BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

COLLECTION OF SENSOR DATA INTO CLOUD

SBĚR SENZOROVÝCH DAT DO CLOUDU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MATEJ ZÁHORSKÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. SVETOZÁR NOSKO**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Computer Graphics and Multimedia (DCGM) Academic year 2019/2020

Bachelor's Thesis Specification

22718

Student:	**Záhorský Matej**
Programme:	Information Technology
Title:	**Collection of Sensor Data into Cloud**
Category:	Embedded Systems

Assignment:

1. Study the literature and approaches for reading data from sensors. Focus on connection of sensors to the cloud, including data protection by signing, timestamping, etc. The examples include temperature/humidity (low data rate), cameras (high data rate), or other sensors.
2. Select a suitable platform for sensor data collection and propose an implementation of data transfer from the selected sensors to the cloud.
3. Implement the selected method and discuss the achieved results.
4. Implement and demonstrate the protection of sensor data (confidentiality, authenticity, and integrity).
5. Discuss the obtained results and present possible continuation of your work.

Recommended literature:

- Based on instructions from the supervisor.

Requirements for the first semester:

- Items 1 to 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Nosko Svetozár, Ing.**
Head of Department:	Černocký Jan, doc. Dr. Ing.
Beginning of work:	November 1, 2019
Submission deadline:	May 28, 2020
Approval date:	May 18, 2020

Abstract

The primary objective of this thesis is data collection from selected sensors to a remote Cloud with the use of STM32MP1. The first part of this thesis includes theory about data collection options, their cryptographic security and functions of the given microcontroller. The second part is about design and implementation of proper software solution for effective and secure data collection from specific sensors. A thorough assessment of this particular solution is described in the last part, which includes cryptographic signing performance in a real world application.

Abstrakt

Primárnym účelom tejto práce je zber dát z vybraných senzorov do vzdialeného Cloudu prostredníctvom platformy STM32MP1. V prvej časti práce je popísaná teória ohľadom možností zberu dát, ich kryptografického zabezpečenia a funkcie daného mikrokontroléra. V druhej časti je navrhnuté a implementované vhodné softvérové riešenie pre efektívne a bezpečné zbieranie dát z vybraných senzorov. Na koniec sú vyhodnotené vlastnosti riešenia, čo zahrňuje i rýchlosť kryptografického podpisovania v reálnej aplikácií.

Keywords

STM32MP1, data signing, hardware acceleration, MySQL, ECDSA, SHA-256, Python, HTU21D, OpenSSL, Google Cloud, I2C, RPMsg, OpenAMP

Kľúčové slová

STM32MP1, podpisovanie dát, hardvérová akcelerácia, MySQL, ECDSA, SHA-256, Python, HTU21D, OpenSSL, Google Cloud, I2C, RPMsg, OpenAMP

Reference

ZÁHORSKÝ, Matej. *Collection of Sensor Data into Cloud.* Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Svetozár Nosko

Rozšírený abstrakt

Úvod

Zber dát zo senzorov a ich následné odoslanie do vzdialeného Cloudu má v dnešnej dobe mnoho implementácii, v ktorých sa väčšinou využíva viacero spolupracujúcich zariadení. Vďaka novej platforme zo série STM32MP1 je možné na jednom mieste čítať, spracovať, zabezpečiť a odoslať dáta do vzdialeného Cloudu. Cieľom tejto práce je oboznámenie sa s danou platformou a navrhnúť adekvátny spôsob pre efektívny a bezpečný zber dát.

Výsledkom práce je softvér schopný zozbierať dáta z jednoduchého teplotného senzoru a z inteligentnej kamery detekujúcej špecifické objekty v obrázkoch. Následne sú zozbierané dáta podpísané a odoslané do Cloudu šifrovanou komunikáciou, čo je možné vďaka mnohým softvérovým a hardvérovým komponentom dostupným nielen z operačného systému Linux bežiaceho na tejto platforme.

Návrh

Z dôvodu prítomnosti dvoch procesorov tvoriacich heterogénny multiprocesorový systém je vhodné rozdeliť implementáciu a vyvážiť ich využitie. Teplotný senzor, ktorý je schopný zbierať dáta aj o vlhkosti využíva I^2C rozhranie na komunikáciu. Senzory s daným rozhraním je možné pripojiť cez GPIO alebo Arduino konektory a s použitím procesora Cortex-M4 zozbierať relevantné dáta.

Tento procesor však nemá prístup k úložnému priestoru a ani k internetu, v dôsledku čoho je nutné zozbierané dáta predať procesoru Cortex-A7. K tomu je možné využiť medziprocesorový komunikačný kontrolér. Softvérová knižnica OpenAMP na strane koprocesoru a framework RPMsg na strane hlavného procesora sú dostupné k vytvoreniu komunikačného kanálu, ktorý zároveň využíva zdieľanú pamäť pre predávanie informácii.

Pred zahájením sieťového prenosu je nevyhnutné dáta zabezpečiť proti krádeži či podvrhnutiu. Knižnica OpenSSL implementuje široký výber algoritmov pre tvorbu podpisov, ktoré zaručia integritu a autenticitu dát i po dokončení sieťového prenosu. Vhodným algoritmom je ECDSA z dôvodu využitia menšieho súkromného kľúča a menších podpisov, čo šetrí procesorový čas, pamäť a dátové úložisko. Súčasťou podpisovania je i hashovanie, ktoré sa dá hardvérovo akcelerovať pomocou dedikovaného hashovacieho modulu.

K zozbieraniu dát z inteligentnej kamery a uloženiu detekcí sú potrebné dodatočné knižnice, ktoré nie sú k dispozícií v štandardnej distribúcii. K tomu je možné využiť distribučný balík, s ktorým sa dajú pridať, odobrať, či upraviť softvérové komponenty podľa potreby a skompilovať novovytvorený obraz systému.

V tomto bode sú dáta zabezbečené a pripravené na odoslanie. Na záver sa musí vybrať vhodná služba poskytujúca vzdialené úložisko, či už vo forme databázy alebo súborového systému. Pre dáta z jednoduchého senzoru, dodatočné metadáta a vygenerované podpisy je vhodné využiť relačnú databázu z dôvodu jednoduchej analýzy dát. Pre veľké množstvo informácií v podobe obrázkov predstavujúcich neštruktúrované dáta je štandardným riešením súborový systém namiesto databázy. Cena za využitú diskovú kapacitu je podstatne nižšia v porovnaní s databázou, pričom sa zároveň predchádza problémom so škálovaním a pomalému spracovaniu dát. Informácia o existencií obrázka v úložisku spolu s metadátami a podpismi sa ukladajú do databázy rovnako ako v prípade teplotného senzoru.

Implementácia

K vyvíjaniu koprocesorového firmvéru je možno použiť softvér STM32CubeIDE, ktorý ponúka konfiguračný nástroj pre automatické generovanie kódu. Vďaka tomu je k dispozícii inicializačný kód pre I^{2}C, piny a aj medziprocesorovú komunikáciu. Hardvérovou abstrakčnou vrstvou HAL je ďalej naimplementovaný zber dát z teplotného senzoru k jednoduchšiemu prenosu kódu medzi rôznymi platformami. OpenAMP knižnica prijíma správy vo forme príkazov od procesora Cortex-A7 a inicializuje zber dát, ktoré nakoniec odošle ako odpoveď.

Zozbierané dáta sú dostupné na strane hlavného procesora, ktorý ich kryptograficky podpíše a pripraví k odoslaniu. V procese podpisovania bola naimplementovaná funkcionalita hardvérovej akcelerácie SHA-256 hashovania použitím užívateľského rozhrania AF_ALG. V kombinácii s OpenSSL vytvára podpisy pre detekcie z inteligentnej kamery, pretože akcelerácia má prínos až pri rozsiahlych dátach. V prípade menšieho množstva dát je použitý softvérový algoritmus pre SHA-256 a abstrakčná vrstva EVP v OpenSSL.

Softvér na strane ARM Cortex-A7 bol naimplementovaný v jazyku Python. Skripty nadväzujú komunikáciu s koprocesorom, prijímajú, podpisujú a odosielajú zozbierané dáta do Cloudu. K správnej funkcionalite boli Pythonu dodané balíky pre komunikáciu s ARM Cortex-M4, MySQL databázou a Google úložiskom. Skripty sú schopné udržiavať automatické pripojenie k databázi a úložisku, pričom v prípade nedostupnosti sú dáta zbierané a ukladané v lokálnom úložisku. Python knižnica pre OpenSSL však nepodporuje ECDSA, preto bola vytvorená zdieľaná knižnica v C a následne importovaná do Pythonu.

Vyhodnotenie

Platforma použitá v tejto práci má obmedzené množstvo výkonnostných testov, a preto je na záver vyhodnotená rýchlosť hashovania a podpisovania v OpenSSL. Počas implementácie riešenia bolo nevyhnutné urobiť určité kompromisy. V prípade hardvérovej akcelerácie je k dispozícii i modul známy ako cryptodev, ktorý ponúka vyšší výkon ako jeho AF_ALG alternatíva. V čase implementácie však nebol schopný akcelerovať SHA-256, a preto bola táto chyba predaná prostredníctvom fóra integračnému týmu v STMicroelectronics. Vďaka rozsiahlej komunite, v ktorej sú i zamestnanci tejto firmy bolo možné odovzdať daný problém zodpovednému personálu. V dôsledku toho bude v novej softvérovej verzii ekosystému plánovanému v júni 2020 tento modul schopný akcelerovať i výpočet SHA-256. Jeho predbežné výsledky boli poskytnuté zamestnancom firmy a sú taktiež popísané v tejto kapitole. Na koniec boli pridané aj výsledky zberu dát z oboch senzorov a ich prítomnosť v Cloude.

Záver

V konečnom dôsledku bolo možné vytvoriť softvérové riešenie, ktoré do vysokej miery využíva potenciál tejto platormy, čo osobne považujem za úspech. V budúcnosti by bolo vhodné využiť aj bezpečné prostredie ARM TrustZone pre zber, zabezpečenie a odosielanie dát. Operačný systém OP-TEE alebo Trusted Firmware-A, ktoré je možno spustiť v bezpečnom prostredí nepodporujú spúšťanie Python skriptov. Z tohoto dôvodu by sa musela implementácia kompletne presunúť do jazyka C, čo vyžaduje alternatívnu databázu využívajúcu SQLite a SFTP úložisko (knižnice sú štandardne podporované na platforme) alebo podporu dodatočných knižníc potrebných pre súčasné Cloudové riešenie (MySQL a Google Cloud Storage).

Collection of Sensor Data into Cloud

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Svetozár Nosko. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Matej Záhorský
May 27, 2020

Acknowledgements

I would like to express my sincere gratitude to Ing. Svetozár Nosko for giving me useful advice, feedback and will to help, my family and friends for their support and everyone in the STMicroelectronics community and staff for their effort to help with some of the issues encountered during the implementation process of this thesis.

Contents

Chapter 1

Introduction

People everyday are surrounded by a wide variety of microcontrollers (MCUs) and sensors that make their lives easier without even knowing about their presence. As the days are passing by, many things are being automated and for that reason, requirements for them are quickly rising.

Many people think of a computer when it comes to the term microcontroller and technically, they are not wrong. As an example, these tiny computers might be controlling electric windows in your car or the automatic brake system, which means that generally, there are many of them in the whole system. Besides car industry, they are also a part of medical devices, office machines, computer peripherals, televisions, alarm clocks and the list could go on for a very long time. However, these systems are built for simple tasks where they often react to an user input, such as a button click and perform the desired operation by sending a signal to the controlled unit. In many cases this is the only thing we want and thanks to simplicity of the device we can achieve our goal while using only a tiny bit of power. With that comes a problem with potential future upgrades and flexibility. Once a decision is made, it is quite expensive to make changes later due to fixed design of the system. Additional upgrades such as statistics collection of the controlled devices state and possible prediction of failures might be too much to process by the MCUs Central Processing Unit (CPU). If we further want to collect the data on a server we need additional components for network connectivity and protection, because once we are online we are in danger of unauthorized access to our system.

If that is what we need, then it is necessary to opt for a more robust system with more power and features, such as an STM32MP1[1] from STMicroelectronics. This MCU offers two CPUs, the ARM Cortex-A7 which is focused on high computational power for more complex tasks and the more efficient ARM Cortex-M4, that is used for small operations in real-time, such as reading data from a sensor. Besides offering high performance, it also provides support for Linux operating system, many useful libraries for development, graphical processing unit for easier interaction and most importantly advanced security features for better protection.

This thesis will focus on how this particular unit works, what are the possibilities of sensor data transfer to the MCU and then to a remote cloud. This needs to be done while maintaining high performance and emphasis on security with data signing to avoid potential data forgery.

[1]STM32MP1 Series. STMicroelectronics. [online]. [cit. 2020-04-23]. Retrieved from: https://www.st.com/en/microcontrollers-microprocessors/stm32mp1-series.html

Chapter 2

Data collection and security

While the more robust and powerful ARM Cortex-A7 is suitable for more complex tasks, the ARM Cortex-M4 co-processors purpose is to provide additional computing power for real-time applications. Due to the reason of both processors having different characteristics, which means that they create a heterogeneous environment, it is important to implement a way for them to communicate and exchange information. This chapter describes methods used for transmitting data from sensors to the STM32MP1 unit using the co-processor and how it is possible to transfer the collected data to the ARM Cortex-A7 master CPU, which is running Linux operating system and has an active internet connection. The master is then capable of running heavier tasks for securing the data with signing and ciphering algorithms and afterwards sending them over the network to a remote cloud, which is also a part of this chapter.

2.1 Sensor communication interfaces

As different sensors require different hardware interfaces for communication with the board, it is important to select the more suitable one based on the characteristics of the particular sensor. As some sensors are intelligent and are capable of communication via Ethernet or USB, they won't be included as every implementation differs from model to model. Following hardware peripherals can be found on many different platforms including the STM32MP1.

UART and USART

One of the most common communication modules is UART (Universal Asynchronous Receiver/Transmitter). Asynchronous communication is done without using a synchronization signal, which means that a different approach is required to let the receiver know when the data are being transmitted. UART does this by creating a data packet, which consists of a start bit, data bits, zero or one parity bit and one or two stop bits. Start bit marks the beginning of the communication between the transmitter and receiver, parity bit is used to detect errors during transmission and a stop bit (or two stop bits) marks the end of the transmission. By specifications of UART present on the MCU, the packet size is programmable for either 7, 8 or 9 data bits [16] plus start, parity and stop bits, which results in high data overhead and relatively slow effective transfer speeds.

It also needs to be ensured that both communicating sides are properly configured to avoid any potential data loss. One of the most important settings is transmission speed, which is called baud rate. Its value represents amount of symbols transmitted per second, which in case of 2 symbols (zeroes and ones) is equivalent to bit rate [2]. Big differences in baud rate can cause one side to receive corrupted data, because it is unable to process them at the same speed as the other side.

To avoid this problem, the synchronous mode can be used as well and it is supported by USART on the MCU. It is able to communicate with the receiver asynchronously like UART, but also synchronously by using a separate clock signal. It can be used for connecting an SPI sensor, which also uses this communication mode. However, this fact does not ensure higher transfer speeds, because start and stop bits are still being sent [16]. The biggest disadvantage of both UART and USART is unsupported communication with multiple devices that SPI and I^{2}C do support, which is the reason why it is not very commonly used with sensors.

I^{2}C

I^{2}C module (Inter-Integrated Circuit) is a serial interface that communicates with devices over a single data channel. As opposed to UART/USART, it is commonly used to control sensors since more of them can be connected at the same time.

The controlling unit, which initializes the communication is called master, while connected devices or sensors are slaves. Because the communication is synchronous, master needs to generate a clock signal to synchronize with its slaves. If UART/USART was to be used, it would use two data channels for communication, one for transmission and one for reception. This is where I^{2}C also differs, because it uses a single data channel to communicate bidirectionally (half-duplex) with all connected slaves. Before the communication with a slave begins, it is first necessary to select it by sending each slave an address and a read or write bit. After the address is received by a slave, it compares the address with its own and responds with an acknowledgement bit if it matches. Afterwards, the master can start sending data to the slave or slave to the master.

However, some sensors have a fixed address, so it is not possible to differentiate them. For that reason, sensors which can have a programmable address or a different model of the sensor should be selected. In addition to that, the modules transmission speed is significantly lower (1 Mbit/s) [16] in comparison to other interfaces, such as SPI (on STM32MP1 up to 100 Mbit/s).

SPI

If the previous modules are not suitable for use, the SPI (Serial Peripheral Interface) can be selected instead. Thanks to a clock synchronization signal it can communicate synchronously just like I^{2}C.

Although that's where the similarities end, since it is a four-wire bus unlike I^{2}C, which is only two-wire. The extra wires are necessary for full duplex communication, because it is achieved using separate data channels for each direction [16].

As both I^{2}C and SPI support multi-slave systems, the selection of a slave is done differently. The fourth channel of SPI module is called a slave select (SS) channel, which is set to a logical 0 and kept in this state until the communication has ended and then the value is restored back to a logical 1. Since this system only controls when the slave can receive or transmit data, a single SS wire is connected to only one slave unlike I^2C. For large amounts of devices it becomes problematic since there are only a limited amount of SS wires. In that case either GPIO pins can be used to act like a SS channel or a decoder has to be obtained, which of course increases the overall cost of the system.

In some specific scenarios, in which there are multiple LEDs or shift registers, a single SS wire can be used. This is known as „daisy-chaining“, [16] where master communicates with all slaves simultaneously. The masters data output is connected to the input of the first slave, which then outputs the data to the next slave. The last slave is then outputting data to the masters input and the chain is closed.

However, daisy-chaining is often not a viable option, because in most cases a direct communication between master and slave is required. This is why the first option with multiple SS wires would need to be used to achieve communication with multiple sensors.

2.2 Inter-processor communication

The main problem in heterogeneous systems is their cooperation and communication, which isn't straight-forward and requires some software solutions in order to accomplish their synergy. The more robust Cortex-A7 in this system is referred to as „host“ or a „master“, which means that the Cortex-M4 (referred to as a „co-processor“ or a „remote“ processor) is initialized, programmed, debugged through this processor and the communication between them is also initiated on the masters side. For this reason, most of the software solutions are present in the kernel of Linux operating system, which offers several drivers and frameworks designed for this purpose. As these solutions can be different on specific platforms, most of the following information in this section are gathered from STMicroelectronics wiki pages, which are created and approved by official employees of this company and they are considered reliable and accurate. Each subsection contains a reference to the respective page for more details.

IPCC

The drivers and frameworks require a hardware peripheral called inter-processor communication controller (IPCC) [10]. This controller provides a communication support on hardware level, where both parties (sender and receiver) have their own register banks and interrupts. The message that is being transmitted is stored in RAM as a shared memory buffer, which is not a part of the IPCC peripheral [16]. In total there are 6 channels available for communication, where each of them consists of two sub-channels:

- Processor 1 to Processor 2 sub-channel
- Processor 2 to Processor 1 sub-channel

Since the memory containing the message is shared, it is important to ensure that the processors do not simultaneously access this memory and cause unwanted errors. For that purpose all 12 sub-channels have a flag for controlling the communication. A flag can be of two values, generating two different interrupts:

- TXF - The sending processor is notified about a sub-channel being ready for new transmission.

- RXO - The receiving processor gets an interrupt about sub-channel being occupied, which indicates an incoming message.

These 6 channels can operate in 3 different modes:

- Simplex - Only one sub-channel is used. The sender checks if its free based on the channel flag, then it posts the data in shared memory and sets the flag to occupied. This generates an interrupt on the receivers side, so it can read the data from memory. Afterwards, the flag is cleared and a free channel interrupt on the senders side is generated for a new transmission to begin.
- Half-duplex - Only one sub-channel is used. The sender stores required data in memory, sets the flag and waits for response. After the message is processed by the receiver, it first stores the response in memory and then clears the flag.
- Full-duplex - Both sub-channels are used, it can be considered as a combination of two simplex channels where both parties are communicating asynchronously, which means they are not waiting for a response.

Following figure illustrates how the channels and their sub-channels can generate an interrupt that notifies the receiving processor about an incoming message or the sending processor about a free channel.

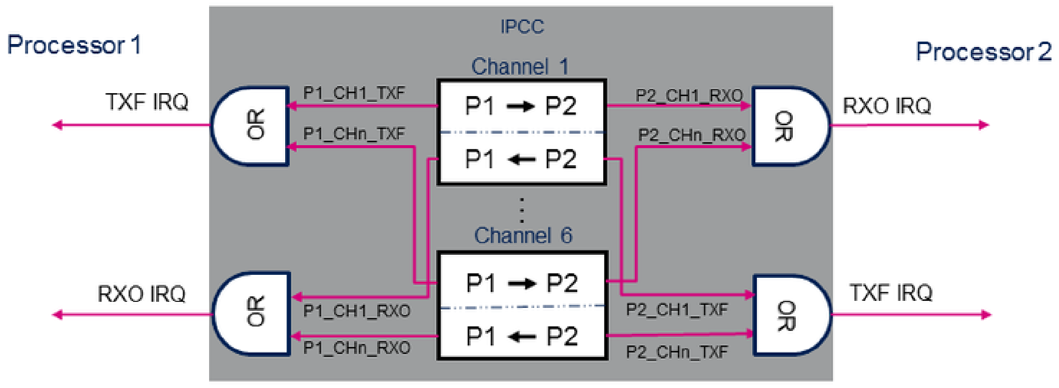


Figure 2.1: IPCC communication channels and interrupts[1].

[1]IPCC internal peripheral. STMicroelectronics. [online]. [cit. 2020-04-23]. Retrieved from: https://wiki.st.com/stm32mpu/nsfr_img_auth.php/thumb/8/8e/IPCC_peripheral.png/800px-IPCC_peripheral.png

Linux mailbox framework

The mailbox framework is used for exchanging messages or signals between the host processor and the co-processor. This framework is defined and operational in the Linux kernel space and it is separated into two parts [11]:

- The mailbox controller is a driver for the IPCC hardware peripheral, which means that it configures and handles the IPCC interrupts.
- The mailbox client is in charge of messages themselves, which are going to be sent or received. A client can be defined by the user, but mostly it is used by other frameworks for message exchange purposes.

In the following figure the mailbox client is the remoteproc (rproc) framework driver, which in turn forwards messages to the RPMsg framework. Both frameworks will be described in the next sections, since they will be key to this thesis.

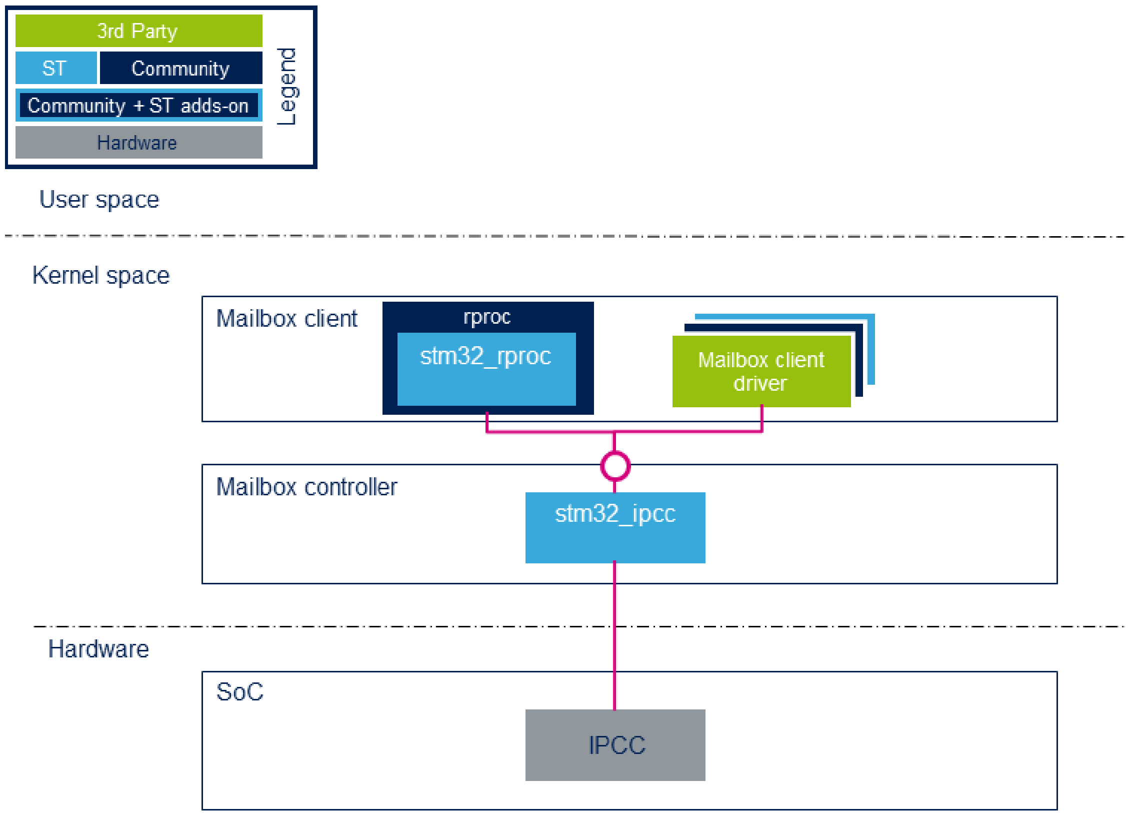


Figure 2.2: Linux mailbox framework[2].

[2]Linux Mailbox framework overview. STMicroelectronics. [online]. [cit. 2020-04-23]. Retrieved from: https://wiki.st.com/stm32mpu/nsfr_img_auth.php/a/aa/Mailbox_overview.png

Linux remoteproc framework

The key software component for developing real-time applications in production mode is the remoteproc framework. It allows the master processor to initialize, configure, debug and monitor the co-processor when an application is deployed [17]. It uses a remoteproc driver to contol the initialization, such as clocks, memories, registers or watchdogs. The framework's generic part is responsible for loading the firmware image into the co-processors memory, parsing the resource table and starting/stopping the execution while offering monitoring and debugging services if they are required. The Linux mailbox framework client is also being used here, since it is responsible for sending notifications to the co-processor. As already stated earlier, the remoteproc framework creates some sort of an interface for sending messages with the RPMsg framework as shown on the following figure.

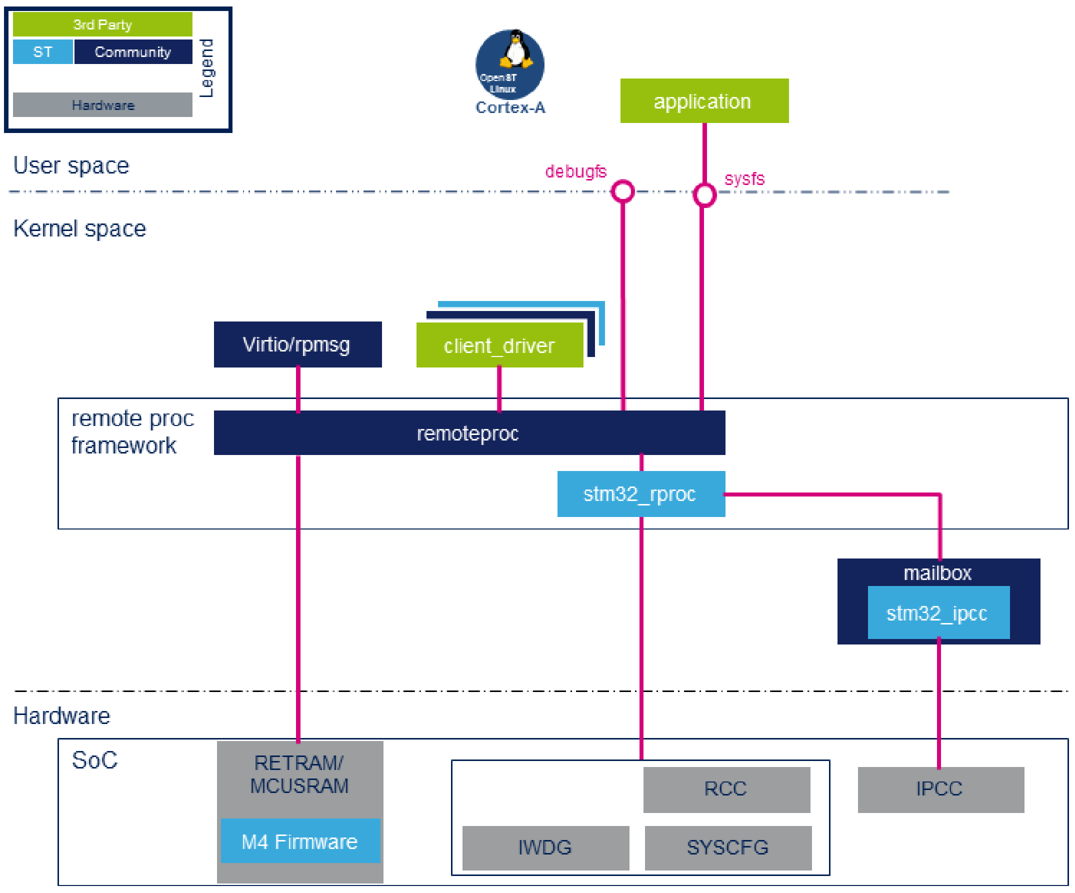


Figure 2.3: Linux remoteproc framework[3].

[3]Linux remoteproc framework overview. STMicroelectronics. [online]. [cit. 2020-04-23]. Retrieved from: https://wiki.st.com/stm32mpu/nsfr_img_auth.php/e/ef/Remoteproc_overview.png

RPMsg framework and/or OpenAMP

If the data which are available on the remote processor (in its memory) need to be further processed, stored on a disk or sent over the network, they first need to be „sent“ to the master processor. The Linux RPMsg framework is designed to provide information exchange mechanism via shared memory buffers [19]. A RPMsg client can be identified, registered and associated to by a textual service name. Once an identical service name is announced by a remote processor, the RPMsg client driver is probed by the framework, communication channel is established and ready to start. The whole process is initiated by the remote processor by creating an endpoint with a unique source address and an associated callback function. However, on remote processor there is no kernel module or framework available, which would be in charge of the communication. A recommended solution by STMicroelectronics is OpenAMP middleware library, which handles all RPMsg related configuration and message exchanging. All the frameworks and drivers described earlier have lead to a somewhat complex solution for communication. It requires the remoteproc framework which forwards services to the mailbox framework, that in turn controls the IPCC peripheral. It is used to send notifications about a new message, which has been stored by the RPMsg framework in a shared memory. On the remote processors side, the IPCC HAL driver is used to implement mailbox framework and the OpenAMP library for RPMsg. The complete diagram of all used software solutions and how they cooperate can be seen on the following figure.

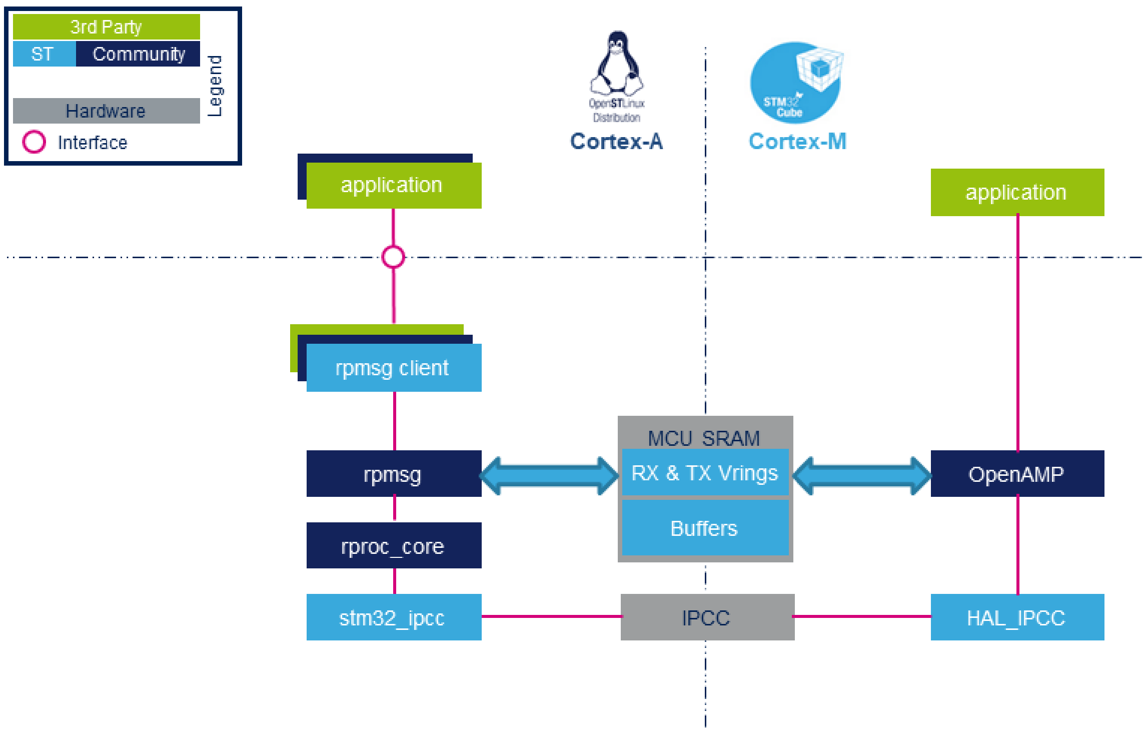


Figure 2.4: Linux RPMsg framework and/or OpenAMP[4].

[4]Linux RPMsg framework overview. STMicroelectronics. [online]. [cit. 2020-04-23]. Retrieved from: https://wiki.st.com/stm32mpu/nsfr_img_auth.php/3/35/Rpmsg_overview.png

2.3 Cloud

Data captured at a certain point by a sensor might not have a meaning by themselves. They need to be correlated and compared to further data that will be obtained sometime in the future or that are currently available on a remote MCU or storage. This section is about bringing them together on a single storage where they can be properly analyzed.

Database or file storage?

A specific sensor have a unique output which defines a lot of factors that need to be considered before selecting a remote storage. Simple sensors that measure temperature, humidity or pressure give us structured data, which can be represented by either float or double data type depending on how precise they have to be. A simple relational database using SQL is capable of storing structured data where data size is known along with additional metadata about their origin.

A single row in a relational database should be as small as possible in terms of its size to keep its price as low as possible. Storing massive data results in scalability issues and low response times [5], which would be easy to accomplish with a camera. Even a tiny image that consists of only white pixels with a resolution of 50x50 is enormous in comparison to a simple decimal number. For this reason, a simple relational database would be very expensive to maintain and a lot of expensive storage would be required. Images themselves are unstructured data, as they do not have a fixed size and a dedicated data type that could represent them. Databases can store large binary objects (BLOB), which can take up to 65 KB of unstructured binary data, which in many cases is not sufficient. As it is not impossible to do this, it is considered a bad practice to insert large unstructured data into a database.

In cases like this it is very common to use a separate storage, which is designed exactly for this purpose. The image or any other binary object is first sent to a specific folder in a Cloud storage with a specific file name and file extension. The information about the objects presence on the Cloud is then sent to the relational database in a form of full path to the file in the storage. Additional metadata captured during the image processing are also being sent to the relational database for future analysis. However, with this solution there is a high chance of inconsistency between the database and storage, so extra precautions need to be done to make sure that the object is really present on the storage if it is referenced in the database.

Cloud services

It is important to know what benefits does cloud computing and cloud storage bring to the table. However, these services are not free, so it is necessary to select the best option for implementation while keeping the price as low as possible.

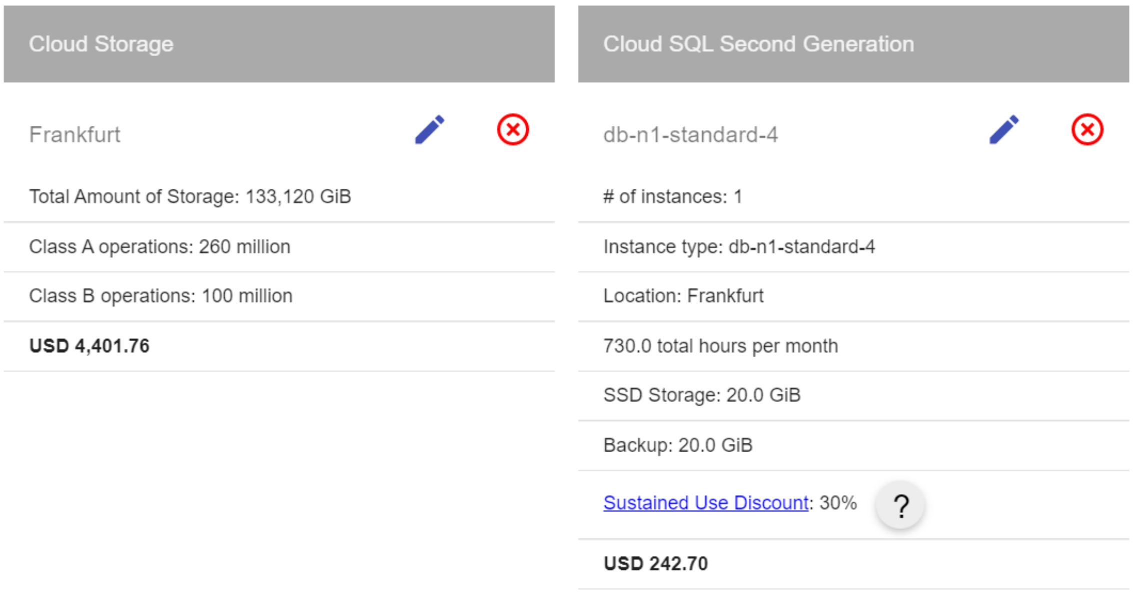
There are many database and storage services that are offered for low prices, but their availability, performance and support are questionable. Among many different companies the most known are Google and Amazon. Both of them offer relational database and storage services that are broadly used by many companies. The price of each service depends on actual usage, either if its storage size, database operations, server region, CPU and RAM utilization or network usage. For a direct comparison between the two services, an identical setup was created and using their proprietary calculators it was simple to figure out required expenses for each service.

It can be assumed that if an imaging sensor is being used, the total cost of the service is going to be higher in comparison to a temperature sensor, which outputs very little data. If a single camera captures an image every second with an average image size of 50 KB, the total storage size needed is around 4,32 GB per day. With every added device the number is multiplied and storage requirements quickly rise. These images become outdated after some time and if a month old images are automatically removed from the cloud, the total requirements for a time span of one month per device is around 130 GB. Database operations are also being taken into account since both services charge extra for them. Each image insertion into database is one POST operation, which means that around 260 thousand insert operations are done on both database and storage monthly.

For storing images on Amazon Web Services a standard S3 storage service has been selected, whereas on Google's side it was a Cloud Storage option. Based on previous calculations, the total storage requirements for 1000 concurrently running cameras is 130 TB of storage, 260 million insert operations (Class A operations) and for purpose of analysis, a 100 million GET operations (Class B operations) was given as an example for comparison, because it depends purely on the application. Except for storage requirements, many operations are required to be done as fast as possible and a server located thousands of kilometers away can cause higher transmission delays. Both companies are offering their services on multiple continents, including Europe, which is why a server based in Frankfurt was selected in their respective calculators. The instance on Google's side had 4 virtual CPUs (vCPUs) and 15 GB of RAM, while on Amazon RDS there is the same amount of vCPUs and 16 GB of RAM. In case of data storage, Google's Cloud Storage price is lower by 5% [2.5] in an equal scenario than Amazon S3 storage [2.6]. Along with storage prices, the relational database from Google [2.5] came out being cheaper by around 24% in a very similar specification than Amazon RDS [2.6].

After researching both of these platforms, it is safe to say that their differences are very minor, which can be also seen on some online blogs[5]. Both are offering very comparable availability and security, so everything comes down to the final price based on the configuration.

[5]Google Cloud vs AWS in 2020 (Comparing the Giants). Brian Jackson. [online]. [cit. 2020-04-23]. Retrieved from: https://kinsta.com/blog/google-cloud-vs-aws/



Cloud Storage

Frankfurt

Total Amount of Storage: 133,120 GiB

Class A operations: 260 million

Class B operations: 100 million

USD 4,401.76

Cloud SQL Second Generation

db-n1-standard-4

# of instances: 1

Instance type: db-n1-standard-4

Location: Frankfurt

730.0 total hours per month

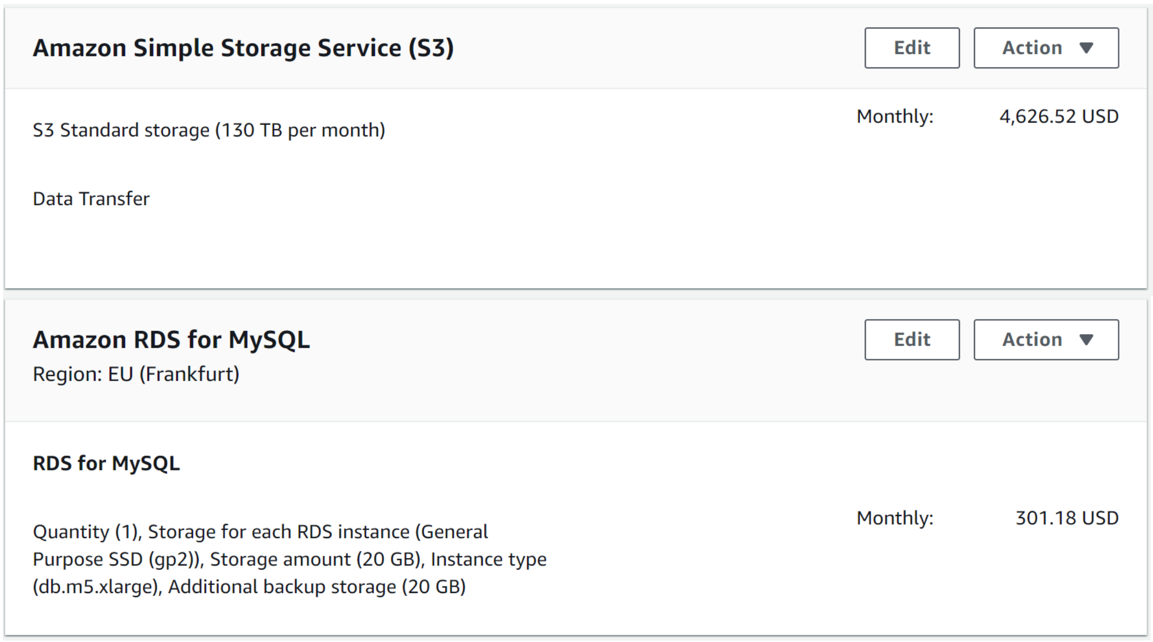
SSD Storage: 20.0 GiB

Backup: 20.0 GiB

Sustained Use Discount: 30%

USD 242.70

Figure 2.5: Google Cloud Storage and SQL pricing.



Amazon Simple Storage Service (S3)

S3 Standard storage (130 TB per month) — Monthly: 4,626.52 USD

Data Transfer

Amazon RDS for MySQL
Region: EU (Frankfurt)

RDS for MySQL

Quantity (1), Storage for each RDS instance (General Purpose SSD (gp2)), Storage amount (20 GB), Instance type (db.m5.xlarge), Additional backup storage (20 GB) — Monthly: 301.18 USD

Figure 2.6: Amazon S3 Storage and RDS database pricing.

2.4 Software

The STM32MP1s software consists of multiple solutions in order to create a working ecosystem, that is capable of using all the resources available while keeping the system and user programs safe, powerful and in perfect synergy. The hardware is without any doubt important, but without software it would be very complicated for developers to work and create something useful on this platform. There are some software implementations, which can simplify the process of developing software solutions. Similarly to section 2.2, this section will mostly contain information from STMicroelectronics wiki pages, which will be linked in each sub-section for more information.

STM32MP1 boot sequence

The platform can be booted in either engineering mode or production mode, where the engineering mode allows us to flash the co-processor firmware on the board and debug it. In this case, the Linux operating system is not booted and the Cortex-A7 is in an infinite loop, but it can be debugged as well if needed. This mode can be entered by switching the boot pin 0 to position 0 and pin 2 to position 1 on the back side of the board [18]. In the production mode, the Cortex-A7 is used to boot the Linux operating system and initialize all necessary features. The boot sequence in this mode consists of multiple stages where all the components are progressively initialized in order to run and use the Linux operating system.

ROM Code

The first stage is executed in both boot modes, where the master processor initializes only the basic system clocks [18]. In engineering mode, the Cortex-A7 allows the co-processor to use some of its functions and also enables full debugging capabilities of both CPUs. The debugger is an external function and it can be used even when only a single CPU has to be debugged. Afterwards the master goes to an infinite loop and the co-processor may begin software execution. In production mode it is needed to find the boot source from which a bootloader will be initialized. The default boot device is the SD card, which can be selected by switching boot pins 0 and 2 to position 1. The STM32MP157C-DK2 board used in this thesis has only two switchable boot pins, 0 and 2, whereas pin 1 is set to 0 by a pull down resistor [23]. This means that if a different boot source needs to be selected, it has to be specified in the primary boot source field of One Time Programmable (OTP) fuse box used for on-chip non-volatile storage [16]. When the clock initialization is done and boot pins are correctly set, the ROM code loads the first stage bootloader (FSBL) from the boot device into embedded RAM. On the trusted boot chain, a loaded image needs to be authenticated before an execution begins to ensure it is legitimate [3]. The initialization procedure by the ROM code can be seen on the following figure.

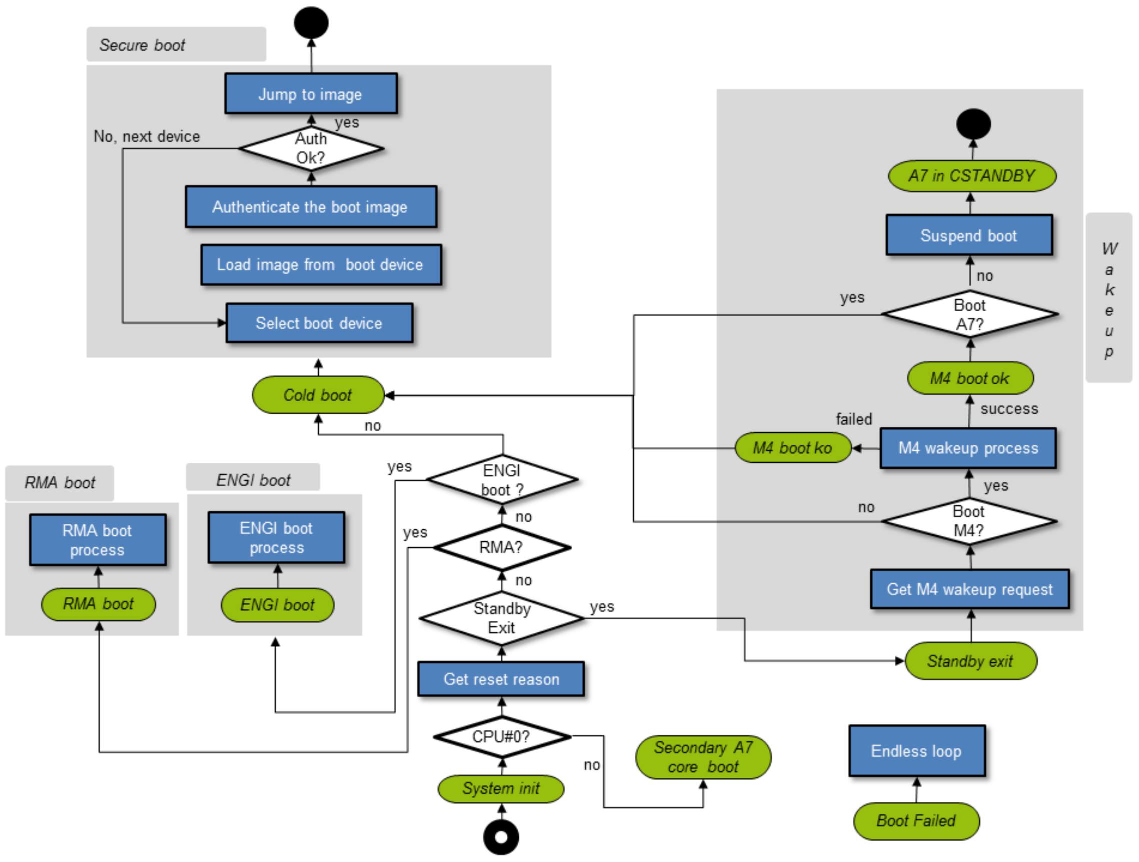


Figure 2.7: ROM Code Overview[6].

FSBL

In the FSBL, the entire clock tree is initialized and external RAM is prepared for loading secondary stage bootloader (SSBL) [3]. The FSBL bootloader depends on the boot chain which has been flashed onto the board. On the trusted boot chain, the Trusted Firmware-A (TF-A) is used as the FSBL to initialize all security features available on this board, which is also the default and recommended boot chain. The basic boot chain is available as well, which provides only a limited set of features. In that case, the U-Boot SPL (Secondary Program Loader) is loaded by the ROM code.

SSBL

The secondary stage bootloader is running in the external RAM, so it is able to load more complex devices and features such as USB, Ethernet or display [3]. U-Boot is commonly used in embedded systems and it is used as SSBL in both trusted and basic boot chains. In this stage, the boot file system is loaded from storage with kernel image and device tree blobs. In addition to Linux operating system, the co-processor firmware running on Cortex-M4 can be initialized and booted by the SSBL, which is referred to as „early boot“.

[6]ROM Code overview. STMicroelectronics. [online]. [cit. 2020-04-23]. Retrieved from: https://wiki.st.com/stm32mpu/nsfr_img_auth.php/1/17/ROMCodeOverview.png

Linux kernel

In the last stage of boot sequence the kernel initializes all peripherals and devices that are needed by the system, which means that the root file system is mounted and the user space initialized [3]. In addition to SSBL, the co-processor firmware can also be booted by the Linux remoteproc framework [17].

Device Tree

During the boot process of operating system, the kernel needs to locate all hardware devices and map them to their respective kernel drivers, so that the system can use them when they are needed. This process is called hardware probing and usually it is done dynamically by ACPI [1]. On embedded systems there are hardware components that can't be probed and without any information about their existence, the operating system wouldn't be able to address and use them. All modules that have been described earlier (UART, SPI, and I^2C) must be registered manually, which can be done by the device tree. According to the device tree specification [7], it is a tree data structure with nodes that describe devices in the system. Each device has its own characteristics which are described by property/value pairs. Each node has exactly one parent except for the root node, which has no parents. Some of the characteristics of a device can include their registers, clock speed, state and many more, which may be relevant for a specific hardware component.

These specifications may need to be changed by developers to match their application needs. The files that contain all the required information about available devices are called device tree source files with a *.dts* file extension or device tree source include files with *.dtsi* file extension. Once modifications are completed, the device tree compiler creates a device tree binary file, which is then parsed to the bootloader during boot sequence.

Each software component (U-Boot, TF-A and Linux kernel) has its own device tree bindings in order to correctly initialize the system. From the description of boot sequence, the trusted firmware (TF-A) is a first stage bootloader, which indicates that it initializes some components first. In this stage, system wide characteristics that apply to all domains can be specified. This is especially important if some resource has to be isolated for the Cortex-M4 domain only or if a secure peripheral must be available for non-secure environment. It is required for the Linux kernel device tree to be configured accordingly, because probing a device which is unavailable to the non-secure environment would cause the system to freeze and the boot sequence fails.

The Yocto Project and OpenEmbedded project

STMicroelectronics offers 3 different packages to use for preparing the software and development environment:

- Starter package[7] - It's purpose is to provide default OpenSTLinux images with 3 different boot chains [3]:
  - Basic boot chain - U-Boot bootloader is used as FSBL with only limited features, which means that also security features are missing from the image.

[7]STM32MP1 Starter Package - images. STMicroelectronics. [online]. [cit. 2020-04-23]. Retrieved from: https://wiki.st.com/stm32mpu/wiki/STM32MP1_Starter_Package_-_images

  - Trusted boot chain - Trusted Firmware-A is used as FSBL, it is a recommended boot chain with many security features.
  - OP-TEE boot chain - It is basically a trusted boot chain with support for OP-TEE secure OS.

- Developer package - It is used by developers for cross-compilation of their applications in a Linux operating system. It sets all necessary variables in a terminal, such as the default compiler or target architecture for easier development.

- Distribution package - The starter package does not provide all software components that might be required, so a distribution package can be used to build a custom distribution image with some packages added or even removed.

OpenEmbedded project is a build framework which enables developers to cross-compile their own custom distribution for embedded devices according to their needs. The starter pack uses the official „st-image-weston" image [14] with a basic Wayland/Weston display framework included on all 3 boot chains. It can be built using a tool named „bitbake" [13], which functions similarly to „make". Instead of a Makefile, it is provided with a recipe or a set of recipes, where each of them represents a software component. Usually, a recipe contains following metadata:

- Source code location - It can be an URL address from which the source code can be downloaded or a local directory.

- Package dependencies - A list of packages which need to be built and deployed along with the selected package for proper functionality.

- Configuration, build, install and remove instructions - Specific options on how the package needs to be treated during each phase (eg. build options).

These recipes are often organised in a collection called a layer. Each layer represents one major software component such as a kernel or a board support package (BSP), which are basically a combination of all the recipes contained in the layer. Developers are also able to create their own layers for their applications and build them into the distribution. A single image can be built from multiple layers, including a core layer provided by OpenEmbedded or platform specific layers such as the BSP from STMicroelectronics.

The Yocto Project is is an open source collaboration project that grew from and works closely with the OpenEmbedded project and it helps developers with creating Linux-based systems for embedded devices. It provides additional software metadata and development tools (devtool[8]) for adding, modifying, building and deploying software without the need of re-flashing the board. Nowadays, the difference between the OpenEmbedded project and Yocto project is insignificant with both focusing on software contributions to the community.

The OpenSTLinux is a distribution based on the OpenEmbedded build framework, that contains the BSP (boot chain, Linux kernel and OP-TEE secure OS) and application frameworks (eg. the Wayland-Weston display framework) [14]. STMicroelectronics develops several layers (like the BSP layer) which are combined with the OpenEmbedded core layer to provide a starting point for developers to create user applications on the platform.

[8]Devtool. Yocto Project. [online]. [cit. 2020-04-23]. Retrieved from: https://www.yoctoproject.org/software-item/devtool/

2.5 Security

Once the sensors are connected and running along with the MCU sending data to a Cloud storage, it is mandatory to ensure secure connection while maintaining data authenticity, integrity and confidentiality to avoid data theft or forgery. There are some specific security measures for accomplishing this goal while securing internal peripherals as well and they will be described in this section.

Extended TrustZone protection controller

As mentioned before, sensors can be connected through various interfaces that are supported by the board. These interfaces are available through either GPIO or Arduino pins and they can be easily isolated from every component and used only by the ARM Cortex-M4 domain. For this purpose, an Extended TrustZone protection controller (ETZPC) [16] can be used to configure security attributes that will allow or block accesses to the specified interface. All peripherals are connected to the Advanced Peripheral Bus (APB) or an Advanced High-performance Bus (AHB), on which they can be configured as:

- Secured - The communication with a secure peripheral can be only done from a secure environment, while other non-secure accesses are being ignored. The OP-TEE secure OS[9] and Trusted Firmware-A[10] (TF-A) running in the ARM TrustZone secure environment can communicate with interfaces configured as secure.
- Non-secured - The Linux operating system and the co-processor firmware are considered as non-secure environments and they can only access non-secure interfaces.
- Isolated - Only the co-processor has access to isolated hardware modules, whereas everything else including both ARM Cortex-A7 domains are blocked.

However, interfaces that are available on the GPIO and Arduino pins can not be set as secure in this context, which is why the focus will be isolation only [16]. The ETZPC security attributes can be configured by modifying the TF-A device tree. Peripherals that are securable are by default available only in the secure environment and isolable modules in the non-secure environment. If the default value needs to be changed, it can be done in this device tree by adding a DECPROT line with 3 parameters, where the first parameter is the required peripheral. The second and third parameters then configure security attributes:

- DECPROT_S_RW - For usage in the OP-TEE OS or TF-A (secure).
- DECPROT_NS_RW - For usage in the Linux OS or the co-processor firmware (non-secure). In this case a 3rd parameter can be set as well:
  - DECPROT_LOCK - Generates an error once the Cortex-M4 attempts to use a peripheral which is already used by Cortex-A7.
  - DECPROT_UNLOCK - In contrast, it allows continuous accesses by both domains, but it may cause errors so a hardware semaphore should be used.
- DECPROT_MCU_ISOLATION - Only the Cortex-M4 can access these peripherals.

[9]Open Portable Trusted Execution Environment (OP-TEE). Linaro Limited. [online]. [cit. 2020-04-23]. Retrieved from: https://www.op-tee.org/

[10]Trusted Firmware - Open Source Secure World Software (TF-A). Linaro Limited. [online]. [cit. 2020-04-23]. Retrieved from: https://www.trustedfirmware.org/

OpenSSL

OpenSSL is an open source software library[11], which can be used in applications that require secure connections or that are manipulating secure data. It is a full featured toolkit for Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols, licensed under Apache License 2.0, which means it can be used for commercial and non-commercial purposes under some conditions. The toolkit contains multiple libraries and tools that can be used for specific needs:

- **libssl** - Contains an implementation of all TLS protocols up to version 1.3.
- **libcrypto** - A cryptographic library, which includes data encryption and signing methods.
- **openssl** - A command line tool used for creating, analyzing and testing certificates, keys, encrypted and decrypted data, message digests and many more.

There are a wide variety of algorithms that can be used in OpenSSL, which can be listed by using the aforementioned utility. To name a few from each type:

- Message digest algorithms - MD4, MD5, SHA-1, SHA-2, SHA-3, BLAKE2, RIPEMD-160, SM3, GOST
- Cipher algorithms - AES, DES, RC4, SM4, SEED, ARIA, CAMELLIA, CAST5
- Public key algorithms - RSA, DSA, DH, EC, HMAC, CMAC, X25519, POLY1305, SIPHASH

Data signing

The data that have been obtained from a sensor and transferred to the master processor need to be further processed and secured before sending them over the network to a remote cloud. The 3rd party which will be reviewing the data must be ensured that their source is legitimate. The data (message), which can be either in form of a textual string or a file in form of a document or an image, must keep its integrity and authenticity in order for them to be successfully verified by the public. The data integrity is only accomplished when a message could not have been possibly altered during transmission. Hashing (message digest) algorithms are one-way functions, which transform messages into unique strings of values that are very hard to decode back to their original form. There is only one hash for a specific message, which means that everyone who computes a hash from the same message will get the same result. When the message is sent over the network, its hash is included with it, so it can be compared and verified when it arrives to the recipient. This only solves the integrity part, because the receiver could still have obtained a message from a different source than expected and a hash alone wouldn't be able to detect it.

As the data can be modified along with the hash, the problem becomes the authenticity. It can be accomplished by using a signature algorithm to modify the hash before it is sent through the network. These algorithms generate a pair of keys in the first stage of signature generation, a private and a public key. The private key is known only to the

[11]OpenSSL. The OpenSSL Project. [online]. [cit. 2020-04-25]. Retrieved from: https://github.com/openssl/openssl

creator of the message and it is used to modify the calculated hash. Giving access to the private key would mean that anyone could generate valid signatures. The public key is, as its name suggests, available to the public for decoding signatures. The verifier then takes the message, signature and public key, decodes the signature using the public key, generates a new hash using the same hash algorithm as the message sender and compares the results. A match of message digests indicates a successful verification, otherwise the message has been tampered with. Following diagrams roughly describe how signatures can be generated and verified through a Cloud database.

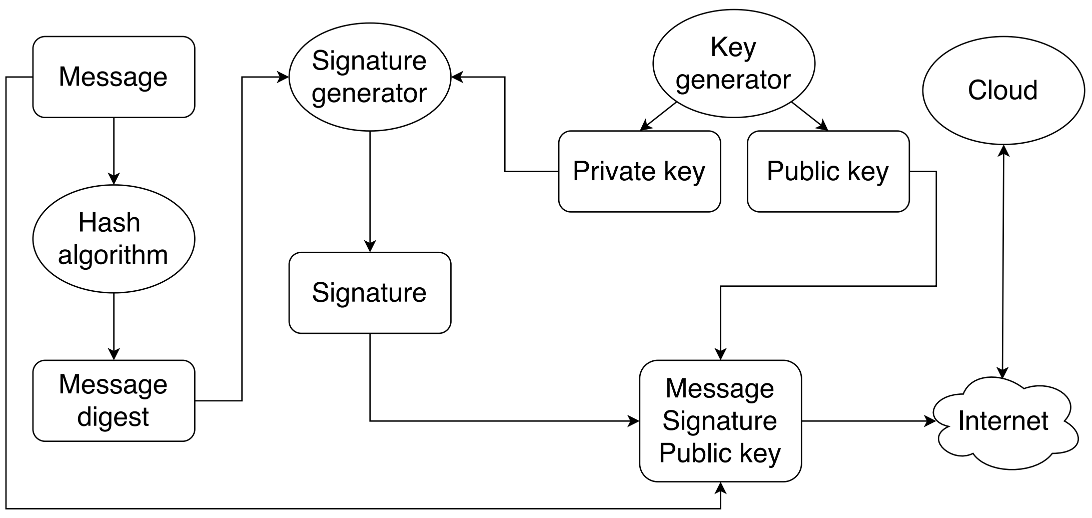


Figure 2.8: Signature generation process.

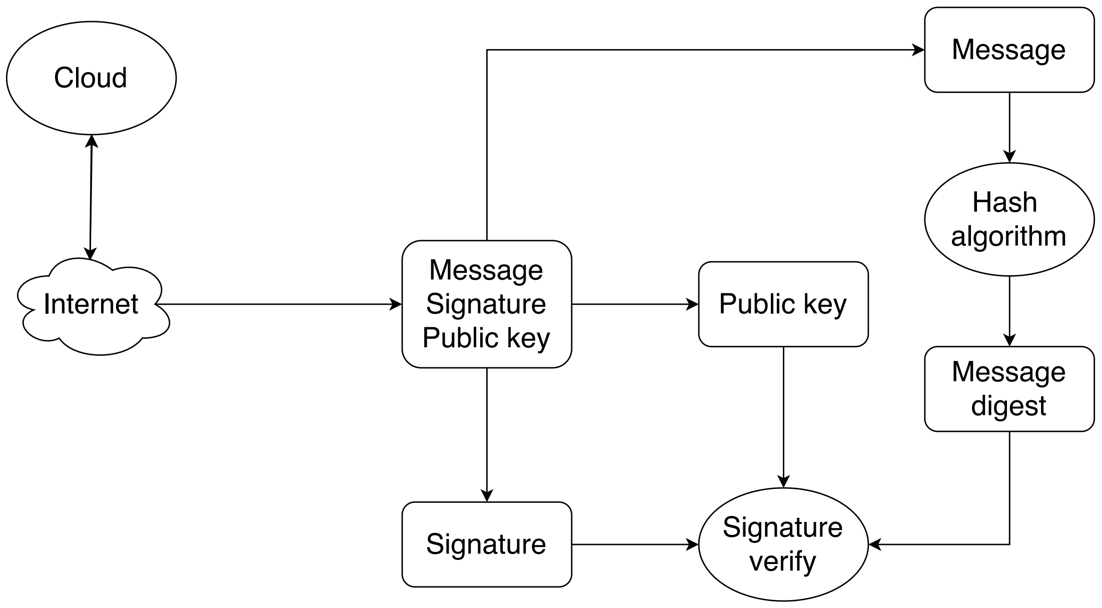


Figure 2.9: Signature verification process.

Hash algorithms

A message which was produced by a sensor can be signed by a signature algorithm using a unique private key, which is generated and stored safely on the platform. However, the data are inconsistent and usually of a significant size, which implies that the signature would be large and slow to generate [4]. A more effective way is to first create a hash of the message, which is of fixed size and it **uniquely** represents the original data. The emphasis on uniquely is important, because some hashing algorithms are obsolete and their insecurity has been proven. Algorithms that are not suitable are MD5 [12] and SHA-1 [21], which were found insecure by using a collision attack. In this kind of attack two different messages can be found with the same hash value. This means that data, which used a hash from one of the aforementioned algorithms could be counterfeit and it would not be detected. The most common algorithms in 2020 are from the SHA-2 family and the fact that the SHA-256 has been adopted by the Transport Layer Protocol in version 1.2 (TLSv1.2) [22] only proves its strength. Size of the output hash from this algorithm is bigger in comparison to MD5 or SHA-1, which means that there is very slim, almost non-existent chance of generating the same hash for different messages and creating a collision. The stronger SHA-384 or SHA-512 algorithms use even larger digests, but are also taking longer to generate. It is important to note that the STM32MP1 is capable of hardware acceleration of SHA-256 algorithm, which improves compute performance and will be described later in this thesis.

Signature algorithms

One of the oldest available methods that can be used to secure data is RSA (Rivest–Shamir–Adleman). It is a cryptosystem which is mostly used for data encryption, where the public key is used for encryption and only the private key can decrypt it. In case of signing, the process is exactly opposite, because it is not confidentiality that is being implemented, but authenticity and integrity. The RSA is based on factorization of large numbers, which means that the larger the private key is, the better protection it offers. The NIST and ECRYPT-CSA recommendations [15] for RSA key size to ensure short term protection (approximately 10 years) in 2020 are at least 2048 and 3072 bits respectively. However, large keys require more memory, storage and CPU cycles for creating a signature in comparison to what ECDSA offers. ECDSA is based on DSA (Digital Signature Algorithm) which is a standard for digital signatures based on the mathematical concept of modular exponentiation and the discrete logarithm problem. Whereas original DSA offered the same key lengths as RSA, the elliptic curve DSA (ECDSA) is using a lot smaller keys while being as secure as the other two mentioned alternatives. A key length of 192 and 256 bits is equivalent to a RSA/DSA key length of 1024 bits and 2048 bits respectively [8]. This makes the requirements for computation a lot smaller and more suitable for embedded systems with limited resources in comparison to personal computers. However, speed tests that have been conducted to compare RSA and ECDSA show that the signature verification procedure is executed faster by the RSA, whereas signature generation is faster by ECDSA [8]. The Montgomery reduction algorithm can be used to optimize the ECDSA sign and verify process by using shift operations instead of modular reductions [8]. ECDSA is also used on the platform for verification of the FSBL before the boot process begins[12].

[12]STM32MP15 secure boot. STMicroelectronics. [online]. [cit. 2020-04-25]. Retrieved from: https://wiki.st.com/stm32mpu/wiki/STM32MP15_secure_boot

Hardware acceleration and Crypto API

The STM32MP1 discovery kit offers many security modules and their respective software components for user applications. On microcontrollers including the STM32MP1 there is generally smaller computation power, so offloading any complex tasks would leave space for other applications that require more CPU power. Fortunately this can be done by taking advantage of hardware acceleration for either ciphering, message digest or CRC calculation by modules referred to as CRYP, HASH and CRC respectively. Their Linux kernel drivers are available to be used by the Crypto API framework, which in turn can be accessed by two user interfaces:

- AF_ALG[13] - This interface is based on a socket user interface, which can be bound to an AF_ALG data structure containing information about the required operation. The communication is then done by either „read“ and „write“ functions or „send“ and „recv“ functions, which are commonly used with BSD sockets. It can also be used as an engine in OpenSSL software library, which makes it more user friendly and easier to implement.
- Cryptodev - It is a kernel module which provides access to the Crypto API and can as well be used as an engine in the OpenSSL software. The main page of the module states that it is faster than aforementioned AF_ALG interface, however the comparison graphs might be outdated at this point[14].

Refer to the following figure for better understanding of how the user interfaces, framework and kernel drivers are connected to provide hardware acceleration for user applications.

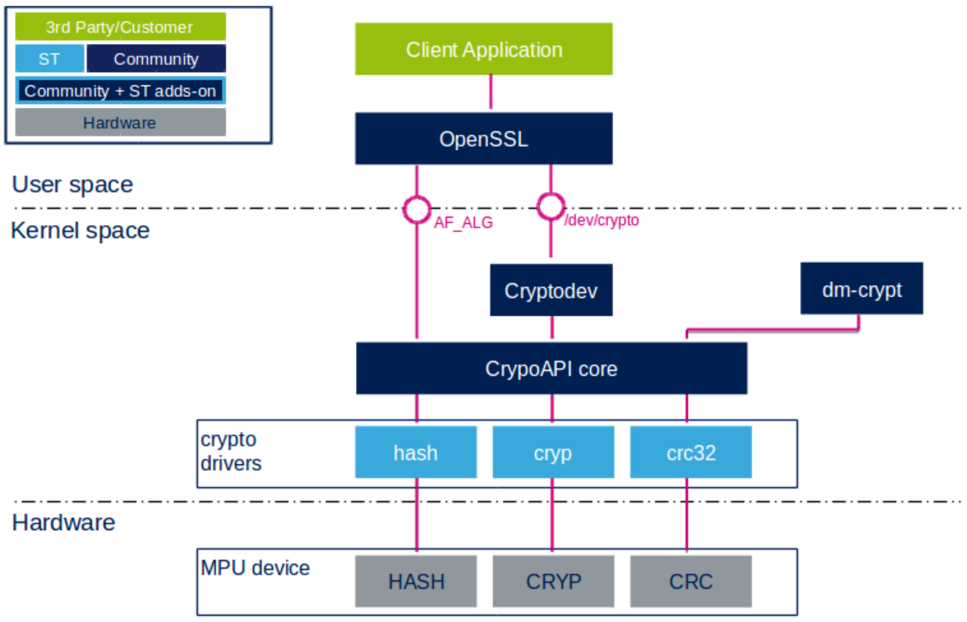


Figure 2.10: Crypto API framework.

[13]User Space Interface - The Linux Kernel documentation. The Kernel development community. [online]. [cit. 2020-04-25]. Retrieved from: https://www.kernel.org/doc/html/v4.10/crypto/userspace-if.html

[14]Cryptodev-linux module [Comparison]. Phil Sutter, et al. [online]. [cit. 2020-04-25]. Retrieved from: http://cryptodev-linux.org/comparison.html

Chapter 3

Design

The goal of this thesis is to implement a data collection software while taking advantage of the STM32MP1s potential. The data shall be collected from various sensors that use different communication techniques, where most of them communicate via common interfaces such as SPI or I^{2}C. It is not unusual to collect data from these sensors with the Cortex-M4 as it is a part of many MCUs. It only makes sense to use it for sensor communication to balance the utilization of computing resources on this platform. The most important part of the solution is cryptographic security to ensure data authenticity, integrity and confidentiality after they are sent over the network. Accomplishing this goal is a computationally intensive task, which means it should be done on the Cortex-A7 instead. For this purpose, there is a built in support for an OpenSSL library, which simplifies the implementation and can take advantage of hardware acceleration to even further optimize the performance. The Cortex-A7 runs a custom distribution of Linux, which means there is a file system for storing collected data as well as internet connection capability, which are key for collecting data to a remote Cloud. Speaking of Cloud, Google provides solutions with comparable specifications for a lower price than Amazon making it the Cloud of choice. The connection can be automatically encrypted with the use of TLS and SSL to ensure data confidentiality during transmission, which is of course supported in their solutions as well. At last, the platforms hardware acceleration as well as signing and verifying performance shall be assessed to gain knowledge of its capabilities in real world applications.

3.1 Software environment

The software environment on the platform consists of bootloaders, Linux kernel, kernel space drivers and frameworks and user space applications. The STM32MP15 starter package offers 3 different builds (boot chains) with Weston/Wayland display framework that can be flashed onto the board, but they are lacking some key components that will be required later in the implementation. Fortunately, the distribution package can be taken advantage of to modify an image with needed software packages. The Yocto project and OpenEmbedded project build system allows developers to select, modify and deploy software components that are not available in the default starter pack or are improperly configured.

STMicroelectronics provide a remote repository that can be used to obtain all available distributions. Every layer and recipe is downloaded to a local directory on a computer for building selected images with included packages. The build environment must be initialized first by an environment setup script included in the repository. The first package required is „opencv“ package, which in this implementation will be used for manipulating raw image data. Another package that needs to be modified is the TF-A software component, specifically its device tree configuration of ETZPC. By default, I^2C peripherals are configured as non-secure, but the one used for collecting data on the Cortex-M4 domain has to be isolated. For signing messages a hash has to be calculated, which can be done with a software algorithm or a dedicated hardware module. The hash peripheral also has to be enabled in the kernel device tree, so its corresponding recipe must be loaded and its device tree modified accordingly.

Since the OpenCV library had to be included, the entire image had to be built before flashing it onto the board by STM32CubeProgrammer software. It is necessary to put the board in programming mode by switching both boot pins to value 0. The software is then able to connect to this board via USB and flash a selected boot chain on the board. The boot chain of choice is „trusted“ boot chain, which uses TF-A secure firmware as the first stage bootloader. After the flashing process is complete, both boot pins have to be set to value 1 for SD card boot and it may be powered up. After a power cable is plugged in, the Linux operating system is booted up and ready to be used.

3.2 Sensor selection

The main factor for choosing a sensor should be the communication interface that it uses. Of course, it also depends on what kind of data and how much of them are expected to be received over the interface, how many sensors will be connected simultaneously, what is its accuracy or price and so on.

For the purpose of this thesis, a HTU21D sensor has been provided for measuring temperature and humidity. The measurement range is from -40 to 125 degrees Celsius and 0 to 100 % relative humidity (RH). It uses the I^2C communication interface, which supports speeds of up to 1 Mbit/s on the STM32MP1. The amount of data received from this sensor are expected to be only a few bytes, in fact the data are sent over 3 8-bit data packets (3 bytes of data), where the last byte is checksum and it can be skipped. The transmission speed is considered to be more than sufficient and it won't cause any slow-downs. However, this particular sensor has a fixed I^2C address, which means it can not be changed and used with multiple identical sensors simultaneously, at least not over the same interface. It is not a problem either as in this implementation there is only a single sensor, but in case more of them are needed, a different type should be selected with an option to change the address or use a SPI sensor instead.

Except for a simple I^2C sensor there is also an intelligent camera present for collecting and transmitting images with some additional metadata over an Ethernet interface. This communication can potentially reach the highest speeds of them all, but implementation will be done on the Cortex-A7 side unlike the first sensor, which is going to be communicating with Cortex-M4. The camera runs a TCP server, which waits for a client connection (from the MCU) and starts sending encoded images in base64 format along with some additional metadata in a JSON structure. It is important to note that the image data are raw. This means that it has to be reconstructed with the use of metadata, which include an image width and height required for correct image shaping. Received images are 8-bit images, so a single pixel consists of 8 data bits resulting in 256 different colors (2^8 possible combinations). The OpenCV library mentioned earlier shall be used to shape the image and save it in a bitmap file format with 8 bits per pixel.

3.3 Connecting sensors

Since the camera sensor is using an Ethernet interface, its connection is quite straightforward. It only requires an Ethernet or Ethernet-over-USB cable connection. However, the I^2C sensor is trickier, because there is no standard cable for connecting to this interface. I^2C needs two wires for communication, but also voltage input and grounding as well. Some GPIO connectors on the board can be used for plugging in this sensor via „I2C1“ or „I2C5“ peripherals. According to HTU21D data sheet[1], the input voltage is typically 3V and it can handle up to 3,6V maximum. GPIO pins can supply either 3,3V or 5V and because 5V is too high, the first variant must be chosen to avoid sensor damage. The user manual provided for STM32MP1 lists all GPIO pins and their functions, so the only thing to do is to wire corresponding pins to the sensor and solder them together. It is better to use an external connector which can be later unplugged if a different sensor has to be used or for protecting aforementioned GPIO connectors.

3.4 ARM Cortex-M4 firmware development

Now that the sensor is connected via an I^2C interface, the next step is to develop software for collecting data and sending them over the RPMsg framework to the master. There are many programs and IDEs that can be used to develop a co-processor firmware, such as SW4STM32 or the more recent STM32CubeIDE. The latter one seems to be a better option, because it contains STM32CubeMX software for automatic code initialization and generation. It has multiple perspectives (window configurations for a specific purpose) including a CubeMX perspective, which can be used to configure pins, peripherals, libraries and clocks in a graphical user interface (GUI). This is a very important feature, because it makes the development a lot quicker by requiring programmers to write only a few lines of code. A hardware abstraction layer (HAL) and OpenAMP libraries are also available for sending and receiving data via I^2C and communication with master via RPMsg respectively.

3.5 OpenSSL and data signing

At this point all data are collected and accessible from Linux. Before they are sent to a remote Cloud, they need to be protected from unauthorized access, data theft and forgery. The first step is to sign the data with a signature algorithm, which is a computationally intensive task. Any opportunity for improving performance and lowering processor usage would be welcome. Fortunately, the board has multiple modules available for hardware acceleration of these tasks. An important part of creating signatures is hashing, which can be hardware accelerated by a dedicated hash peripheral. Two user interfaces that have access to this peripheral have already been described and a better option seems to be the „cryptodev“ module due to offering higher performance than its „AF_ALG“ alternative often by a significant margin. It can also be used by OpenSSL as an engine, which means that OpenSSL has access to its functions and can use the hash peripheral for hash calculation. The OpenSSL C library allows for defining an engine in the program, which can afterwards be used by the high level abstraction interface called „EVP“. With this interface it is quite simple to create a pair of keys, generate hashes and sign data with only a few

[1]HTU21D Sensor – Miniature Relative Humidity and Temperature Sensor. Measurement Specialties. [online]. [cit. 2020-04-28]. Retrieved from: https://www.cdiweb.com/datasheets/te/htu21d.pdf

lines, just like with the HAL library for peripheral communication.

As both SHA-1 and MD-5 algorithms are vulnerable to collision attacks and are not recommended for creating hashes today, a stronger alternative has to be selected. The SHA-256 algorithm matches this requirement and it is also commonly used in the TLS protocol since version 1.2. It is less intensive for computation and requires less storage than SHA-384 or SHA-512, which makes it ideal for use in embedded systems. ECDSA signature algorithm is being used on the platform for verifying the first stage bootloader and it uses smaller keys equivalent to a lot larger RSA keys in terms of their strength. ECDSA relies on a secure elliptic curve key and thanks to OpenSSL there is a wide selection of them. Elliptic curves used for signing and verifying the FSBL on the platform are NIST P-256 and Brainpool 256 [20]. The NIST P-256 curve is also listed in Federal Information Processing Standards (FIPS) under recommended curves making it an ideal choice [9].

The OpenSTLinux (version 1.2.0) distribution comes with OpenSSL C libraries and Python version 3.5 installed by default. Having more alternatives gives developers more choices, since there are multiple software wrappers for OpenSSL including one for Python. The problem with official Python wrappers is lack of support for ECDSA signature algorithm. Hardware acceleration functionality is also complicated to confirm in these wrappers. The best option would be to use the original C library which already has support in the default distribution and contains every functionality that might be necessary.

There are 3 scenarios of data collection, which means that received data and their signatures are always different. When a user chooses to collect both temperature and humidity, it is more efficient to sign both values at once and store only one signature instead of two. With that in mind, it is always necessary to select both values from the database for successful verification. For this reason there are 3 different options, so if only humidity or temperature is selected separately, they can be collected, signed and stored separately as well.

3.6 Relational database design

Collected data and their signatures have to be stored either in a file on a remote storage or a database. Relational databases are capable of data analysis in order to obtain a useful information. It is important to design the system to be as efficient as possible with no redundancy. It is already known that collected data and their metadata have to be stored, but also signatures and public key/keys for verification. There is also an expectation of simultaneous data collection by multiple MCUs, which requires a table containing unique device IDs. Created signatures are also unique, but they correspond to only a single row. It means that creating separate table would not only have no benefit, it would also be ineffective due to the need of joining multiple tables. On the other hand, public key is expected to be only one and storing it in each row would cause the database to scale very quickly.

Since there is a choice for collecting data at once or separately, all database tables must be created accordingly. A single table would be able to do the trick, with a possibility of a „null“ value in both temperature and humidity columns. If a mistake happens and the data are missing, there would be rows with only signatures and metadata. A database trigger can be created to check if there is at least one value, which would be executed after every single insert operation, thus impacting performance. Even with a trigger, the developer would always have to check for a „null“ value in either of these columns, which seems ineffective as well. If both columns are occupied with data strings, they have to be concatenated and only then they may be verified, because that is how they were signed in the first place. Signing them separately would require twice as much storage for two separate signatures. To simplify this process with no performance or storage impact, the solution can be done with 3 tables for each scenario. The design might seem redundant, but in terms of performance and simplicity it seems to be the best option. In conclusion, there are 5 tables in total:

- Device - Contains a unique ID of a device that collects data to the database and its public key. In case a private key is lost and a new one has to be generated along with a new public key, there may be more entries for the ability to verify older signatures as well as new ones.
- Temperature, Humidity and SensorData - All 3 tables for 3 scenarios of collecting data from the HTU21D sensor.
- Image - A single row contains an image path in storage, its metadata and a signature.

3.7 Cloud setup

In chapter 2.3 some services offered by Amazon and Google for Cloud storage were described and compared. It may be assumed that the better choice of those two is Google due to it being offered for a lower price on an almost identical configuration. At the time of software implementation, Google also provided free credits worth of 300$ in a time span of one year, that can be used on most of their services including database and storage options. This seemed like a good opportunity to learn about how it works without the risk of financial loss. After all credits are spent or the time limit passes, the price per month is expected to be similar to the initial estimation in Figure 2.5.

The Cloud SQL service is their name for relational databases, which can be based on either PostgreSQL, MySQL or SQL Server 2017. Unfortunately, SQLite that has an available C library on the STM32MP1 can not be selected. This is also not an option on Amazon RDS services, so a good alternative could be MySQL. Data stored in the database are expected to be of a relatively insignificant size. Temperature and humidity or image path with additional metadata such as timestamps do not take a lot of storage space. Queries where this kind of data will be retrieved with a public key and a signature can also be considered as simple, since they only require a single join operation. This kind of analysis can be easily handled by both MySQL and PostgreSQL, but in simple queries MySQL is faster and PostgreSQL also consumes a lot of memory per connection [6]. Since the database system is expected to have only a small amount of tables with simple analytical operations executed on them and hundreds of connected devices, the MySQL seems to be a better option.

The database can be set to only allow secure SSL connections, which are encrypted and thus allowing data to be confidential during transmission. However, connections from public IP addresses are disabled and only devices with an address in exceptions list can connect. For this reason, it would be better if the MCU had a static IP address, so the setting would have to be done only once. If only SSL connections are enabled, the device has to provide SSL server certificate, client certificate and client private key. Only then the connection may be established and data collection can begin.

In terms of remote storage, securing the connection works similarly. Only a user with authorization key can obtain data from this storage, because otherwise anyone could have access to collected data. A file with authorization key can be created and downloaded from the cloud management system and loaded by the Google Cloud storage library when the program is executed. The connection is also encrypted and secured via TLS, so even images remain confidential during transmission.

3.8 Data collection

The cloud is prepared, database tables are ready and authentication is enabled. Data, metadata and signatures are ready to be sent over the network. That is a simple task with the use of MySQL and Google storage libraries, but the connection may not be stable. It can time out and be unreachable for an unknown amount of time. All collected data can be stored offline during the service unavailability. They can be simply stored in local files in defined directories and sent over the network once a connection is established. For that purpose, a multiprocessing capability needs to be implemented for simultaneous data collection and connection manipulation. The main process, which is called a parent process can be in charge of connection and in the meantime all processes it creates (child processes) can be used to collect data. However, child processes have their own memories and they can only access their own variables. This means that a process, which was collecting data before a connection was established has no information about it, so it is unable to send the data. It could use a shared memory to communicate or keep writing them into a file. The parent process then retrieves them from shared memory or accesses written files in a selected directory. However, it may cause errors and inconsistencies if they are accessed simultaneously. The control can be implemented with semaphores, which allow only a single process at a time to execute some part of code. If one process is writing into a file, the second one should wait until writing is complete and only then access the file. This way there is a perfect synergy where each process takes care of its own code. A better option is to save the data to a file instead of using a shared memory, because if connection times out it has to be written locally anyway.

3.9 Assessment of results

Reviews of hardware acceleration support on this platform are limited, so it is important to assess its performance. The „cryptodev“ module is supposed to be faster than its AF_ALG user space alternative and it should be properly analyzed to see what the differences really are. Better results with higher computation speeds shall improve the overall signing performance allowing for many more generated and verified signatures per second. The OpenSSL command line utility can be used to measure speeds of different software algorithms for hashing, ciphering or CRC calculation. As both user space interfaces can be used as engines in OpenSSL, the hardware acceleration performance can be also measured with the utility. If engines are not available, the AF_ALG interface can be simply implemented with the use of sockets in a separate program and tested manually.

In addition to hashing and signing performance, the results of data collection should also be assessed at the end. With some minor effects on the temperature sensor, it can be seen how it reacts to environment changes and how the database looks after the collection is done. For the camera sensor, some results with collected images on a remote storage can be displayed as well.

Chapter 4

Implementation

The key part of this thesis is the software solution implementation, which includes preparation of software environment and cloud, data reading, securing transmission over the network and more. Every single part will be described in this chapter under following sections.

4.1 Custom distribution image

During the implementation, there was a need for compromises due to several issues that have been encountered. One of the issues was a problematic building and installation of external packages, that are not included in the distribution package. The MySQL library that is needed for connecting to a database and the Google Cloud Storage library for remote storage connection have to be built from an external source. Unfortunately, this process has always failed and a different approach had to be chosen. The distribution includes Python by default, which had to be used as an alternative. Installation process of both Python packages was simple and a connection could be successfully established. These packages are not installed by default, but they might be downloaded with Python package installer (pip). However, it is not a requirement as packages can be installed manually, but since the image has to be built with OpenCV and modified device trees anyway, adding one more package would not make a big difference.
As already mentioned, there is a repository that can be used to download all distributions and modify them. The environment setup script is included and it needs to be run with some environment variables for proper workspace initialization:

- **DISTRO** - Defines a distribution with a selection of images that can be built and flashed on the machine [14]. The default distribution is „openstlinux-weston“, which features the Weston display framework and is used in the starter package. The framework support is not necessary for the implementation, but it may be useful in the future for displaying information on the screen.
- **MACHINE** - Selects a series of devices for which an image from the distribution will be built. It has to be set to „stm32mp1“ since the platform used is an STM32MP157C-DK2.

After running the environment script a workspace is initialized and changes can take place. For the default distribution an „st-image-weston“ image can be used, which is also the same as the image used in starter package. A basic core image without the Weston framework support can be selected as well, but as mentioned earlier it might be needed later. At this stage some additional software components can be added. The setup script generated a few directories and files based on distribution and machine selection. In order to include additional packages, a file called „local.conf“ located in „conf“ directory of the workspace must be modified. Both „python3-pip“ and „opencv“ recipes for Python package installer and OpenCV respectively are included in this configuration file separated with a white space.

The last line of this file should look like this:

```
IMAGE_INSTALL_append += "opencv python3-pip"
```

Since the OpenCV is required and also can not be deployed afterwards, the image must be built even if the Python package installer was skipped and all Python libraries were manually added. This process takes a considerable time, since there are over 8000 recipes, or more than 8000 software components. Thankfully, this only needs to be done once, every other time only dependencies or affected components are rebuilt to support a new package, which is a lot faster process. If its not the first time an image is being flashed, it is important to backup everything beforehand in order to avoid loss of sensitive data. Some packages can be also built separately, without the need of building and flashing the entire image. As already mentioned, this approach was not successful with pip, because Python version 3 is already included without some pip dependencies. Some applications that are not in any way dependent to other packages such as OpenSSL can be deployed while the operating system is running.

At this point all necessary packages are included and and a build process may be started:

```
$ bitbake st-image-weston
```

The devtool utility can be used to setup all package sources into a workspace and afterwards edited, built and deployed onto the boards root file system or flashed through the STM32Programmer utility. Preparation of software environment includes setting up the ETZPC isolation of I^2C peripheral, which the sensor will be connected to. Security features including ETZPC are only available in „trusted“ and „optee“ builds, but since the OP-TEE operating system won’t be used, the „trusted“ version will be flashed instead. All settings related to ETZPC are done in the TF-A device tree source file, which needs to be edited using devtool.

The exact package name can be found by issuing a search command in the devtool utility:

```
$ devtool search tf-a*
```

There are usually many results when searching with partial names. However, it is quite simple to figure out that the package is named „tf-a-stm32mp“. If it is not entirely clear, all recipes can be found online to read more about their content.

To add a package to the workspace and modify its source, the modify command can be used:

```
$ devtool modify tf-a-stm32mp
```

All device tree source files are now present in a „fdts“ sub-directory of the TF-A source folder. Name of the device tree source file is identical to the boards model name, which is STM32MP157C-DK2. When this file is opened and reviewed, the device tree configuration can not be found. Everything is inherited from STM32MP157A-DK1, because it has the same TF-A device tree. After examination of this included file, there are already some settings present for other peripherals under ETZPC section, which are required for successful boot of the Linux operating system. It's important to note, that if the original source file is to be used, the entire ETZPC configuration has to be copied from the included file. All peripheral settings in the original file have precedence over settings in included files, which means they are overwritten. However, in this implementation it is simpler to modify the included device tree file instead, because it only requires adding one line and its content does not have to be preserved.

Isolation of a peripheral can be simply done with a single line:

```
DECPROT(STM32MP1_ETZPC_I2C5_ID, DECPROT_MCU_ISOLATION, DECPROT_UNLOCK)
```

The sensor is connected to an „I2C5“ peripheral (refer to section 4.3 for more information), which is why the first parameter contains its ID. The other two parameters have been described earlier to properly set up isolation.

Now the file can be saved and built using devtool:

```
$ devtool build tf-a-stm32mp
```

The output of build process including device tree blobs is now in the workspace. However, the final image does not contain these changes as they have not been deployed yet. This is where the bitbake utility is used to update a distribution image, which has already been built with OpenCV and Python package installer.

The right command to deploy modified software has following syntax:

```
$ bitbake tf-a-stm32mp -c deploy
```

The TF-A software is modified and deployed, but the hash peripheral is still needed for hardware acceleration. Since it is disabled by default, it can be enabled now in the device tree of Linux kernel before flashing. The process of this operation is very similar to modification of the TF-A, except that the modified software component is different and its device tree file as well. Name of the recipe is „linux-stm32mp“ and the source file with device tree configuration is **arch/arm/boot/dts/stm32mp157c.dtsi**, which is a device tree include file. The hash peripheral can be found under „hash1“ section, where only its status needs to be changed. Setting its value to „okay“ will enable the peripheral for use in Linux. It is important to note, that this specific hashing module is enabled by default in ETZPC for the non-secure environment, so if for some reason its assignment is missing in TF-A device tree configuration, it needs to be added for the system to boot and use it. Now the Linux kernel and its device tree can be built with the same devtool and bitbake commands that were used for TF-A.

At this point, the ETZPC isolation is configured, hash peripheral ready for hardware acceleration, Python package installer and OpenCV library included in the image. To flash this image, the board has to be put in programming mode, connected via USB to a computer and the STM32CubeProgrammer utility started. Besides images there are also flash layout files for all boot chains, STM32MP1 boards and the SD card boot device. They define partitions, their sizes and offsets for all software components. After a layout is chosen, the source directory has to be selected as well and flashing may begin.

The isolation can be confirmed by enabling „I2C5“ in the Linux kernel device tree, which will of course cause the system to hang. In that case, its configuration was successful and isolation is working. Since the hash peripheral is now running as well, its proprietary drivers for hashing algorithms of a name beginning with „stm32“ are also loaded.

Available algorithms can be listed by executing this command:

```
$ cat /proc/crypto
```

4.2 Cloud database and storage

The next step before starting data collection is to create a database instance. There may be more instances that would separate multiple MCUs, but it would result in problematic data correlation and analysis. One database instance for data collection from both types of sensors will be enough and support for multiple connected devices will be done through table management and design instead. In the instance creation form some changes have to be made from default values. Server region should be changed based on the MCU location to have as low latency as possible. Some other settings such as disk size, RAM size and CPU core count can be changed as well, but are not mandatory for simpler queries with low data volume. Disk size is dynamically resizing when usage is approaching the limit anyway, so this problem resolves itself. Timing of backups can be also changed to some time during night when usage is minimal, but this can be modified later if required.

Now that the instance is ready and running, the database itself has to be created. This process is very simple and requires only a database name and character set, which by default is UTF-8 and it supports all possible characters that will be stored. In the „Overview“ page of the database instance the IP address should be saved, because it will be used for remote connections. However, if a connection was attempted right now, it would be ignored and it would simply time out. First, the network to which a MCU is connected has to be authorized by adding its IP address in the „Connections“ page. It can also be noticed in the SSL section of this page that unsecured connections are allowed, which of course is not recommended. Enabling secure only connections will require all devices to provide a server SSL certificate and a client SSL certificate. A client SSL certificate is composed of two files, the client certificate and client private key, that can be downloaded after they are created. Select a unique name for each client and download all generated files. By default, these files should be stored in „auth“ directory, which is a part of the software implementation root directory. It can be changed to a different path, but it has to be defined in the main script that will be collecting data as well.

In the SQL directory there is a file with 5 table definitions for data storage. All Python scripts are programmed to work with these tables:

- **Device** - Each device that connects to the database and wants to collect data needs to be uniquely identified. Both scripts for data collection have a „device“ variable with a unique name to avoid potential inconsistency. The second column is also unique and it contains a public key for verifying signatures. There may be more public keys for a single device in case a private key gets accidentally deleted. In that case, the public key would also have to change, but previous keys are still needed for verifying older signatures. All keys are stored in a „PEM“ format for simplifying the verification process, but it could be optimized in case of limited bandwidth or for lowering monthly costs. This table is expected to contain only a few rows with one device, so the size of it should not be an issue. However, with increasing amount of devices and sensors, every opportunity to optimize is important, whether its due to finances or database performance.

- **SensorData** - The sensor chosen for this thesis is collecting data about temperature and humidity, which are in range of -40 to 125 degrees Celsius or 0 - 100% relative humidity (RH) respectively. Data type for this column could be float or double, but there is a problem with the format in which they are stored. The value is always converted to a string with 2 decimal numbers and it is also signed in this form. For example, a value of „10.10“ would be stored in the database as „10.1“ if float or double data type was used, which would result in a failed verification. For this reason all values are stored as strings instead, so trailing zeroes are not removed and data can be successfully verified. As there is a possibility of choice between both temperature and humidity or only one of them, there are two more tables named **Temperature** and **Humidity** for this purpose. The co-processors command with all related configuration on the masters side is initialized by a command line parameter passed to the Python script. The „Signature“ column must contain a signature string encoded in base64 format with its size usually being 97 characters, but it may not be the case all the time. There is some overhead to avoid potential problems, so the column size has been set to 110 characters. Some extra metadata for the time and date of collection are stored in the „Created“ column with its „datetime“ data type. To verify a signature, a public key is needed as well, which is referenced in the last column to a row in the „Device“ table.

- **Image** - Since images are usually of a couple kilobytes in size, it is more effective to store only a path to the image in a database row. Database storage is a lot more expensive than a dedicated storage designed for unstructured data, so this approach seems to be more reasonable. However, the database has no knowledge about the presence of images in storage, which can lead to inconsistency.

The location type has been changed to a single region in Frankfurt to lower the total cost, but if there's a need of backups, multiple regions can be used as well. The standard storage class is the option of choice since the access frequency is going to be on daily basis. Retention policy can be enabled as well to make sure that images will not be removed until they are no longer required. Fine grained access can be used to create permissions for specific objects, which may not be necessary, but the other option becomes permanent after some time and changing back would not be possible later. At last, a globally unique name of the bucket has to be specified for accessing it remotely. However, the access will be limited to only authorized users with a special token.

This token needs to be generated and downloaded to all devices that will access this storage. On page „Service Accounts“ of „IAM & Admin“ section some specific access permissions may be defined. A service account can either have full access, edit access or view access only. As images will be written, there is a need for at least edit access permissions. It is important to create a private key for such access type and download it in a JSON format. All storage operations relevant to the implementation are in the „gstorage“ script inside „include“ folder. The „client“ function creates an object used for accessing remote buckets and the authentication file path has to be passed to this function as an argument.

4.3 Co-processor firmware

Before the software implementation begins, the sensor has to be physically connected. Both figures shown underneath can be used to help with proper connection of all related wires. It can be seen that pin number 1 provides voltage input of 3,3V and a ground wire can be connected to pin 6. Since the HTU21D operates in range from 3,0V to 3,6V, those two pins are ideal for power supply. Figure 4.1 labels pin 1 as „CN2 pin 1“, which is on the bottom left corner of the GPIO connector block.

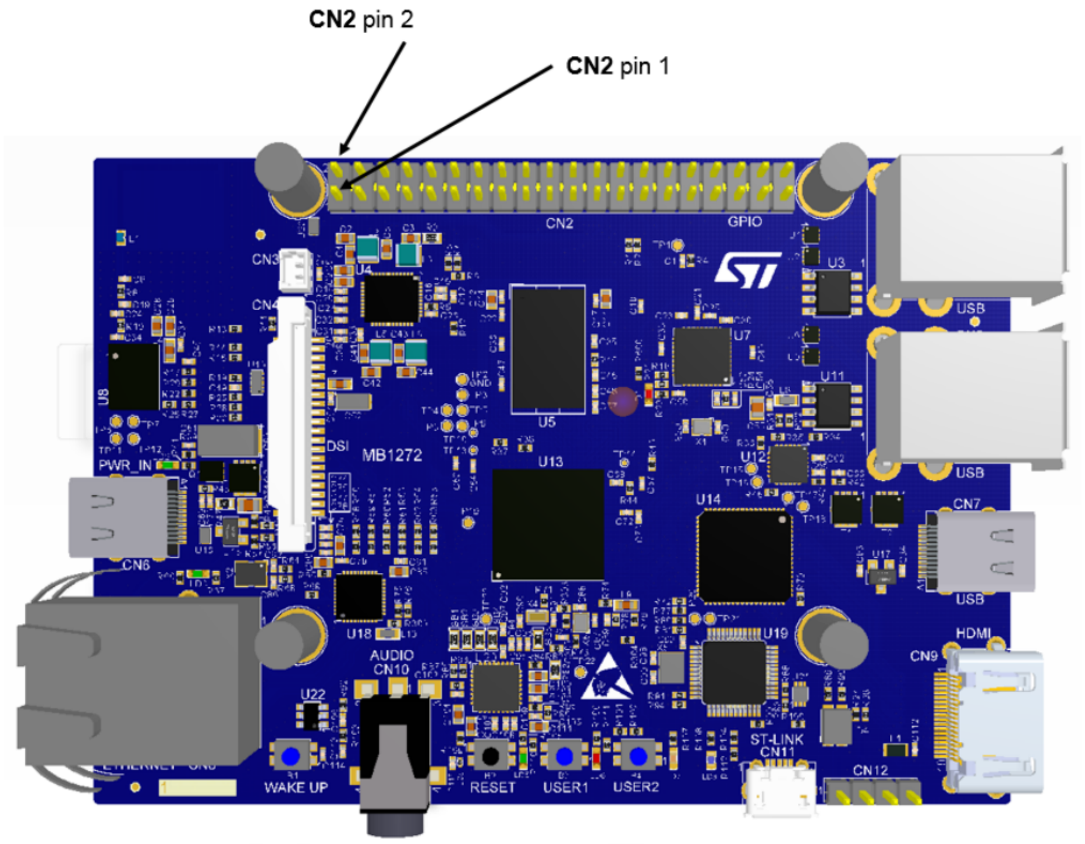
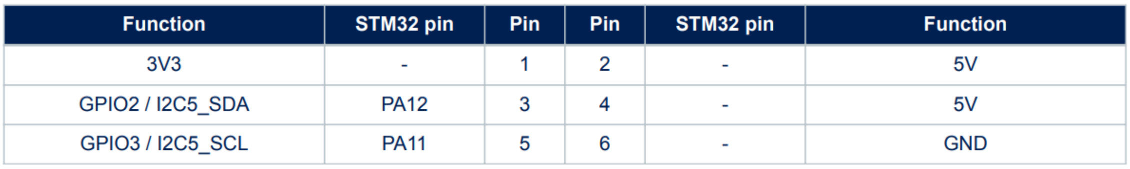


Figure 4.1: GPIO connectors [23].



Function	STM32 pin	Pin	Pin	STM32 pin	Function
3V3	-	1	2	-	5V
GPIO2 / I2C5_SDA	PA12	3	4	-	5V
GPIO3 / I2C5_SCL	PA11	5	6	-	GND

Figure 4.2: GPIO connectors pinout [23].

Now it is time to head to the STM32CubeIDE and create a new project. The IDE has to download and prepare the boards firmware package containing all drivers and middlewares, so a proper model needs to be selected first. Afterwards, the project can be created and a device configuration tool, otherwise known as CubeMX perspective is displayed. All necessary components and peripherals for data collection can be configured and enabled in this perspective.

The I^2C interface can be used on multiple pins, so to obtain correctly generated code, the GPIO pins that are physically connected to the sensor have to be manually selected. Following figures show how the CubeMX tool can be used to generate necessary initialization code for the I^2C peripheral.

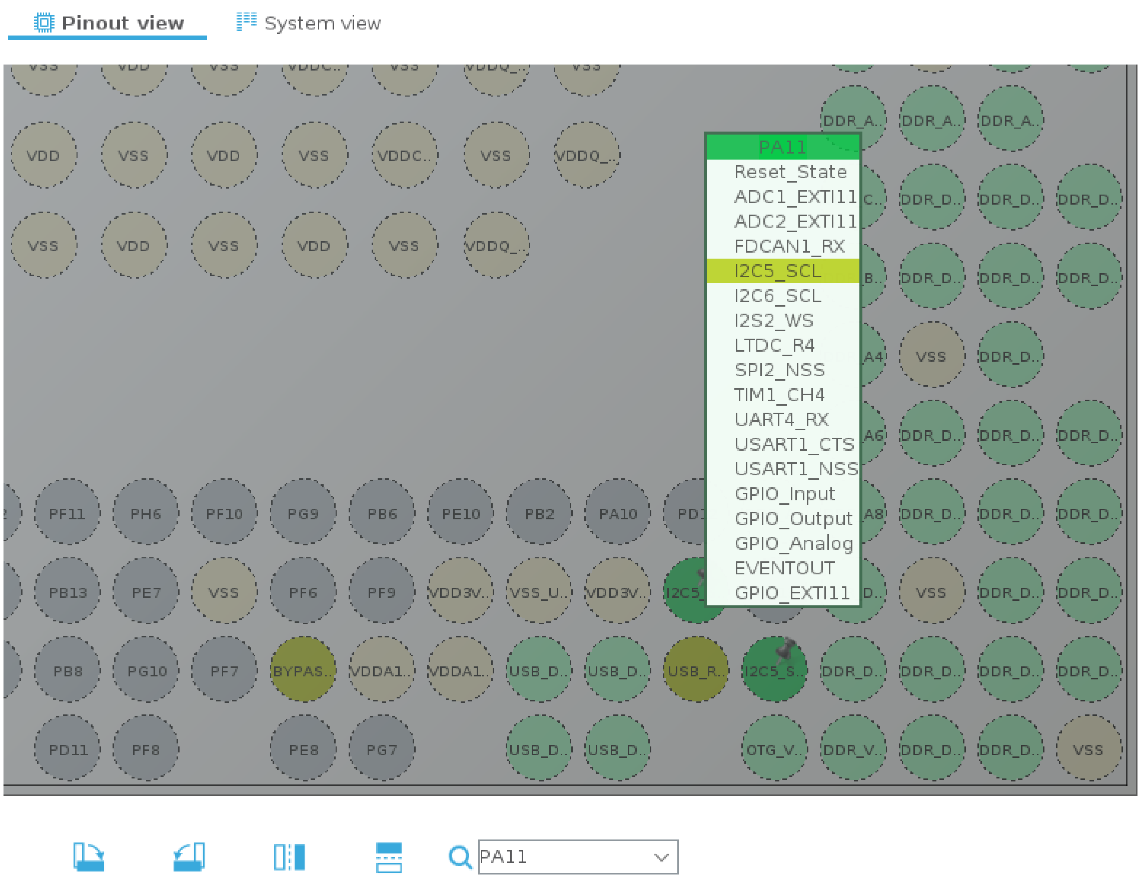
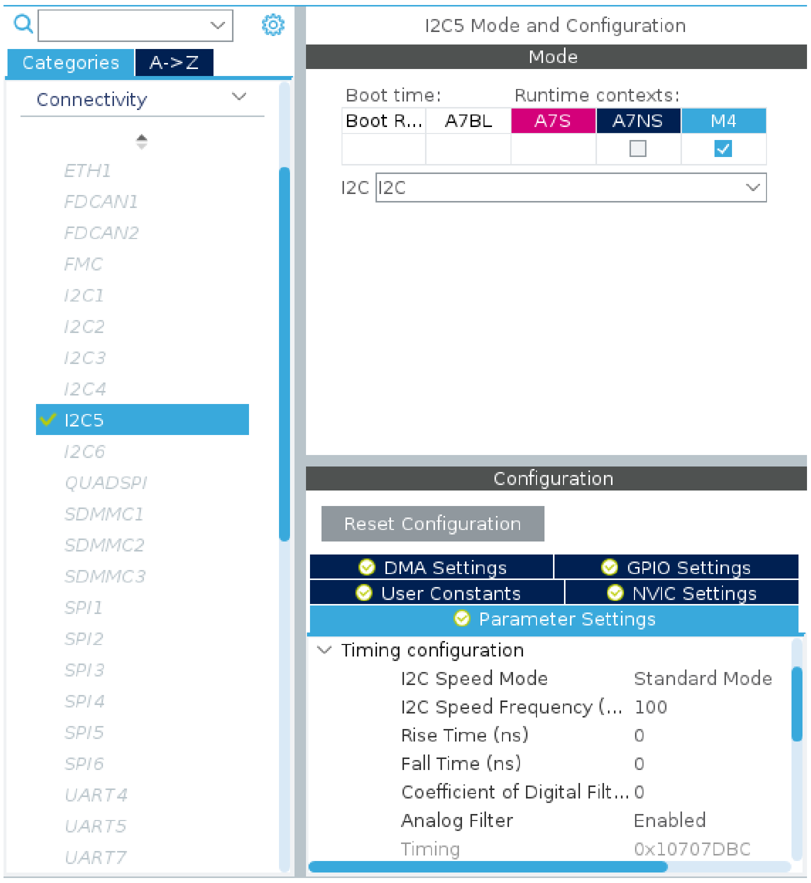


Figure 4.3: The CubeMX perspective pinout configuration.

Both pin names can be found after searching for them in the pinout view search box. According to figure 4.2, the pin name of a „I2C5“ clock wire is „PA11“. Now its correct functionality can be selected, which in this case is „I2C5_SCL“. The same process has to be repeated for the data pin and afterwards the peripheral may be enabled as shown on the following figure.



I2C5 Mode and Configuration

Boot time:		Runtime contexts:		
Boot R...	A7BL	A7S	A7NS	M4
			☐	☑

I2C: I2C

Timing configuration

I2C Speed Mode	Standard Mode
I2C Speed Frequency (...	100
Rise Time (ns)	0
Fall Time (ns)	0
Coefficient of Digital Filt...	0
Analog Filter	Enabled
Timing	0x10707DBC

Figure 4.4: The CubeMX perspective peripheral selection and configuration.

The left side of the perspective provides a peripheral selection and configuration window. All available hardware modules are listed in logical categories and they can be enabled on multiple runtime contexts (domains). In this case, the „I2C5“ peripheral can be enabled only for Cortex-M4 domain or Cortex-A7 non-secure domain, which means that the secure environment does not have a direct access to it. It was already mentioned that GPIO and Arduino pin functions are not securable by ETZPC. Some other settings may be done in the configuration section, but those have been kept on their default values as changes are not needed.

As for other drivers and components, there is also a need for HAL, IPCC peripheral and OpenAMP middleware library. While HAL is available to use right away, both IPCC peripheral and OpenAMP library have to be enabled from this list. For correct usage of OpenAMP, both IPCC interrupts need to be enabled first, because RPMsg communication relies on them. Afterwards, both IPCC hardware module and OpenAMP library can be initialized. After all changes are saved, the program asks if code should be generated. After that, it is up to developers to implement data collection and communication with the Cortex-A7.

The code generator automatically inserts code for initialization of system clocks in engineering mode, GPIO pins according to the CubeMX pinout and all components that have been enabled with the use of Hardware Abstraction Layer (HAL). At this point the sensor can receive a command and obtain response on read request.

The HTU21D data sheet lists all necessary information for implementation. It states that the sensors address is 0x80 and it can respond to following commands:

- **0xE3** - Requests a temperature measurement. The data wire is held for this communication, so the master can not exchange data with other slaves (hold master).
- **0xE5** - The humidity measurement command is issued while the data wire is held as well.
- **0xF3** - Similar to 0xE3, but its possible to communicate with other sensors during temperature measurement.
- **0xF5** - Humidity measurement command without holding the data wire.
- **0xE6** - Data may be written to the user register for adjusting configuration, such as measurement resolution (accuracy).
- **0xE7** - Reads the user register configuration.
- **0xFE** - Issues a soft reset command.

The HAL transmit function can be used to send two data packets, where one of them contains a slaves I^2C address with a write bit and a second packet is one of the commands described above (in this implementation its either 0xE3 or 0xE5). Afterwards, the HAL receive function sends a read request by sending a data packet with sensors address and a read bit. The communication is held by the sensor until the measurement is done. It responds with 3 bytes of data and returns them to the master, who accepts the response and writes received data into a memory buffer. The 3rd byte is only checksum and it may be skipped. First 2 bytes are holding measured data and they have to be converted with a specific formula based on the measurement type (temperature or humidity) to get the correct value. Last two bits of the response are only status bits and they must be cleared prior to conversion. At this stage all data are collected in memory and they can be passed to the Cortex-A7. The I^2C communication sequence implemented in this thesis is shown on the following figure.

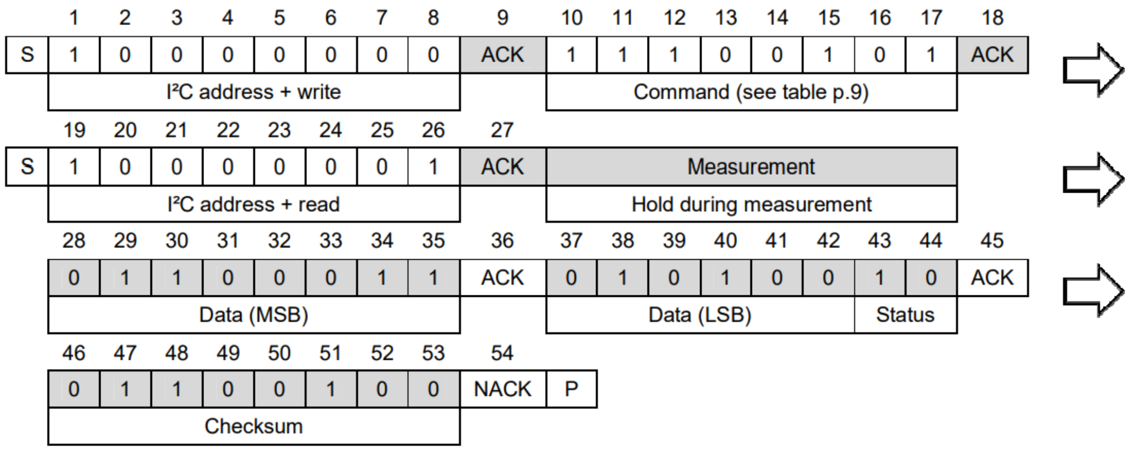


Figure 4.5: Communication sequence with hold master[1].

[1]HTU21D Sensor – Miniature Relative Humidity and Temperature Sensor. Measurement Specialties. [online]. [cit. 2020-04-28]. Retrieved from: https://www.cdiweb.com/datasheets/te/htu21d.pdf

4.4 Inter-processor communication

Collected data are now present in a memory buffer on the Cortex-M4 domain and are ready to be signed. Even if the Cortex-M4 processor has access to the hash peripheral, it is not enough to create a signature. Signing will be done on the Cortex-A7s side, which needs to have access to the data. The RPMsg framework on Linux side and OpenAMP library on the co-processors side can be taken advantage of to accomplish this goal. The STM32CubeIDE software can prepare the OpenAMP library and generate some initial configuration code. Its up to programmers to establish a communication channel with the RPMsg framework and read incoming messages from the master. There are many ways to implement this functionality and one of them includes these functions:

- **VIRT_UART_Init** - An OpenAMP endpoint is created and a communication channel is established.
- **VIRT_UART_RegisterCallback** - An incoming message can be processed in a callback function, which has to be registered before use.
- **OPENAMP_check_for_message** - As its name suggests, it simply checks for an incoming message in the Mailbox, which has been initialized by the HAL_IPCC driver.

The last function is called in an endless while loop, in which the co-processor waits for an incoming message. Once a new message has been received, the callback function is used for analyzing it and controlling data collection. In this implementation following messages can be used as a command for reading data:

- **TH** or **HT** - These messages trigger collection of both temperature and humidity from the sensor and are sent together in a response, where the first value is always temperature and second value humidity.
- **H** or **T** - Only humidity or temperature respectively is being collected and sent as a response.

If some other string of characters is received, an error message is sent stating that it is unable to recognize the command. An error message can also be received if the data collection fails, but this is possible only if the RPMsg communication is functional. If an error occurs in the early stages before a message can be sent, an LED has been configured to blink in an endless loop.

The Linux RPMsg framework establishes the connection on its own as soon as the co-processor creates an endpoint. The user interface for sending messages via this framework is mounted as a virtual terminal. A simple „echo“ can send a message via this virtual terminal and the „cat“ utility can read responses. The communication type seems to be half-duplex since the remote processor was unable to send a message before a request arrived from master. The Cortex-A7 has memory access first and it always initiates the communication, which only makes sense to only receive data when they are requested. However, in some cases a full-duplex communication is relevant, because some sensors are capable of notifying us about an event. For example, when a temperature rises above or below a defined threshold, a notice about such event should be received as soon as possible. It is not a part of implementation since the HTU21D does not support this functionality.

At this point, implementation of the Cortex-M4 firmware is complete. Before getting to data signing, there is a need to address some design choices and compromises. As already stated before, there is only a library for SQLite database communication, but that was not an option on neither Google or Amazon cloud services. The database is running on MySQL for which an external library could not have been built. Since it was not a viable option and an unknown 3rd party library would not be ideal to use, there is still Python as an alternative. The Python language has some advantages and disadvantages over C, but the installation process of all necessary packages was successful and it was ready to be used. With that in mind, there is yet another issue. The official Python wrapper for OpenSSL does not support ECDSA signatures and if a 3rd party ECDSA library would be used, it would still not support hardware acceleration. Either a compromise has to be done to skip hardware acceleration and use a 3rd party Python package, or the Python „ctypes“ library can be imported and used. It allows for usage of C libraries in Python, which could be created to implement signature generation and verification via OpenSSL with hardware acceleration enabled. The C language is also faster in code execution and the process of signing can be modified, which is also a disadvantage of wrappers.

In addition to the first issue, the original choice of „cryptodev“ kernel module could not be used in the implementation due to lack of support for hash algorithms. The usage of hash acceleration needs to be implemented on a lower level by using AF_ALG sockets combined with the ECDSA signature generation process in OpenSSL. Even though it does not perform as well as „cryptodev“, it still provides higher calculation speed than a software hashing algorithm with less CPU utilization in some particular cases.

So, the co-processor firmware is ready, but its not running yet. As already mentioned, it could be initialized manually with the „echo“ tool prior to running the script, but it is very simple to implement it in Python. In the „include“ directory there is a script with name „firmware“, which controls the remoteproc framework. With basic file functions commands like „stop“ and „start“ can be executed along with the firmware name that can be found in the „/lib/firmware“ directory. This is a default path where the remoteproc framework searches for firmware files, but a new path may be defined if needed [17].

4.5 Signature generation and verification

Since the hardware acceleration and ECDSA procedures can only be done in C, functions for signing and verifying will be called from a custom library that can be imported with the „ctypes“ package and used in Python scripts.

The co-processor firmware is now capable of sending responses with measured data, they just need to be accessed from the program. The „pyserial“ package for Python can be installed and imported for opening a communication via a virtual terminal, from which the RPMsg messages can be obtained. After that, a command of choice may be written to the virtual terminal with a simple write method. Each response contains a new line character at the end, so it can be read line by line and stripped from white characters. Functions that control the communication fill an array with data passed through an argument and return a string, which has to be signed. In case of „TH“ command, temperature and humidity are simply concatenated together and passed to a function for creating signatures. The array is filled with data that will be later sent to the database.

There are multiple C functions that can be called from within the script. There are default algorithms for signing and verifying strings and files implemented with the „EVP“ abstraction layer. These functions use a software algorithm for hash calculation. Signing strings can only be done with this implementation, since hardware acceleration would be slower with only around 5 to 20 bytes of data [5]. On the other hand, files are usually of a greater size and hardware acceleration would be useful. The „cryptodev“ kernel module does not support SHA-256 hardware acceleration, which is why it must be implemented via the AF_ALG user interface instead. Functions that use this interface does not cooperate with the „EVP“ layer as it works on a different basis. Instead, the AF_ALG socket is created, bound to a structure defining the algorithm to access a corresponding peripheral and finally it is opened. Then, the file content is pushed via a socket „send“ function to the peripheral with a flag „MSG_MORE“, which is necessary for updating the hash multiple times. This has to be done due to the fact that the entire file might be too large to be loaded and sent at once, which means it has to be divided into smaller parts called „chunks“. When all data are sent, the „recv“ socket function can finalize the calculation and provide the final hash. The only thing remaining is to allocate memory for the signature, generate it, convert it to a base64 format and return it to the Python script. However, the procedure is not finished yet, because data had to be allocated on a heap and now the memory has to be freed. The signature was returned as a pointer, so it has to be cast to a string and decoded for further use. All data are now saved in a Python variable and the received pointer can be passed to a function for freeing the allocated memory in the C library.

Verification functions have also been created to verify strings and files via „EVP“. They have to be called with 3 parameters, which is again a string of data or a file name, signature and a public key. The signature is converted back to its original binary format from base64, the public key is loaded to an OpenSSL structure for keys and the „EVP“ layer is used to calculate a hash and verify the signature. Since there is an AF_ALG alternative for signing, it can also be used in verification. It uses the same principle of hash calculation as in the signing procedure and it passes all 3 arguments with a hash to an OpenSSL verification function without the use of „EVP“.

4.6 Data transmission

The last part of data collection is to send them securely to the remote cloud. Python libraries used to connect to the database and storage were „pymysql“ and „google-cloud-storage“ respectively. The official MySQL connector for Python was not suitable due to an exception that could not have been fixed in the phase of disconnecting. Its alternative is working without any issue and it supports secure connections with the use of SSL certificates. A timeout can also be configured in case the database is not reachable, so it is possible to process data offline until the connection is established again. All data, metadata and a signature are saved in a data array, which is simply passed to a function that executes an insert query to the database. However, data can be obtained offline without the knowledge of a public key ID. Due to this problem, the public key is saved in a local file and its ID has to be obtained once a connection is established. In case of image files, the storage is updated prior to the database, because the other way around the connection could be lost and an image would not be present in the storage. However, if the file is manually removed from the storage, it has to be removed from the database as well, because otherwise it has no knowledge about such operation. It is possible to make a synchronization procedure on program startup, but with thousands of files it would take an unreasonable amount of time. It is in the interest of developers to not lose any important data, so a backup can be automatically scheduled in the Cloud console. In terms of remote storage, the Python library provided by Google can be used to connect and send or receive files securely with

TLS. The storage connection is initiated once a user is authorized with a private token obtained after creating a service account. It can be accessed through a client object, which executes all requests on the storage. First of all, it is important to create an object of a unique name on the remote storage, which represents a directory to which the program will upload files. Afterwards, the content of a local file can be sent to the storage. Its full path is stored in a data array with metadata such as a timestamp or a date and time of data arrival on the MCU. The image file that has been uploaded was also signed with the use of hardware acceleration and the signature along with the public key ID is saved in the array as well. As already mentioned, the file is first uploaded to the storage and only then the database is updated.

4.7 Offline processing

In order to collect data even when a connection is not available, they need to be kept somewhere until the connection is available again. In order to accomplish that, the program needs to do two things at once, which requires a multiprocessing technique. While one of the processes attempts to connect to the cloud and maintains this connection, the other process (or processes) can take care of data collection independently. This is due to the fact that the camera is automatically sending images and the process could be busy with a different operation. This is why there are multiple nested processes, which are forked after the initialization of all components. At first, the parent process creates a child process for collecting data, which in turn creates a new process every time a new detection is received on a socket (in case of camera). In this kind of environment, there is a high risk of simultaneous access to shared resources, which is why there is a need for a semaphore. If this semaphore is open and the child process receives data, it creates a new child process, which locks the semaphore and writes all data and metadata into a file. This file is considered a shared resource and it can not be accessed by any other process until the semaphore is unlocked. Before the process finishes, it unlocks the semaphore for another process to continue, which can be either another child process with received data, or the parent process that wants to synchronize with the Cloud. With this solution, child processes can simply keep writing data even if the parent process is unable to synchronize due to unavailable connection. To simplify the implementation, data are always saved into a file and the parent process periodically checks for new files in a directory for synchronization. The camera can also get disconnected, so the first child process that was created after initialization attempts to connect to this camera again and it keeps trying periodically until its successful.

Chapter 5

Assessment

The STM32MP1 series of microcontrollers is relatively new and there are limited performance reviews regarding its hardware acceleration capabilities. Besides performance assessment, final results of data collection into a remote cloud are also reviewed in this chapter.

5.1 Hashing and signing performance

Since a part of the implementation is a signature generation and verification process, it is a good opportunity to take advantage of the hash peripheral available on this platform. However, in some cases with low data volume it is not the most effective option, because the time it takes to transfer data to an external peripheral is longer than it takes a software algorithm to calculate a hash. Before proceeding to signing performance with ECDSA, the hashing performance needs to be assessed first. All hash calculation results are described in the following table.

Block size	Software SHA-256 (OpenSSL)	Hardware SHA-256 (AF_ALG)
16 bytes	4.062 MB/s	0.189 MB/s
256 bytes	16.825 MB/s	2.229 MB/s
1024 bytes	21.040 MB/s	8.159 MB/s
8192 bytes	22.694 MB/s	38.398 MB/s
16384 bytes	22.817 MB/s	52.447 MB/s

Table 5.1: Hash calculation assessment.

After conducting numerous tests there are a few notes to be made. It is interesting to see how much slower the acceleration is compared to a software algorithm with data blocks of size 1024 bytes and less. At the same time, a drastic improvement can be seen with blocks of 8192 bytes and larger and at some point it becomes infeasible to opt for a software solution. The OpenSSL speed assessment has been conducted using its integrated speed tester and it does not include blocks of 4096 bytes. The AF_ALG test has been created for the purpose of this thesis and it can be run with any block size defined. With 4096 byte blocks, the speed is equivalent to 24.971 MB/s, which means that it pulls ahead of software hashing at this point. The camera used in this thesis produces images of size between 3 to 5 KB, which is unfortunate as there is no major improvement for this particular case. However, the calculation speed is not everything as hardware acceleration also keeps the

CPU utilization considerably lower even with the smallest block size of 16 bytes. It starts at around 43% and goes down to 18% with increasing block size, which makes it a more viable option if processing time is limited.

It has been mentioned that the „cryptodev“ engine does not work with SHA-256 as of now, but it has been discussed and resolved with the help of STMicroelectronics community and integration team and a fix is going to be available in the next major software update. The next software ecosystem release version 2.0.0 is scheduled to be available at the end of June 2020, which is again unfortunate as it can not be implemented in this thesis. However, an early test has been conducted by STMicroelectronics staff to show the performance of „cryptodev“ in OpenSSL. These results are not official and they may or may not be the same in the final release, so it has to be taken into consideration before looking at the following table.

Block size	Hardware SHA-256 (cryptodev)
16 bytes	1.656 MB/s
256 bytes	39.314 MB/s
1024 bytes	109.589 MB/s
8192 bytes	747.110 MB/s
16384 bytes	1461.288 MB/s

Table 5.2: Early results of SHA-256 hardware acceleration with „cryptodev“.

If these results are accurate, there would be no advantage of software hashing other than the smallest 16 byte blocks, which is why both temperature and humidity are always signed with the default OpenSSL software algorithm.

As one of the most important factors for signing performance is the hash calculation, the number of created signatures per second also changes depending on hashing and the next table only proves this fact.

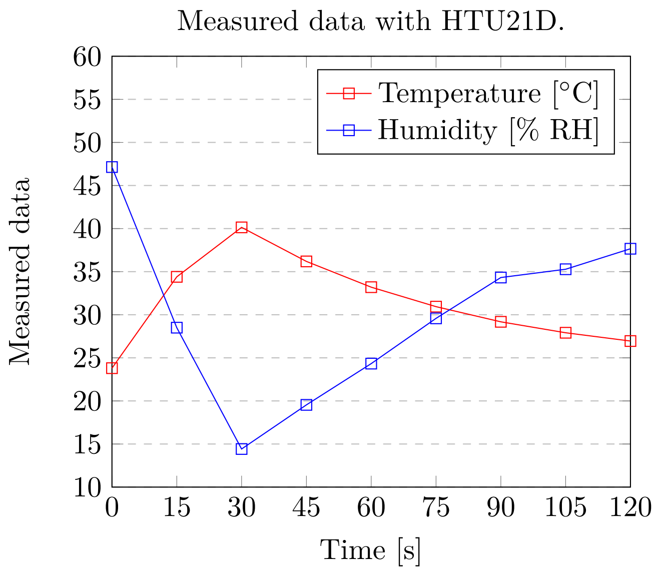
Signatures per second	HTU21D data (10 B)	Camera image (5 KB)	200 KB file
Generated (OpenSSL)	725	547	87
Verified (OpenSSL)	241	217	70
Generated (AF_ALG)	-	466	133
Verified (AF_ALG)	-	201	97

Table 5.3: ECDSA signing performance comparison.

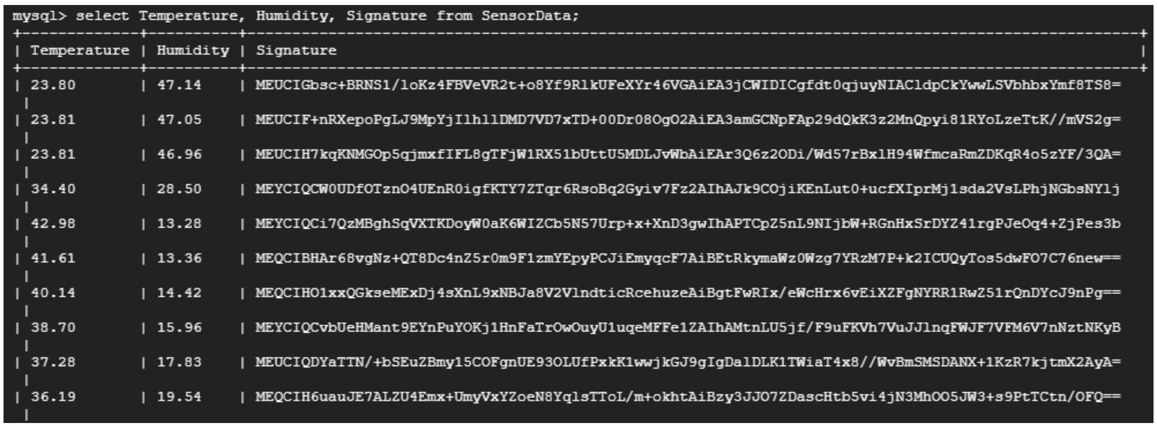
Results from the table above are from assessments where 4096 byte data blocks have been used to get comparable performance. In case of a tiny textual string, the block is almost empty and from the performance analysis of hashing it would be inefficient to create signatures with AF_ALG as there is an immense difference in speed. Unfortunately, a camera image is still signed and verified faster with a software algorithm, which may be due to the fact that the implementation required an external C library. In the implementation all socket settings have to be done in every function call, thus decreasing overall performance. In the end, there are still more signatures created and verified using a software algorithm rather than with a hardware acceleration support. The usage of acceleration only starts to make sense with larger files and larger data blocks, where the performance can be more than twice as high. In terms of CPU utilization this has been harder to measure as hashing is a lot faster process than signing. Signature generation or verification process after a hash is done can not be further accelerated, which means that the CPU had been almost always fully utilized. During these tests, the average utilization is roughly 93% with hardware acceleration support and 100% without it. However, in case of a large file that has to be divided into smaller blocks, the utilization drops to around 55%.

5.2 Data collection results

The HTU21D sensor can be used to monitor temperature and humidity periodically and in this implementation, a measurement is done once per 5 seconds. The following graph displays measurement of data during which the sensor has been affected by hot air.



In a time span of 2 minutes, there have been a total of 25 measurements made. As a result of hot and dry air there was a spike in both temperature and humidity at the beginning of measurement, following by slow recovery. All data and metadata with their signatures have been collected to a remote database table named **SensorData**. A part of its content is displayed in the next figure.



```
mysql> select Temperature, Humidity, Signature from SensorData;
+-------------+----------+-------------------------------------------------------------------------------------------------+
| Temperature | Humidity | Signature                                                                                       |
+-------------+----------+-------------------------------------------------------------------------------------------------+
| 23.80       | 47.14    | MEUCIGbsc+BRNS1/loKz4FBVeVR2t+o8Yf9RlkUFeXYr46VGAiEA3jCWIDICgfdt0qjuyNIACldpCkYwwLSVbhbxYmf8TS8=
 |
| 23.81       | 47.05    | MEUCIF+nRXepoPgLJ9MpYjIlhllDMD7VD7xTD+00Dr08OgO2AiEA3amGCNpFAp29dQkK3z2MnQpyi81RYoLzeTtK//mVS2g=
 |
| 23.81       | 46.96    | MEUCIH7kqKNMGOp5qjmxfIFL8gTFjW1RX51bUttU5MDLJvWbAiEAr3Q6z2ODi/Wd57rBxlH94WfmcaRmZDKqR4o5zYF/3QA=
 |
| 34.40       | 28.50    | MEYCIQCW0UDfOTznO4UEnR0igfKTY7ZTqr6RsoBq2Gyiv7Fz2AIhAJk9COjiKEnLut0+ucfXIprMj1sda2VsLPhjNGbsNYlj
 |
| 42.98       | 13.28    | MEYCIQCi7QzMBghSqVXTKDoyW0aK6WIZCb5N57Urp+x+XnD3gwIhAPTCpZ5nL9NIjbW+RGnHxSrDYZ41rgPJeOq4+ZjPes3b
 |
| 41.61       | 13.36    | MEQCIBHAr68vgNz+QT8Dc4nZ5r0m9F1zmYEpyPCJiEmyqcF7AiBEtRkymaWz0Wzg7YRzM7P+k2ICUQyTos5dwFO7C76new==
 |
| 40.14       | 14.42    | MEQCIHO1xxQGkseMExDj4sXnL9xNBJa8V2VlndticRcehuzeAiBgtFwRIx/eWcHrx6vEiXZFgNYRR1RwZ51rQnDYcJ9nPg==
 |
| 38.70       | 15.96    | MEYCIQCvbUeHMant9EYnPuYOKj1HnFaTrOwOuyU1uqeMFFe1ZAIhAMtnLU5jf/F9uFKVh7VuJJlnqFWJF7VFM6V7nNztNKyB
 |
| 37.28       | 17.83    | MEUCIQDYaTTN/+bSEuZBmy15COFgnUE93OLUfPxkK1wwjkGJ9gIgDalDLK1TWiaT4x8//WvBmSMSDANX+1KzR7kjtmX2AyA=
 |
| 36.19       | 19.54    | MEQCIH6uauJE7ALZU4Emx+UmyVxYZoeN8YqlsTToL/m+okhtAiBzy3JJO7ZDascHtb5vi4jN3MhOO5JW3+s9PtTCtn/OFQ==
 |
```

Figure 5.1: Collected temperature and humidity in the database.

The intelligent camera that sends detections in images is connected via an Ethernet interface. The STM32MP1 connects to this cameras TCP server and processes images as they are received. Images are reconstructed using metadata and signed with ECDSA. Afterwards, they are sent to the Google Cloud Storage while metadata are stored in a database table named **Image**. Following figures display the camera and resulting storage bucket with some collected images.

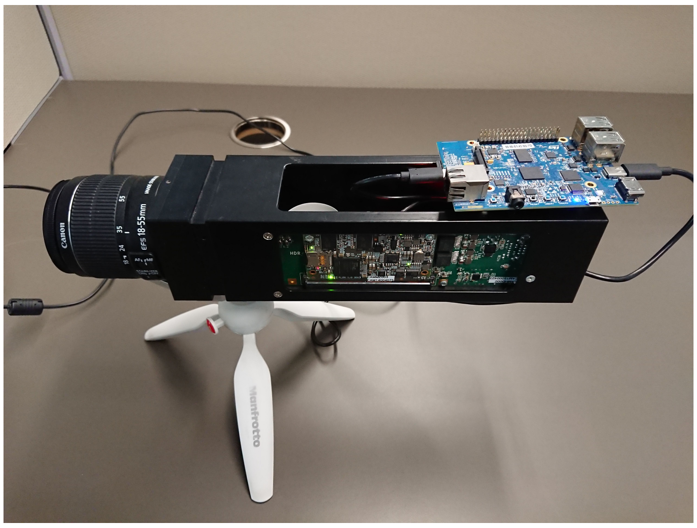
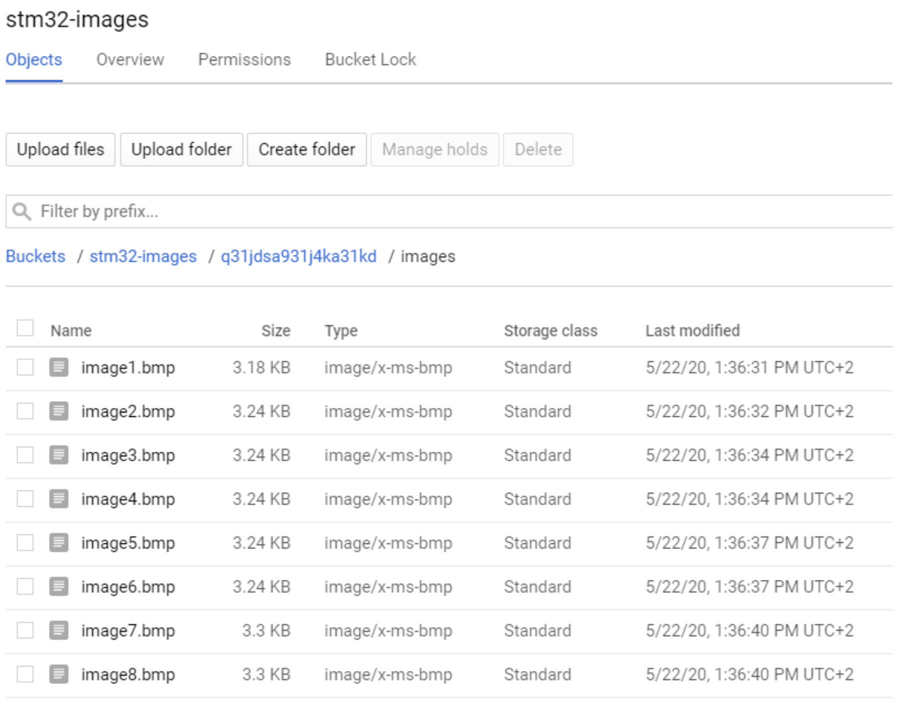


Figure 5.2: Intelligent camera sensor connected to the STM32MP1.



stm32-images

Buckets / stm32-images / q31jdsa931j4ka31kd / images

Name	Size	Type	Storage class	Last modified
image1.bmp	3.18 KB	image/x-ms-bmp	Standard	5/22/20, 1:36:31 PM UTC+2
image2.bmp	3.24 KB	image/x-ms-bmp	Standard	5/22/20, 1:36:32 PM UTC+2
image3.bmp	3.24 KB	image/x-ms-bmp	Standard	5/22/20, 1:36:34 PM UTC+2
image4.bmp	3.24 KB	image/x-ms-bmp	Standard	5/22/20, 1:36:34 PM UTC+2
image5.bmp	3.24 KB	image/x-ms-bmp	Standard	5/22/20, 1:36:37 PM UTC+2
image6.bmp	3.24 KB	image/x-ms-bmp	Standard	5/22/20, 1:36:37 PM UTC+2
image7.bmp	3.3 KB	image/x-ms-bmp	Standard	5/22/20, 1:36:40 PM UTC+2
image8.bmp	3.3 KB	image/x-ms-bmp	Standard	5/22/20, 1:36:40 PM UTC+2

Figure 5.3: Google Cloud Storage with collected images.

Chapter 6

Conclusion

The purpose of this thesis was to explore capabilities of the STM32MP1 series MCUs and find a solution for collecting data to a remote cloud while maintaining data security. Even with a heterogeneous multiprocessing environment, the processing power is somewhat limited compared to more robust computing systems, such as personal computers. For this reason, the main goal was to use a solution with as small impact on processor to ensure the highest performance possible. The data must have been secured by creating a unique cryptographic signature that ensures their authenticity and integrity. The platform comes with broad selection of hardware and software components for implementing security measures. Since creating signatures is a computationally intensive task, the dedicated hashing peripheral seemed to be a great advantage of this platform that made the signing process faster in some occasions.

In the end, the data collection mechanism on this platform has been successfully implemented. Thanks to the code generation tool, hardware abstraction layer and OpenAMP library, the co-processor firmware was the easiest part. On the other hand, the software on masters side was slightly more complex. Some of the required libraries for establishing a connection to the Cloud are not available in the standard distribution. For this reason, the Python language has been chosen for implementing the second part of data collection as building and importing additional packages was successful. All imported packages had required functionality except the official OpenSSL Python wrapper, which does not support elliptic curves at this time. Due to this fact, the signing part of implementation has been done in C with the original OpenSSL library. As large data volumes have a bigger profit of the hash hardware acceleration, it is also available as an option for signing files.

There are a lot of issues that need to be resolved in the future, but thanks to the STMicroelectronics community and integration team, the „cryptodev“ user space interface for accessing the hash peripheral is supposed to be capable of SHA-256 calculation in a new software release version 2.0.0 [5.1]. If the implementation could move from Python to C, it would not only improve performance, but it would also create an opportunity to use the software inside the ARM TrustZone secure environment. The OP-TEE OS or the Trusted Firmware-A are capable of executing C applications securely, which is not possible for Python scripts at this time. To be able to use only C, the database and storage libraries would need to be built from source, or the SQLite engine alternative would have to be used on a different Cloud service with a SFTP file server instead of the Google Cloud Storage.

Bibliography

[1] Unified EFI Forum, Inc. *Advanced Configuration and Power Interface Specification* [online]. version 6.2. 2017 [cit. 2020-04-24]. Available at: https://uefi.org/sites/default/files/resources/ACPI_6_2.pdf.

[2] Frenzel, L. E. *Principles of Electronic Communication Systems*. 4th ed. New York: McGraw-Hill Education, february 2015 [cit. 2020-01-27]. ISBN 9780073373850.

[3] STMicroelectronics. *Boot chains overview* [online]. 2019. revised 22. 1. 2020 [cit. 2020-04-23]. Available at: https://wiki.st.com/stm32mpu/wiki/Boot_chains_overview.

[4] Gauravaram, P. *Cryptographic Hash Functions: Cryptanalysis, Design and Applications*. Brisbane, Australia, 2007. [cit. 2020-04-25]. Dissertation. Queensland University of Technology. Available at: http://eprints.qut.edu.au/16372/1/Praveen_Gauravaram_Thesis.pdf.

[5] Winand, M. *Performance Impacts of Data Volume* [online]. N.d. [cit. 2020-04-23]. Available at: https://use-the-index-luke.com/sql/testing-scalability/data-volume.

[6] Yigal, A. *Sqlite vs. MySQL vs. PostgreSQL: A Comparison of Relational Databases* [online]. 2018 [cit. 2020-05-05]. Available at: https://logz.io/blog/relational-database-comparison/.

[7] Linaro Ltd. et al. *Devicetree Specification* [online]. version 0.3. 2020 [cit. 2020-04-24]. Available at: https://github.com/devicetree-org/devicetree-specification/releases/tag/v0.3.

[8] Imem, A. A. Comparison and evaluation of digital signature schemes employed in NDN network. *International Journal of Embedded systems and Applications (IJESA)* [online]. 1st ed. june 2015, vol. 5, no. 2, [cit. 2020-04-25]. DOI: 10.5121/ijesa.2015.5202. ISSN 1839-5171. Available at: http://airccse.org/journal/ijesa/papers/5215ijesa02.pdf.

[9] Standards, N. I. of and Technology. *Digital Signature Standard (DSS)* [online]. 2013 [cit. 2020-05-12]. Available at: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[10] STMicroelectronics. *IPCC internal peripheral* [online]. 2019. revised 22. 1. 2020 [cit. 2020-04-23]. Available at: https://wiki.st.com/stm32mpu/wiki/IPCC_internal_peripheral.

[11] STMicroelectronics. *Linux Mailbox framework overview* [online]. 2019. revised 30. 1. 2020 [cit. 2020-04-23]. Available at: https://wiki.st.com/stm32mpu/wiki/Linux_Mailbox_framework_overview.

[12] Dougherty, C. R. *MD5 vulnerable to collision attacks* [online]. 2008. revised 21. 1. 2009 [cit. 2020-04-25]. Available at: https://www.kb.cert.org/vuls/id/836068/.

[13] STMicroelectronics. *OpenEmbedded* [online]. 2019. revised 29. 1. 2020 [cit. 2020-04-24]. Available at: https://wiki.st.com/stm32mpu/wiki/OpenEmbedded.

[14] STMicroelectronics. *OpenSTLinux distribution* [online]. 2019. revised 10. 10. 2019 [cit. 2020-04-24]. Available at: https://wiki.st.com/stm32mpu/wiki/OpenSTLinux_distribution.

[15] BlueKrypt. *Cryptographic Key Length Recommendation* [online]. 2020. revised 8. 4. 2020 [cit. 2020-04-25]. Available at: https://www.keylength.com/.

[16] STMicroelectronics. *STM32MP157 advanced Arm®-based 32-bit MPUs* [online]. Reference manual, 4th ed. February 2019, revised 18. 2. 2020 [cit. 2020-04-23]. Available at: https://www.st.com/content/ccc/resource/technical/document/reference_manual/group0/51/ba/9e/5e/78/5b/4b/dd/DM00327659/files/DM00327659.pdf/jcr:content/translations/en.DM00327659.pdf.

[17] STMicroelectronics. *Linux remoteproc framework overview* [online]. 2019. revised 28. 11. 2019 [cit. 2020-04-23]. Available at: https://wiki.st.com/stm32mpu/wiki/Linux_remoteproc_framework_overview.

[18] STMicroelectronics. *STM32MP15 ROM code overview* [online]. 2019. revised 19. 2. 2019 [cit. 2020-04-23]. Available at: https://wiki.st.com/stm32mpu/wiki/STM32MP15_ROM_code_overview.

[19] STMicroelectronics. *Linux RPMsg framework overview* [online]. 2019. revised 31. 1. 2020 [cit. 2020-04-23]. Available at: https://wiki.st.com/stm32mpu/wiki/Linux_RPMsg_framework_overview.

[20] STMicroelectronics. *STM32MP15 secure boot* [online]. 2019. revised 20. 06. 2019 [cit. 2020-04-24]. Available at: https://wiki.st.com/stm32mpu/wiki/STM32MP15_secure_boot.

[21] Stevens, M. et al. *The first collision for full SHA-1* [online]. N.d. [cit. 2020-04-25]. Available at: https://shattered.io/static/shattered.pdf.

[22] Dierks, T. and Rescorla, E. *The Transport Layer Security (TLS) Protocol Version 1.2* [online]. 2008 [cit. 2020-04-25]. Available at: https://tools.ietf.org/html/rfc5246.

[23] STMicroelectronics. *Discovery kits with STM32MP157 MPUs* [online]. User manual, 1st ed. March 2019 [cit. 2020-04-23]. Available at: https://www.st.com/content/ccc/resource/technical/document/user_manual/group1/d6/59/df/e0/8f/e7/45/8f/DM00591354/files/DM00591354.pdf/jcr:content/translations/en.DM00591354.pdf.