Temperature and humidity readings from a small office.

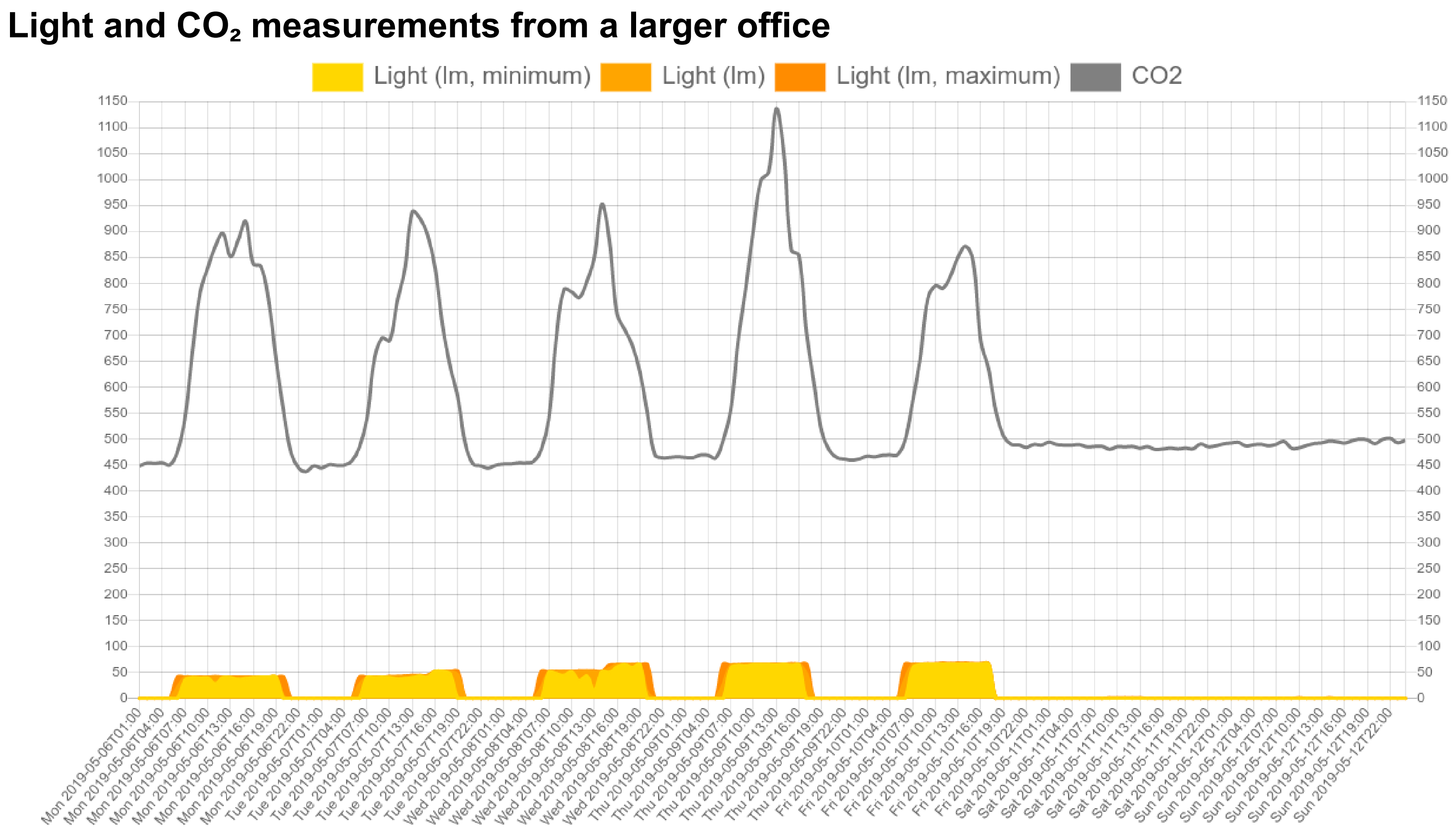


Figure 14. Light and CO_2 measurements from a larger office.