UMinho | 2021

Marcelo Borges

Robust Software Services for IoT Embedded Systems

Universidade do Minho
Escola de Engenharia

João Marcelo Mendes Borges

Robust Software Services for IoT Embedded Systems

dezembro de 2021

Universidade do Minho
Escola de Engenharia

João Marcelo Mendes Borges

Robust Software Services for IoT Embedded Systems

Dissertação de Mestrado
Mestrado em Engenharia Eletrónica Industrial e Computadores
Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Jorge Cabral

dezembro de 2021

Agradecimentos

Em primeiro lugar, agradeço ao meu orientador, Professor Doutor Jorge Cabral, por me ter dado a oportunidade de realizar este projeto e todo o conhecimento transmitido durante estes cinco anos.

A todos os meus professores do curso de Engenharia Eletrónica da Universidade do Minho pela excelência da qualidade técnica de cada um.

Aos meus pais, por todo o apoio, atenção, carinho ao longo desta caminhada e por todo o investimento em mim. Aos meus irmãos, Célia e Paulo, pela amizade e atenção dedicadas sempre que precisei. Um especial obrigado à minha namorada por toda a ajuda, compreensão e apoio em todos os momentos, e sobretudo por não se cansar de me ouvir falar de eletrónica todo o dia. Ao Carlos, Pedro e Ingrês, obrigado pela vossa amizade e apoio ao longo destes anos.

Aos meus colegas de laboratório *Embedded Systems Research Group* que sempre me acompanharam no desenvolvimento deste projeto. Especialmente ao Rui Almeida, obrigado por todo o incentivo e aconselhamento ao longo desta etapa e pela dedicação do seu escasso tempo a discutir e rever o documento.

Por fim, a todos que me ajudaram neste percurso, o meu maior obrigado!

Project "(Link4S)ustainability - A new generation connectivity system for creation and integration of networks of objects for new sustainability paradigms [POCI-01-0247-FEDER-046122 | LISBOA-01-0247-FEDER-046122]" is financed by the Operational Competitiveness and Internationalization Programmes COMPETE 2020 and LISBOA 2020, under the PORTUGAL 2020 Partnership Agreement, and through the European Structural and Investment Funds in the FEDER component.

STATEMENT OF INTEGRITY

Resumo

Serviços de Software Robusto para Sistemas Embebidos IoT

O aumento do número de dispositivos conetados e o aparecimento de novas tecnologias *LPWAN*, como o *NB-IoT*, permite que a nova geração de dispositivos *IoT* apresente tempo de vida superior a 10 anos, com recurso a uma bateria de dimensões AA convencionais. Contudo, durante este tempo, os dispositivos *IoT* designados de *end-devices*, devem conseguir manter o devido funcionamento do sistema, economizando o máximo de energia possível. Neste tipo de dispositivos o uso de reprogramação *Over-The-Air (OTA)* também é necessário, pois, este permite mudar o comportamento do dispositivo caso o paradigrama da aplicação mude ou *software bugs* sejam encontrados. Os *end-device* são normalmente controlados por uma "entidade" superior chamada de *cloud* que com eles interage.

O objetivo desta dissertação é dar continuidade ao trabalho realizado no laboratório do ESRG, nomeadamente na dissertação de mestrado [1] que se focou no desenho de um dispositivo com um tempo de vida superior a 10 anos. A solução final foi um dispositivo com um *software bare-metal* e com uma estimativa de 17 anos de tempo operacional. Esta foi analisada e foram encontradas possíveis melhorias na pilha de *software*. Assim sendo, este foi redesenhado para o uso de um sistema operativo *Azure RTOS ThreadX*. Neste redesenho vários módulos foram otimizados e adicionados como encriptação, *OTA*, e comandos. Estes novos módulos permitem a alteração de parámetros do dispositivo, tais como, o tempo de amostragem dos sensores, o tempo de envio e atualizações incrementais do *software*. Para realizar as atualizações (OTA) foi desenvolvido o algoritmo diferencial DeltaGen. Em todo o desenvolvimento foram feitos testes unitários e usados analisadores estáticos para prevenir erros antes da colocação dos dispositivos no terreno.

Foi desenvolvida a *cloud* que recebe a informação dos dispositivos e os controla utilizando uma arquitetura baseada em microserviços que aumenta a flexibilidade e agilidade do sistema.

Em condições normais o dispostivo tem um tempo de operação estimado de 23 anos, sendo que em condições ideais esta aumentaria para 30 anos. A aplicação possui comunicações encriptadas utilizando os algoritmos de encriptação RSA e AES, atualizações *OTA* e reconfiguração a partir de comandos enviados da *cloud*.

Palavras-chave: baixo-consumo, OTA, Azure RTOS ThreadX, design, NB-IoT, IoT

Abstract

Robust Software Services for IoT Embedded Systems

Following the rising of connected IoT devices, the usage of LPWAN technologies, such as NB-IoT, allow end-devices to achieve ultra-low-power consumption, reaching increasingly higher lifetimes powered by a battery cell of standard dimensions AA. Their long lifetime requires these devices to operate properly while saving energy. Since the application paradigm can change in the extended device lifespan or software bugs can be found, there is a need to make these devices autonomous and connected to the cloud, allowing for reconfiguration without physical maintenance. The cloud controls the end-devices and receives the transmitted end-devices data.

This dissertation aims to continue the work accomplished by the ESRG laboratory, in the master's thesis [1], which developed and further analysed both power consumption and performance of NB-IoT monitoring end-device, targeting its optimisations through a software/hardware co-design to achieve ten years of operation with a single battery cell. It resulted in a bare-metal software device with 17 years of estimated operational time.

This work aimed to improve further the work done at ESRG by implementing new features to make this device more autonomous. The application was re-designed with the operating system Azure RTOS ThreadX. In the re-design, several modules architecture were optimised, encrypted communications were added, and the ability to change the run-time device settings as the sensors sampling time, transmission interval, and update to new firmware through incremental OTA updates. To perform these updates was developed the differing algorithm DeltaGen. In the development phase, unit tests and static analysers were completed. The cloud using a microservice architecture was implemented, being responsible for controlling the devices and receiving the collected data.

In normal conditions, the final solution estimated lifetime is 23 years, and it can reach 30 years without battery leakage. It contains end-to-end secure communications using the RSA and AES encryption algorithms, OTA updates, and can be reconfigure through the cloud's commands in run-time.

Keywords: ultra-low-power, OTA, Azure RTOS ThreadX, design, NB-IoT, IoT

Contents

List of Figures

List of Tables

Code Snippets

Acronyms

3GPP	3rd Generation Partnership Project.
ADC	Analogue to Digital Conversion.
AES	Advanced Encryption Standard.
API	Application Programming Interface.
ASCII	American Standard Code for Information Interchange.
AWS	Amazon Web Services.
BOM	Bill of materials.
CB	Control Block.
CI	Continuous Integration.
CoAP	Constrained Application Protocol.
COTS	Commercial Off-The-Shelf.
DASA	Differencing Algorithm based on Suffix Array.
DES	Data Encryption Standard.
DRX	Discontinuous Reception.
ECC	Elliptic-curve Cryptography.
ECDH	Elliptic-curve Diffie-Hellman.
ECDSA	Elliptic Curve Digital Signature Algorithm.
EEPROM	Electrically-Erasable Programmable Read-Only Memory.
eSIM	embedded-SIM.
ESRG	Embedded Systems Research Group.
EXTI	External Interrupt.
FBC	Fixed Block Comparison.
FOTA	Firmware Over-The-Air.

Gbps	Gigabits-per-second.
GPIO	General Purpose Input/Output.
HA	Height Array.
HAL	Hardware Abstraction Layer.
I2C	Inter-Integrated Circuit.
IC	Integrated Circuit.
IoE	Internet of Everything.
IoT	Internet of Things.
IP	Intellectual Property.
ISR	Interrupt Service Routine.
IV	Initial Vector.
JSON	JavaScript Object Notation.
LCP	Longest Common Prefix.
LDO	Low-Dropout Regulator.
Link4S	(Link4S)ustainability.
LL	Low Level.
LoRa	Long Range.
LPMG	Low Power Mode Governor.
LPTIM	Low-power Timer.
LPUART	Low-power UART.
LPWAN	Low-Power Wide Area Network.
LTE	Long Term Evolution.
M2M	Machine to Machine.
mAh	Milliamps Hours.
Mbps	Megabits-per-second.
MCU	Microcontroller Unit.
MD5	Message-Digest algorithm 5.
MISRA	Motor Industry Software Reliability Association.
MPU	Memory Protection Unit.
MQTT	Message Queuing Telemetry Transport.

MSB	Most Significant Bit.
MSI	Multispeed Internal.
NAS	Non-Acess Stratum.
NB-IoT	Narrowband IoT.
OFDMA	Orthogonal Frequency-Division Multiple Access.
OS	Operating System.
OTA	Over-The-Air.
P2M	Person/People to Machine.
P2P	Person/People to Person/People.
PCB	Printed Circuit Board.
PSM	Power Saving Mode.
RA	Rank Array.
RAM	Random Access Memory.
RFID	Radio Frequency Identification.
RM	Rate Monotonic.
RO	Read-only.
ROM	Read-Only Memory.
RSA	Rivest-Shamir-Adleman.
RTC	Real Time Clock.
RTOS	Real-Time Operating System.
SA	Suffix Array.
SC-FDMA	Single-Carrier Frequency-Division Multiple Access.
SIM	Subscriber Identification Module.
SoC	System-On-a-Chip.
SPI	Serial Peripheral Interface.
TCP	Transmission Control Protocol.
UART	Universal Asynchronous Receiver-Transmitter.
UDP	User Datagram Protocol.
URC	Unsolicited Result Code.

USART	Universal Synchronous and Asynchronous Receiver-Transmitter.
USB	Universal Serial Bus.
UUID	Universally Unique Identifier.
WFI	Wait for Interrupt.

Chapter 1: Introduction

This chapter starts by presenting this dissertation's contextualization, motivation, and goals. First, the contextualization gives an essential perspective of the frame of the dissertation. Then the motivation for conducting a study on this topic following the dissertation objectives. Finally, this chapter concludes by guiding the reader through the work done, presenting the document structure.

1.1 Contextualization and Motivation

This dissertation continues the work done in ESRG laboratory [1]. It is inserted in the (LINK4S)ustainability project from the Link4S consortium, which is committed to working together in the generation of new scientific knowledge to design, develop, construct and test novel Smart Embedded Connected Devices and associated software platforms, aiming at the integration of networks of objects and social networks in the context of mobility and energy ecosystems.

In the Masters's thesis titled "Software/Hardware Co-Design for NB-IoT Low-Power Applications: Consumption and Performance Analysis" [1], the purpose was to analyze both power consumption and performance of NB-IoT monitoring end-device, targeting its optimizations through a software/hardware co-design, in order to achieve ten years of operation with a single battery.

The end-device is a monitoring system with multiple sensors: methane, acceleration, luminosity, temperature and humidity, which can continuously sample in low-power mode, enabling an internal interrupt mechanism to detect if the sampled value is within a defined threshold. Therefore, these sensors' interrupt signals can wake up the end-device microcontroller, thens signalling an alert. Furthermore, the microcontroller can enter low-power sleep and react to asynchronous events.

In terms of hardware, a previous benchmark of the development board provided by the partner NOS, which helped explore what was required to implement an NB-IoT end-device, later allowing the design of a custom solution. The resulting custom board developed reached an 11-year battery lifetime under ideal operating conditions. However, the Low-Dropout Regulator chosen had a power quiescent current prejudicial during the device's sleep period. From the knowledge acquired with the measurements a new board was developed, and the software was also improved, allowing to achieve a 17-year battery lifetime under ideal conditions.

In terms of software, a comparison between a bare-metal and a freeRTOS approach, where the difference measured was not relevant for the device's overall operation time, concluding that both approaches were viable. However, it was unfeasible to improve the freeRTOS version because of its memory footprint. Moreover, were not implemented security mechanisms in the communications, such as encryption, because of unavailability from the cloud side.

This dissertation aims to guarantee software-wise that the developed device can survive for all its hardware lifetime, improving the application software without degradation the end-device operation lifetime.

1.2 Objectives

The main goal is to improve the software of a monitoring NB-IoT end-device, making it robust and reliable enough to maintain the 10-years battery autonomy. This goal can be achieved with the following objectives:

- **Re-Design the bare-metal application to an RTOS based one**

  The re-design of the application to use an RTOS adds flexibility to the design and reliable operation, and as the system becomes more complex, it simplifies software integration. In the application re-design, the ability of the cloud to change settings of the end-device, a new sensors' architecture to optimize space, and auto-recovery should be added. The communication module can be generalized to better modular software.

- **Test and validation the application**

  Test and validation of the application are required to prevent unexpected code behaviour in run-time.

- **Implementation of encryption algorithms to the communications**

  The LTE protocols as the NB-IoT have the message's payload already encrypted in the control plane, being natively encrypted in the radio interface by Non-Acess Stratum (NAS) messages [2]. However, to achieve end-to-end data privacy it is required to use appropriate cryptographic solutions in data exchange.

- **Implementation of the cloud server for the end-devices**

  The reception of data from the end-devices, its storage, and run-time device control should be done from the cloud server. The data should be saved in a database allowing further post-process by other applications.

- **Enable Firmware Over-The-Air (FOTA)**

  The end-devices can be located in remote and even inaccessible places, so adding new features and resolving bugs or security vulnerabilities is impossible without recovering the devices or going into their location and downloading new firmware. As a solution, Firmware Over-The-Air (FOTA) is required because it allows upgrading the firmware wirelessly without physical interaction, preventing long maintenance downtimes.

- **Deployment of the end-devices**

  The end-devices and cloud software need to be ready for the LINK4S project use cases deployment.

1.3 Document Structure

The document contains six chapters that will be briefly described next. After this introductory chapter is the state-of-the-art, it introduces concepts used during the dissertation. It starts with the traditional Internet of Things (IoT) device's architecture and covers wireless technologies. Then, an overview of the Firmware Over-The-Air (FOTA), from the new firmware to the process of creating the Delta and sending it to the end-device. Afterwards, some Real-Time Operating System (RTOS) concepts are introduced, mainly focusing on solutions such as FreeRTOS, Zephyr, and Azure RTOS ThreadX. The Azure RTOS ThreadX time and thread management is explained in detail. Then, the microservices cloud architecture is introduced, and an overview of the cryptography.

Chapter three presents the system specification, which explains the dissertation start software and hardware in the end-device, its functional and non-functional requirements, and use cases.

Chapter four contains the Improvement Opportunities that focus on finding the end-device software and cloud solution improvements.

Chapter five contains the design, which is divided into three main topics: (i) End-device presents the new software re-design adding new architectures to optimize or improve the current software taking advantage of the improvements opportunities found in the previous chapter; (i) Cloud shows the microservices architecture used in the cloud to handle the end-devices; (i) Delta presents the DeltaGen algorithm, which is a differencing algorithm used to generate the delta to perform incremental updates.

Chapter six shows a comprehensive description of how the delta, end-device and cloud were implemented and presents some development phase helper scripts.

Chapter seven contains the results of the end-device power consumption in the several implementation stages. In the end, it is estimated the power consumption in a real application scenario.

The last chapter concludes with the work developed during the thesis and proposes future work to improve the final solution.

Chapter 2: State-of-the-Art

In this chapter, several important concepts to the development will be discussed, starting with an overview of the Internet of Things, which is a concept that enables the connecting of all devices that can be located in remote places. Consequently, Firmware Over-The-Air is used to fix bugs of the development phase and add more functionalities. Next, an overview of the RTOS advantages, disadvantages, utility, and an overview of available RTOSes as FreeRTOS, Zephyr and the Azure RTOS ThreadX. Afterwards, the microservices architecture used in the cloud is illustrated, and, finally, cryptography to secure communications between the device and the cloud is addressed.

2.1 Internet of Things (IoT)

The term Internet of Things (IoT) was invented in 1999, initially to promote Radio Frequency Identification (RFID) technology. It describes the network of physical objects (things), which are embedded with sensors, software, and other technologies with the purpose of connecting and exchanging data with other devices [3, 4, 5, 6]. This term was to refer Machine to Machine (M2M) communications [7]. A few years later, the Internet of Everything (IoE) [8] appeared to describe interrelated elements of a whole system, including people. IoE entails not only in M2M but also Person/People to Person/People (P2P) and Person/People to Machine (P2M) communications.

The Internet of Things is already part of our lives. According to Gartner, there will be online by 2020, 20.4 thousand millions of IoT devices and by 2025, and the number is expected to rise to 75 thousand millions devices [9]. Where a device's purpose could be monitoring environmental variables for several years in a remote place [10], supporting IoT devices (e.g. wearable medical device) [11], actuating in a safety-critical or mission-critical situation. There are distinct applications domains, such as home automation [12], environmental monitoring [10, 13], healthcare [11, 14, 15], smart cities [16], smart energy [17], agriculture [18], smart oceans [19, 20], among others. As a result, IoT devices' characteristics vary from large scale to low-cost design, resource constraints to device heterogeneity, opt for functions instead of security. Consequently, security, privacy [21], and reliability [22] issues are major concerns when designing an IoT system.

2.1.1 Traditional Architecture

A traditional IoT device architecture is depicted in Figure 2.1. It comprises Sensors, Power Management, Microcontroller Unit (MCU), Actuators and Wireless Communication Technologies. The sensors and actuators depend on the device application. Some devices only have sensors, or actuators, or both [23].

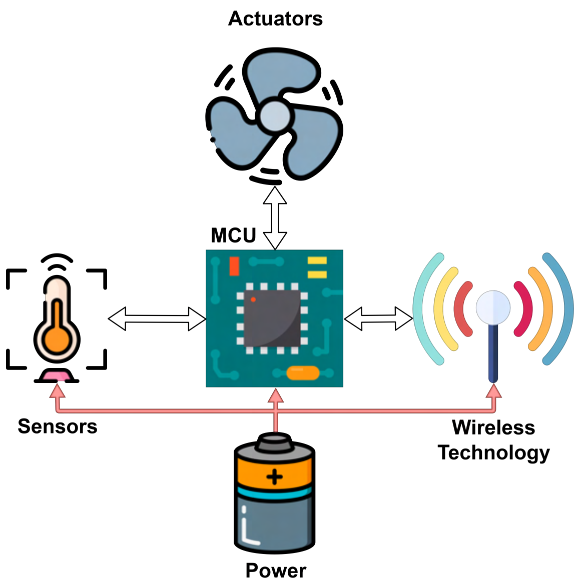


Figure 2.1: Traditional IoT device architecture.

The Power Management is responsible for supplying the device's components and can have multiple forms. Batteries can be used as as power supply in [24, 10], or by energy harvesting from the environment, or a hybrid solution that can have both. Also, the power does not require a battery or harvesting as described previously if a power source is available. A device is power constrained when it is limited to use a limited amount of energy.

The Sensors and Actuators enable the device to interact with the real world. Through the sensors, the device can monitor, sense, and listen to the environment using variables such as temperature, humidity, pressure, acceleration. The actuators allow the device to take action, for example, a General Purpose Input/Output (GPIO) pin output set to high. The actuation can be done by driven MOSFETs, relays, motors, among others.

The Wireless Communication Technology is responsible for connecting the device with its controller. Its purpose is to allow M2M or P2M communications. It can take many forms and some of the techonologies are described in detail in Section 2.1.2.

The Microcontroller Unit (MCU) is the "brain" of the end-device. It is responsible for executing the IoT device's purpose and managing all the devices's components according to the application.

2.1.2 Wireless Technologies

Different wireless techonologies are used depending on the IoT application scenario [25]. Figure 2.2 illustrates the data rate and power consumption per range of different technologies. It can be seen that increasing the data rate also increases the power consumption but does not influence the range. These various technologies such as RFID, Wi-Fi, Bluetooth, Zigbee, Z-Wave, EnOcean, NB-IoT, LoRa, among others, are chosen according to the application requirements.

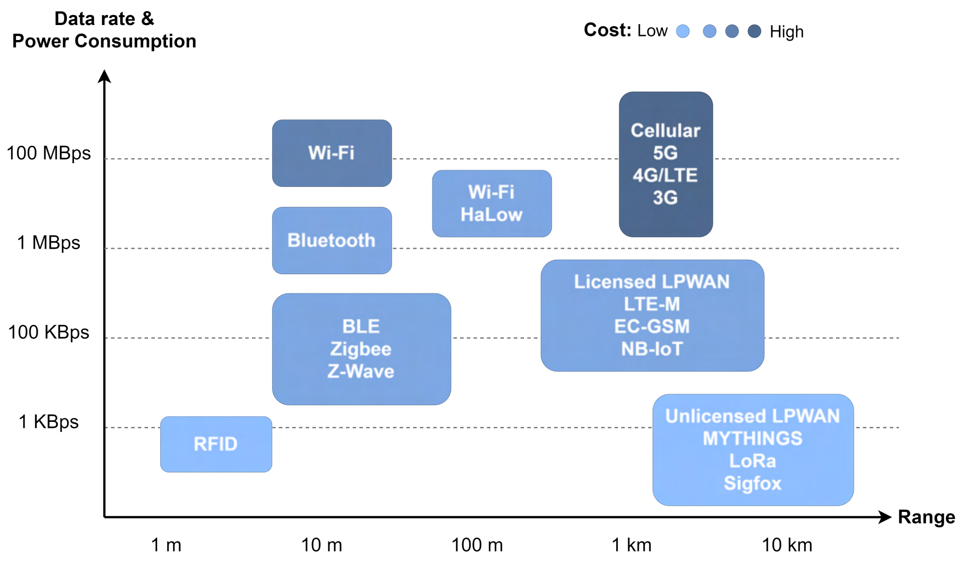


Figure 2.2: Data rate and Power Consumption per range of multiple technologies (adapted from [26]).

From the low range technologies as RFID, which does not require a battery in the tags [27] into cellular 5G, which has a long-range and data rates greater than 100 Megabits-per-second up to a peak of 20 Gigabits-per-second. Focusing on the low-end constrained devices, which generally are power constrained, Low-Power Wide Area Network (LPWAN) techonologies are a perfect fit for it due to its long-range and power consumption.

Low-Power Wide Area Network (LPWAN)

LPWAN represents a new trend in the evolution of IoT technologies. Unlike 2G/3G/4G or Wi-Fi, these techonolgies do not focus on minimizing latency or enabling high data rates per device. Instead, the key performance metrics defined for LPWAN are coverage [28], scalability, and energy efficiency. LPWAN technologies ensure a long transmission range, low energy consumption, and low-cost deployment solution. It allows up to 40 km as communication range in rural zones and 10 km in urban zones [29], up to 10 years of battery lifetime [30, 10], less than $\$ 5$ of device cost and less than $\$ 1$ per device per year of operator subscription cost [31]. It was mainly designed for IoT applications that require transmitting few tiny messages per day in long radio range [32] representing a market opportunity for communication service providers [25].

LPWAN has multiple technologies, focusing on NB-IoT and Long Range (LoRa). The latter technology was developed by Semtech and is the most widely used technology for LPWAN in the sub-GHz unlicensed band [33]. Due to the utilization of unlicensed bands, the LoRa network is open to customers who lack authorization from radio frequency regulators. As a result, the LoRa network is easy to deploy over a range of more than several kilometres and serves customers with minimum investment and maintenance costs.

On the other hand, NB-IoT, which is a new narrow-band IoT system built from existing Long Term Evolution (LTE) functionalities announced by the 3rd Generation Partnership Project (3GPP) in 2016, promises to provide improved coverage for a massive number of low-throughput, low-cost devices with low device power consumption in delay-tolerant applications.

Compared to the LoRa technology, NB-IoT reuses the LTE design extensively, including the numerologies, downlink Orthogonal Frequency-Division Multiple Access (OFDMA), uplink Single-Carrier Frequency-Division Multiple Access (SC-FDMA), channel coding, rate matching, interleaving, and infrastructures, for example, in Portugal, the telecommunication company NOS infrastructures. This significantly reduces the time required to develop full specifications [34] and increase the technology coverage.

2.2 Firmware Over-The-Air (FOTA)

The raising of IoT devices and the various applications made the access and maintenance of them to be a concern. The development can follow code guidelines and reviews, but event so bugs still exist in the design or implementation. Therefore, the firmware must be kept in constant development to remove bugs and improve functionality. An over-the-air firmware update system is preferable as it allows for faster updates and encourages accessibility for the development.

FOTA raises other problems, such as the integrity of the firmware received can be compromised and lack of security. In order to achieve integrity, there are several algorithms as checksums, MD5 and SHA. Also, to a secure system, as [35] proposed an Over-The-Air firmware update to many nodes simultaneously, creating a bootloader that allowed a sensor node to boot to the updated firmware or the existing one assuring the security using Advanced Encryption Standard (AES) encryption.

Moreover, there are problems of update time and battery drain, for example, in a Low-Power Wide Area Network (LPWAN) (with standards such as LTE-M and NB-IoT) a costumer can have more than 20000 IoT modules [36], with the LTE-M connectivity, it can take until an hour to update just ten devices, taking this as the average speed, it would take more than 83 days to update the entire fleet while burning battery and data plans [36]. As a result, incremental firmware update solutions are developed to reduce the data size transferred and compressions methods, e.g. [37].

The incremental update is done by sending a Delta to the board. The Delta is a sequence of commands that from the old firmware, the new firmware can be reconstructed following the commands. These commands are normally:

- **ADD** ([38]) – Adds to the current new firmware the received bytes in the add command;
- **COPY** ([38]) – Receives an old firmware address and copy the respectively received length from the old firmware to the new;
- **REPEAT** ([38]) – Used when the new data have specific patterns with small differences;
- **PAD** ([39]) – It pads the new firmware memory with a specific data received;
- **RUN** ([40]) – The update module copies the data that is associated with the instruction a finite number of times resulting in the more efficient transmission of repeated data [41];

The incremental firmware update process of the new firmware source to the device running has multiple stages, as illustrated in Figure 2.3.

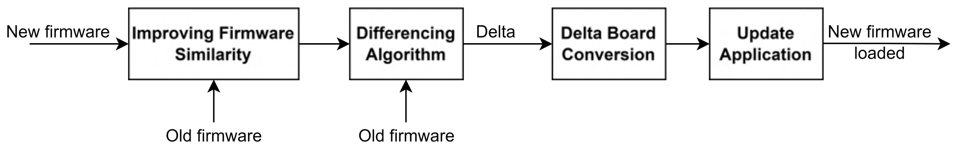


Figure 2.3: Essential stages of the firmware update process. (Adapted from [39]).

The first stage receives the new and old firmware and improves the firmware similarity. This is used because firmware can introduce slight modifications that can result in a disproportional increase in the size of the delta script produced. For example, if a new function is added, the delta should be a COPY command from address 0x0000 to the function address, ADD the new function code, and COPY from the function end to the end of the firmware. The problem is that the new code is not just reallocated in the memory (shifted). The instructions that call functions will use different target addresses (the address of the called function in the flash memory) in the two versions. Therefore, since these modified target addresses will be added using the ADD instructions, the size of the generated delta will be significantly large. This is also true when a new global variable is defined or a previously defined variable is removed, as other global variables may need to relocate. Hence, the instructions that reference them will have different target addresses. The authors in [41] present some of the following techniques used to improve the similarity of firmware images:

- **Slop regions** – It was first introduced in [42], and it consists of leaving a slop region defined as free space located immediately after the function's code. It enables the function to grow or shrink in the pre-allocated slop free area, not requiring to shift other functions. A disadvantage of the slop regions is that an excessive fragmentation of the memory space can occur, as some regions may contain code while others remain clear. Also, it uses flash memory inefficient, but fragmentation can increase energy consumption because the control circuitry needs to activate many memory regions [41];

- **Position independent code** – Position independent code is a code that, being placed somewhere in the primary memory, executes properly regardless of its absolute address. The function address needs to be loaded and then jumped to it, adding indirection. Normally, the address of the function is already present in the instruction stream;

- **Interrupt Service Routine pinning** – Interrupt Service Routine (ISR) are functions that are invoked by software or hardware to answer to signals that need attention—for example, a timer overflow. The interrupts vector table is normally placed at the beginning of the program memory. When an interrupt occurs, it goes to a predefined vector position to handle the interrupt. Changing the firmware can relocate these ISRs callbacks, affecting the memory addresses contained in the interrupt vector table. In [38], it is solved by pinning the routines in fixed locations at the linking stage;

- **Relocatable code** – When building a runnable program, it is compiled and linked together to construct the final executable. The modules are referred to as relocatable are firstly made pseudo-addresses, and when linked together, the linker assigns its values. Nevertheless, after the final program is created, it should also execute from different memory addresses. Therefore, a relocatable code is software whose execution address can be dynamically moved around the available address space and loaded in multiple addresses [41]. The authors in [43] use this technique to mitigate the effect of function and variable shifts;

- **Global variables' address pinning** – It was firstly introduced in [44], and it intends to ensure that the addresses of the global variables are the same in both software versions. Hermes uses a slop region between the *.data* and *.bss* to avoid address shifting of the undefined variables when *.data* expands or shrinks;

There are several techniques to preserve the similarity between firmware. After having the new firmware as similar as possible, the old and new firmwares are fed to perform the differing algorithm that generates the Delta. The differencing algorithms can be of two types: block-level or byte-level according to the granularity level that the new firmware is compared with the old one.

The block-level algorithms, e.g. [45, 46], split the firmware images into fixed block sizes and detect the segments that are not common. Therefore, its accuracy depends on the block size. On the other hand, the byte-level algorithms, e.g. [47, 48], can find common segments inside the block size using blocks of variable lengths and utilising more fine-grained approaches in order to achieve better accuracy. Nextly, will be explain the FBC ([45]), Rsync ([46]), R3diff ([48]), and DASA ([47]) differencing algorithms.

The Fixed Block Comparison (FBC) [45] is the simplest method of comparison between two firmware. This algorithm splits both firmware into fixed block sizes and then compares each block. If the block matches, it adds a COPY command. If not, it adds an ADD command. The main benefit of this algorithm is low time, and space overhead and easy implementation [41].

The Rsync algorithm [46] is used by incremental programming as [38] to compute the common segments between the firmware. It is a block-level algorithm that detects matching segments between firmware by splitting the firmware images into fixed-size blocks and then uses a sliding window with a size equal to the block size. For each block size, is calculated a pair of checksum and MD4 algorithms. The unmatched blocks are accumulated in the delta. Although Rsync can find subsequences with higher accuracy than FBC, it still faces similar drawbacks since its granularity depends on the window size used; thus, being unable to detect common segments with a size smaller than that of the window used [41].

The R3 [48] is a byte-level comparison algorithm that computes the hash values of every three continuous bytes in the current image. It is chosen three bytes because copying smaller segments is not beneficial because of the COPY command size. This algorithm iterates through all the possible prefixes computing the optimal ADD command or CPY, adding to the delta script the smallest.

Differencing Algorithm based on Suffix Array (DASA) [47] is a byte-level comparison algorithm that focuses on minimizing space and time complexity for computing the optimal delta script. This is accomplish using an efficient data struct, the Suffix Array (SA) [49]. A Suffix Array is a sorted array of all suffixes of a string. It is a data structure used in, among others, full-text indices and data compression algorithms. DASA starts by joining both firmware, adding padding extension formats “$-#”, then calculates and sorts all SA. With the SA is calculated the Height Array (HA). The HA contains the length of the Longest Common

Prefix (LCP) of each suffix with the next one in sorted order. Using the HA is possible to get the LCP of any two suffixes in linear time. Afterwards, it computes the optimal delta script using the previous arrays as the R3 computes the optimal delta command with the smallest COPY or ADD command. As expected, DASA outperforms Rsync in terms of delta script size [41, 47] since the Rsync is a blocked-size algorithm.

The Delta can be generated using the different algorithms described previously. Afterwards, the delta is fed to the Board Compatibles Conversion stage. This stage is responsible for converting the delta script generated to a pre-defined standard update protocol, assigning values to the delta commands. With the compatible board packages, they need to be sent to the board. This transmission can have security problems that can be protected with encryption and hashing keys as done by the authors [50]. It was proposed that the server sign the firmware image and its metadata using Elliptic-curve Cryptography (ECC).

2.3 Real-Time Operating System (RTOS)

A Real-Time Operating System (RTOS) is a software, preemptive and deterministic [51], that manages the memory, I/O, data, and processors of a computer system, guaranteeing all timing constraints are satisfied [52]. It adds flexibility and reliability operation and, as the system becomes more complex, it facilitates the software integration. An operating system consists of two parts: kernel space and user space.

There are several types of kernels (monolithic, micro-kernel and exo-kernel) and it provides the lowest-level abstraction layer for the resources. The monolithic kernel was prominent in the early days, with millions of lines of code and is often known as the spaghetti code. Where no information is hidden, and the system is a collection of procedures. In the micro-kernel, the code is moved as much as possible to the user space, making it easier to extend and port to new architectures. Consequently, because less code runs in the kernel, it is more reliable and secure. Finally, the exo-kernel is usually used where strong hardware interaction is required because it separates hardware from the application using conceptual models such as file systems, virtual address spaces, and sockets. [51]

In a real-time system, operations performed must meet logical correctness and be completed on time. Non-real-time systems require logical correctness but have no timing requirements. Therefore, the tolerance of a real-time system towards failure to meet the timing requirements can be classified as hard real-time, firm real-time and soft real-time. When missing a deadline is unacceptable, the system is called hard real-time. In a firm real-time, the value of an operation completed past its timing constraint is considered zero but not harmful. Finally, in a soft real-time system, the value of an operation diminishes the further it completes after the timing constraint. [51]

Embedded devices have various applications and constraints, so an RTOS needs to be versatile. Therefore, specific characteristics of it are developed/studied to the extreme, concerning power consumption as [53] did in their proposed two mechanisms to achieve it: (1) Low Power Mode Governor (LPMG), which is used to select an optical low power mode after computation of retention time in idle mode and (2) Power-Off Mode, which disconnects power to the board when idle mode lasts for a long time. Similarly, [54] achieve better power consumption by developing an efficient low power scheme with the prediction of the expected execution time of real-time tasks and using the idle time of the system for scheduling these tasks in low power modes. In the process, there were trade-offs of missing deadlines. As a result, it is not a good operating system for hard real-time systems.

2.3.1 Rate Monotonic Algorithm

In 1973, Liu et al. [55] proposed the Rate Monotonic (RM) algorithm for preemptive scheduling tasks with fixed priorities in a single processor and hard-real-time environment. It consists in assigning the task priority according to their relative period. Therefore, the highest frequency task or the task with the shortest period will get the highest priority. Then after assigning its priorities, the next highest frequency task will get the next highest priority, and so on. In the end, the lowest frequency task or the task with the longest period is the one that is going to have the lowest priority. The rationale behind this priority assignment algorithm is that the least frequent task may span high-frequency deadlines.

2.3.2 RTOS Overview

With the increase in IoT devices, the vendors are pushing further down into small-footprint OSs and focusing on RTOS [56]. According to [57], the fourth RTOSes most used in constrained devices and edge nodes are Linux, FreeRTOS, Windows, and Zephyr with a percentage of 43, 35, 31, and 8, respectively. Also, as of 2017, according to marketing research firm VDC Research, the ThreadX RTOS has become one of the most popular RTOSes in the world, having been deployed in over 6.2 billion devices [58]. Since Linux and Windows do not fit in ultra-low-power constrained devices because of their power consumption and size, this section will present an overview comparing the FreeRTOS, Zephyr and Azure RTOS ThreadX.

FreeRTOS

FreeRTOS was initially developed by Richard Barry around 2003 from one of his consulting projects and has grown gradually over 18 years [59]. It is open-source and written in the C language. It has a microkernel architecture that provides robustness against bugs in the components [60]. In terms of memory size, it has a minimal ROM, RAM and processing overhead. Typically an RTOS kernel binary image will be in the 6K to 12K bytes [59]. It has plenty of services as tasks, mutexes, queues semaphores, task notification, stream and message buffers, software timers, event groups.

In 2017 FreeRTOS went under the aegis of Amazon Web Services (AWS), having its libraries implement clients for AWS IoT-specific value-added cloud services, including Over-The-Air (OTA). These libraries are suitable for building smart microcontroller-based devices that connect to the AWS IoT cloud.

If the power consumption is constrained, the FreeRTOS has tickless mode available. This mode takes advantage of the idle task hook to place the microcontroller into its low-power state, saving energy in the process. In [61], the LM3S3748 microcontroller sleep and deep sleep modes were used to use FreeRTOS low-power mode, concluding that the best solution is to have the microcontroller sleeping tickless. Moreover, FreeRTOS has a migration path to SafeRTOS, which includes certifications for the medical, automotive and industrial sectors [59].

Figure 2.4: FreeRTOS logo.

Zephyr

Zephyr [62] was first released in 2017, and it is a small RTOS for connected, resource-constrained and embedded devices supporting multiple architectures. It has a monolithic kernel with flexible configuration and builds a system for compile-time definition of required resources and modules. The Zephyr main advantage is a built-in set of protocol stacks as IPv4, IPv6, Constrained Application Protocol (CoAP), among others. Also, it supports a virtual file system interface with several flash file systems for non-volatile storage and management, and device firmware update mechanisms [62].

Figure 2.5: Zephyr RTOS logo.

Azure RTOS ThreadX

Azure RTOS ThreadX was initially named ThreadX and developed and marketed by Express Logic of San Diego, California, United States. Express Logic was purchased for an undisclosed sum by Microsoft on April 18, 2019, renaming it to Azure RTOS ThreadX and converting it to open-source. The name ThreadX is derived from the threads used as the executable elements, and the letter X represents context switching, i.e., it switches threads.

Azure RTOS ThreadX provides priority-based, fast interrupt response, preemptive scheduling, memory management, interthread communication, mutual exclusion, event notification, and thread synchronization features. The major distinguishing technology characteristics of ThreadX include preemption-threshold, priority inheritance, efficient timer management, fast software timers, picokernel design, event-chaining, and small size: minimal size on an ARM architecture processor is about 2 KB.

The preemption-threshold ease some of the inherent problems of preemption. It allows a thread to specify a priority ceiling for disabling preemption. Threads with higher priorities than the ceiling are still allowed to preempt, while those less than the ceiling are not allowed to preempt. For example, suppose a thread of priority 25 only interacts with a group of threads that have priorities between 20 and 25. During its critical sections, the thread of priority 25 can set its preemption-threshold to 20, preventing preemption from all of the threads that it interacts with. This mechanism still permits critical threads (priorities between 0 and 19) to preempt this thread during its critical section processing, resulting in much more responsive processing.

Event-chaining is typically useful when a single thread must process multiple synchronization events. For example, the application can register a notification routine for each object instead of having separate threads suspended for a queue message, event flags, and a semaphore. The application notification routine can resume a single thread, interrogating each object to find and process the new event when invoked. It generally results in fewer threads, less overhead, and smaller RAM requirements. It also provides a highly flexible mechanism to handle the synchronization requirements of more complex systems.

The priority inheritance within its mutex services allows a lower priority thread to temporarily assume the priority of a high-priority thread waiting for a mutex owned by the lower priority thread. This capability helps the application avoid nondeterministic priority inversion by eliminating preemption of intermediate thread priorities. Of course, preemption-threshold may be used to achieve a similar result.

The picokernel architecture is possible because instead of layering kernel functions on top of each other like traditional microkernel architectures, ThreadX services plug directly into its core. This results in the fastest possible context switching and service call performance.

ThreadX has extensive safety certifications from Technischer Überwachungsverein (TÜV, English: Technical Inspection Association) and UL (formerly Underwriters Laboratories) and is Motor Industry Software Reliability Association MISRA C compliant.

The ThreadX is the foundation of Express Logic's X-Ware Internet of things (IoT) platform, which also includes embedded file system support (FileX), embedded UI support (GUIX), embedded Internet protocol suite (TCP/IP) and cloud connectivity (NetX/NetX Duo), and Universal Serial Bus (USB) support (USBX). ThreadX has won high appraisals from developers and is a very popular RTOS. The ThreadX RTOS has become one of the most popular RTOSes globally, having been deployed in consumer electronics, medical devices, data networking applications, and SoCs.

Figure 2.6: ThradX logo.

2.3.3 Bare-metal to RTOS

The bare-metal conversion to use an RTOS is one of the more extensive application re-design. The author Paiva [1] used the FreeRTOS and resulted in the same lifetime of the bare-metal application but had only 1 KB of RAM from the 20 KB of the device. Therefore, the RTOS should have the smallest footprint possible and use little RAM, and in the application re-design, circular buffers and shared block of memories are used.

Since several RTOSes are available, it was searched for the smallest footprint, faster, safe and open-source. The small footprint is required due to the memory constraints and the fast to save battery energy. The end-device will operate for more than ten years without human contact, so it is advantageous to have safety certifications to maintain the RTOS proper operation, not compromising the application. It is good for the RTOS to be open-source, enabling the addition of low-power modes and learning with its kernel.

The Azure RTOS ThreadX [63] fulfils all these requirements for the application's re-design. In the re-design, binary semaphores will be used as mutexes because they are faster and smaller. Preemption-threshold will be used to avoid nondeterministic priority inversion. Since the application is not a complex system, it will be disabled the event-chaining, increasing the RTOS performance and reducing the memory footprint.

The Azure RTOS ThreadX version 6.1.3 does take advantage of the MCU low-power modes. Without it, the application can not run for more than ten years. Therefore, a low-power mode needs to be added to the kernel. This master's thesis will extend the ThreadX kernel to use it. In order to be able to accomplish that, it is required to understand the ThreadX thread states and how it implements time and application timers. Therefore, the timer in Azure RTOS ThreadX will be explained in the next section.

2.3.4 Azure RTOS ThreadX Thread States

The ThreadX have five distinct thread states: ready, suspended, executing, terminated. The Figure 2.7 represents the thread state transition diagram for ThreadX.

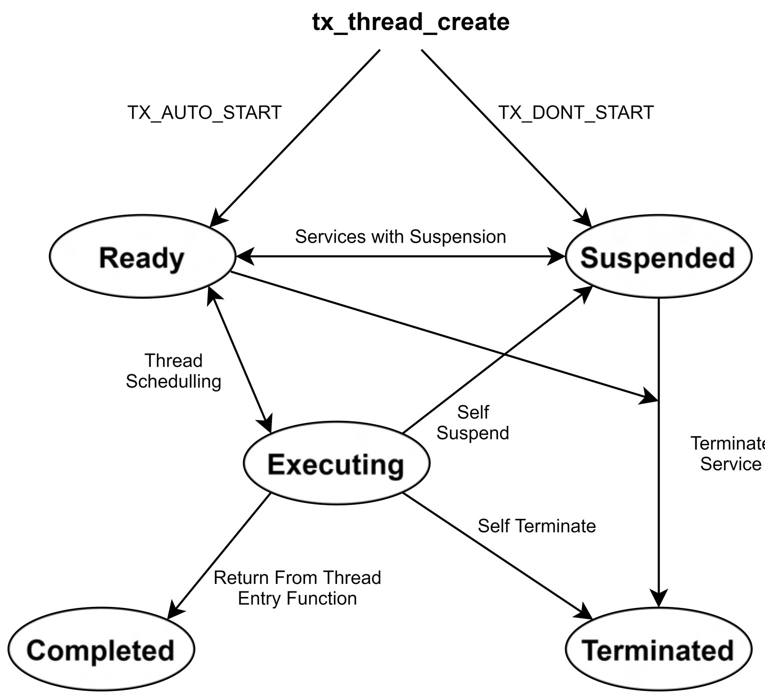


Figure 2.7: ThreadX thread state transition (adapted from [64]).

A thread is in a ready state when it is ready for execution, and it is not executed until it is the highest priority thread in the ready state. ThreadX executes the thread when this happens, changing its state to executing. If a higher-priority thread becomes ready, the executing thread reverts to a ready state, and the newly ready high-priority thread is then executed. It changes its logical state to executing. This transition between ready and executing states occurs every time thread preemption occurs.

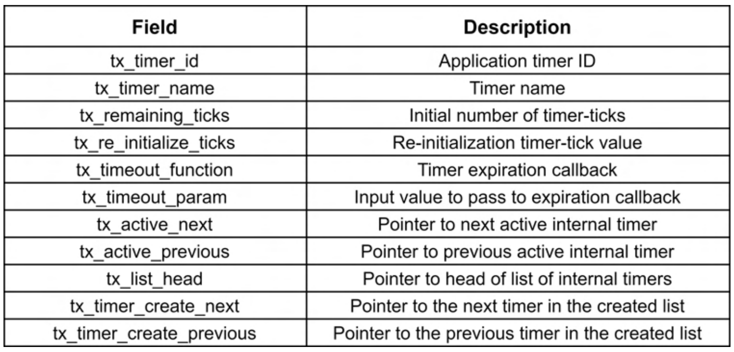
At any given moment, only one thread is executing because a thread in the executing state has control of the underlying processor. Threads in a suspended state are not eligible for execution. The reasons for being in a suspended state include suspension for time, queue messages, semaphores, mutexes, event flags, memory, and basic thread suspension. After the cause for suspension is removed, the thread is placed back in a ready state.

A thread in a completed state is a thread that has completed its processing and returned from its entry function. Finally, a thread is in a terminated state because another thread or the thread itself called the tx_thread_terminate service. A thread in a terminated or completed state cannot execute again.

The application should enter in low-power mode when all the threads are in a suspended state. For example, a thread is sleeping or suspending waiting for an event flag to be set. The way ThreadX manages time needs to be taken into account to design a low-power mode and will be explained in the next section.

2.3.5 Azure RTOS ThreadX Timers

The ThreadX has application timers (TX_TIMER) similar to hardware timers, except the hardware implementation (usually a single periodic hardware interrupt is used) is hidden from the application. Such timers are used by applications to perform time-outs, periodic, and or watchdog services. The TX_TIMER has a Control Block (CB), as attributes are illustrated in Figure 2.8. It has the timer ID, name, initial number of timer-ticks to count, number of timer-ticks for all timer expirations after the first (if one-shot timer, it is 0), the timer expiration callback function, the input parameter of the expiration callback, pointers of a double linked list to manage the timer activated, and two pointers of a linked list to manage the timers' creation.



Field	Description
tx_timer_id	Application timer ID
tx_timer_name	Timer name
tx_remaining_ticks	Initial number of timer-ticks
tx_re_initialize_ticks	Re-initialization timer-tick value
tx_timeout_function	Timer expiration callback
tx_timeout_param	Input value to pass to expiration callback
tx_active_next	Pointer to next active internal timer
tx_active_previous	Pointer to previous active internal timer
tx_list_head	Pointer to head of list of internal timers
tx_timer_create_next	Pointer to the next timer in the created list
tx_timer_create_previous	Pointer to the previous timer in the created list

Figure 2.8: Attributes of the control block of a ThreadX application timer.

The application timers are used in the ThreadX services. For example, a thread has in its control block an application timer. When it calls the thread_sleep, it activates the thread timer with the respective sleep time and when the timer timeouts, the expiration function resumes the thread suspended.

When a TX_TIMER data type is declared, a Timer CB is created and added to a doubly-linked circular list, as illustrates in Figure 2.9. The pointer named tx_timer_create_ptr points to the first control block in the list.

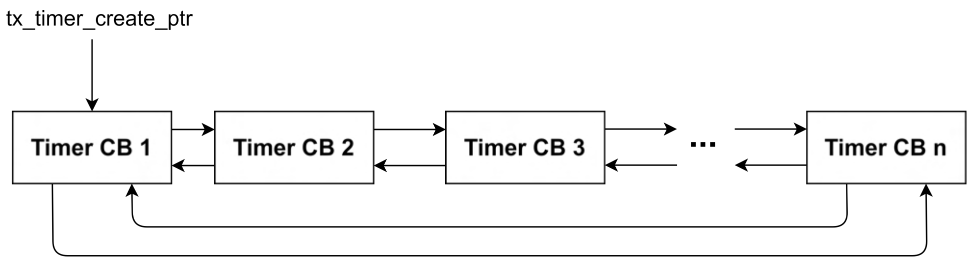


Figure 2.9: Created application timer list.

ThreadX can still function even without a periodic interrupt source. However, all-timer related processing is disabled, including time-slicing, suspension timeouts, and timer services. This periodic interrupt is used to call the ThreadX _tx_timer_interrupt.

2.4 Microservices Architecture

Microservices architecture is a cloud-native architecture that aims to realize software as packages services, as illustrated in Figure 2.10. Each service is independently deployable on a potentially different platform and technological stack [65]. It can run its own process communicating between then using mechanisms as TCP-IP or RESTful.

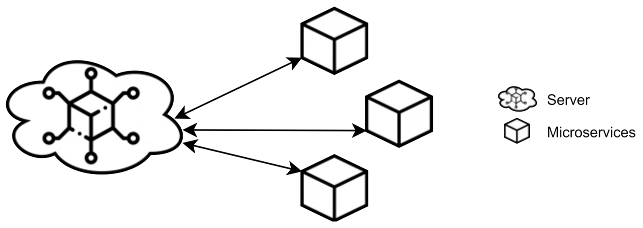


Figure 2.10: Cloud system overview.

Migrating monolithic architectures to microservices-based ones carries many advantages: adaptability to technological changes, reduced time-to-market, and better service structure [65, 66].

2.4.1 Cryptography

The fundamental objective of cryptography is to enable two devices, A and B, to communicate over an insecure channel in a way that a stranger C that is listening to the channel can not understand [67].

This channel can be a network or phone line. The message that A wants to send to B is called plaintext. After A encrypts the plaintext, using a predefined key, it sends the resulting ciphertext to B. B receives the ciphertext and decrypts resulting in the plaintext of A. C that does not have the key is not able to decrypt the ciphertext. The presented scheme is depicted in Figure 2.11.

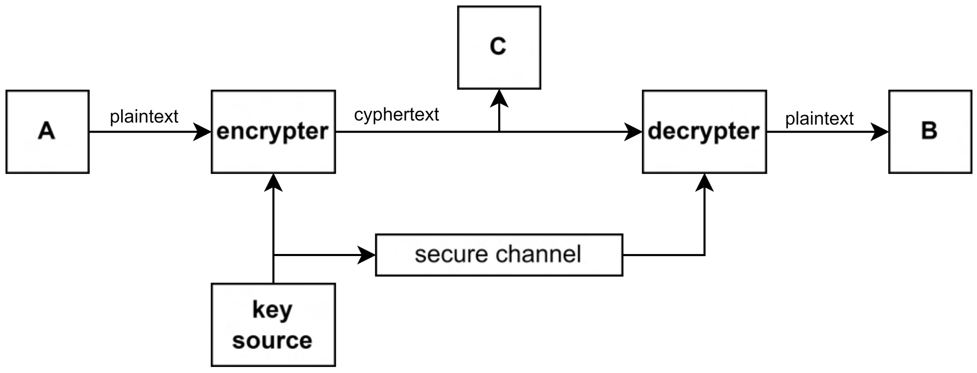


Figure 2.11: Classical cryptography scheme. (Adapted from [67])

The cryptography encryption algorithms overview is illustrated in Figure 2.12. These algorithms can be divided into two categories: key-based and keyless. In the latter, the relation between the plaintext and the ciphertext is exclusively dependable on the encryption algorithm. As a result, keyless encryption is generally less secure than key-based [68] because anyone who has access to the algorithm will be able to decrypt every message.

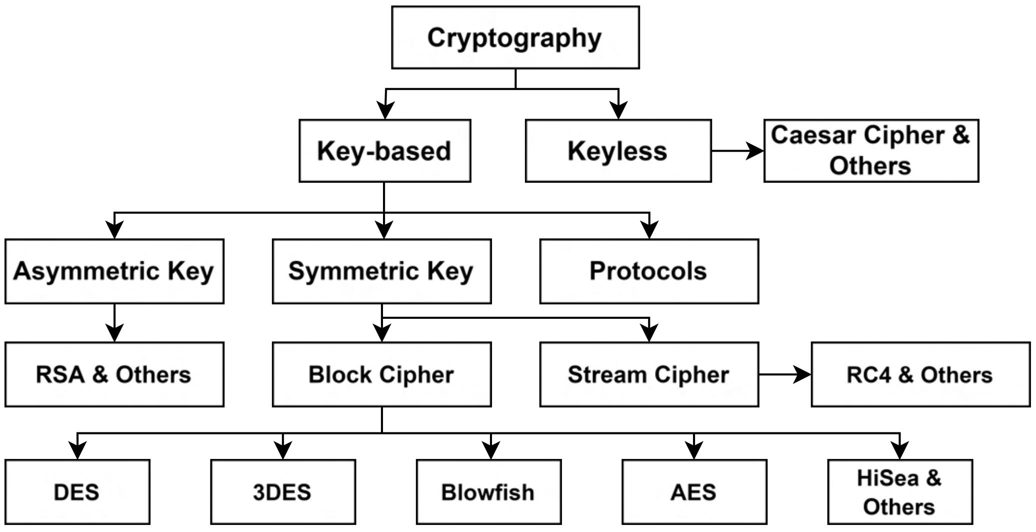


Figure 2.12: Cryptography encryption algorithms overview. (Adapted from [68])

The key-based encryption algorithms can be divided into asymmetric key, symmetric key. The asymmetric key-based encryption is composed of pairs of keys. Each pair consists of a public key (which may be known to others) and a private key (which may not be known by anyone except the owner). The key generation depends on the algorithm, and it can be the RSA algorithm [69]. Using asymmetric encryption, any person can encrypt a message using the receiver's public key, but only the receiver's private key can decrypt the encrypted message. This allows for a client-server program where the client requests the public keys, and then the server can then decrypt the received messages with its secret private key.

The symmetric key encryptions are different to the asymmetric by having the same key to encrypt and decrypt. Therefore, the key needs to be kept secret and known by the sender and receiver. Changing the key can increase the security, but the strength of symmetric encryption depends on the secrecy of encryption and decryption keys [68]. The symmetric encryption can be divided into the block and stream cyphers according to the grouping of the message bits. In a block cypher, an array of characters are encrypted all at once and sent to the receiver. The block size for the stream cypher is one character [68].

Moreover, there are several symmetric key encryption algorithms such as Data Encryption Standard (DES) [70, 69], 3DES [71, 69], Advanced Encryption Standard (AES) [69], BLOWFISH [72], HiSea [73], RC4 [74], among others.

According to [75], symmetric encryption algorithms are almost 1000 times faster than asymmetric algorithms because they require less processing power for computations. As a result, asymmetric encryption may be too heavy for IoT constrained devices. In contrast, symmetric encryption as AES is feasible, as proven in [76]. The constrained resources of the IoT devices makes asymmetric encryption not suitable for ultra-low-power applications, as stated in [77]. However, it can be solved by research in lightweight implementations or hardware accelerators, as represented in [77].

Chapter 3: System Specification

This master's thesis continues the master's thesis work of the author Sofia Paiva [1], and, as such, a review of the work done is presented in this chapter. The author developed a bare-metal monitoring NB-IoT end-device with an estimated battery lifetime of 17-years proving that an NB-IoT device can have a lifetime greater than ten years. This dissertation provides better software architecture, with an Real-Time Operating System (RTOS), Firmware Over-The-Air (FOTA), new application design, and ensuring that the software is operating correctly. However, ensuring the same ultra-low-power behaviour and lifetime greater than 10-years.

The top-level overview is shown in Figure 3.1. It is composed of two major domains: the cloud and the NB-IoT end-devices. The end-device is a monitoring system with a typical IoT architecture, as shown in Section 2.1.1 without actuators. It has multiple sensors capable of monitoring the environmental variables, an NB-IoT transceiver to communicate with the cloud maintaining the low-power consumption, and a microcontroller as the system brain. The cloud has a microservice structure, controls the end-devices, and receives the collected data.

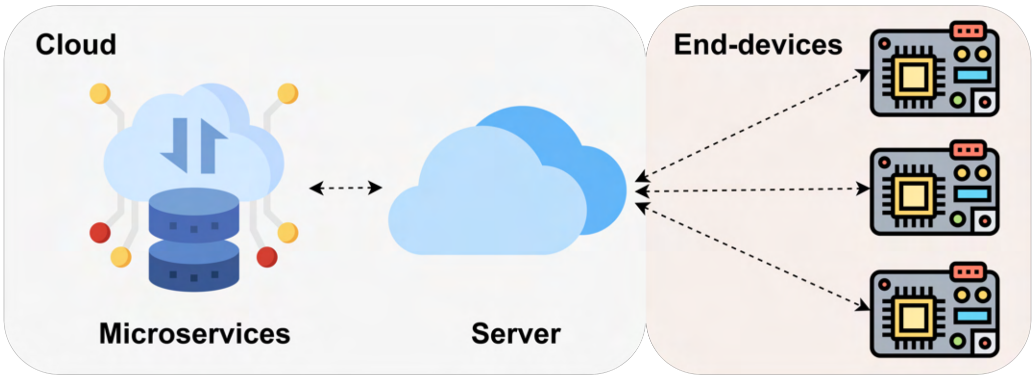


Figure 3.1: Top-level overview.

The end-device stays most of its lifetime in sleep mode to save energy, waking only to sample the sensors, transmit the respective samples, and when any sensor wakes up the MCU signalling an anomaly. The latter is possible due to the sensors having the ability to, in low-power mode, do continuous sampling and signalling the MCU.

3.1 Functional and Non-Functional Requirements

Since this project continues the author Sofia master's thesis, it includes its requirements and adds new ones. As usual, every project development has its own functional and non-functional requirements that must be followed in order to have a final product. Non-functional requirements define system attributes, while functional describe what the system should do. Listed below are the requirements and constraints concerning our system.

Functional Requirements

- Collect environmental data;
- Transmit collected data to the cloud;
- Signal abnormal behaviour;
- Receive commands from the cloud;

Non-functional Requirements

- Maintain operation for more than ten years;

3.2 Hardware

The board hardware is represented in Figure 3.2. It is composed of multiple sensors capable of measuring temperature (HDC2080 [78]), humidity (HDC2080 [78]), acceleration (BMA400 [79]), and light (OPT3002 [80]). Also, it has the ATECC608A [81] chip, which has multiple encryption algorithms, useful for ensuring secure communications to the cloud. These communications are possible with the Quectel BC66 [82], allowing the system to use the NB-IoT technology. All the external chips are controlled by the Microcontroller Unit (MCU) (STM32L081KZ [83]), which is from the low power line of STMicroelectronics. It also has incorporated hardware encryption engines, making it faster and easier to encrypt data as the ATECC608A. All the components are powered by the Saft LM17500 [84] battery. The battery voltage is dropped to $1.8\ V$ through a TPS7A02 [85] LDO, powering the MCU and sensors.

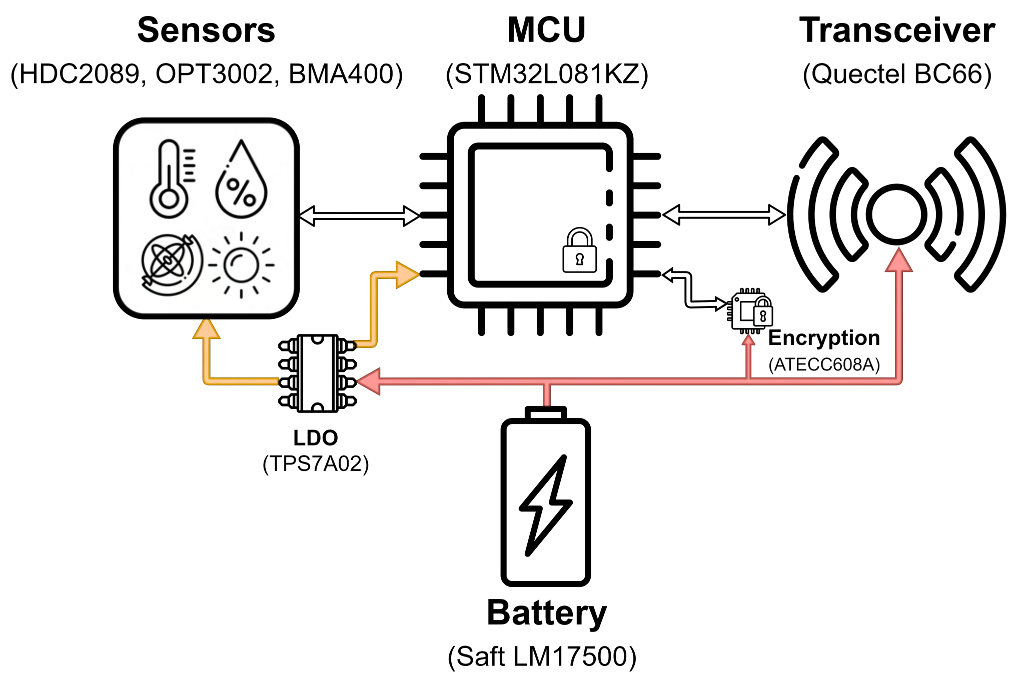


Figure 3.2: System hardware overview.

The Link4S board 3D representation front and back is depicted in Figure 3.3.

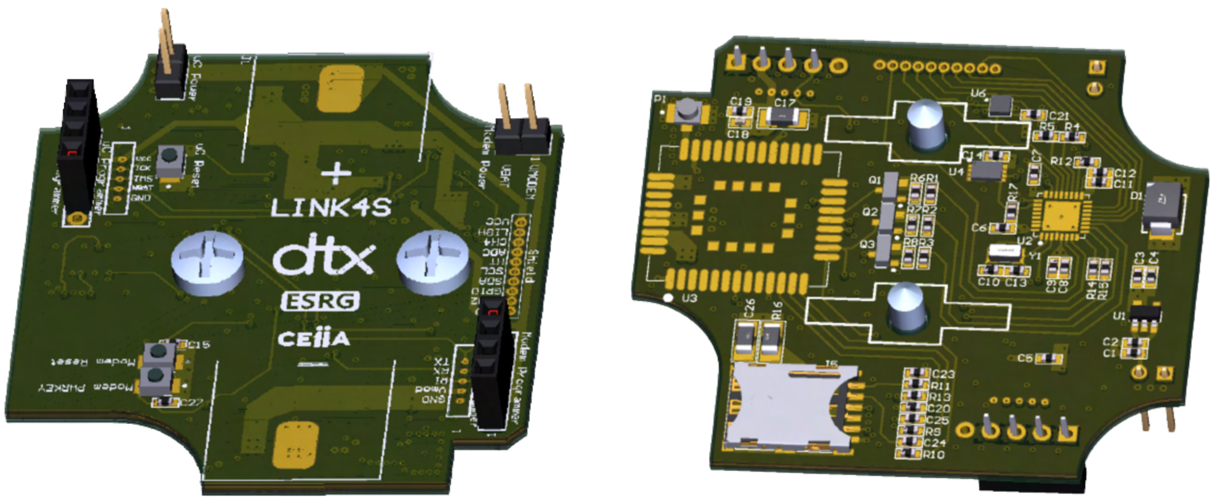


Figure 3.3: Link4S board 3D representation.

3.2.1 Sensors

The sensors allow the system monitor the environment according to its purpose. The end-device has three sensors to measure light, temperature, humidity, and acceleration. The sensors selected have low-power modes that enable continuous sampling and trigger an external pin, signalling that the sampling value is greater or less than the respective set threshold value.

Table 3.1 represents the sensors power supply, power consumption, size, interfaces, and low-power modes specifications.

Table 3.1: Sensors specifications.

Sensor	Power Supply (V)	Power Consumption	Size (mm^2)	Interface	Low-power Modes
HDC2080	1.62 - 3.6	$650\ uA/550\ uA$ @ Humidity/Temperature Measurement $0.55\ uA$ @ 1 measurement/second of Humidity and Temperature $160\ nA$ @ Sleep Mode	3x3	I^2C, INT Pin	Sleep Auto triggered
OPT30002	1.6 - 3.6	$3.7\ uA$ @ Full-scale active $1.8\ uA$ @ Dark Active $400\ nA$ @ Shutdown	2x2	I^2C, INT Pin	Shutdown Single-shot Continuous
BMA400	1.72 - 3.6	$3.5\ uA$ @ OSR-0 Normal Mode $850\ nA$ @ OSR-0 Low-power Mode $160\ nA$ @ Sleep Mode	2x2	I^2C, SPI, 2 INT Pin	Sleep Low-power Normal

Temperature and Humidity (HDC2080)

The HDC2080 [78] is an integrated humidity and temperature sensor with high accuracy measurements with very low power consumption. The capacitive-based sensor includes new integrated digital features and a heating element to dissipate condensation and moisture. The HDC2080 digital features include programmable interrupt thresholds to provide alerts and system wakes up without requiring a microcontroller to monitor the system continuously. Combined with programmable sampling intervals, a low power consumption, and support for a 1.8-V supply voltage, the HDC2080 is designed for battery-operated systems. Table 3.2 illustrates the sensor sampling rate, measurement variable, and power consumption according to the sensor modes.

Table 3.2: HDC2080 power consumption and sampling rate according to mode and variable measured.

Mode	Sampling Rate (Hz)	Measurement Type	Power Consumption (uA)
Normal	-	Temperature	550
Sleep	-	-	0.05
Continous	1	Temperature	0.3
Continous	1	Temperature & Humidity	0.55
Continous	0.5	Temperature & Humidity	0.3
Continous	0.1	Temperature & Humidity	0.105

Luminosity (OPT3002)

The OPT3002 [80] light-to-digital sensor provides the functionality of an optical power meter within a single device. This optical sensor greatly improves system performance over photodiodes and photoresistors. The OPT3002 has a wide spectral bandwidth, ranging from 300 nm to 1000 nm. Measurements can be made from $1.2\ nW/cm^2$ up to $10\ mW/cm^2$, without the need to manually select the full-scale ranges by using the built-in, full-scale setting feature. This capability allows light measurement over a 23-bit effective dynamic range. The results are compensated for dark-current effects, as well as other temperature variations. It has an interrupt pin, which can summarize the measurement result with one digital pin. The power consumption is represented in Table 3.1. As can be seen, it is very low, allowing the OPT3002 to be used as a low-power, battery-operated, wake-up sensor when an enclosed system is opened.

Accelerometer (BMA400)

Bosch Sensortec´s triaxial BMA400 [79] is an ultra-low-power acceleration sensor. It is a 12-bit digital triaxial acceleration sensor with smart on-chip motion and position-triggered interrupt features. The BMA400 is especially suited for ultra-low-power devices, which need a long-lasting battery lifetime due to its auto low power mode and auto wake-up. Further, it can distinguish between critical situations and false signals, avoiding false alarms. The sensor's power varies according to the mode used, as represented in Table 3.1.

3.2.2 Transceiver

The transceiver is responsible for the communications with the cloud server, being the system component with greater power consumption. In order to reduce the power consumption, the application uses the NB-IoT technology explained in Section 2.1.2. The NB-IoT modem used was the Quectel BC66 [82]. It has a high-performance, multi-band LTE Cat NB1 module with extremely low power consumption ($3.5\ \mu A$ in PSM, $350\ \mu A$ at Idle (DRX - 2.56s), and $110\ mA$ at LTE Cat NB1 (23 dBm, single-tone)).

The Quectel BC66 is interfaced with the MCU by UART communication and AT commands interface, having at own software accelerating the development process. The modem has the limit of each transmission to the cloud is limited of 1024 bytes and reception of 512 bytes. Also, it is already integrated with different protocol stacks such as UDP, TCP, MQTT, and LwM2M, facilitating the development process in the MCU. The BC66 has a build-in eSIM reserved for future use, which will allow the removal of the SIM external interface, reducing the number of components needed and the system power consumption.

3.2.3 MCU

The MCU is the STM32L081KZ [83], which incorporates the high-performance Arm Cortex-M0+ 32-bit RISC core operating to a max frequency of 32 MHz, a Memory Protection Unit (MPU), high-speed embedded memories (up to 192 Kbytes of Flash program memory, 6 Kbytes of data EEPROM and 20 Kbytes of RAM) plus an extensive range of enhanced I/Os and peripherals. It operates from a 1.8 to 3.6 V power supply (down to 1.65 V at power down) having multiple power-saving modes:

- **Sleep mode** – only the CPU is stopped. All peripherals continue to operate and can wake up the CPU when an interrupt/event occurs. Sleep mode power consumption is down to $140\ \mu A/MHz$.
- **Low-power run mode** – This mode is achieved with the Multispeed Internal (MSI) RC oscillator set to the low-speed clock (max 131 kHz), execution from SRAM or Flash memory, and internal regulator in low-power mode to minimize the regulator's operating current. Also, the clock frequency and the number of enabled peripherals are both limited. It can achieve a power consumption down to $8\ \mu A$.
- **Low-power sleep mode** – This mode is achieved by entering Sleep mode with the internal voltage regulator in low-power mode to minimize the regulator's operating current. In this mode, both the clock frequency and the number of enabled peripherals are limited; a typical example would be to have a timer running at 32 kHz. When an event or an interrupt triggers wake-up, the system reverts to the run mode with the regulator on. It has a power consumption down to $4.5\ \mu A$.
- **Stop mode** – All clocks are disabled except the RTC source clock (LSE or LSI) if desired. The MCU can wake up by any external interrupt line. This mode has the lowest power consumption with state retention (RAM and registers), with power consumption down to $0.4\ \mu A$ without RTC and $0.8\ \mu A$ with it.
- **Standby mode** – This mode has the lowest power consumption, achieving power consumption down to $0.28\ \mu A$ without RTC and $0.65\ \mu A$ with it. However, RAM and registers data are lost.

The application stays multiple of its lifetime in a sleep state and wakes up to sample, send the samples, or react to unnormal events. Therefore, the stop mode is essential to extend the system battery and, consequently, the system lifetime.

In order to connect the MCU to other components, for example, sensors and the transceivers, it has embedded standard and advanced communication interfaces: up to three I2Cs, two SPIs, four USARTs, a Low-power UART (LPUART). Moreover, it has several analogue features, one 12-bit ADC with hardware oversampling, two ultra-low-power comparators, several timers, one Low-power Timer (LPTIM), four general-purpose 16-bit timers and two basic timers, one RTC and one SysTick, which can be used as timebases. It also features an AES hardware encryption engine, two watchdogs, one watchdog with independent clock and window capability and one window watchdog based on a bus clock.

3.2.4 Encryption

The NB-IoT has the communications encrypted in the transmission layer. However, to have end-to-end secure communications, it is needed to use encryption algorithms. The MCU, as described above, has AES encryption engines that can be used, but the board also has the ATECC608A [81] chip. It operates at a voltage level from 2 to $5.5\ V$ and integrates Elliptic-curve Diffie-Hellman (ECDH) security protocol, along with Elliptic Curve Digital Signature Algorithm (ECDSA), as well as accessories and consumables authentication and more. This Integrated Circuit (IC) also offers an integrated AES hardware accelerator with a typical operating current of $1\ mA$ and less than $150\ nA$ of sleep current. It is interfaced with the MCU through the I2C protocol.

3.2.5 Power Supply

The board power supply is responsible for powering all the components. This board features two voltage ranges: $1.8\ V$ and $3.0\ V$ (battery voltage). The 1.8 V are needed since the MCU and sensors need it and the and it is obtained using the Low-Dropout Regulator (LDO) TPS7A02 [85]. It is an ultra-small, ultra-low quiescent current ($25\ nA$) LDO that can source $200\ mA$ with excellent transient performance. The system battery is the Saft LM17500 [84]. This battery has a nominal voltage of $3\ V$ with a capacity of $3\ Ah$ with leakage of 1%.

3.3 Software

The state machine of the application that the author Sofia Paiva finished its master's thesis [1] is presented in Figure 3.4.

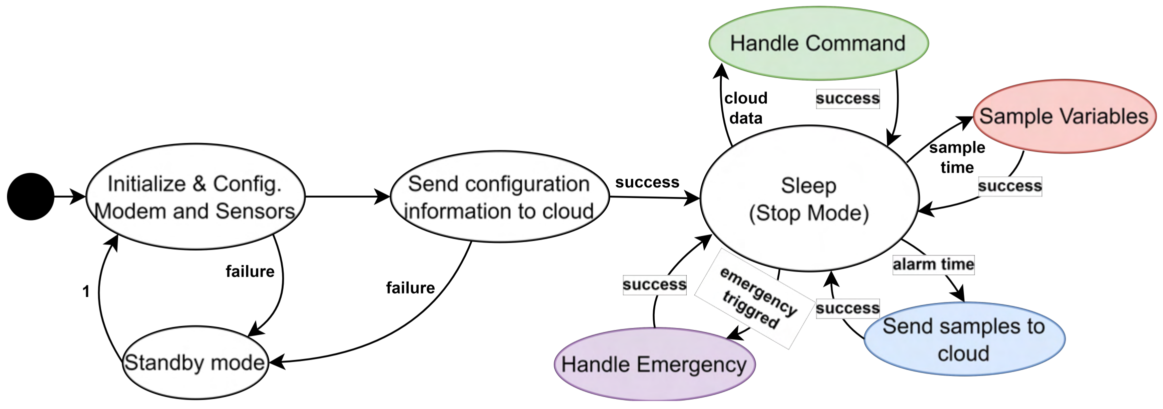


Figure 3.4: First application state machine.

It starts by initializing and configuring the modem and the sensors. Then, it sends the system configuration information to the cloud and enters the infinite application loop. In the infinite application loop, it is in the MCU stop mode power mode saving energy and wakes up for:

- **Sample the variables** – Triggered by the RTC wake up functionality, and when it wakes up, it samples the variables. The variables are considered the physical measurements of which sensor can be made. For example, the HDC2080 sensor has two variables because it can sample the temperature and humidity;
- **Send the samples made to the cloud** – Triggered by the RTC alarm functionality, and it signals to send the variables samples made;
- **Handle an emergency** – As described in Section 3.2.1, the sensors have signalling capabilities to detect when a value is greater or less than a threshold previously configured. This signalling in the application is called an emergency. When an emergency happens, the application reads all the sensors and alerts the cloud server.

When something fails in the application, it enters the MCU standby power mode resetting.

In order to implement the state machine of Figure 3.4, it was used by the author [1] the software stack represented in Figure 3.5. From the bottom to the top, it is composed of the Hardware, the Hardware Abstraction Layer (HAL), the Low-Level Portable Layer, and the Application.

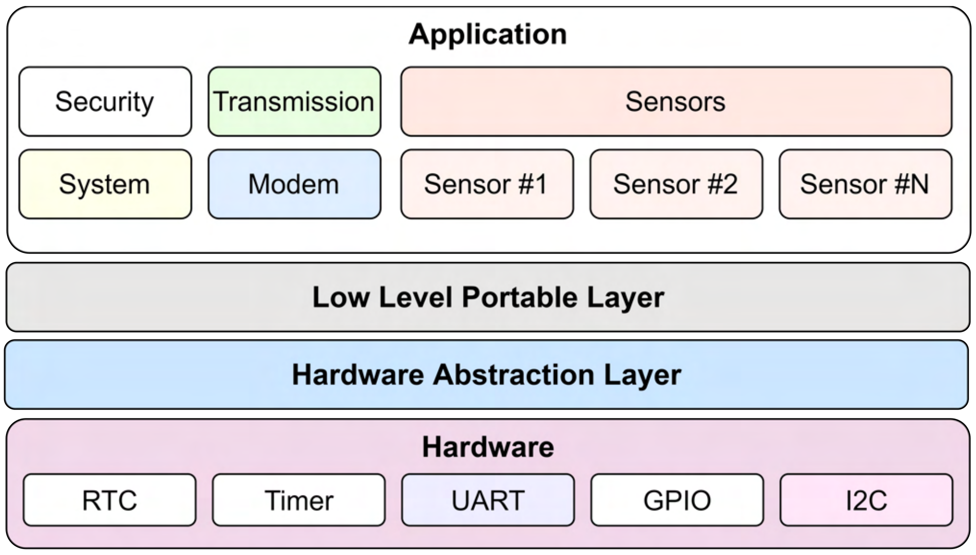
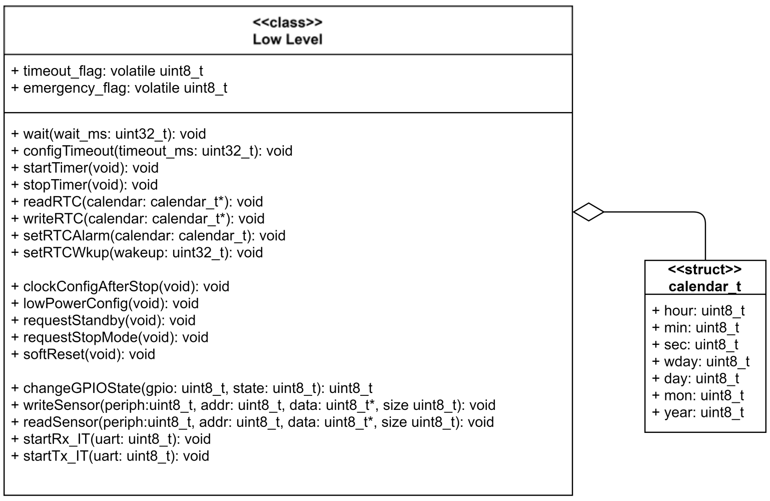


Figure 3.5: Startup application software stack.

3.3.1 Low-Level Portable Layer

The low-level portable layer, referred to as low-level, is responsible for an abstraction of the Hardware Abstraction Layer (HAL), giving an Application Programming Interface (API) to the application, illustrated in Figure 3.6. In this case, the HAL used is already available for STMicroeletronics' MCU. This layer is beneficial for providing portability to the software stack. This way, if the hardware changes, only the low-level layer is needed to be re-implemented.



<<class>>
Low Level
+ timeout_flag: volatile uint8_t
+ emergency_flag: volatile uint8_t
+ wait(wait_ms: uint32_t): void
+ configTimeout(timeout_ms: uint32_t): void
+ startTimer(void): void
+ stopTimer(void): void
+ readRTC(calendar: calendar_t*): void
+ writeRTC(calendar: calendar_t*): void
+ setRTCAlarm(calendar: calendar_t): void
+ setRTCWkup(wakeup: uint32_t): void
+ clockConfigAfterStop(void): void
+ lowPowerConfig(void): void
+ requestStandby(void): void
+ requestStopMode(void): void
+ softReset(void): void
+ changeGPIOState(gpio: uint8_t, state: uint8_t): uint8_t
+ writeSensor(periph:uint8_t, addr: uint8_t, data: uint8_t*, size uint8_t): void
+ readSensor(periph:uint8_t, addr: uint8_t, data: uint8_t*, size uint8_t): void
+ startRx_IT(uart: uint8_t): void
+ startTx_IT(uart: uint8_t): void
<<struct>>
calendar_t
+ hour: uint8_t
+ min: uint8_t
+ sec: uint8_t
+ wday: uint8_t
+ day: uint8_t
+ mon: uint8_t
+ year: uint8_t

Figure 3.6: Class diagram of Low-level portable layer.

The methods of low-level portable layer API can be divided into three sections:

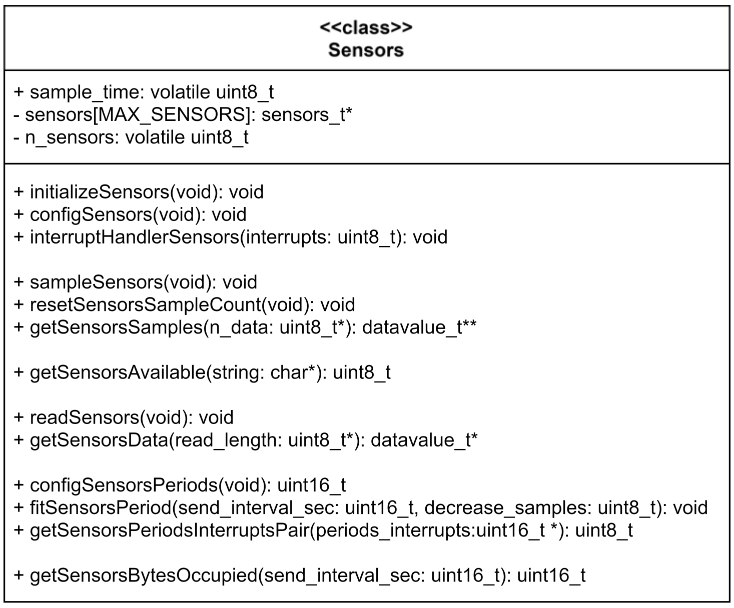
- **Timers** – The timers allow the application to wait specific periods and give the ability to use a timer, for example, as a timeout timer. Also, it provides a straightforward API to use the RTC with the structure calendar_t. This structure facilitates reading, writing the RTC and setting up alarms.
- **System control** – The system control is what actuates in the MCU's clocks, power modes, being the direct API to its needed internals.
- **MCU external interfaces** - The external interfaces are responsible for handling the General Purpose Input/Output (GPIO) and its communication peripherals as I2C and UART.

3.3.2 Application Layer

The Application Layer uses the API offered by the Low-Level Portable Layer, and it is responsible for executing the application purpose. It can do it by separating itself into multiple modules such as Security, System Control, Transmission, Modem, and Sensors. These modules will explained in more extensive depth in the following sections.

Sensors

The sensors module is responsible for the manager of the sensors, for example, initializing, configurting, and reading them. Figure 3.7 illustrates the sensors module, it has initialize, configuration, sample, read functions.



```
<<class>>
Sensors

+ sample_time: volatile uint8_t
- sensors[MAX_SENSORS]: sensors_t*
- n_sensors: volatile uint8_t

+ initializeSensors(void): void
+ configSensors(void): void
+ interruptHandlerSensors(interrupts: uint8_t): void

+ sampleSensors(void): void
+ resetSensorsSampleCount(void): void
+ getSensorsSamples(n_data: uint8_t*): datavalue_t**

+ getSensorsAvailable(string: char*): uint8_t

+ readSensors(void): void
+ getSensorsData(read_length: uint8_t*): datavalue_t*

+ configSensorsPeriods(void): uint16_t
+ fitSensorsPeriod(send_interval_sec: uint16_t, decrease_samples: uint8_t): void
+ getSensorsPeriodsInterruptsPair(periods_interrupts:uint16_t *): uint8_t

+ getSensorsBytesOccupied(send_interval_sec: uint16_t): uint16_t
```

Figure 3.7: Sensors module interface.

The sensors module was designed so adding a new sensor would be easy. To achieve this, all sensors' drivers have the API to initialize, configure, read, and handle its interrupts. A sensor structure is represented in Figure 3.8. It comprises a sensor configuration, model, type, and units string, protocol (I2C, USART, SPI), status, an array of pointers to the sensordata_t, the period, the counter, and counts.

The sensordata_t has the name of the sensor's data, for example, TMP for temperature or HUM for humidity, and the samples is an array of the struct datavalue_t, the number of samples.

The sample periods are decomposed in counts and counter so that it is possible to have multiple periods to different sensors data. Firstly, the counts are calculated by finding the sensors' minimum period (configSensorsPeriods) and setting a common multiple to the other periods. Secondly, from the minimum period is determined the counts (fitSensorsPeriods) (number of times the MCU wakes up to sample) needed to achieve the respective period. Every time the MCU wakes up to sample, the counter variable in all data sensors is incremented. When it matches the counts, the sensors data is sampled, and the counter variable is reset.

The sensors sampling has two modes of operation: simple and average. No sample post-processing is done in simple mode, and every sample is saved as a single sample. In average mode, a predefined number of samples is required; when the sampling ends, the average, minimum and maximum values are saved.

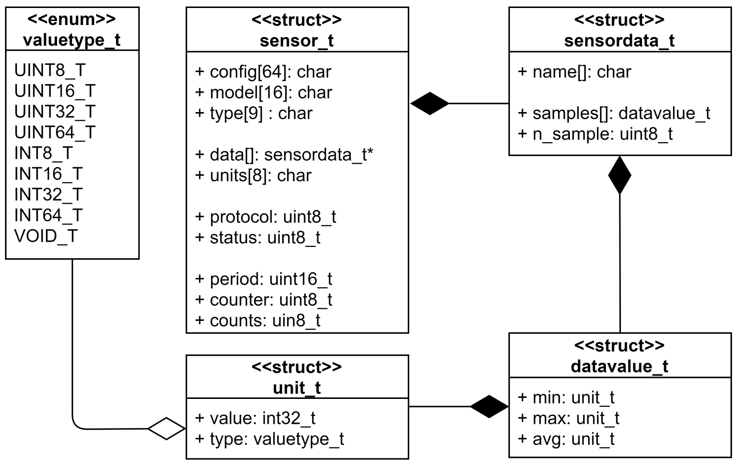
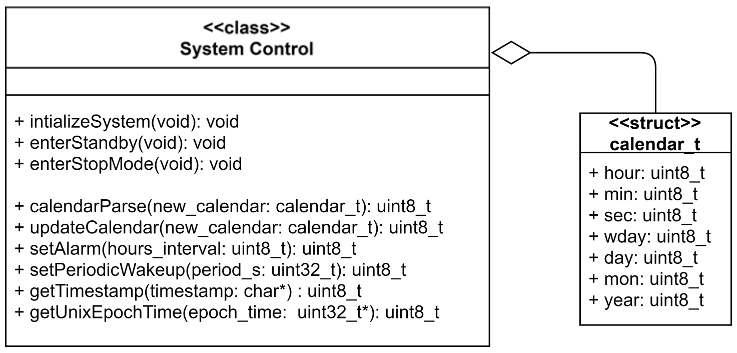


Figure 3.8: Sensors data structure.

Due to the two modes of operation, the author of [1] came up with the datavalue_t struct type to achieve a generic approach saving the samples. It is composed of the minimum, maximum, and average of type unit_t. The unit_t is a struct with an integer value and the valuetype_t enumeration. The enumeration is useful to know how to use the saved value. For example, if it is a 32-bits signed or a 16-bits unsigned value.

System Control

Figure 3.9 represents the System Control interface. It controls all functions related to the MCU, such as power modes (enterStopMode and enterStopMode) and the RTC (setPeriodicWakeup and setAlarm). Also, it features a timestamp get function that can convert the RTC time to UNIX (getUnixEpochTime) or string format.

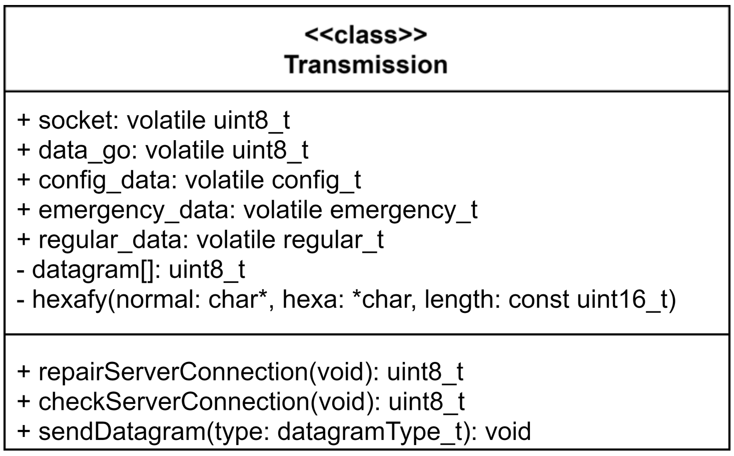


<<class>>
System Control
+ intializeSystem(void): void
+ enterStandby(void): void
+ enterStopMode(void): void
+ calendarParse(new_calendar: calendar_t): uint8_t
+ updateCalendar(new_calendar: calendar_t): uint8_t
+ setAlarm(hours_interval: uint8_t): uint8_t
+ setPeriodicWakeup(period_s: uint32_t): uint8_t
+ getTimestamp(timestamp: char*) : uint8_t
+ getUnixEpochTime(epoch_time: uint32_t*): uint8_t
<<struct>>
calendar_t
+ hour: uint8_t
+ min: uint8_t
+ sec: uint8_t
+ wday: uint8_t
+ day: uint8_t
+ mon: uint8_t
+ year: uint8_t

Figure 3.9: System control module interface.

Transmission

The transmission module interface is represented in Figure 3.10. It is composed of three functions: repairServerConnection, checkServerConnection, and sendDatagram. The repairServerConnection repairs the connection with the cloud server. The check server connection checks if it is possible to establish a server connection, and the sendDatagram is responsible for serializing and sending the respective application structure to the cloud. The external module variables are the socket, the data_go flag, and the transmission structures: the configuration, the emergency and the regular. As for internals, there is the datagram array and the hexafy function.



```
<<class>>
Transmission

+ socket: volatile uint8_t
+ data_go: volatile uint8_t
+ config_data: volatile config_t
+ emergency_data: volatile emergency_t
+ regular_data: volatile regular_t
- datagram[]: uint8_t
- hexafy(normal: char*, hexa: *char, length: const uint16_t)

+ repairServerConnection(void): uint8_t
+ checkServerConnection(void): uint8_t
+ sendDatagram(type: datagramType_t): void
```

Figure 3.10: Transmission module interface.

The data received through the public structures represented in Figure 3.11 is serialized using the message pack and transmitted to the cloud server. The public transmission structures represent the possible application transmissions: emergencies, configurations, and regulars.

- **config_t struct** – It has the device id, the transmission timestamp, the sensors enabled string, the modem configuration string, and the periods and interrupts state of the sensors in pair in a 16-bit unsigned array. The periods and interrupts pair, for example, [20, 1, 30, 0, 10, 2], and can be understood that the first sensor data period is 20 seconds and the interrupt value 1; the second has a period of 30 seconds and no interrupt; the third one has a period of 10 seconds, and an interrupt value of 2.

- **emergency_t struct** – It is composed of the device id, timestamp in the transmission, the flag that triggered the emergency, the sensors samples in the emergency moment and the samples made number.

- **regular_t struct** – It is formed of the device id, the transmission timestamp and signals quality, a pointer to the sensor data array and the number of sensors data structures.

The socket variable saves the socket modem number that is used. The data_go indicates when it is time to send to the cloud. The hexafy function is responsible for converting an array of characters to a hexadecimal one.

The message pack is an efficient binary serialization format. It lets data exchange among multiple languages like JSON, but it is faster and smaller.

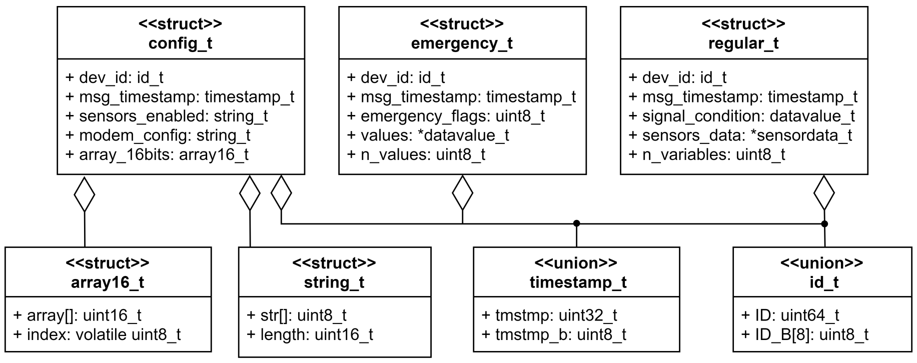
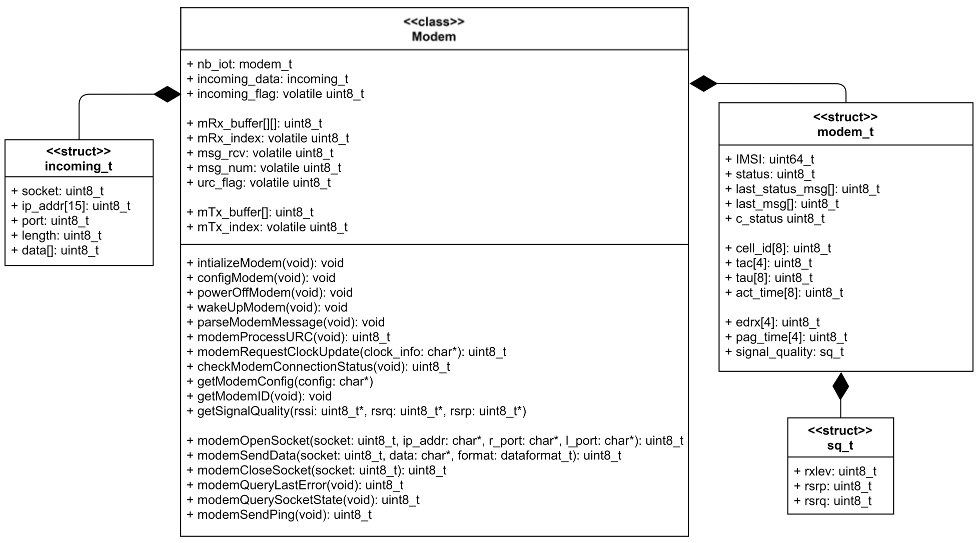


Figure 3.11: Transmission structures.

Modem

The modem is responsible for the NB-IoT technology, having its own communication interface with the MCU through UART using AT commands, accelerating the development process. The driver for it is represented in Figure 3.12.



<<class>> Modem
+ nb_iot: modem_t
+ incoming_data: incoming_t
+ incoming_flag: volatile uint8_t
+ mRx_buffer[][]: uint8_t
+ mRx_index: volatile uint8_t
+ msg_rcv: volatile uint8_t
+ msg_num: volatile uint8_t
+ urc_flag: volatile uint8_t
+ mTx_buffer[]: uint8_t
+ mTx_index: volatile uint8_t
+ intializeModem(void): void
+ configModem(void): void
+ powerOffModem(void): void
+ wakeUpModem(void): void
+ parseModemMessage(void): void
+ modemProcessURC(void): uint8_t
+ modemRequestClockUpdate(clock_info: char*): uint8_t
+ checkModemConnectionStatus(void): uint8_t
+ getModemConfig(config: char*)
+ getModemID(void): void
+ getSignalQuality(rssi: uint8_t*, rsrq: uint8_t*, rsrp: uint8_t*)
+ modemOpenSocket(socket: uint8_t, ip_addr: char*, r_port: char*, l_port: char*): uint8_t
+ modemSendData(socket: uint8_t, data: char*, format: dataformat_t): uint8_t
+ modemCloseSocket(socket: uint8_t): uint8_t
+ modemQueryLastError(void): uint8_t
+ modemQuerySocketState(void): uint8_t
+ modemSendPing(void): uint8_t
<<struct>> incoming_t
+ socket: uint8_t
+ ip_addr[15]: uint8_t
+ port: uint8_t
+ length: uint8_t
+ data[]: uint8_t
<<struct>> modem_t
+ IMSI: uint64_t
+ status: uint8_t
+ last_status_msg[]: uint8_t
+ last_msg[]: uint8_t
+ c_status uint8_t
+ cell_id[8]: uint8_t
+ tac[4]: uint8_t
+ tau[8]: uint8_t
+ act_time[8]: uint8_t
+ edrx[4]: uint8_t
+ pag_time[4]: uint8_t
+ signal_quality: sq_t
<<struct>> sq_t
+ rxlev: uint8_t
+ rsrp: uint8_t
+ rsrq: uint8_t

Figure 3.12: Modem class interface.

The modem interfaces present initialization, configuration, power on and off, get, and communication functions. The communication is done through UDP or TCP having the open and close socket functions. It is also possible to request the current clock time and check if the modem is connected to the network.

As public variables, the modem has the mRx_buffer and mTx_buffer buffers used to receive and transmit data between itself and the MCU. Also, the modem struct saves modem data, as its status, cell id, and signal quality. The application is not prepared to receive data from the cloud, but the modem module saves it to the incoming_t struct.

The modem offers functions as parseModemMessage and modemProcessURC used to be called in the bare-metal infinite main loop when the respective flags msg_rcv and urc_flag are set.

Security

The security was not implemented due to time and incompatible cloud implementation.

This chapter aimed to specify the end application of the master's thesis [1] because it is being continued. Therefore, it was started by explaining the big picture by a top-level overview, functional and non-functional requirements, and system hardware and software specifications. The application hardware is an MCU (STM32L081KZ), sensors (HDC2080, OPT3002, BMA400), power supply (LDO TPS7A02 and the battery Saft LM17500), encryption (ATECC608A and MCU encryption engines) and the transceiver (Quectel BC66). The software application is composed of the HAL, the Low Level, Sensors, Transmission, Modem, System Control and Security. The next chapter will explore the possible improvement opportunities.

Chapter 4: Improvement Opportunities

This chapter will present the general system overview, starting with the overall end-device architecture that will be analysed, finding where the software can be improved or new features added. Then, the cloud and use cases will be presented.

4.1 Architectural Improvements

The application system architecture is presented in Figure 4.1. It is composed of three layers: the end-device, the network infrastructure and the cloud services. As explained in Section 3, the end-device is an ultra-low-power device that stays most of its life sleeping and wakes up to measure environment variables or signals when an anomaly occurs, communicating with a cloud server through the NB-IoT technology. The network is one of the main advantages of the NB-IoT technology since it uses mobile antennas as the LTE technology. This way, the end-device can be deployed where the mobile network delivers antenna coverage. The mobile network used in this project is NOS [86] because they are project sponsors. Finally, the cloud is composed of the server and the microservices, as the author Paiva proposed in [1], but they were not implemented as itnot in its thesis scope. It should receive the end-devices data, parse it, and save it in a database.

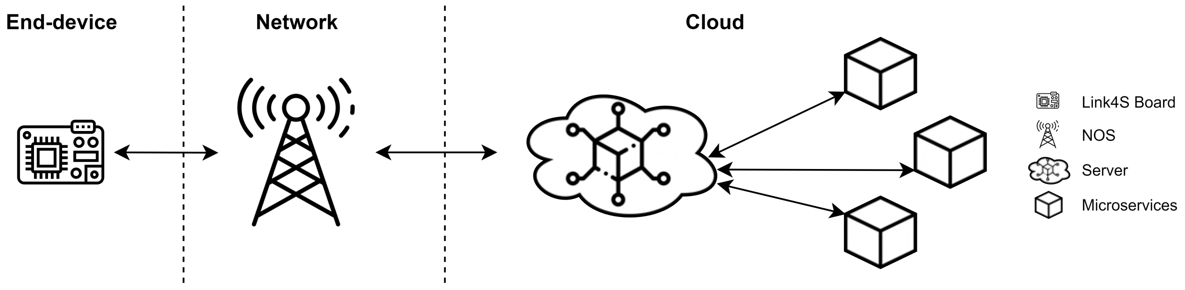


Figure 4.1: System overview.

4.1.1 End-device

There is two main software and architectural improvements in the end-device: the first is the addition of commands from the cloud, enabling alter and retrieve device parameters in run-time, for example, setting the enabled sensors, sampling rates and interrupt status, or getting a sensor configuration or the transmission time interval. The second is the conversion of the application from a bare-metal design to an

RTOS based one because it adds flexibility to the design and operation reliability. As the system becomes more complex, it simplifies software integration [87]. Also, the RTOS can take advantage of the end-device communication delays improving efficiency, consequently increasing the code binaries and the amount of RAM space occupied due to internal variables and the threads stacks.

Figure 4.2 is the end-device system overview illustrating how the several software modules, explained in Section 3.3, interact with each other.

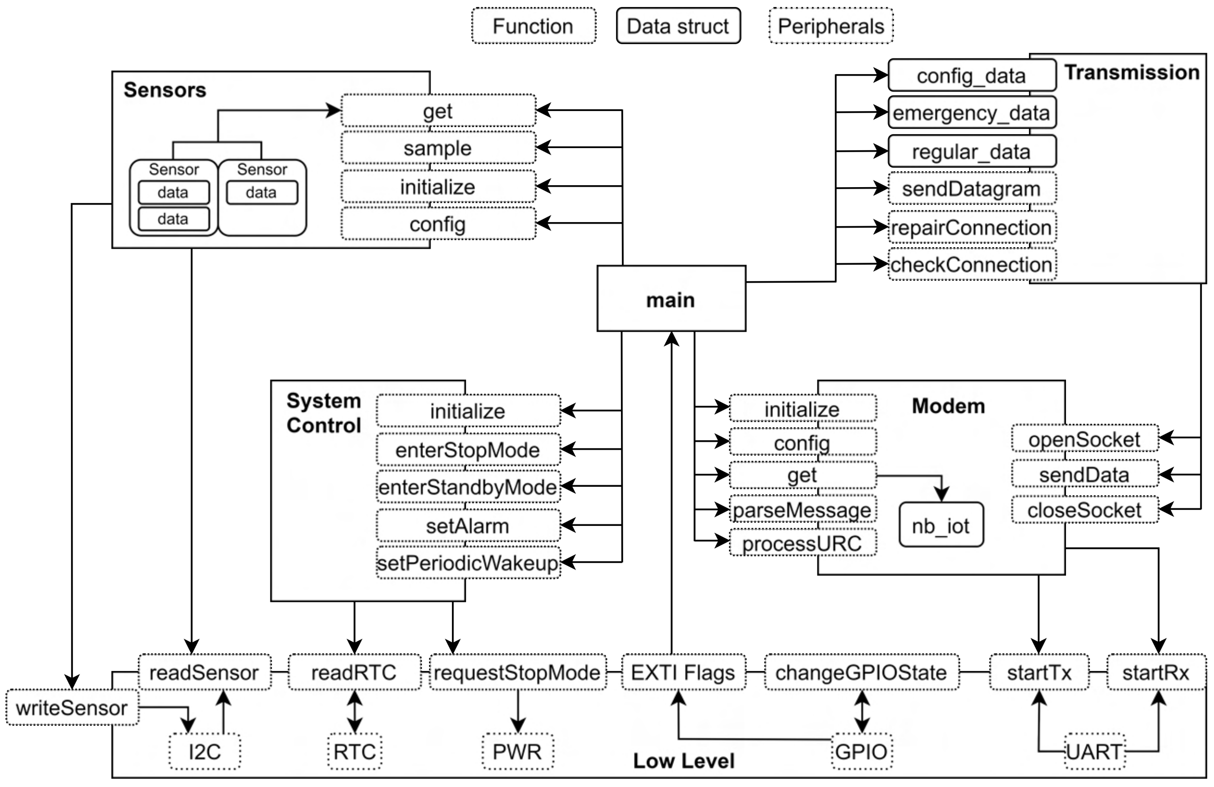


Figure 4.2: End-device system overview.

Low-Level

The low-level portable layer is beneficial in the software stack since it allows the hardware to change, and only this layer is required to be remade, wrapping the MCU HAL. Therefore, a better name for it is HAL Wrapper.

In order to have a good software stack, the HAL Wrapper should provide a set of peripherals interfaces needed by the upwards modules, including the flags and buffers. For example, the modem internal variables in the class diagram of Figure 3.12 have the mRx_buffer and the mTx_buffer. These buffers are used in the UART transmissions to the modem, so they should be allocated to the UART. Then, the modem module uses that peripheral interface, consequently the peripheral buffers. Another example is in the External Interrupt (EXTI) peripheral. When an interrupt occurs, the callback sets flags from the modules above. For example, the sensors external interrupts or the sample_time flag located in the sensors module. These flags should be linked to the respective MCU pins, and the application uses them as required.

Sensors

The sensors module was designed to be as generic as possible. Hence, its drivers were implemented with a set of required functions:

- initialisation – check if the sensor is available;
- configuration – configures the sensor sampling rate, interrupt mechanism, power modes;
- read – read the sensors data;
- interrupt handler – if the sensor has an interrupt mechanism, handle if needed; for example, some sensors after an interrupt need to a register be read to be able to be triggered again;

The ability to turn on or off the interrupt and sensor could be added to this set of functions. Also, be able to change the sensor interrupt value set.

With the addition of commands from the cloud server, it is needed to add new configuration variables, for example, to enable or disable the interrupt, sampling, and sensor to the sensor structure represented in Figure 3.8.

Since the same sensor can have multiple variables, the temperature and humidity sensor (HDC2080) can measure the humidity and temperature, having the sensor two variables. The way the sensor structure and the sensors module interpret the sensors data could be different. The variable approach is easier to control with the new commands architecture and gives a clear data relationship.

Due to the generic approach in sensor samples save, the structure datavalue_t was used. This structure is composed of three unit_t structures allowing the average sensor mode, and each unit_t struct has a 4-byte integer and a 4-byte valuetype_t resulting in 8 bytes per unit_t. Since every datavalue_t has three unit_t, each occupies 24-bytes.

Suppose the sensor maximum samples number is set to 16, the datavalue_t array occupies 384 bytes in memory. If only the samples are saved with a sensor sample size of 8-bit, it will take, in simple and average mode, 16 and 48 bytes, respectively. Changing to a sensor sample size of 16-bit will take 32 and 96 bytes, respectively. Also, because the samples are saved in the array of structures in the sensordata_t, the number of samples is chosen in compile-time, being impossible to have different sensordata_t array sizes.

Moreover, with the RTOS, the sensor each sampling could be different. A thread for each sensor could not be possible due to RAM constraints, but maybe re-design in a way that is easier to achieve the pretended generic approach.

To sum up, all the sensors module architecture can be improved, optimising in space, using and functionality.

Modem

The modem has two transmission modes: hexadecimal and text mode. The implementation of author Paiva [1] uses the hexadecimal mode. This mode decreases the data transmission by half, so changing the mode to text mode is necessary.

The bare-metal implementation receives data asynchronously from the modem. Therefore, adding the operating system will be very advantageous to enchant the modem driver.

Transmission

The transmission module is responsible for the communication between the end-devices and the cloud server. The main module fills the public transmission structures and calls the sendDatagram function to send a regular, emergency, or configuration message. In order to add modularity, the communication with the cloud could be generic, not having specific structures to each type of communication. For example, if a new parameter wants to be added in the configuration message, the struct config_data, the fill struct code in main, and the struct read and structure of the datagram in transmission need to be changed. It would be better if only the main module could append a new datagram object.

The application transmission bottleneck is the limit from the modem of transmission of 1024 bytes. Hence, the transmission was designed to send a restricted number of 1024-bytes for each transmission, restraining the data collected size between transmissions. Nevertheless, the transmission could be designed to can part the message in 1024-bytes packets, restricting the data collected to the device memory available to store.

The modem module is used by the main and the transmission module. In order to encapsulate it, it can be controlled and managed by the transmission module, thus, increasing the architecture modularity.

Also, the transmission module has two buffers: (1) the buffer used to fill the datagram with the public communication structs of size 512 bytes, and the hexadecimal version of the datagram with 1024 bytes of length. The hexadecimal datagram buffer is due to the use of the modem in hexadecimal mode. Also, in the modem implementation, it copies the hexadecimal datagram to an internal command buffer. It causes many buffers that can be shared to save RAM.

Adding the commands to can change end-device parameters in run-time will change the module purpose, making the module able to send and receive messages. Therefore, the module name will be changed to communication.

Software Stack

The software stack can improve by separating the HAL and the external MCU drivers (sensors, modem), resulting in the modem, and the sensors drivers are not in the application layer. Consequently, the application layer stays only the needed modules to execute the application purpose. The Low-Level Portable Layer should include the Hardware Abstraction Layer (HAL). Also, to add modularity, the application modules should encapsulate the peripherals in the lower layers. For example, the Transmission module can encapsulate the modem and security module, and the Sensors module the sensors drivers.

4.1.2 Cloud

The cloud was not in the scope of the master's thesis of the author [1], so it did not design and implemented. On the other hand, this master's thesis will design and implement since it continues the author's work. The cloud architecture proposed is of microservices. Due to the project scalability, the microservices architecture is beneficial for continuous development. Because it is easier to build, maintain the application, add flexibility, and improve productivity, scalability, and speed. It is composed of a microservice server and microservices clients represented on the right side and left side of Figure 2.10, respectively.

4.2 Use Cases

The end-device application overall use case is represented in Figure 4.3. The end-device stays most of its life sleeping and wakes up to get the environmental data by sampling the sensors or sensors external interrupts and transmitting them to the cloud through regular datagram transmissions. Also, when it sends data to the cloud, it can receive commands that alter the system behaviour.

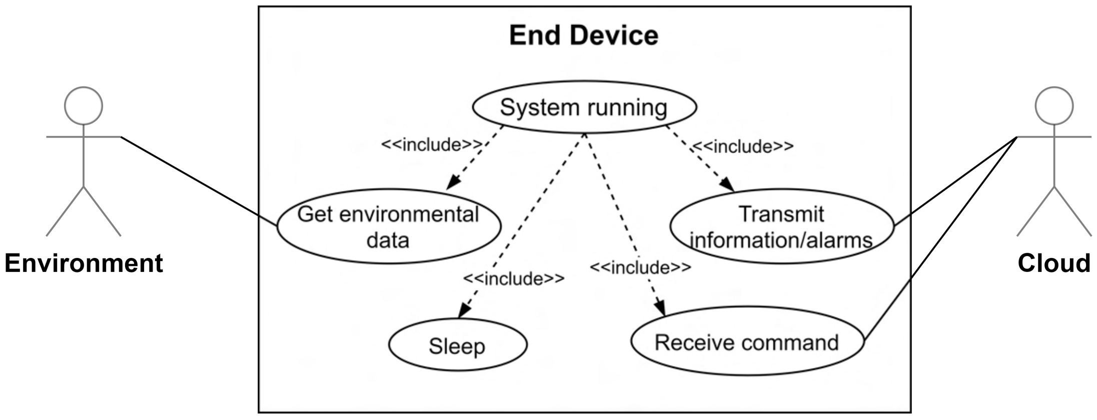


Figure 4.3: Start application use cases.

Chapter 5: Design

This chapter will present the Delta differencing algorithm for FOTA, then the end-device and cloud use cases followed its designs. The device will be re-designed from a bare-metal architecture to use an RTOS. Then new sensors and transmission architectures will be improved. New modules such as the OTA, Commands, and the Trace will be introduced. Afterwards, the cloud microservices design using the Blackwing framework and commands will be present.

5.1 Use Cases

The system design will be improved with the use cases illustrated in Figure 5.1. It is similar to Figure 4.3, but the command reception, encryption, and decryption in the communication data are added. When it sends data to the cloud, it can receive commands that alter the system behaviour.

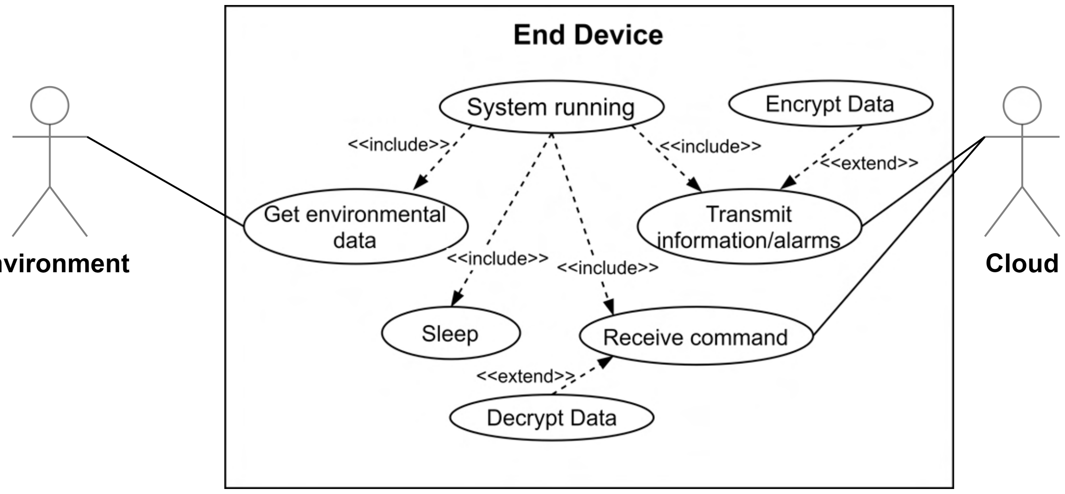


Figure 5.1: End-device use cases.

The cloud use case interaction with the end-device is represented in Figure 5.2. The end-device sends its configuration, emergency, regular, and command response messages to the cloud. If they have any command, the cloud microservices send back to the end-device.

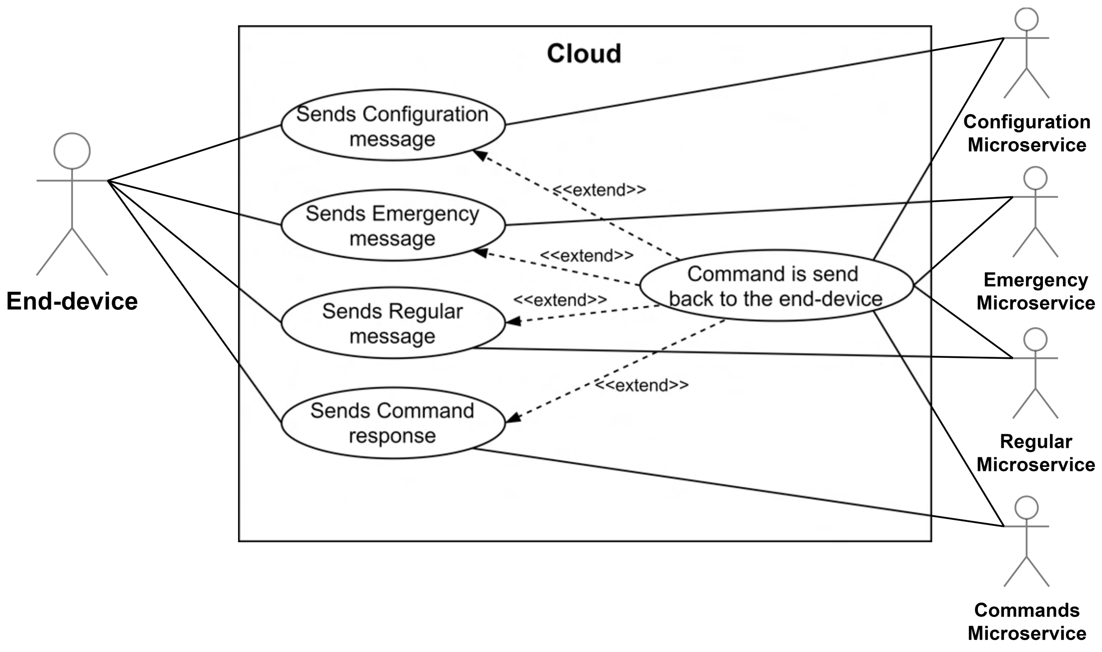


Figure 5.2: Cloud use case.

The cloud microservices interactions with the database are represented in Figure 5.3. The microservices insert the message received from the end-device and query the device's commands waiting.

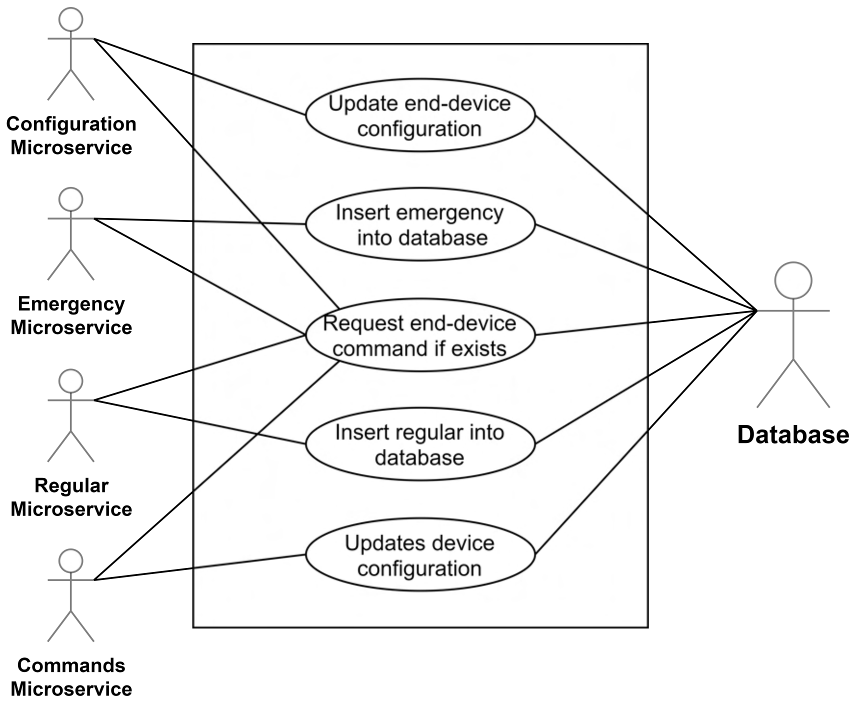

Configuration Microservice
Emergency Microservice
Regular Microservice
Commands Microservice
Update end-device configuration
Insert emergency into database
Request end-device command if exists
Insert regular into database
Updates device configuration
Database

Figure 5.3: Database use cases.

5.2 Firmware Over-The-Air Algorithm

This section explains how the new firmware to be updated Over-The-Air will be generated using the Delta algorithm. It starts with an overview of the delta generation through a block diagram, following the delta commands and finalizing with the delta algorithm.

Figure 5.4 presents an overview of the OTA delta. Each one of the blocks will be described next in detail. It starts by receiving the old and new firmware in Intel Hex [88] or binary format and converting them to arrays to enter in the differencing algorithm named DeltaGen. DeltaGen generates the delta commands, and these commands are converted to compatible board packages in the last stage.

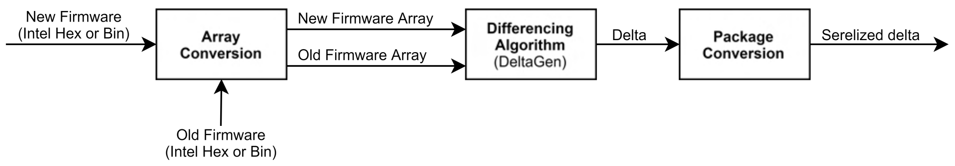

New Firmware (Intel Hex or Bin)
Array Conversion
Old Firmware (Intel Hex or Bin)
New Firmware Array
Old Firmware Array
Differencing Algorithm (DeltaGen)
Delta
Package Conversion
Serelized delta

Figure 5.4: OTA delta block diagram.

The Array Conversion stage, as described above, receives the new and old firmware in Intel Hex or binary formats and converts them to arrays. The hex file format conveys binary information in American Standard Code for Information Interchange (ASCII) text form. Each text line contains hexadecimal characters that encode multiple binary numbers. The binary numbers may represent data, memory addresses, or other values, depending on their position in the line and the type and length of the line [88]. The binary format is the representation of the memory in binary.

These arrays are fed into the Differencing Algorithm upon converting into arrays. This algorithm is responsible for finding the common parts from the new and old firmware that can be copied. If it can not, it add as new. The differencing algorithm generates the delta, which is composed of the following commands:

- ADD – Adds a byte sequence of a specified length;
- COPY – Copies a byte sequence from the old code to the new code;
- END – Indicates the end of the delta;

The following form is used to describe the commands and their fields, where the subscript number in each field represents the number of bytes it occupies:

$$\mathtt{END}_1 \tag{5.1}$$

The END command has the opcode number 0 and a fixed size of one byte. This command is useful when the delta is split into multiple packages since it signals its end.

$$\mathtt{COPY}_1\ \mathtt{n}_{1-2}\ \mathtt{old\_address\_offset}_{1-4} \tag{5.2}$$

The COPY command has the opcode number 1, followed by the number of bytes to copy (n) from the old firmware and the old firmware address offset to start copy.

$$\mathtt{ADD}_1\ \mathtt{n}_{1-2}\ < \mathtt{Byte}_1, \mathtt{Byte}_1 >_n \tag{5.3}$$

The ADD command has the opcode number 2 and the length (n) of new bytes received from the cloud to add to the new firmware sequence.

Since the commands opcode values range from 0 to 2, it only requires two bits to be represented. The remaining six bits from a byte will be used, as shown in Figure 5.5, to represent the number of bytes needed in each parameter, reducing the delta size and having generic copy offsets (old address offset) and length (n). The commands opcode bits 2-4 have the size of length (n) in bytes (i.e. one equals 1 byte), and in the CPY command, the bits 7-5 are the size of the copy address. Figure 5.5 is depicted the described command structure. In this figure example, in the CPY command, the number of bytes representing the old address and length (n) is 1 (one) and 2 (binary 10 equals 2 in decimal), respectively. The parameter length (n) of the command ADD number of bytes is 1 (one).

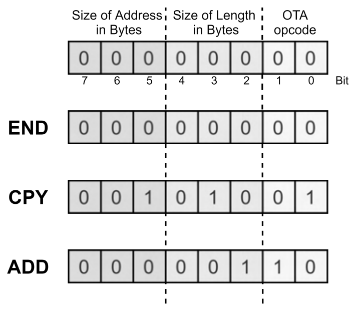


Figure 5.5: OTA command structure.

As a result, all the delta bytes have meaningful information and does not have, for example, zeros to fill a four bytes word to identify the old address offset.

The delta algorithm is similar to the Differencing Algorithm based on Suffix Array (DASA) represented in [47], which proposes an optimal differencing algorithm that uses a byte-level comparison to generate the smallest delta size employing an efficient data structure called suffix array [49]. A Suffix Array is a sorted array of all suffixes of a string. It is a data structure used in, among others, full-text indices and data compression algorithms. In this master's thesis is proposed the DeltaGen algorithm, whose core is similar to the DASA algorithm using the suffix array and a byte-level comparison.

The DeltaGen block diagram is represented in Figure 5.4. It starts by receiving the new and old firmware arrays and adding the sentinels $ and # into one array. The sentinels' values can not be present in the firmware arrays, and the value of $ has to be less than #.

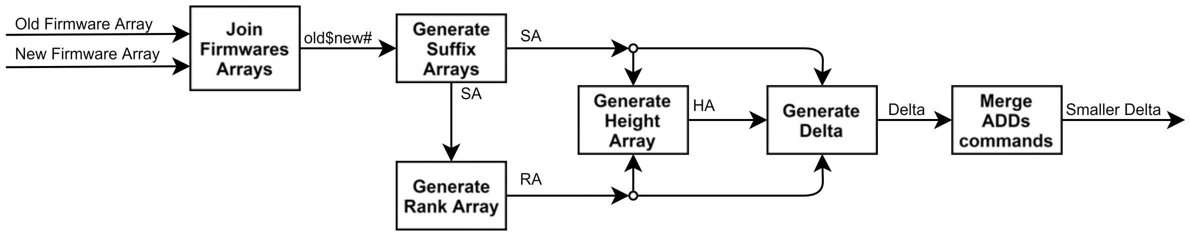
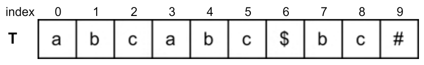


Figure 5.6: DeltaGen block diagram.

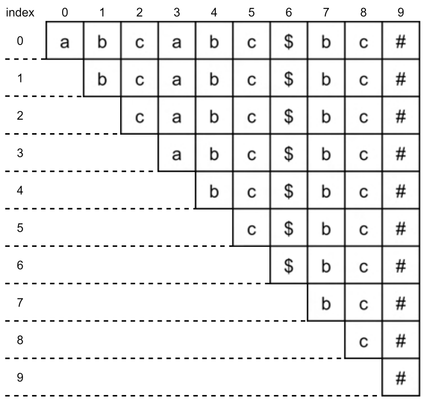
The DeltaGen starts by joining two strings forming the T (text) array illustrated in Figure 5.7. For example purposes, the string “abcabc” is the old firmware, and the “bc” is the new firmware.



index	0	1	2	3	4	5	6	7	8	9
T	a	b	c	a	b	c	$	b	c	#

Figure 5.7: Join of old and new strings.

The suffix array, as described above, it is an alphabetically sorted array of all suffixes of a string. From the string T, the unsorted suffix arrays are exemplified in Figure 5.8.



index	0	1	2	3	4	5	6	7	8	9
0	a	b	c	a	b	c	$	b	c	#
1		b	c	a	b	c	$	b	c	#
2			c	a	b	c	$	b	c	#
3				a	b	c	$	b	c	#
4					b	c	$	b	c	#
5						c	$	b	c	#
6							$	b	c	#
7								b	c	#
8									c	#
9										#

Figure 5.8: DeltaGen unsorted suffix arrays.

Figure 5.9 represents the sorted suffixes arrays. The array at the left side of the figure has the index of the respective sorted sub-array in the T array. This way, there is no need to save all the sub-strings and just the start index in the T string. This array of indexes forms the Suffix Array (SA), representing the suffixes arrays index in alphabetic order.

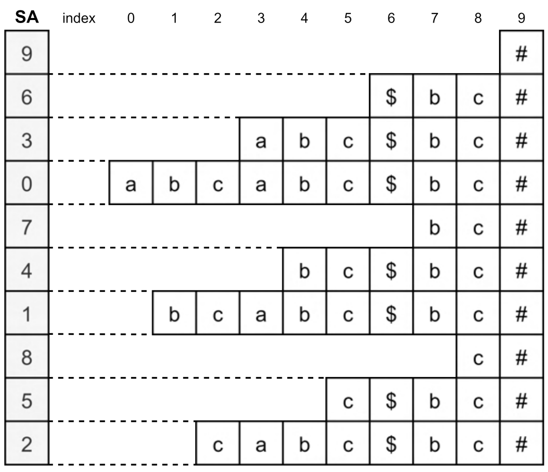


Figure 5.9: DeltaGen sorted suffix arrays.

The Rank Array (RA) contains the index of the respective sub-array index in the SA array, illustrated in Figure 5.10. For example, the suffix array that starts at index 0 in T has index 3 in alphabetic order. Therefore, the first value of the RA array is 3. Moreover, the suffix “abc$bc#” with index 3 in T is the index 2 in the SA. As a result, the rank array will have in position three the index 2.

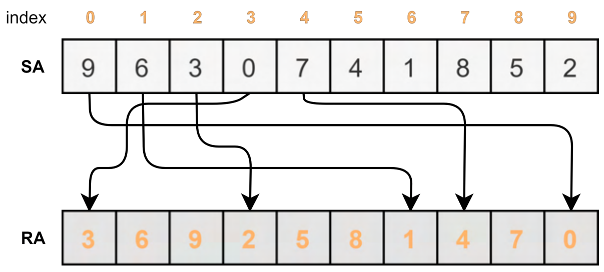


Figure 5.10: DeltaGen rank array.

To summarize, the RA has the position in the suffix array of the sub-arrays indexes represented in T, resulting in Expression 5.4.

$$RA[SA[i]] = i \tag{5.4}$$

The Suffix Array and the Rank Array are used to calculate the Height Array (HA). The Height Array contains the Longest Common Prefix (LCP) between the ordered Suffix Array. The LCP between two sub-arrays is the characters match length from the beginning of two arrays.Therefore, the first index of the HA is 0, and the remaining values are the length of the Longest Common Prefix between the current and previous suffix array as represented highlighted as green in Figure 5.11. For example, the suffix array that starts in index 6 ("$bc#") and 3 ("abc$bc#") in T starts by comparing the "$" with "a", which does not equal, so the Height Array value in position 2 is 0. Moreover, the next comparison is the sub-arrays "abc$bc#" with "abcabc$bc#". It will match the "abc" sub-string of length 3. Therefore, the Height Array value in position 3 is 3.

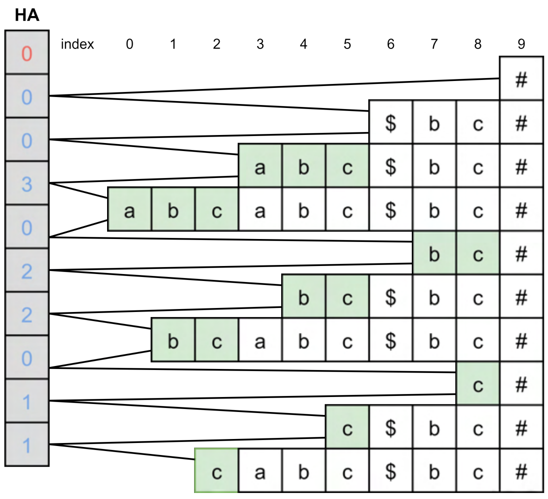


Figure 5.11: DeltaGen height array.

The height value is given by the Equation 5.5

$$Height = \begin{cases} 0, & i = 0 \\ LCP(i-1, i), & i > 0 \end{cases} \tag{5.5}$$

The SA, RA, and HA are used in generating the delta. This delta generation step flowchart is represented in Figure 5.12. It iterates through the new string suffix arrays and finds the Longest Common Prefix (LCP) in the old string. If the LCP value is greater than 0, the COPY command is added with the LCP value to the new string index. On the other hand, when the value equals 0, an ADD command is added with the new string character index and incremented the new string index.

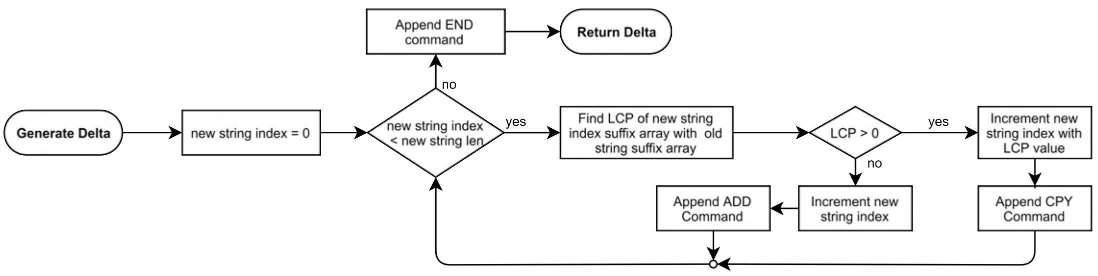

Generate Delta
new string index = 0
new string index < new string len
yes
no
Append END command
Return Delta
Find LCP of new string index suffix array with old string suffix array
LCP > 0
yes
no
Increment new string index with LCP value
Append CPY Command
Increment new string index
Append ADD Command

Figure 5.12: Delta Generation flowchart.

Since the new string “bc” delta only uses a CPY command, the string “bceh” will be used to demonstrate the ADD command. Therefore, the generated delta block output of the new strings “bc” and “bceh” is represented in Figure 5.13. The new string “bc” delta is composed of a CPY and END command. Therefore, to reconstruct the new string, it must copy two bytes from the start in index four of the old string (“abcabc”). On the other hand, the string “bceh” delta has the same copy command as the “bc” string, but it has two more ADD commands to add the character “e” and “h”.

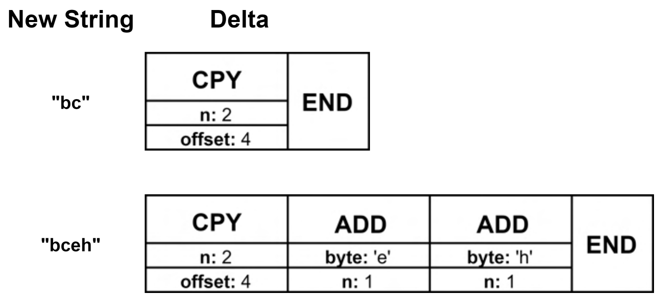


Figure 5.13: Delta output of the new string “bc” and “bceh”.

The delta output from the Generate Delta step enters the Merge ADDs commands, which purpose is to reduce the delta size by merging the ADD commands that are followed. From the example in Figure 5.13, the new string “bceh” has two additional commands with the bytes “e” and “h”. These two ADD commands can be merged into one, resulting in the delta commands shown in Figure 5.14.

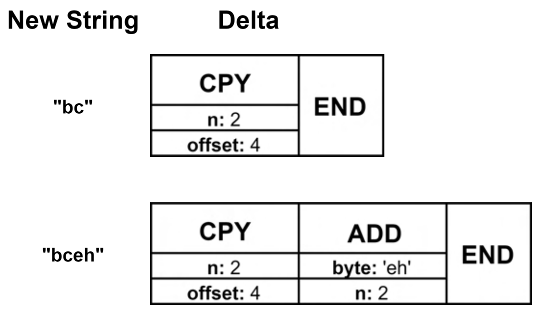


Figure 5.14: Delta output to the new string “bc” and “bceh” with ADD commands merged.

The ending delta is a sequence of commands to reconstruct the new string using the old string. In order to use this algorithm with the real board firmware, in the DeltaGen Join Firmware Arrays step, an offset of 255 to all positions is added. Since the firmware byte values range from 0 to 255, the offset is required because the sentinels’ values need to be less than the array’s values.

The last stage of the delta is the Package Conversion. This stage is used to convert the commands to serialized data that can be received by the end-device applying the command structure represented in Figure 5.5.

Because the serialized data can be greater than the maximum package length that the board communication module can handle. Using the LINK4S project as an example, the modem (Quectel BC66) can only receive 512 bytes at a time. Therefore, the serialized delta needs to be split. A package conversion function will be done to solve this. The parameters for this function will be:

- **Header size** – The length of the first payload package that can be filled with the delta is reduced according to the OTA header size received;
- **Payload size** – Length that can be filled with the delta commands;

The package conversion function will return a list of byte arrays with the respective delta, ready to send to the end-devices. The delta needs to be sent through the cloud to the end-device, and the end-device needs to parse and implement the commands as explained in Sections 5.4.4 and 5.3.10, respectively.

5.3 End-device

After the analysis in section 4, opportunities were found to add features, improve or re-design their architecture. For example, add a new commands module that allows the device to be changed in run-time or a more generic communication approach. In the application re-design will be presented new architectures to increase the modularity, memory optimization, functionality, and flexibility of the application. Also, coding guidelines in variables, functions, enumerations, and structures adapted from the MISRA C specification will be introduced.

Since an RTOS could have several advantages, it was searched for an RTOS that fits the application constraints and then was re-design the modules taking advantage of it. The addition of the RTOS will increase the system modularity without compromising the system lifetime and RAM. Furthermore, the bare-metal conversion to use an RTOS is one of the more extensive application re-design. The author Paiva [1] in her master's thesis used the FreeRTOS resulting in the same lifetime as the bare-metal application but had only 1 KB of RAM left from the 20 KB of the device. Therefore, the RTOS should have the smallest footprint possible and use little RAM. As a result, the Azure RTOS ThreadX was chosen.

5.3.1 Code guidelines

The Motor Industry Software Reliability Association (MISRA) C advise using a specific code specification. It was adapted to use in the end-device codebase to the following. All variables will use the snake case code guidelines, meaning words are separated by underscores, for example, alarm_counter variable. Also, the variables suffix will end with the variable type. For example, if a uint8_t alarm_counter is declared, its name will be alarm_counter_u8. Table 5.1 exemplifies suffixes for several variable types.

Table 5.1: Variable type suffixes.

Variable Type	Suffix
int8_t	_s8
uint8_t	_u8
int32_t	_s32
*int32_t	_ps32
**uint32_t	_ppu32
char	_c
const char	_cc
uint8_t array	_au8

The functions will be using the camel case code guidelines. For example, the send function in the communication will be named communicationSendDatagram. Instead of using the underscore as the snake case separating words, it uses upper case letters. It is helpful to distinguish variables from functions. Also, the functions return will have the return type suffix represented in Tables 5.1, 5.2, and 5.3. These suffixes allow knowing right away the function return in its call.

Table 5.2: Functions type suffixes.

Function Return	Suffix
void	_v
uint8_t	_u8
int32_t	_s32
enum	_e

Specific C types such as enumerations, structs, unions, or any scalar will also have a suffix type in the end, as represented in Figure 5.3. For example, to create the struct alarm, it will be called alarm_st, and if a variable alarm is variable of the type alarm_st is declared, it will be named alarm_s.

Table 5.3: Specific C type suffixes.

C Type	Type Suffix	Variable Suffix
enum	_et	_e
struct	_st	_s
union	_ut	_u
any_scalar	_t	

Since it will be using an RTOS, Table 5.4 represents the RTOS variable type suffixes. For example, if it is a binary semaphore, the variable will have the suffix _bsem.

Table 5.4: Specfic RTOS type suffixes.

RTOS Variable Type	Suffix
Binary Semaphore	_bsem
Counter Semaphore	_csem
Thread	_thread
Mutex	_mux
Event Flags	_eventflags
Event Flag Enumeration	_ef
Queue	_queue

Moreover, if any variable or function is private in the module (static), it starts with an underscore "_". The code guidelines are useful to help develop of software programs and thereby reduce errors. If the coding standards are followed, the code is consistent and easily maintained because anyone can understand it and modify it at any point in time.

5.3.2 Software Stack

A new proposal for the software stack is presented in Figure 5.15. Comparatively to the one in Figure 3.1, the Azure RTOS ThreadX layer above the hardware was added; the Low-Level Portable Layer was renamed to HAL Wrapper, and it encapsulates the Commercial Off-The-Shelf (COTS) HAL; the External Peripherals HAL and the Middleware Layer was added; the security and transmission module will be renamed to Cryptography and Communication respectively.

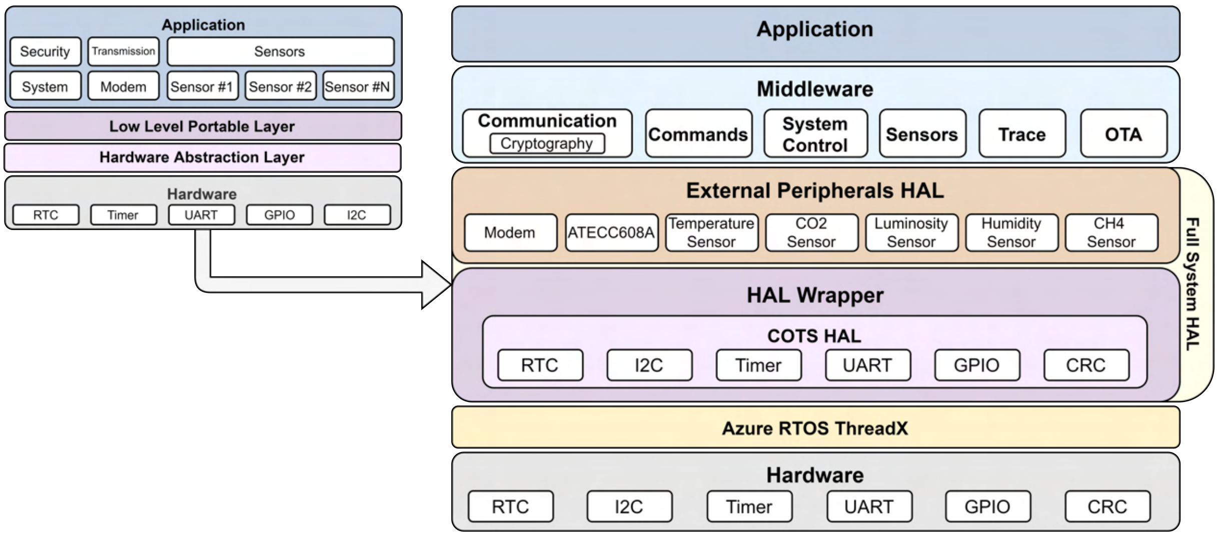


Figure 5.15: Re-designed system software stack.

The HAL Wrapper and the External Peripheral HAL forms the Full System HAL. The HAL Wrapper encapsulates the MCU HAL providing a generic HAL to the application. As a result, changing the MCU would only need reimplementation of this layer. The External Peripherals HAL are the drivers of the peripherals external to the MCU, for example, the Modem, the encryption chip and the multiple sensors.

The Middleware contains the modules that control the lower layers needed by the Application (the layer above). In this layer was added three modules: Commands, Trace, and OTA. The Commands module will be responsible for altering the application parameters such as sensors sampling period and sending to the cloud interval in run-time. The Trace module will be responsible for tracing the application to help in the development phase. The OTA module will be responsible for parsing and handling the firmware updates from the cloud.

The Transmission module was renamed Communication because it will incorporate transmission and reception. It will be re-designed to have a generic transmission datagram encapsulating the datagram from the application. Also, it will be responsible for managing the modem and the cryptography.

The System Control module will be responsible for answering the RTOS low-power modes and refreshing the RTC time. Moreover, the functions needed to the Commands module will be added to alter it.

The sensors module will be re-designed to optimize space and enable run-time parameters changes.

Overall, the improvements to the software stack aim to increase the software modularity and the application design, enabling easier continuous development. The following section will present the Azure RTOS ThreadX low-power mode kernel extension and the Middleware layer specification.

5.3.3 Azure RTOS ThreadX Low-power mode

When all the threads are suspended and the kernel has no thread to execute, it goes to a kernel state that, if activated, calls the assembly instruction Wait for Interrupt (WFI). The problem of only using the WFI instruction is that it does not take advantage of the MCU low-power modes, and the kernel keeps waking up due to the tick interrupt. Therefore, the kernel should enter a low-power mode when all the threads are suspended, disabling the kernel timer tick and using the MCU low-power modes.

This low-power mode kernel extension is represented in Figure 5.16. It starts by searching in all timers and finds the timer ticks from the next timer to expire. Then, it calls a user implemented function (configPreSleepMode) that receives the number of timer ticks knowing how much time it can sleep. The user application sets up a timer to wake up the MCU in the given time and disable the timer giving the kernel tick, for example, the systick. Afterwards, the user-implemented function (sleepMode) is called to enter the MCU low-power state. In low-power mode, the MCU will be awake from the set timer or an interrupt. The kernel calls the configPosSleepMode functions to know how long the system has spent sleeping to adjust its timers.

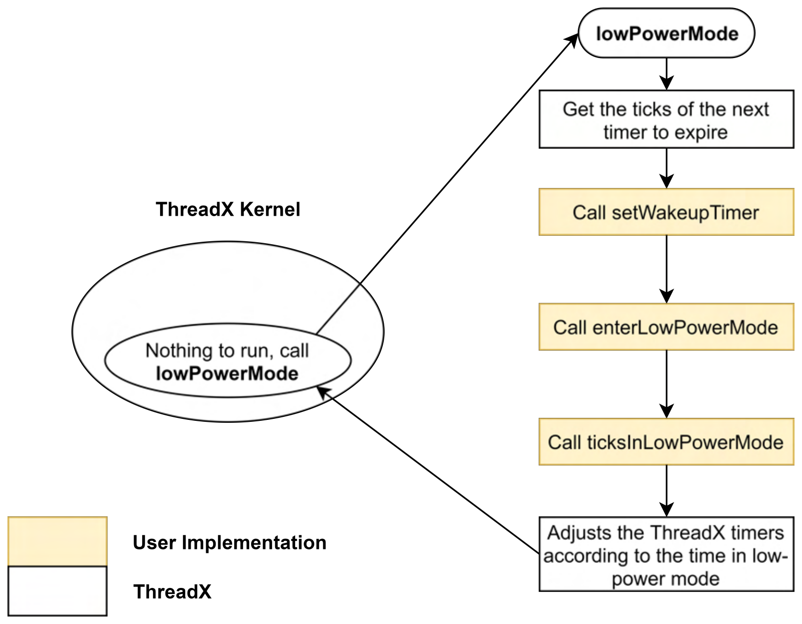

lowPowerMode
Get the ticks of the next timer to expire
Call setWakeupTimer
Call enterLowPowerMode
Call ticksInLowPowerMode
Adjusts the ThreadX timers according to the time in low-power mode
ThreadX Kernel
Nothing to run, call lowPowerMode
User Implementation
ThreadX

Figure 5.16: ThreadX low-power mode extension.

Since the ThreadX has the timer list _tx_timer_list with timer linked lists in each position, searching for the next timer expiration is possible by going through all the linked lists in the timers list comparing the timer ticks.

The timer adjustment process is represented in Figure 5.17. It uses an auxiliary timer list to save all the timers, clear the original, and then subtract the time spent in low-power mode and insert them into the original list.

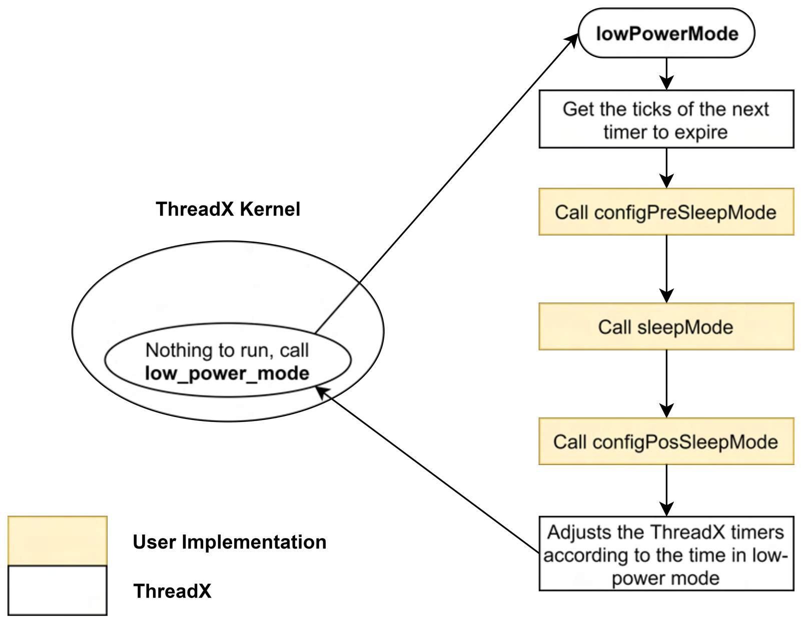

lowPowerMode
Get the ticks of the next timer to expire
Call configPreSleepMode
Call sleepMode
Call configPosSleepMode
Adjusts the ThreadX timers according to the time in low-power mode
ThreadX Kernel
Nothing to run, call low_power_mode
User Implementation
ThreadX

Figure 5.17: ThreadX low-power mode timer adjustments flowchart.

The low-power mode extension of the ThreadX will allow the kernel to enter in the deepest low-power modes of the MCU, saving energy not needing the constant tick if no thread is ready to run.

5.3.4 System Control

The system control provides time capabilities and energy manager controlling the RTC and the MCU power drivers.

It was added the alarm concept. An alarm, represented by the structure in Figure 5.18, is a configured time in a day that will be signalled when it is reached. A new transmission signal architecture will be used in the regular datagram. It has two modes: the time interval and alarms mode. The time interval enables the board to send every configured time. For example, every hour, it sends the regular datagram. The alarms mode lets set up several alarms in a day, and when the time is reached, the regular datagram is sent. For example, setting alarms to 6:30, 12:15, 18:30 will send the datagram at 6:30, 12:15, and 18:30, respectively. The alarms structure has the time_interval to indicate the mode in use, the respective arrays of time, and the respective index in use.

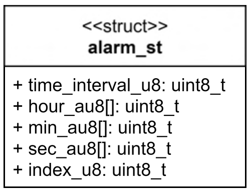


Figure 5.18: System control alarm structure.

The new alarm architecture allows being configured using cloud commands. For this purpose was added the following commands:

- Set/Get alarms – the set receives the alarms mode and, according to it, gets the regular send interval or the send alarms. The get alarm sends the board alarms and mode to the cloud;
- Set/get RTC refresh time – sets the frequency of updating the RTC clock or returns the current period;

The system control class diagram is represented in Figure 5.19. The **systemControlTXAppDefine** function purpose is to create the system control module's ThreadX services (threads, semaphores). The **systemControlInitialize** sets the MCU low-power modes and the default system control settings. The function **systemControlSetCurrentDatetime** will be used to set the RTC time and date. The **system-ControlGetUnixEpochTimestamp** returns the timestamp in the UNIX format. Finally, the remaining functions are used to activate, set, and get the alarms.

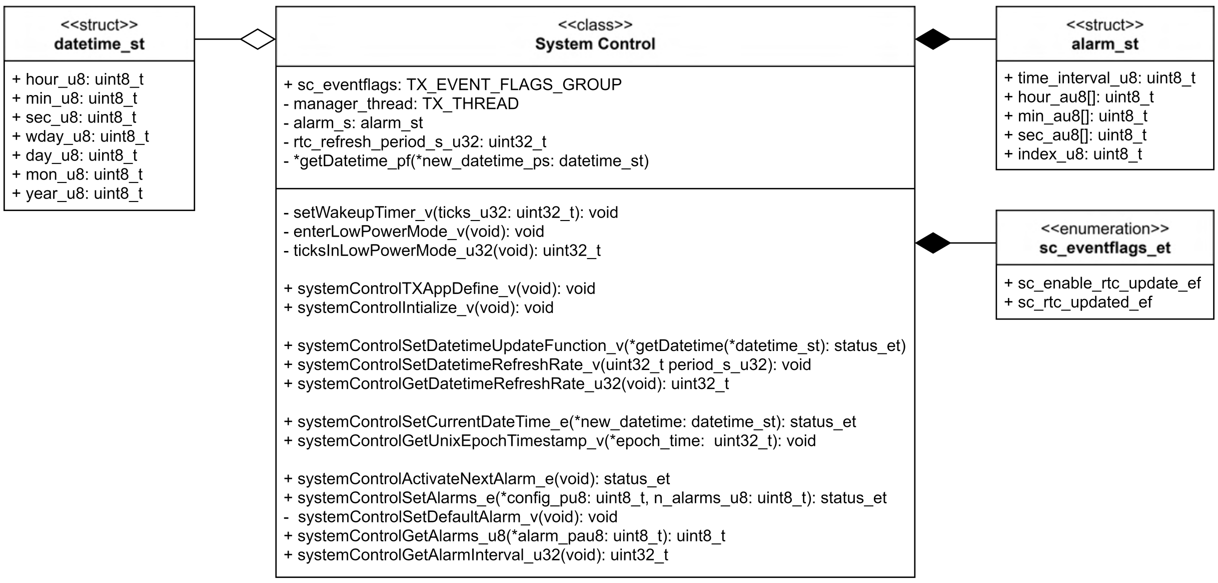

<<struct>>
datetime_st
+ hour_u8: uint8_t
+ min_u8: uint8_t
+ sec_u8: uint8_t
+ wday_u8: uint8_t
+ day_u8: uint8_t
+ mon_u8: uint8_t
+ year_u8: uint8_t
<<class>>
System Control
+ sc_eventflags: TX_EVENT_FLAGS_GROUP
- manager_thread: TX_THREAD
- alarm_s: alarm_st
- rtc_refresh_period_s_u32: uint32_t
- *getDatetime_pf(*new_datetime_ps: datetime_st)
- setWakeupTimer_v(ticks_u32: uint32_t): void
- enterLowPowerMode_v(void): void
- ticksInLowPowerMode_u32(void): uint32_t
+ systemControlTXAppDefine_v(void): void
+ systemControlIntialize_v(void): void
+ systemControlSetDatetimeUpdateFunction_v(*getDatetime(*datetime_st): status_et)
+ systemControlSetDatetimeRefreshRate_v(uint32_t period_s_u32): void
+ systemControlGetDatetimeRefreshRate_u32(void): uint32_t
+ systemControlSetCurrentDateTime_e(*new_datetime: datetime_st): status_et
+ systemControlGetUnixEpochTimestamp_v(*epoch_time: uint32_t): void
+ systemControlActivateNextAlarm_e(void): status_et
+ systemControlSetAlarms_e(*config_pu8: uint8_t, n_alarms_u8: uint8_t): status_et
- systemControlSetDefaultAlarm_v(void): void
+ systemControlGetAlarms_u8(*alarm_pau8: uint8_t): uint8_t
+ systemControlGetAlarmInterval_u32(void): uint32_t
<<struct>>
alarm_st
+ time_interval_u8: uint8_t
+ hour_au8[]: uint8_t
+ min_au8[]: uint8_t
+ sec_au8[]: uint8_t
+ index_u8: uint8_t
<<enumeration>>
sc_eventflags_et
+ sc_enable_rtc_update_ef
+ sc_rtc_updated_ef

Figure 5.19: System control class diagram.

A thread will update the RTC periodically due to the RTC shift from real-time. This thread's flowchart is represented in Figure 5.20. It is a loop that starts by checking if the RTC update event flag is enabled. If it is, it will read the alarms set interval, and call the getDatetime function pointer. If it is successful, set the alarms, the RTC with the new DateTime, and sleeps the RTC refresh time. In case of error, it triggers the get DateTime error, waits for the error to get fixed and goes to the getDatetime function call.

This module will have a pointer to a function that receives a DateTime struct and gets the current date and time (systemControlSetDatetimeFunction). This thread will be controlled by setting or clearing the sc_enable_rtc_update_ef event flag, and the system controls signals when the RTC is updated by setting the sc_rtc_updated_ef event flag.

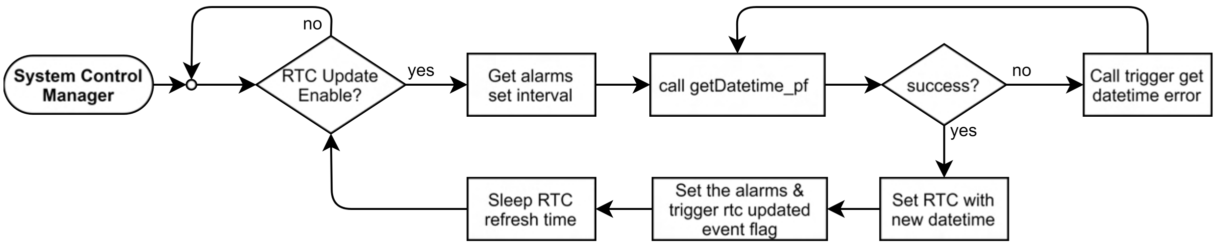


Figure 5.20: System control manager thread's flowchart.

The system control is also responsible for enabling the RTOS low-power modes, receiving the time that the system can sleep and returning to the RTOS kernel the time spent sleeping in order to the RTOS recover from it.

The RTC wake-up mode will be used as the wake-up timer with a resolution of milliseconds up to 36 hours. The sleepMode will be done by calling the enterStopMode function in HAL Wrapper. The system's time in low-power mode will be calculated by subtracting the timestamp reading in the wake-up timer set and the configPosSleepMode function.

5.3.5 Sensors

A new sensor architecture is presented that allows better control of the sensors by allowing run-time commands and optimizing variable storage and variable granularity control. This new architecture structure is represented in Figure 5.21 and provides easy addition of a sensor to the system as creating its sensor struct and variables structs and sample arrays. The sensor struct has the generic function pointers to the initialization, configuration, sample and handles its interrupt. The pointers to functions allow creating generic algorithms in the sensors module.

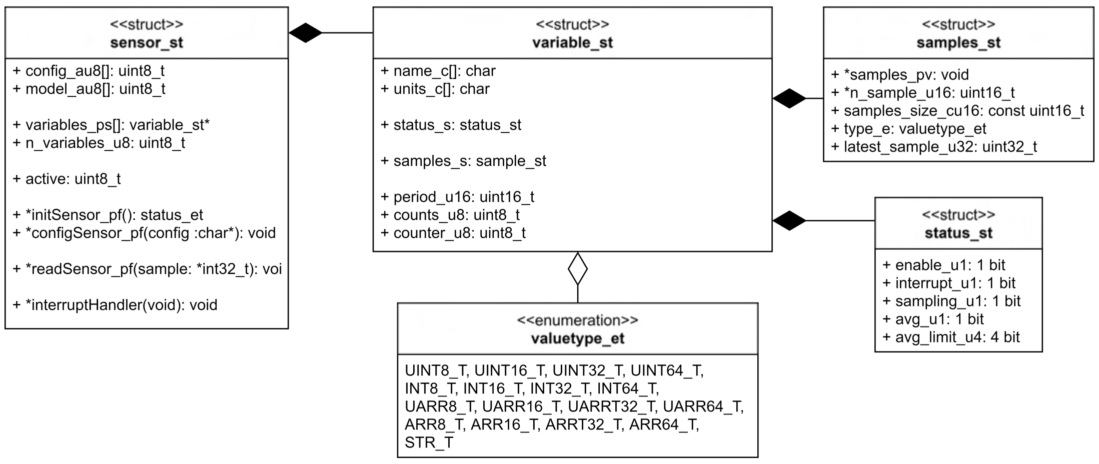


Figure 5.21: Sensor and variable structures class diagrams.

The variable struct has the status struct that contains the current status information of the variable, which can be changed through cloud commands. Enabling the possibility of changing the sampling, interrupt status and change the sampling mode, including its average limit. Using the HDC2080 sensor as an example, it can measure two physical variables, temperature and humidity. Therefore, this sensor will have two variables (temperature and humidity). A variable will be composed by name, units, a sampling period, counts, counter samples, and status. Figure 5.21 represents the new sensors structures.

The samples are encapsulated in the struct samples_st. This struct has sample index, sample array size, sample array type, the latest sample, and a generic pointer to the samples array. The latter generic pointer allows allocating only the bytes needed for each variable. For example, if a sample is 8-bits long with the old storage architecture using the datavalue_t structure, it was allocated for the minimum, maximum and average two 32-bits for every sample (sampling and sampling type). Using the array, if it stores 32 samples, it is only to be used 32 bytes. Equally, it is compatible with the simple and average sample mode. This new way of saving the variables allows different sample sizes, not constraining into one.

The sensors module will have three threads:

- **Sensors Manager** – Responsible for sensors' initialization, configuration and everything that requires managing the sensors;
- **Sample Variables** – Sample the variables if the sampling is enabled and if the variable counts reach the counter value;
- **Emergency Manager** – Handle the physical emergencies triggered by the sensors;

Figure 5.22 shows the sensors manager thread's flowchart. It waits for the sensors' init event flag to be set. When set, it initializes the sensors, and if any are alive, it configures them and signals that the sensors were initialized through the initialized event flag. If no sensors are "alive", it will sleep for the "no sensors" timeout value.

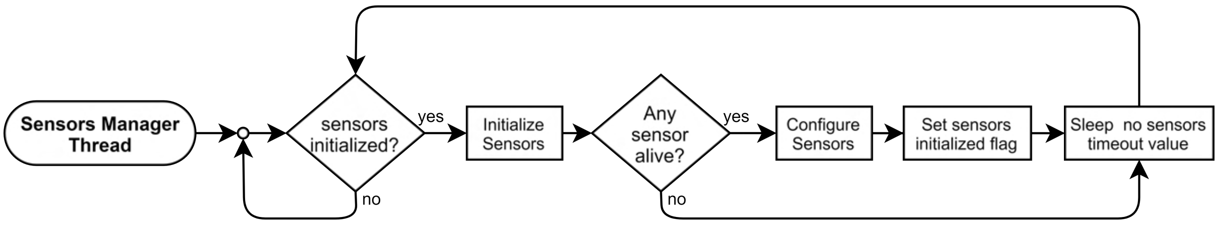


Figure 5.22: Sensors manager thread's flowchart.

The sample variables thread is represented in Figure 5.23. If the sensors were initialized and the sampling enabled. It increases the variables counts and checks if they equal the counter. When it does, it samples the variables. When all the sensors' variables were iterated, it sleeps for the minimum common sampling values between variables periods.

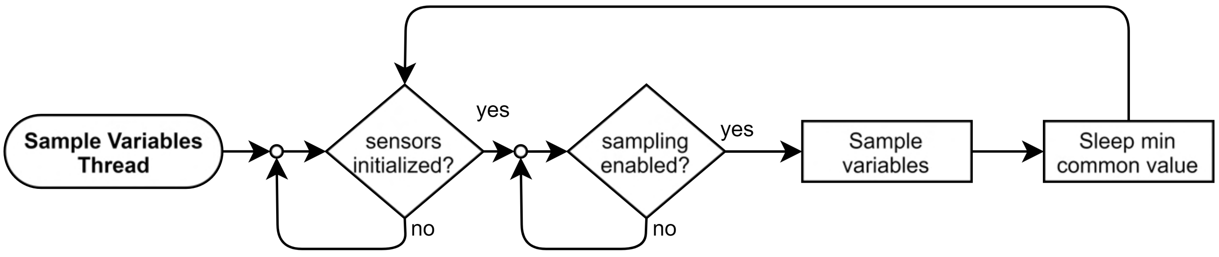


Figure 5.23: Sample variables thread's flowchart.

The emergency thread is illustrated in the flowchart of Figure 5.24. If the sensors are initialized, it waits for external interrupts signalled by HAL Wrapper event flags having. When an external interrupt from the sensors is triggered, it handles the sensor. If an emergency was enabled and triggered, the application is signalled that an event flag that an emergency occurred. Moreover, the application can read it through the `getSensorsEmergencys_v` function.

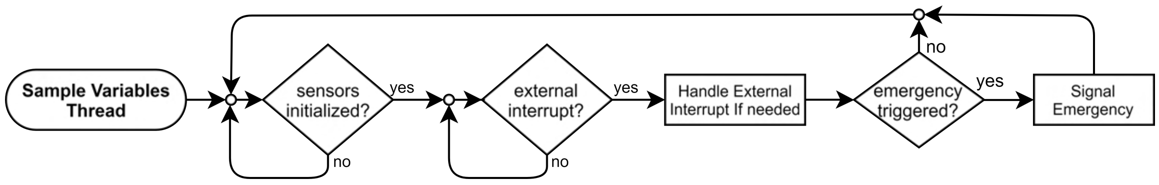
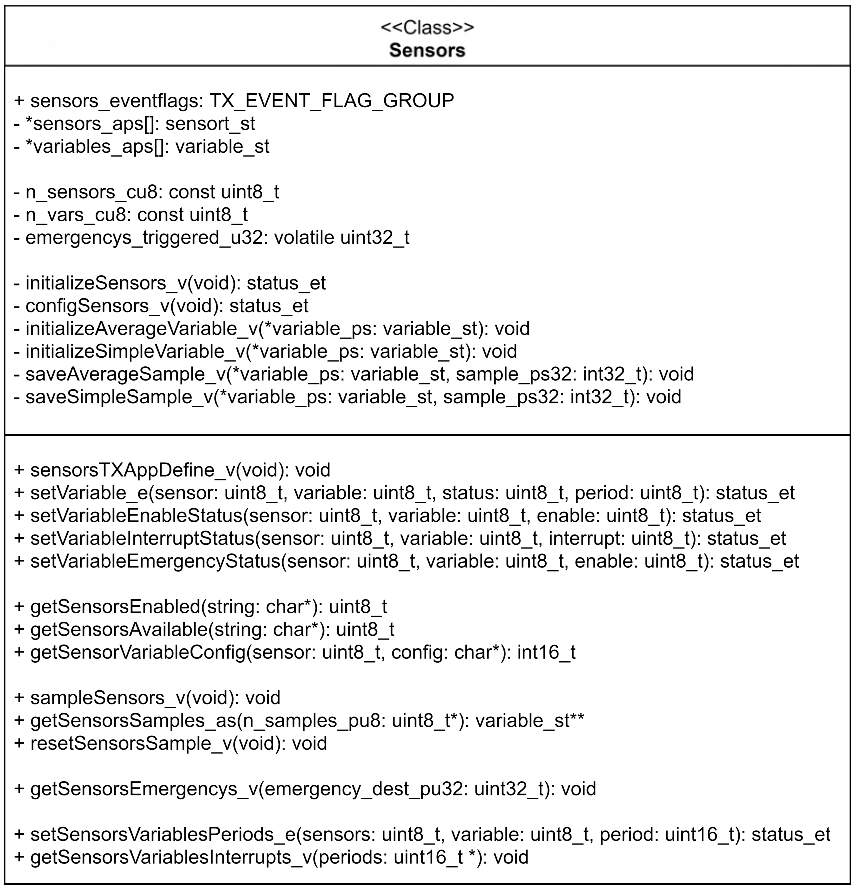


Figure 5.24: Emergency thread's flowchart.

The resulting sensors module interface is represented in Figure 5.25. External functions have the functions needed to enable the commands to set and get the variables status as the interrupt, sampling, operation mode, and period. Also, it has functions to sample and get the sampled values. The module has the sensors_eventflags used to enable or disable the sample and trigger the sensors' initialization. Internally, the module has multiple arrays of pointers to structs to manage all sensors, variables, and samples structs more easily.


<<Class>>
Sensors

+ sensors_eventflags: TX_EVENT_FLAG_GROUP
- *sensors_aps[]: sensort_st
- *variables_aps[]: variable_st

- n_sensors_cu8: const uint8_t
- n_vars_cu8: const uint8_t
- emergencys_triggered_u32: volatile uint32_t

- initializeSensors_v(void): status_et
- configSensors_v(void): status_et
- initializeAverageVariable_v(*variable_ps: variable_st): void
- initializeSimpleVariable_v(*variable_ps: variable_st): void
- saveAverageSample_v(*variable_ps: variable_st, sample_ps32: int32_t): void
- saveSimpleSample_v(*variable_ps: variable_st, sample_ps32: int32_t): void

+ sensorsTXAppDefine_v(void): void
+ setVariable_e(sensor: uint8_t, variable: uint8_t, status: uint8_t, period: uint8_t): status_et
+ setVariableEnableStatus(sensor: uint8_t, variable: uint8_t, enable: uint8_t): status_et
+ setVariableInterruptStatus(sensor: uint8_t, variable: uint8_t, interrupt: uint8_t): status_et
+ setVariableEmergencyStatus(sensor: uint8_t, variable: uint8_t, enable: uint8_t): status_et

+ getSensorsEnabled(string: char*): uint8_t
+ getSensorsAvailable(string: char*): uint8_t
+ getSensorVariableConfig(sensor: uint8_t, config: char*): int16_t

+ sampleSensors_v(void): void
+ getSensorsSamples_as(n_samples_pu8: uint8_t*): variable_st**
+ resetSensorsSample_v(void): void

+ getSensorsEmergencys_v(emergency_dest_pu32: uint32_t): void

+ setSensorsVariablesPeriods_e(sensors: uint8_t, variable: uint8_t, period: uint16_t): status_et
+ getSensorsVariablesInterrupts_v(periods: uint16_t *): void

Figure 5.25: Sensors class diagram.

5.3.6 Cryptography

Since the communication protocol with the cloud server uses the AES and the RSA encryption, the cryptography module provides the mechanisms for this type of encryption. Also, it offers a simple API to the MD5 hash algorithm.

The module class diagram is represented in Figure 5.26. Hardware-wise the platform contains the ATECC608A and the MCU encryption engines to implement the AES encryption. The RSA algorithm that will be used is the PKCS #1 v1.5, and it does not have hardware acceleration. Therefore, it will be implemented in software with the encryption library available from STMicroelectronics. Furthermore, the MD5 will be implemented recurring to the encryption library and an open-source project adapted implementation.

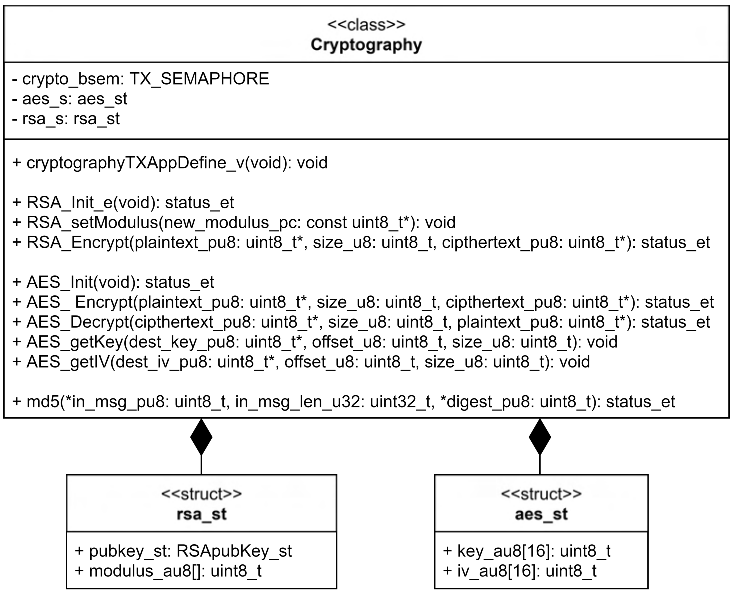

<<class>>
Cryptography
- crypto_bsem: TX_SEMAPHORE
- aes_s: aes_st
- rsa_s: rsa_st
+ cryptographyTXAppDefine_v(void): void
+ RSA_Init_e(void): status_et
+ RSA_setModulus(new_modulus_pc: const uint8_t*): void
+ RSA_Encrypt(plaintext_pu8: uint8_t*, size_u8: uint8_t, cipthertext_pu8: uint8_t*): status_et
+ AES_Init(void): status_et
+ AES_ Encrypt(plaintext_pu8: uint8_t*, size_u8: uint8_t, cipthertext_pu8: uint8_t*): status_et
+ AES_Decrypt(cipthertext_pu8: uint8_t*, size_u8: uint8_t, plaintext_pu8: uint8_t*): status_et
+ AES_getKey(dest_key_pu8: uint8_t*, offset_u8: uint8_t, size_u8: uint8_t): void
+ AES_getIV(dest_iv_pu8: uint8_t*, offset_u8: uint8_t, size_u8: uint8_t): void
+ md5(*in_msg_pu8: uint8_t, in_msg_len_u32: uint32_t, *digest_pu8: uint8_t): status_et
<<struct>>
rsa_st
+ pubkey_st: RSApubKey_st
+ modulus_au8[]: uint8_t
<<struct>>
aes_st
+ key_au8[16]: uint8_t
+ iv_au8[16]: uint8_t

Figure 5.26: Cryptography class diagram.

The cryptography interface has the application define function that creates the binary semaphore (crypto_bsem) to protect the concurrent use of the encryption. Then, there are functions to the RSA and the AES, both initialization and encryption. The RSA has the modulus set, and the AES decrypt, get the key, and Initial Vector (IV) functions. Internally, the module will have the aes_s and rsa_s structures that save the data needed for the algorithms. Also, it will be possible to choose the AES encryption be implemented by the ATEC and the MCU encryption engines by a macro in compile time to be easier to compare them.

5.3.7 Communication

The Communication will be responsible for the transmission and reception of data to and from the cloud. It will have a new communication architecture that adds a generic datagram structuring and command reception. Also, it will be responsible for handling the asynchronous modem events and network connection manager.

Transmission

The cloud reception datagram protocol is explained in Section 5.4. The payload is the message sent by the end-devices. It is serialized using message pack. In the old application, the sensors sample was constrained by the amount of data the communication could send in the interval between transmissions and the buffers that allocated the samples—forcing the application to have fixed low sampling periods. Also, the communication had predefined structures that were shared with the application. If a new parameter to a type of transmission is added, or even if a new type of transmission is necessary, it would be necessary to make the new construct of the datagram in the communication module.

Additionally, in the old transmission module, the cloud received a stream of data that is not identified, consisting of numbers and or strings. If it goes out of order or has more than expected, the cloud could not parse it. Therefore, adding a new parameter also will change the cloud parsing. As a solution, a new communication architecture will be used. The datagram is composed of datagram objects, as illustrated in Figure 5.27. Since the transmission is generic, adding a new object to a datagram type does not require the re-implementation of communication module or cloud microservices.

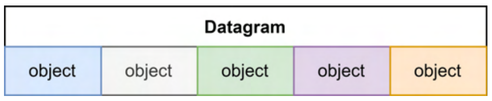


Figure 5.27: New communication architecture datagram.

Figure 5.28 illustrates the transmission of a regular datagram. Each datagram object will be composed of a string identifier and the respective data. For example, the board id identifier is “ID” and is followed by the respective value. The temperature variable has the identifier “sTMP”, which indicates that it is in simple mode followed by its samples.

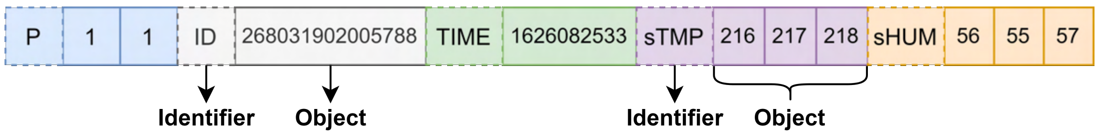


Figure 5.28: New communication architecture datagram.

All the transmission datagrams contain in the start the parts object identified by “P” in Figure 5.28. This object data is the current part and the total datagram parts. If the payload does not fit in the modem transmission limit of 1024 bytes because it is 1500 bytes long, it will send in two modem transmissions. The first datagram will have the object “P” with the data 1, 2 indicating that it is the first package of 2. When it is in the last transmission, it will be sent with 2,2, indicating that the message is the current part 2, with the total parts of 2. Moreover, to identify the datagram is useful to have the board id and timestamp in every transmission. It will be accomplished by passing the number two to the communicationSendDatagram repetition to repeat the first two objects (board id and timestamp).

The datagram objects can be encapsulated in the communication module giving the following generic interface to the application:

- **communicationSetupNewTransmission** – It prepares everything to start a new transmission;
- **communicationSetupMicroService** – Sets the server microservice identifier, if it has one;
- **communicationAppendDatagramObject** – Receives the information needed to fill a datagram object and append it to the datagram;
- **communicationSendDatagram** – Receives the number of objects that will be repeated in case of the datagram exceeds the max transmission length and sends the datagram objects to the cloud;
- **communicationEndTransmission** – Finishes the transmission, giving the possibility for the start of another;

Moreover, since the header does not change, it will enable static encryption by changing macros in compile-time. The static encryption consists of the microservices header being already encrypted and saved in the FLASH memory. Therefore, when filling the cloud header, it will just be copied.

Reception

The possibility to exchange commands adds the need to receive the message from the cloud. Therefore, it needs to handle the reception of data and other asynchronous events such as the server closing the communication socket. The communication will have the **Communication Asynchronous Events** thread, represented in Figure 5.29. It will wait for asynchronous modem requests and will handle them.

An Unsolicited Result Code (URC) is a modem message not requested from the MCU. It can be the modem signalling that received cloud data or the cloud closed the socket. If the server closes the socket, the MCU has to close the modem socket.

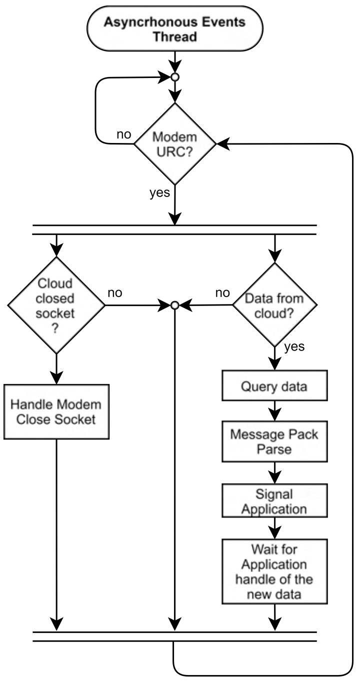


Figure 5.29: Communication Asynchronous Events Thread's flowchart.

When receiving data from the cloud, it is queried, parsed, signalled the application, and it will wait for the application to handle the new data to have the buffers free to get new data from the modem. The cloud data will follow the structure represented in Figure 5.32, it has, in the beginning, the Message-Digest algorithm 5 (MD5). The MD5 [89] is a message-digest algorithm used as a hash function producing a 128-bit value hash value. Since it is a fast hash algorithm, the data received has an MD5 checksum to verify the data integrity, adding safety to the data received.

The communication will use the datagram_t struct to receive the commands. According to the message pack objects received, it will readjust the pointers inside the datagramobject_st. . For example, if the communication receives a byte array and a 32-bits unsigned integer, the datagram_st struct will have two objects: the first one will have the type UARR8_T, array length and pointing to the start o the array; the second one will have the type UINT32_T and a pointer to the unsigned integer.

The communication will give the application an API to interact with the data received. It will signal an event flag when data is received, and then the application can use the following functions:

- **communicationSetupNewReception** – Gets the datagram thread synchronization service;
- **communicationReceiveSize** – Returns the number of objects in the datagram struct;
- **communicationGetReceivedObject** – Receives the index of the datagram object and the pointers that will be set to enable access to it;

- **communicationEndReception** – Frees the datagram struct allowing it to be used for transmission or another reception;

The buffer used in the transmission will be used in the reception, saving RAM but, in contrast, can not transmit and receive simultaneously.

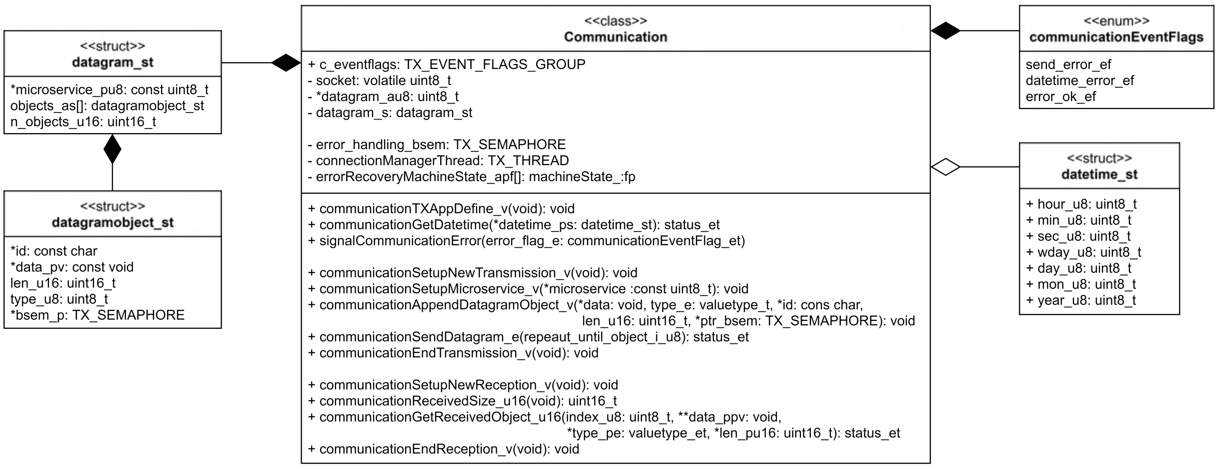

<<struct>>
datagram_st
*microservice_pu8: const uint8_t
objects_as[]: datagramobject_st
n_objects_u16: uint16_t
<<struct>>
datagramobject_st
*id: const char
*data_pv: const void
len_u16: uint16_t
type_u8: uint8_t
*bsem_p: TX_SEMAPHORE
<<class>>
Communication
+ c_eventflags: TX_EVENT_FLAGS_GROUP
- socket: volatile uint8_t
- *datagram_au8: uint8_t
- datagram_s: datagram_st
- error_handling_bsem: TX_SEMAPHORE
- connectionManagerThread: TX_THREAD
- errorRecoveryMachineState_apf[]: machineState_:fp
+ communicationTXAppDefine_v(void): void
+ communicationGetDatetime(*datetime_ps: datetime_st): status_et
+ signalCommunicationError(error_flag_e: communicationEventFlag_et)
+ communicationSetupNewTransmission_v(void): void
+ communicationSetupMicroservice_v(*microservice :const uint8_t): void
+ communicationAppendDatagramObject_v(*data: void, type_e: valuetype_t, *id: cons char, len_u16: uint16_t, *ptr_bsem: TX_SEMAPHORE): void
+ communicationSendDatagram_e(repeaut_until_object_i_u8): status_et
+ communicationEndTransmission_v(void): void
+ communicationSetupNewReception_v(void): void
+ communicationReceivedSize_u16(void): uint16_t
+ communicationGetReceivedObject_u16(index_u8: uint8_t, **data_ppv: void, *type_pe: valuetype_et, *len_pu16: uint16_t): status_et
+ communicationEndReception_v(void): void
<<enum>>
communicationEventFlags
send_error_ef
datetime_error_ef
error_ok_ef
<<struct>>
datetime_st
+ hour_u8: uint8_t
+ min_u8: uint8_t
+ sec_u8: uint8_t
+ wday_u8: uint8_t
+ day_u8: uint8_t
+ mon_u8: uint8_t
+ year_u8: uint8_t

Figure 5.30: New communication class diagram.

From the class diagram of Figure 5.30, the **communicationTXAppDefine** is used to create the ThreadX variables in the kernel initialization. The **communicationGetDatetime** purpose is to set the DateTime struct with the current time if it was successfully got it. The **signalCommunicationError** is used to signal an error in the communication to resolve it.

Connection Manager

The communication will have the connection manager thread. It will be responsible for the modem initialization, configuration, and server connection, including its error handling. This thread will be implemented as the state machine in Figure 5.31. It starts by default and has six states:

- **Initialization** – Checks if the modem is alive, and if it is not, it tries to recover it;
- **Configuration** – Does the modem need configurations and enables the modem connection to the network if needed and waits until it is connected;
- **Check Network Connection** – Checks if the modem is connected to the network and set the after network connection modem low-power savings configurations;
- **Check Server Connection** – Pings the server checking if it can establish a connection with it, meaning that it is alive;

- **Open Socket** – If no socket open, it forms a socket connection with the server;
- **Wait for error** – Waits for a signal of error in communication;

During execution, a failed state will rollback to the previous successful state. Multiple failures will force the MCU to a sleep state and try to solve the error later. Upon a predefined number of failures, it will go back to a previously successful state to guarantee that the error did not propagate. For example, the send error event flag is triggered if the transmission fails and the state machine goes to the Open Socket state. If it cannot open a socket with the server, it goes to the previous state and checks if it has a server connection; if it is not successful in pinging the server, it goes to the state before pinging the server, checking if the modem is connected to the network. If the modem is connected to the network, it will go to the Check Server Connection state. If it is not successful, it means that the server is down. Therefore, it will sleep for a specific time and will try again. When the tries increase, the sleeping time will increase too. After a predefined number of unsuccessful attempts, it will go to the Check Network Connection state to check if the modem did not lose connection when sleeping. When the server is available again, the Check Server Connection state will be able to ping the server, then in the Open Socket state, a socket will be open and will be signal that the error is fixed, and in the case of the send error, it will send the datagram again. The connection manager thread will be again in the Wait for error state, waiting for an error to occur. If the modem goes down, the Initialization state will try to recover it by resetting it and initializing.

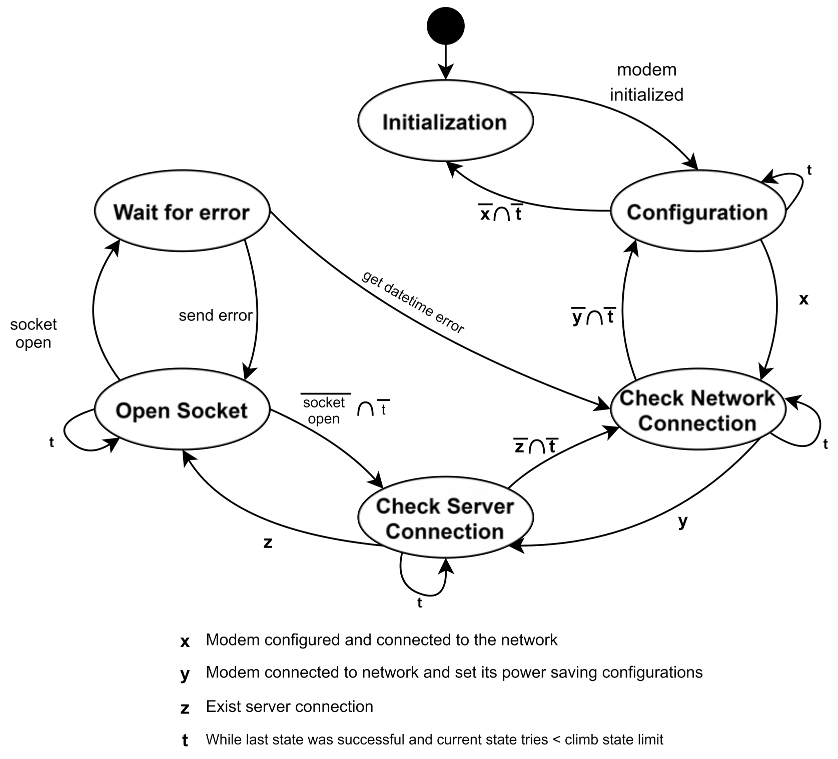

Initialization
modem initialized
Configuration
t
$\overline{x}\cap\overline{t}$
x
$\overline{y}\cap\overline{t}$
Wait for error
get datetime error
socket open
send error
Open Socket
t
$\overline{\text{socket open}}\cap\overline{t}$
Check Network Connection
t
$\overline{z}\cap\overline{t}$
Check Server Connection
t
z
y
x Modem configured and connected to the network
y Modem connected to network and set its power saving configurations
z Exist server connection
t While last state was successful and current state tries < climb state limit

Figure 5.31: State machine of the connection manager thread.

5.3.8 Commands

The Commands module allows the user to dynamically change the system and interact with it through specific messages. Hence, firstly, the board parameters that should be changed or gotten were identified and resumed in Table 5.5. Then the commands architecture.

The device can only receive data from the cloud when it transmits. After the device transmission, the server has a small window to send a message back. Therefore, the commands will be sent to the device when the device transmits data to the cloud, such as when a regular configuration, emergency, and command response datagram are sent.

Table 5.5: Commands according to the module.

Module	Command	Opcode	Description
System Control	Set/get alarms	0	Set or get the board alarms
System Control	Set/get standby	1	Set or get the standby time
System Control	Set/get RTC refresh time	2	Set or get the system control manager thread's period
Sensors	Set/get sensors variables config	3	Set or get sensors variables configurations
Sensors	Set sensors variables enable status	4	Set or get sensors variables configurations
Sensors	Set sensors variables periods	5	Set or get sensors variables periods
Sensors	Set/get sensors variables interrupt thresholds	6	Sets or gets the variable interrupt thresholds (manual and automatic)
Application	OTA Update	7	Receives a chunk of data of the new application and saves it in the FLASH memory
Application	Get software version	8	Gets the board software version
Application	Set/get EEPROM	9	Writes or reads from the EEPROM

The commands datagram reception architecture is represented in Figure 5.32. They have an opcode associated, and new commands can be added by adding new opcodes. The set commands after the opcode have the new data to set the respective command parameters. The get command is a set command with the Most Significant Bit (MSB) set.

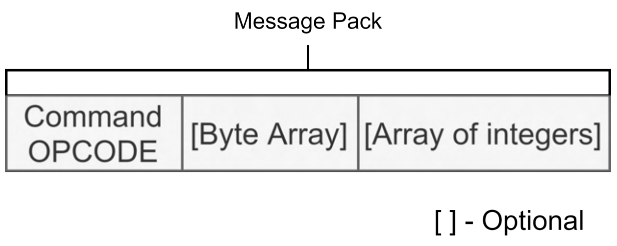


Figure 5.32: Commands datagram reception architecture.

Figure 5.33 represents the command manager thread. This thread waits for the data from the cloud received event flag. Then it gets the command opcode, and if it is valid, verify if it is a set command. If it is, it handles it according to its opcode by calling its function. A set command has a value ranging from 0 to the length of the commands opcodes, and a get command has the same opcode as the set command, but the bit 16 is set. If it is a valid command, it will call the get command function, append the set to the transmission, and get command results as the board id and current timestamp. The data is ready to be sent, so it set the command response event flag and waits for the command response to be sent. It returns to the beginning, waiting for the cloud data event flag.

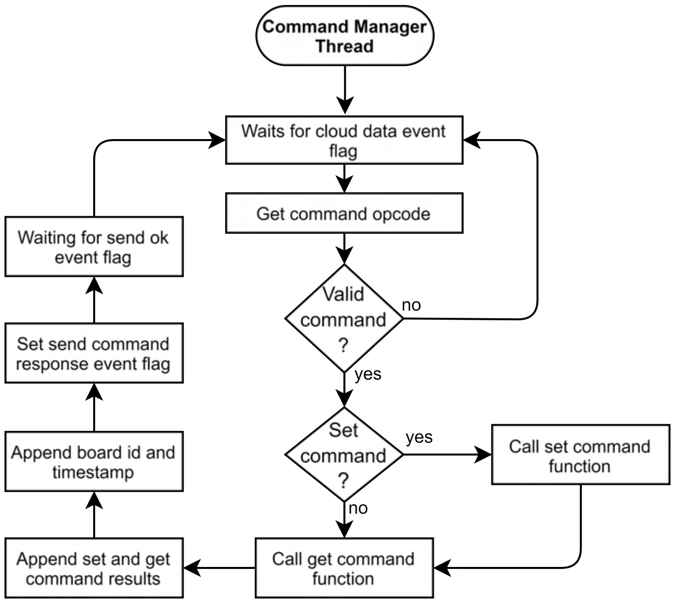
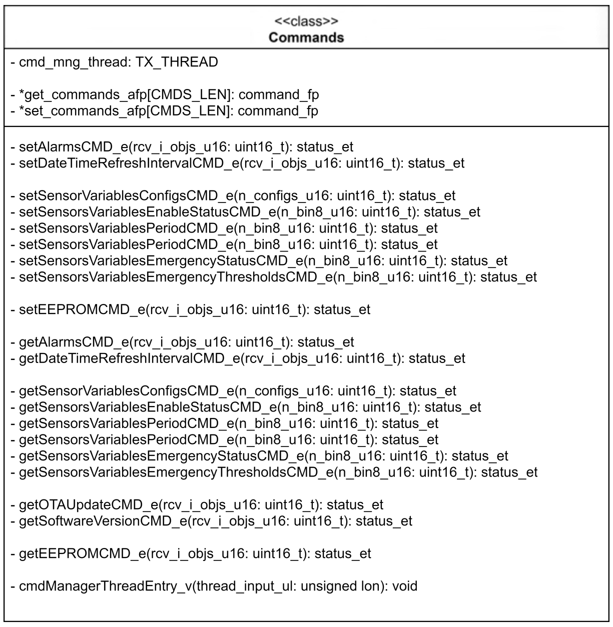

Command Manager Thread
Waits for cloud data event flag
Get command opcode
Valid command ?
no
yes
Set command ?
yes
Call set command function
no
Call get command function
Append set and get command results
Append board id and timestamp
Set send command response event flag
Waiting for send ok event flag

Figure 5.33: Commands manager thread's flowchart.

The commands class diagram is represented in Figure 5.34. It is composed of all the command functions to implement the commands. Notice that the set OTA and software version are not defined because these commands do not have a set functionality. It will return false in the command result instead.



```
<<class>>
Commands

- cmd_mng_thread: TX_THREAD

- *get_commands_afp[CMDS_LEN]: command_fp
- *set_commands_afp[CMDS_LEN]: command_fp

- setAlarmsCMD_e(rcv_i_objs_u16: uint16_t): status_et
- setDateTimeRefreshIntervalCMD_e(rcv_i_objs_u16: uint16_t): status_et

- setSensorVariablesConfigsCMD_e(n_configs_u16: uint16_t): status_et
- setSensorsVariablesEnableStatusCMD_e(n_bin8_u16: uint16_t): status_et
- setSensorsVariablesPeriodCMD_e(n_bin8_u16: uint16_t): status_et
- setSensorsVariablesPeriodCMD_e(n_bin8_u16: uint16_t): status_et
- setSensorsVariablesEmergencyStatusCMD_e(n_bin8_u16: uint16_t): status_et
- setSensorsVariablesEmergencyThresholdsCMD_e(n_bin8_u16: uint16_t): status_et

- setEEPROMCMD_e(rcv_i_objs_u16: uint16_t): status_et

- getAlarmsCMD_e(rcv_i_objs_u16: uint16_t): status_et
- getDateTimeRefreshIntervalCMD_e(rcv_i_objs_u16: uint16_t): status_et

- getSensorVariablesConfigsCMD_e(n_configs_u16: uint16_t): status_et
- getSensorsVariablesEnableStatusCMD_e(n_bin8_u16: uint16_t): status_et
- getSensorsVariablesPeriodCMD_e(n_bin8_u16: uint16_t): status_et
- getSensorsVariablesPeriodCMD_e(n_bin8_u16: uint16_t): status_et
- getSensorsVariablesEmergencyStatusCMD_e(n_bin8_u16: uint16_t): status_et
- getSensorsVariablesEmergencyThresholdsCMD_e(n_bin8_u16: uint16_t): status_et

- getOTAUpdateCMD_e(rcv_i_objs_u16: uint16_t): status_et
- getSoftwareVersionCMD_e(rcv_i_objs_u16: uint16_t): status_et

- getEEPROMCMD_e(rcv_i_objs_u16: uint16_t): status_et

- cmdManagerThreadEntry_v(thread_input_ul: unsigned lon): void
```

Figure 5.34: Commands class diagram.

The command module has the commands thread and two arrays of pointers to functions called set and get commands. The command opcode is the correspondent index in these arrays.

5.3.9 Trace

The Trace module aims to provide a debug entity that allows faster problem solving during the development phase. It will trace the thread stack's free space using a chunk of the EEPROM to save its data enabling the user to read with the MCU programmer after extensive tests.

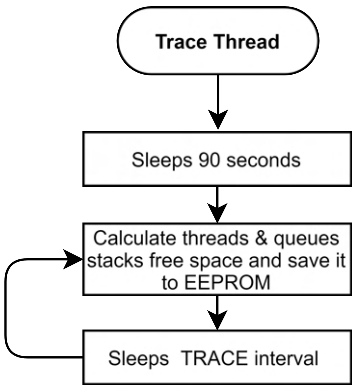

Trace Thread
Sleeps 90 seconds
Calculate threads & queues stacks free space and save it to EEPROM
Sleeps TRACE interval

Figure 5.35: Trace thread's flowchart.

It will have the trace thread, represented in Figure 5.35. It starts by sleeping for 90 seconds, giving time to the user can connect the MCU programmer and not overwrite the old data. Then it enters a loop that starts by calculating the threads and queue stack free space, saves it to EEPROM, and sleeps for a compile-time value. Also, the ThreadX can be be compiled with stack checking. The stack checking calls a callback function when a thread stack overflows. This callback will be implemented in the Trace module. It receives a pointer to the overflowed thread, and with that will be saved in the EEPROM the thread's name.

Moreover, the trace module has the log space used to save objects to the EEPROM. This object's structure represented in Figure 5.36 is composed of an identifier, and according to it, it is of different types. The types will be for strings, and binary semaphore gets services, but it can scale by increasing the identifier number. The string object is composed of the identifier described previously and the respective string. This string ends with the character "
0". The binary semaphore gets the object as the string has its identifier, followed by the service variable address and the called line number and module name.

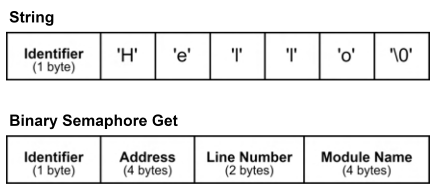


Figure 5.36: Trace log file objects structure.

5.3.10 OTA

In order to update the device Over-The-Air, the end-device receives the delta and handles it, reconstructing the new firmware through the old firmware. Therefore, this section will show how the device memory is prepared to tackle updates and handle them.

The MCU memory is partitioned into two applications areas and a bootloader to enable the delta update. The two applications areas have one running the current application, and the other memory space is the destination of the reconstruction of the new firmware from the old one. The bootloader, designed in Section 5.3.11, is required to be able to change the currently running application to the new one. Therefore, the memory is organized as illustrated in Figure 5.37. Considering the default memory constraints of the STM32L081KZ [83], which has 192 KBytes of flash memory, resulting in three partitions of 8KBytes to the bootloader, and 92 KBytes for each application.

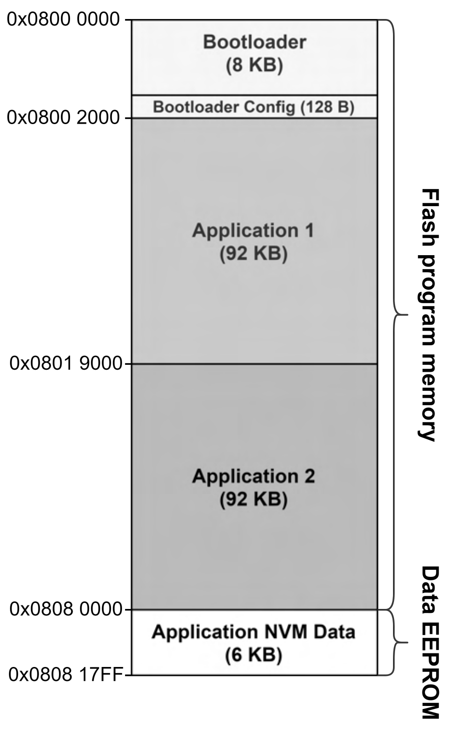


Figure 5.37: MCU memory organization, adapted from STM32L081KZ [83].

This memory layout allows new applications to be written into memory without corrupting the current application. This process is done by receiving the delta from the cloud and handling it. The delta handle in the end-device is interpreted as a command, represented in Figure 5.63. This command is received through the communication reception module, which converts the raw data from the modem into datagram objects. Then the commands module handles this datagram object by identifying the OTA opcode and calling the OTA command. This delta path from communication into the OTA module is represented in Figure 5.38.

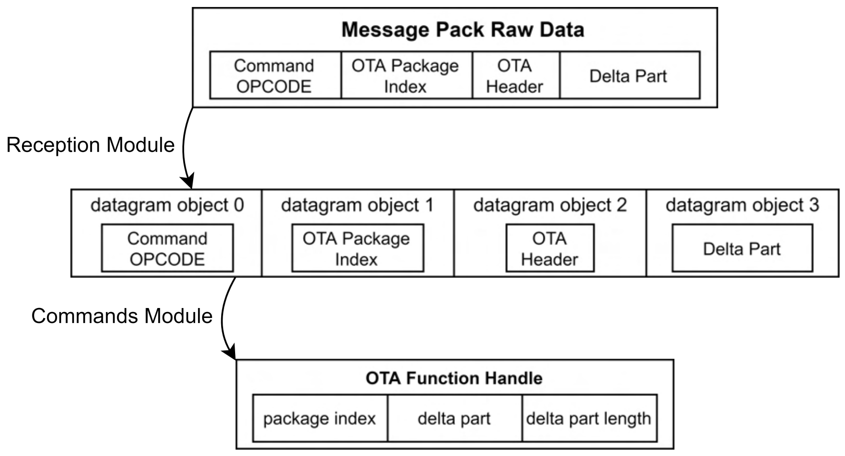


Figure 5.38: Delta path from communication into OTA module.

It has already been explained how the delta arrives at the command module. That being said, the remaining section will explain how the deltas are handled through a state machine that will be briefly explained. Then flowcharts of each state will be presented, finalising with the class diagram of the module.

The OTA module handles the deltas using the state machine represented in Figure 5.39. The OTA Begin state waits for a valid header and parses the new application start address, end address, and checksum. If a valid header is received, it erases the new application flash space in the Erase New App Flash Space state. Then, it enters the Program New App Flash Space, which stays most of the update since the most important memory operations are made in this state. When it receives an END delta command, all delta packages have been already received and written into the flash. After, it moves to the New App Checksum state, which iterates through the new firmware checking the various datagram checksums. If the checksum matches the received in the header, it enters the OTA End state, which writes the magic key and new application address into the bootloader config. Then, it signals the application that a correct firmware update has been done, and the application can restart. If something fails in the OTA update, it aborts the update operation and returns to the Begin state.

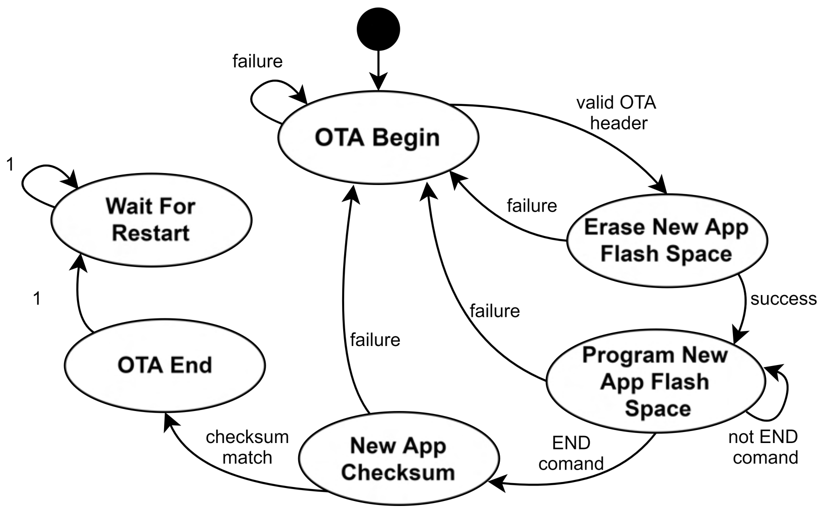


Figure 5.39: OTA state machine.

The OTA Begin state flowchart is represented in Figure 5.40. Its primary purpose is to parse the OTA header and validate its values. It checks if the package index is zero and the header byte array length is not less than nine bytes required to have a valid header (5.62). If it passes the verification, it saves the new application begin address, end address, and checksum. Then it checks if it is a valid new application address, and if it is, it sets the new application current address to the new application first address, clears the index of the word window, and returns S_NEW_APP_FLASH_ERASE as the next state. If either one of the validation fails, it returns the S_FAILURE state.

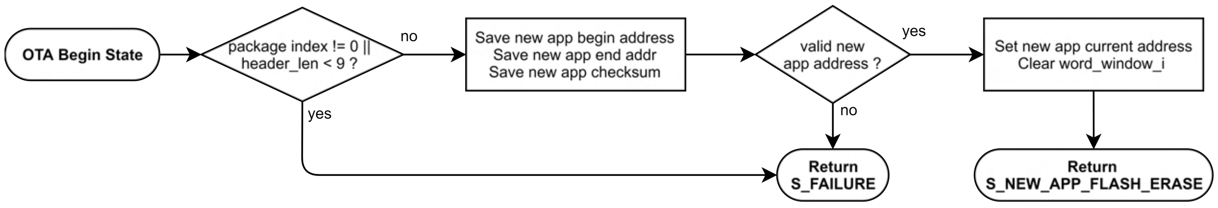


Figure 5.40: OTA begin state flowchart.

The New App Flash Erase State, illustrated in Figure 5.41, starts by calculating the number of pages to erase from the new application range from the begin and end address divided by the page size (128 bytes). If the application consumes more than the maximum page size, the update is aborted, returning the S_FAILURE state.

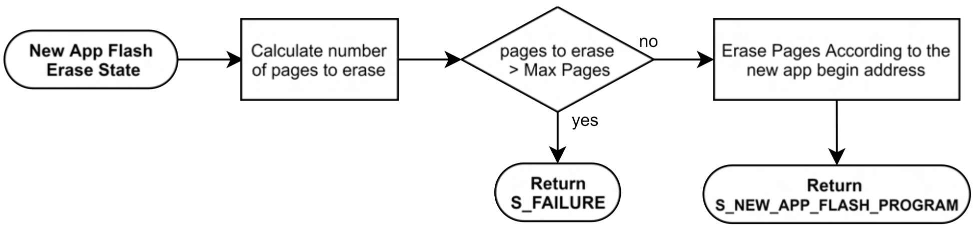

New App Flash Erase State
Calculate number of pages to erase
pages to erase > Max Pages
no
yes
Erase Pages According to the new app begin address
Return S_FAILURE
Return S_NEW_APP_FLASH_PROGRAM

Figure 5.41: Erase new application flash memory state flowchart.

The core of the OTA update is the New App Flash Program State because it is where the OTA commands are parsed, and the new firmware is reconstructed to the flash memory. This state starts by checking if the new package equals the expected index value. If it is, it parses all the commands in the new delta. The OTA commands are explained in Section 5.2, but the following itemise is the board behaviour to them:

- **END** – all the delta packages were received and parsed. Returns the S_NEW_APP_CHECKSUM state;
- **CPY** – copies from the old firmware flash space into the new application space;
- **ADD** – adds the received bytes presented in the update command into the new firmware;

If it parsed all the OTA commands and did not encounter an END command, this state returns the S_NEW_APP_FLASH_PROGRAM state to continue in the same state. On the other hand, if it encounters an invalid command, it returns the S_FAILURE state.

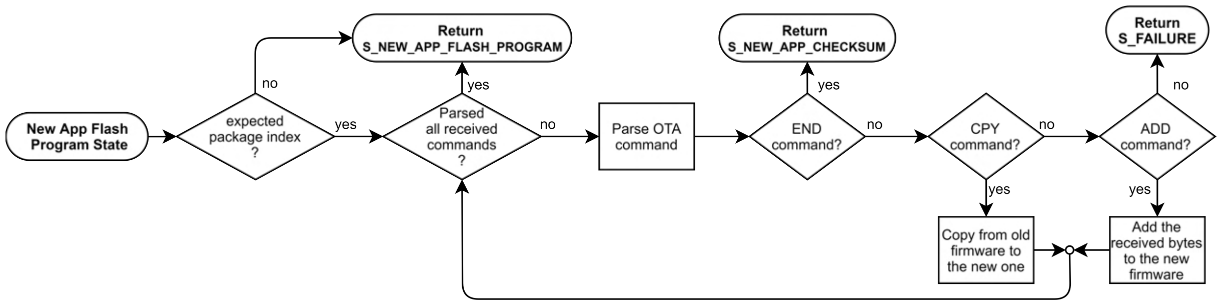

Return S_NEW_APP_FLASH_PROGRAM
Return S_NEW_APP_CHECKSUM
Return S_FAILURE
no
yes
New App Flash Program State
expected package index ?
yes
Parsed all received commands ?
no
Parse OTA command
END command?
no
CPY command?
no
ADD command?
yes
yes
Copy from old firmware to the new one
Add the received bytes to the new firmware

Figure 5.42: Program new application flash memory state flowchart.

The New App Checksum state, represented in Figure 5.43, iterates through all the bytes in the new firmware, summing them. Then, it calculates the checksum, given by the equation in 5.6. Since it uses the mod of 256, the checksum value ranges from 0 to 255.

$$checksum = \Sigma new\_firmware\_bytes \; \% \; 256 \tag{5.6}$$

After the checksum calculation, if it is equal to the header checksum, the OTA update is valid and returns the S_END_OTA state. If it is not, it returns the S_FAILURE state.

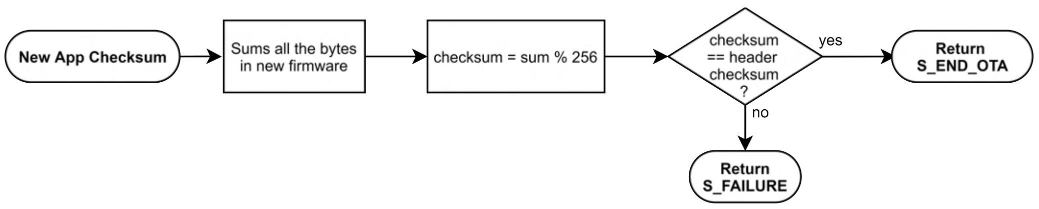


Figure 5.43: New firmware checksum calculation state flowchart.

The End OTA state, shown in Figure 5.44, according to the current application set in the bootloader config flash page, the new application address and the magic key. This state ends by signalling the application that an OTA update has been realised successfully.

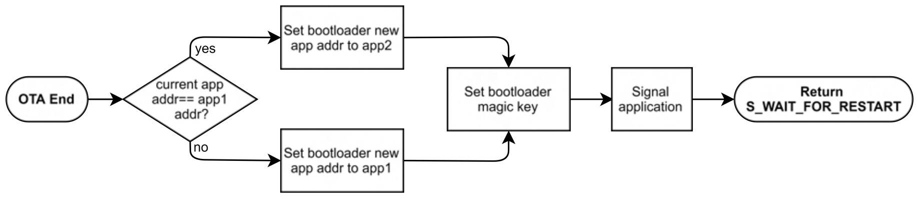

OTA End
current app addr== app1 addr?
yes
no
Set bootloader new app addr to app2
Set bootloader new app addr to app1
Set bootloader magic key
Signal application
Return S_WAIT_FOR_RESTART

Figure 5.44: End OTA state flowchart.

The last OTA machine state is for ensuring the new application integrity. Since it already signalled the application that a newer firmware is ready, this state prevents a new firmware change through the reception of a new delta update—keeping the current new firmware intact. As the OTA End state, it signals the application that a new firmware is ready to restart.

The resulting OTA class diagram is represented in Figure 5.45. It isolates the updates by only exposing the OTA_e and getOTA_u32 functions. The state machine described above is implemented using the private functions otaBeginState_t, newAppFlashEraseState_t, etc. The getOTA_u32 command returns the expected package index from the OTA module.

The word_window_au8 and word_window_i_u8 variables are used to solve the problem of unaligned memory access. For example, if the COPY command has an unaligned offset address and length of 2, it will be copied to the word window array the two bytes from the flash. The following command will fill the rest of the word. As a result, when the word is filled, the memory writes are aligned with the flash space. If the following command is an END command, the word window will be padded with zeros.

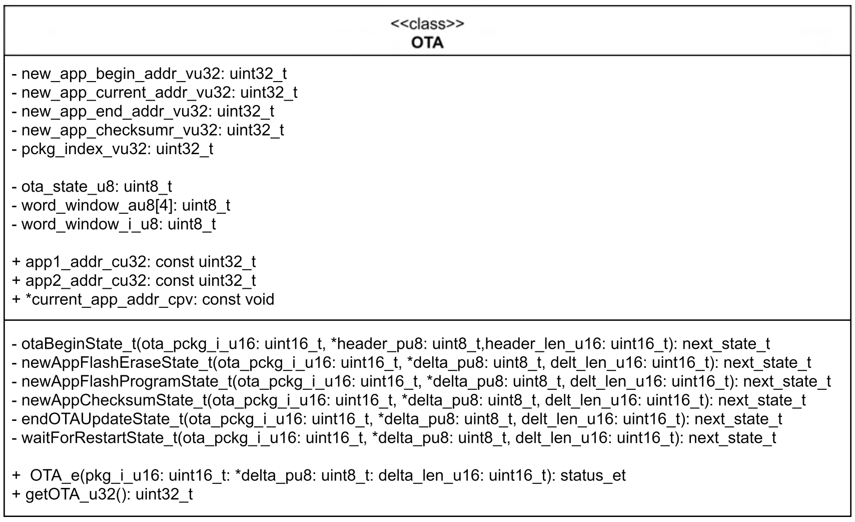

```
<<class>>
OTA

- new_app_begin_addr_vu32: uint32_t
- new_app_current_addr_vu32: uint32_t
- new_app_end_addr_vu32: uint32_t
- new_app_checksumr_vu32: uint32_t
- pckg_index_vu32: uint32_t

- ota_state_u8: uint8_t
- word_window_au8[4]: uint8_t
- word_window_i_u8: uint8_t

+ app1_addr_cu32: const uint32_t
+ app2_addr_cu32: const uint32_t
+ *current_app_addr_cpv: const void

- otaBeginState_t(ota_pckg_i_u16: uint16_t, *header_pu8: uint8_t,header_len_u16: uint16_t): next_state_t
- newAppFlashEraseState_t(ota_pckg_i_u16: uint16_t, *delta_pu8: uint8_t, delt_len_u16: uint16_t): next_state_t
- newAppFlashProgramState_t(ota_pckg_i_u16: uint16_t, *delta_pu8: uint8_t, delt_len_u16: uint16_t): next_state_t
- newAppChecksumState_t(ota_pckg_i_u16: uint16_t, *delta_pu8: uint8_t, delt_len_u16: uint16_t): next_state_t
- endOTAUpdateState_t(ota_pckg_i_u16: uint16_t, *delta_pu8: uint8_t, delt_len_u16: uint16_t): next_state_t
- waitForRestartState_t(ota_pckg_i_u16: uint16_t, *delta_pu8: uint8_t, delt_len_u16: uint16_t): next_state_t

+  OTA_e(pkg_i_u16: uint16_t: *delta_pu8: uint8_t: delta_len_u16: uint16_t): status_et
+ getOTA_u32(): uint32_t
```

Figure 5.45: OTA module class diagram.

5.3.11 Bootloader

The bootloader is the first program in memory and that the MCU initializes. It occupies the first 8 KBytes in the flash, as represented in Figure 5.46. Also, it is responsible for deactivating the board peripherals and jumping to the application. The application jump can be the current running or a new application area if the magic key and the new application address are set. The magic key is a simple identifier that verifies who made the firmware update.

The bootloader has two hook variables that the application can change to signal a new firmware update and its specific address. These variables are presented in the last flash page of the bootloader and are the magic key and the new application address as represented in Figure 5.46. Moreover, the bootloader saves the current application address in address offset 0x1FF4. If it has no address set (zero value), it jumps to application one default address by default. The default address is the memory space after bootloader, which has 0x2000 of offset.

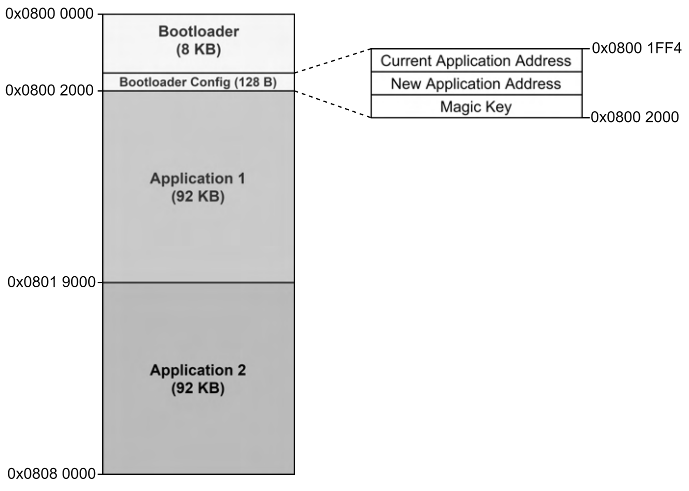


Figure 5.46: Bootloader config page.

When the main application wants to change the application area, it writes to the magic key and the respective new application address to the new application address bootloader variable and restarts. Then, the bootloader reads the magic key address. Suppose that an update has been done successfully, and it is the restart to change to the new application. In this case, the magic key value will match “ESRG”, so it will deactivate the MCU peripherals, clean the magic key value, save the new application address in the current application address, and jump to the new application as represented in Figure 5.47. If the magic key does not match, it will jump to the current application running.

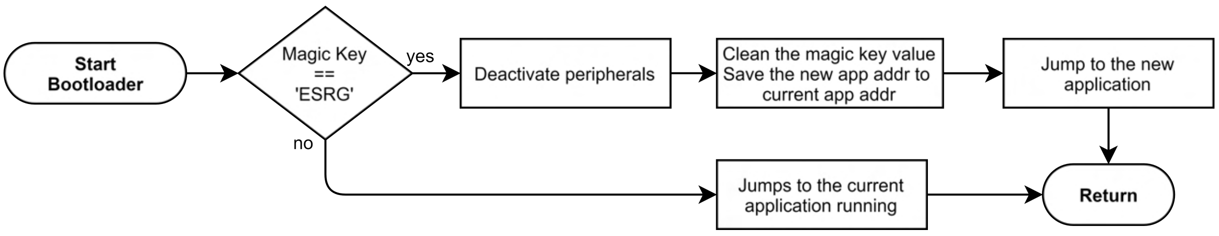

Start Bootloader
Magic Key == 'ESRG'
yes
no
Deactivate peripherals
Clean the magic key value Save the new app addr to current app addr
Jump to the new application
Jumps to the current application running
Return

Figure 5.47: Bootloader's flowchart.

5.3.12 Application

The application layer is composed of a single thread represented in Figure 5.48. It starts by waiting for the communication initialization. After, it sets the system control get DateTime update function pointer used in the system control to refresh the RTC and enables the RTC update.

Since the communication is initialized, the modem is already initialized, so the board id (the modem SIM IMSI) is got. It waits for the RTC to be updated and checks if the sensors were initialized. If yes, it enables the sampling and sets the regular alarm. Afterwards, the command manager thread is started and sets the event flag to trigger a configuration transmission. Finally, it enters the Send and Application Error Handling Loop, illustrated in Figure 5.49.

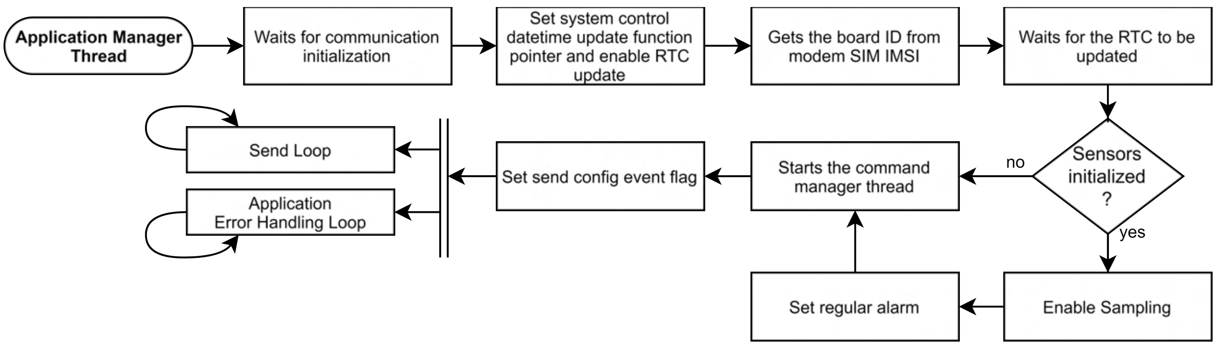

Application Manager Thread
Waits for communication initialization
Set system control datetime update function pointer and enable RTC update
Gets the board ID from modem SIM IMSI
Waits for the RTC to be updated
Sensors initialized ?
no
yes
Enable Sampling
Set regular alarm
Starts the command manager thread
Set send config event flag
Send Loop
Application Error Handling Loop

Figure 5.48: Application Manager thread's flowchart.

The Send Loop is responsible for creating messages to be sent to the cloud. By monitoring the several flags triggered by the middleware modules, this thread creates the corresponding messages, such as configuration, emergency, regulars, and command response. The command response is the location for the command's response. It receives if the command was successful and the respective set or get values.

The application error handling monitors the error messages of all the modules. For example, if the system control does not successfully call the get DateTime function, it triggers the getting DateTime error. The error handle triggers a get DateTime communication error.

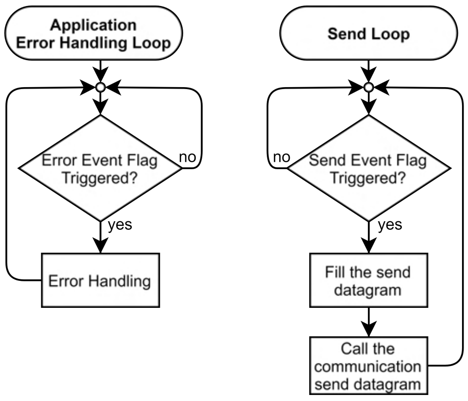

Application Error Handling Loop
Error Event Flag Triggered?
no
yes
Error Handling
Send Loop
Send Event Flag Triggered?
no
yes
Fill the send datagram
Call the communication send datagram

Figure 5.49: Application Manager thread send and error handling loop flowcharts.

As mentioned before, the send loop threads are responsible for sending messages to the cloud. Each message has a specific layout, having the board id and the timestamp in common. The message layout is described in the figure below.

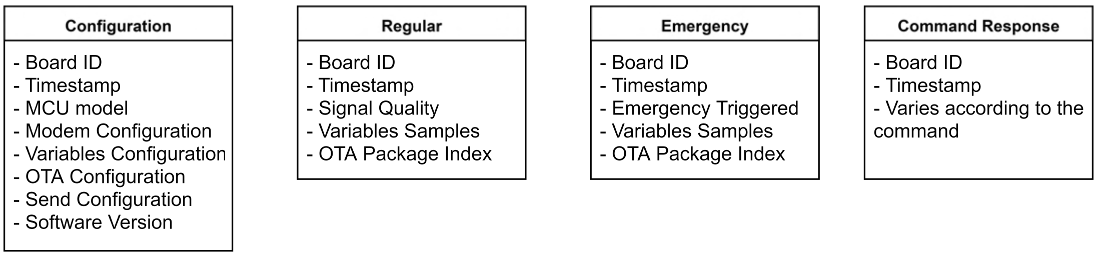


Figure 5.50: Application Message according to the type.

The Configuration datagram will comprises the MCU's model, Modem Configuration, Sensors' Variables Configuration, OTA Configuration, Send Configuration, and Software Version. The Modem Configuration is the modem cell ID, its tac and tau; the Variables Configuration are the sensors' variables status, name, period, and interrupt status; the OTA Configuration is the application one, application two, and the current application running addresses; the Send Configuration have the transmission mode (alarms or time interval), in alarms mode, it has all alarms, in time interval it has the time interval; the Software Version is the current software version.

The Regular datagram will have the Signal Quality and the Sensors' Variables Samples. The signal quality is the modem signal quality in the transmission time; The sensors' variables samples are the samples between transmissions.

The Emergency datagram will have the Emergency Triggered and Variables Samples. The emergency triggered identifies the interrupt that triggered the emergency; the Variables Samples are the sensors' variable sampled after the emergency trigger;

The Regular and Emergency datagrams have the OTA package index if an OTA update was started.

The Command Response datagram has the following parameters that vary according to the command. For example, a set alarms command will return the alarms mode (time interval or alarms), and the respective values according to the mode. If a set of variables interrupt status or period, it will have the sensors' variables interrupt or the period, respectively.

The application threads execution over time, priority and priority preemption threshold is represented in Figure 5.51. The lower the thread number, the higher the thread priority is. The thread priority has been given using the RM algorithm explained in Section 2.3.1.

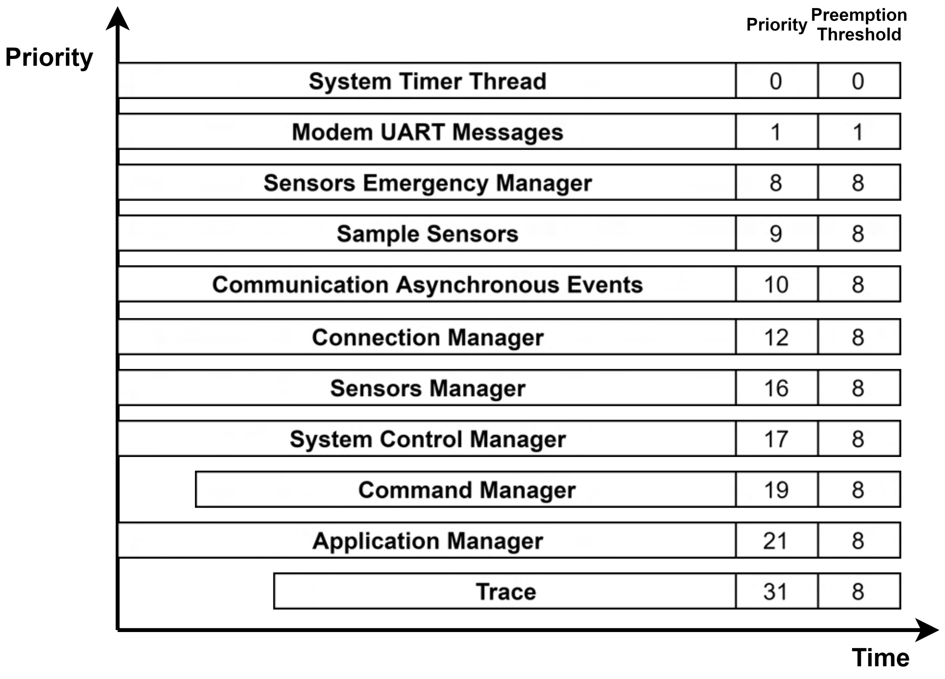


Figure 5.51: Threads priorities and preemption threshold values (lower priority number higher the thread priority).

Since the communication with the modem is a core service, the Modem UART Message handler is the thread with the higher priority, followed by the ThreadX timer thread. Because the threads Sensors Emergency Manager, Sample Sensors, Communication Asynchronous Events threads are fast, they have priority 8, 9, and 10, respectively, followed by the Connection Manager with 12. The Sensors Manager have a priority of 16 compared to 17 of the System Control Manager because it is more important to configure the sensors to update the current date and time. The Command Manager has priority (19) is greater than the Application Manager (21) to prioritize the command reception. Finally, the Trace thread has the lowest priority because it is a debug thread and does not change the other system's threads behaviour.

The preemption threshold values will be assigned according to the shared services between threads. Figure 5.52 represents the application shared binary semaphores.

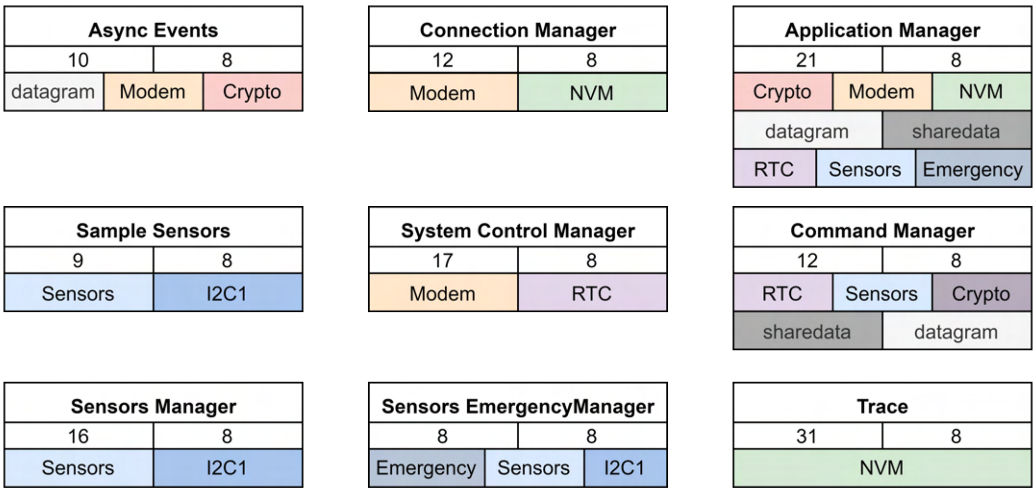


Figure 5.52: Application shared binary semaphores.

Every thread except the Modem UART Messages and the System Timer thread has the preemption threshold 8 to avoid binary semaphore deadlocks. This mechanism helps ease some of the inherent problems of preemption and concurrent accesses.

5.4 Cloud

This section presents the cloud microservices using microservice architecture and its database structure. Then a commands module will be proposed to be used to interface the database with high-level applications.

The cloud design overview is represented in Figure 5.53. It uses the Blackwing framework to manage the microservices and uses the MongoDB database. The server is responsible for receiving the end-device messages, transferring the message to the respective microservice. The microservice then will parse the message and save it into the database. In case that the respective board has a command waiting, the microservices send it to the server, and the server sends it to the end-device

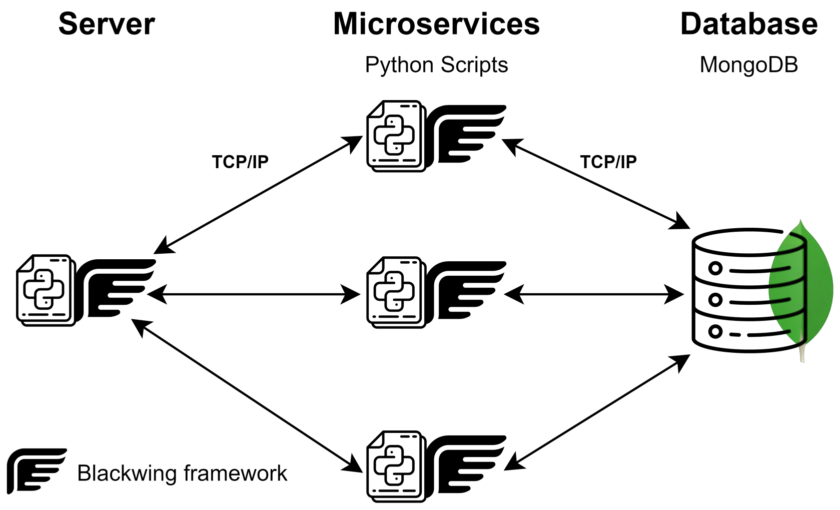


Figure 5.53: Cloud overview design.

The Blackwing is a python framework created by S. Camões. It is all written in python and handles the microservices intercommunications, freeing the user to only implement the microservice code. The server and the microservices are python processes, and the server communicates with the microservices by TCP/IP. The server has a folder with the microservices settings needed to identify and establish connections with them.

The Blackwing framework has a specific packet format presented in Figure 5.54. The message type identifies if the header is encrypted or not. In case of not be encrypted, the header only will have the microservice opcode. If it is encrypted, the header will be encrypted with the RSA algorithm having the AES key and AES IV in it. Also, the payload will be encrypted using the AES algorithm using the key and IV present in the header.

The AES key and IV of the header enable the cloud to decrypt the message payload (where the board message is). Since the header is encrypted with the RSA algorithm, only the cloud can decrypt it. Consequently, only the cloud can decrypt the message payload.

The microservices will be responsible for parsing the received payloads and sending, if available, a command to the respective board.

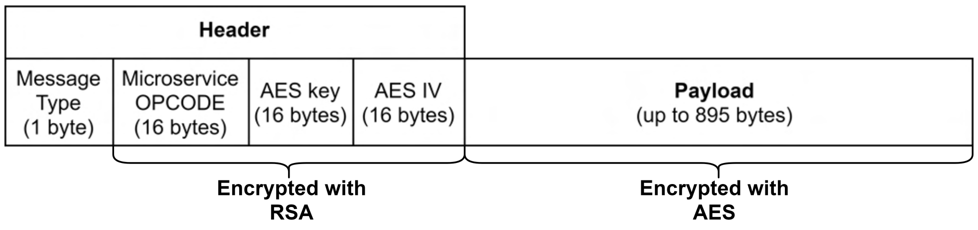


Figure 5.54: Blackwing protocol.

The AES key and IV of the header enable the Blackwing server to decrypt the message payload (where the board message is). Since the header is encrypted with the RSA algorithm, only the server can decrypt it. Consequently, only the cloud can decrypt the message payload.

In case of the microservice has a response to the device, it sends the response to the server, the server encrypts using the AES key and IV and sends it to the end-device.

5.4.1 Microservices

The cloud will have four microservices: Configuration, Regular, Emergency, and Command Response. Since the end-device communication is generic, all microservices have a similar structure illustrated in Figure 5.55. It will start by forming pairs with the received message, which will pair the communication key identifier with the respective values. Then, it is a microservice-specific code, followed by a database update or insertion of the data received. Finally, the microservice reaches the checking for command response to the board by calling the response function. This function queries the database for a command with the received board id. If a command is queried, the microservice will return to the Blackwing server the command or none if not.

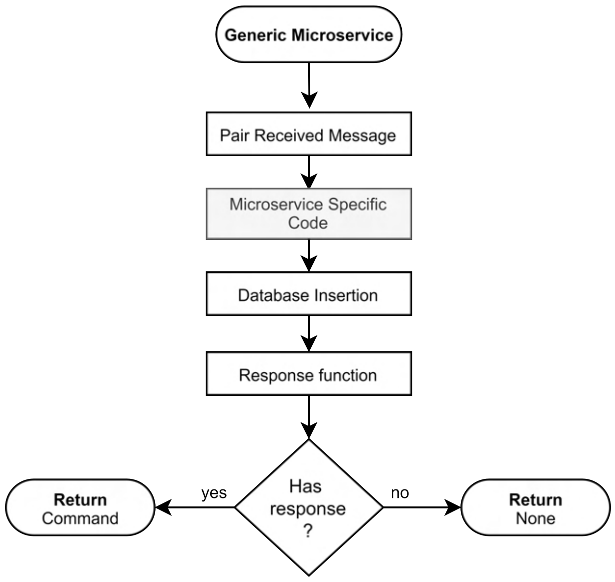


Figure 5.55: Generic microservice's flowchart.

The configuration will have to do a second parse into the received modem, sensors' variables, and send configurations. The modem configuration will separate the cell id, the tau, and the tac; the sensors' variable will unify all the configurations. For example, if it receives the variable TMP, it will group its period, interrupt id, interrupt status and state; in the send configuration, it will parse it according to the transmission mode the alarm or time interval received;

The regular microservice will assign the respective RSSI, RSRP, and RSRQ names to the signal quality and parses the sensors variables samples.

The emergency response does not have any post-processing modifications.

According to the command, the command response microservice will update the board configuration and save the response into the command response database. For example, if the transmission mode is changed from alarms to a time interval.

5.4.2 Commands

The commands query data from the end-devices and save it into the database. Furthermore, the device behaviour can be changed using commands in the database and since its response is saved. The end-devices can be controlled by gathering data and settings and inserting commands into and from the database. Therefore, it is possible to develop a high-level application that queries the database and controls the end-devices using the Command module, as illustrated in Figure 5.56.

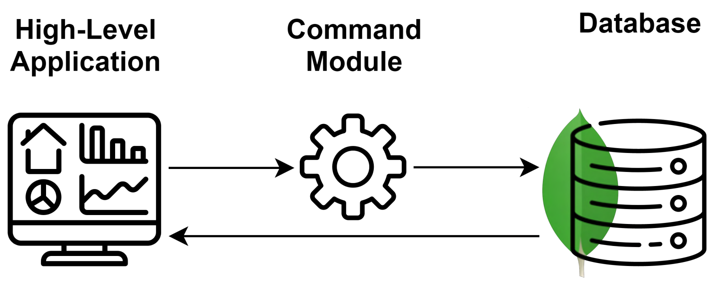
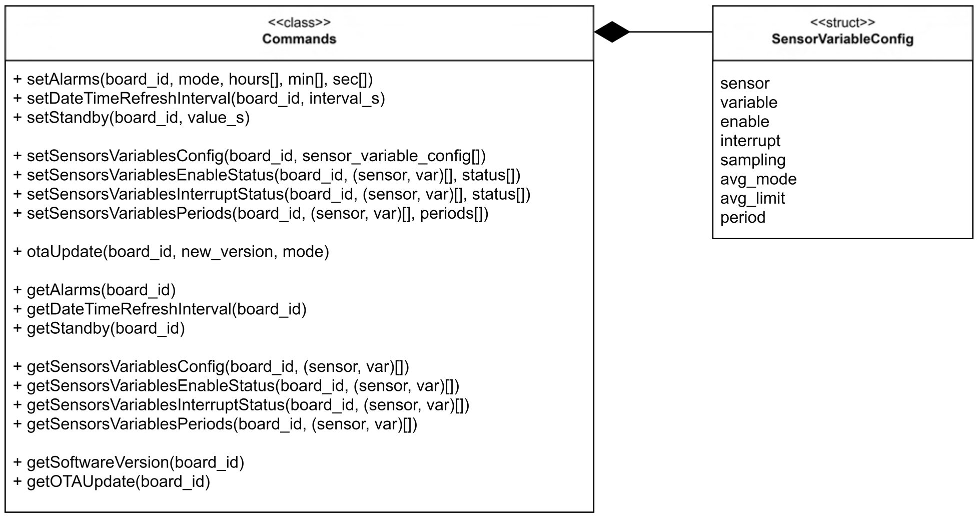


Figure 5.56: Commands module interface by a high-level application.

In this master's thesis is proposed the Command Module, it receives the command the parameters settings and prepares and inserts the command into the database. Its class diagram is represented in Figure 5.57.



```
+ setAlarms(board_id, mode, hours[], min[], sec[])
+ setDateTimeRefreshInterval(board_id, interval_s)
+ setStandby(board_id, value_s)

+ setSensorsVariablesConfig(board_id, sensor_variable_config[])
+ setSensorsVariablesEnableStatus(board_id, (sensor, var)[], status[])
+ setSensorsVariablesInterruptStatus(board_id, (sensor, var)[], status[])
+ setSensorsVariablesPeriods(board_id, (sensor, var)[], periods[])

+ otaUpdate(board_id, new_version, mode)

+ getAlarms(board_id)
+ getDateTimeRefreshInterval(board_id)
+ getStandby(board_id)

+ getSensorsVariablesConfig(board_id, (sensor, var)[])
+ getSensorsVariablesEnableStatus(board_id, (sensor, var)[])
+ getSensorsVariablesInterruptStatus(board_id, (sensor, var)[])
+ getSensorsVariablesPeriods(board_id, (sensor, var)[])

+ getSoftwareVersion(board_id)
+ getOTAUpdate(board_id)
```

```
sensor
variable
enable
interrupt
sampling
avg_mode
avg_limit
period
```

Figure 5.57: Commands module class diagram.

All the commands module functions have as first parameter the board id. Also, when the commands have two squared brackets means that it is receiving an array.

The commands module has the SensorVariableConfig data structures as an auxiliary to configure sensors variables. Finally, the get functions insert the get commands into the database. For example, the setSensorsVariablesPeriods receives an array of new periods to be set according to the sensor-variable tuple array received. According to the sensor-variable tuple array in the function parameter, the get function inserts the command that will retrieve the periods.

Command Response

The command Response function is used to query the database for a command for the respective board. It will receive the database cursor, board id, and name of the microservice calling it. Then, it will query the database for a command for the respective board, prioritizing the OTA commands. If a command is found, it will delete it from the Commands collection if it is not an OTA.

5.4.3 Database

In order to save the samples, configurations, emergencies, commands responses from the end-devices, a database is required. Also, it is the interface between end-devices and a high-level application.

The database in use is MongoDB, which uses an alternative to traditional relational databases. No-relational databases are quite helpful for working with large sets of distributed data.

This section will present all the database collections: Configuration, Regular, Emergency, Commands, Commands Response, and Deltas.

All the collections table that saves packages from the end-device (Configuration, Emergency, Regular, and Command Response) have the parameters "p", board_id, board_timestamp, and system_timestamp. The "p" parameter is the diminutive for parts. Since the transmission packages can be sent in multiple packages, these arrays contain the number of the current and total packages. The board_id is the Universally Unique Identifier (UUID) used for device identification. The board and system timestamp is in UNIX time format when the board transmits, and the system receives, respectively. The common elements in all collections were explained. The rest of the section will explain specific parameters from each collection.

The Configuration collection, represented in Figure 5.58, has the end-device configurations. The end-device configurations are: the alarms parameter has the time interval between regular transmissions or the alarms through the day that it will transmit; the modem configuration is composed of the cell id, tau and tac of the modem; the OTA dictionary has the addresses of the application 1 and 2 areas and the current running (now). Also, it has the OTA packaged index and the total flash size of the end-device: the sensors variables present in the board, its respective status, and the board software version.

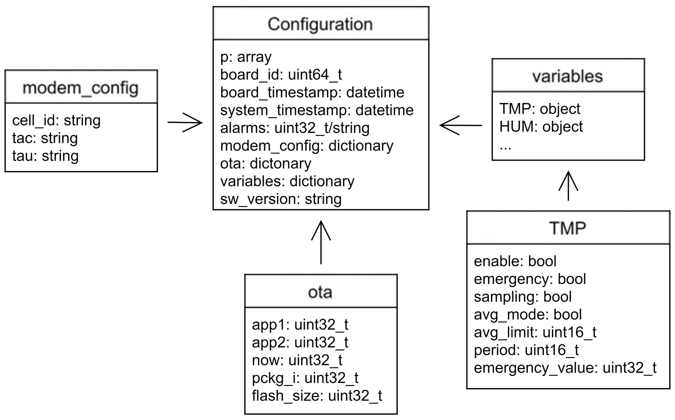


Figure 5.58: Configuration collection.

Figure 5.58 illustrates the Regular and Emergency collection. The Regular collection has the signal quality and the sensors samples. The Emergency collection has the triggered emergency flags and the sensors' variables samples in the emergency time.

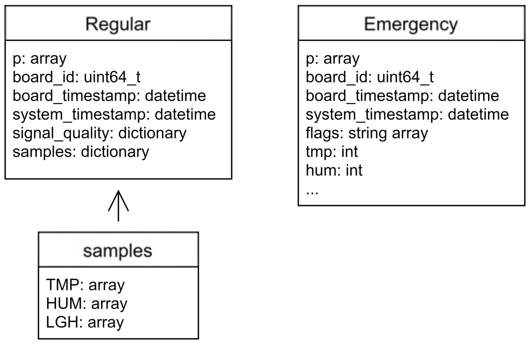


Figure 5.59: Regulars and emergencies collections.

The commands are split into two collections: Commands and Commands Response. The Commands collection has the commands waiting to be sent to the end-device, and the Commands Response has the end-device to the commands. Both have the commands opcode, microservice to respond to, and the MD5 status. The differences between these collections are that the Commands collection has the command datagram to send to the board and its description, which the Command Response does not have. Also, the Command Response has the result of the command (set and get) and data according to the command. For example, the get interrupts command has the interrupts values. The collections are exhibited in Figure 5.59.

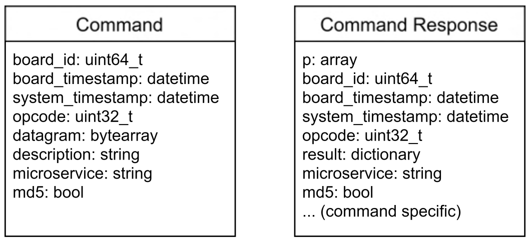


Figure 5.60: Commands and Commands Response collections.

The deltas that enable the firmware update Over-The-Air are saved in the Delta collection depicted in Figure 5.61. This collection is used by the OTA command when a command is inserted. The old and new version parameters are used to find the respective delta. Also, it has the new and old addresses memory addresses, the total number of packages and the respective delta packages.

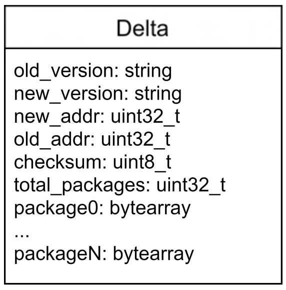


Figure 5.61: Deltas collection.

The database is the data core since it is where the information is saved, and the board interactions wait to be sent to the board. For this reason, if it does not exist, the creation of the commands abstraction layer would not be possible.

5.4.4 OTA

As described above in Section 5.2, the last stage of the delta generation is the package conversion. The package conversion is a function that receives the header size and the maximum payload size returning the delta in multiple packages according to the payload size received. The first package's payload is reduced to embed to the datagram an OTA header represented in Figure 5.62.

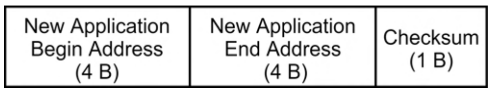

New Application Begin Address (4 B)
New Application End Address (4 B)
Checksum (1 B)

Figure 5.62: OTA header.

The OTA header is a byte array of nine bytes, composed by the new application begin address, end address and checksum.

This header and the delta are included in the board's package. This package structure is represented in Figure 5.63. The packages start by having the OTA command opcode, followed by the OTA package index and the respective delta byte array. The OTA package index is required because the packages need to be received in the end-device ordered and since the modem can only receive 512 bytes at a time. As described above, the first package has the OTA header of Figure 5.62 and, as the following packages, the delta part. Notice that the last delta part has the delta command END at the end.

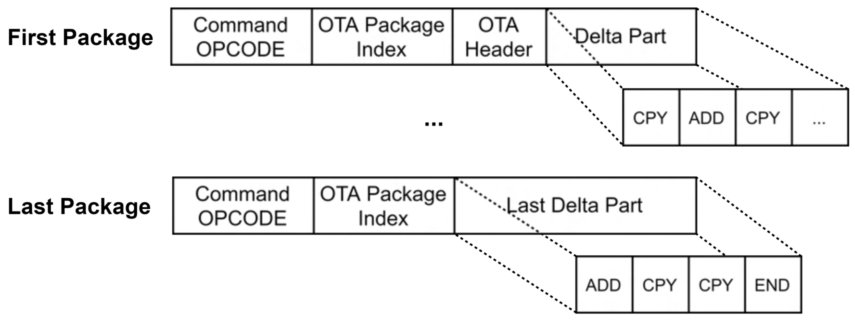


Figure 5.63: Package example of OTA updates.

The cloud server can only send the next delta datagram when the board receives the previous one successfully. Therefore, the board will reject out-of-order packages.

The packages generated from the delta will be saved in the Delta collection, described in Section 5.4.3.

Chapter 6: Implementation

The implementation phase is the realization of the design phase. This chapter presents a comprehensive description of how the Delta, end-device and cloud was implemented.

Not all the code will be shown since it became a large codebase. Therefore, some parts will be hidden to better readability and understanding, having most of the hidden code in Appendix A.1.

Programming Languages and Tools

The programming languages used were C and Assembly for the end-device and Python for cloud microservices and scripts. The development environment used for the end-device was the Keil MDK IDE, which is the complete software development environment for a range of Arm Cortex-M based microcontroller devices. Also, the STM32CubeMX was used for project generation. It is a graphical tool that allows a straightforward configuration of STM32 microcontrollers. Furthermore, the Ceedling framework was used to perform unit testing on the resulting software and the Cppcheck and clang-tidy as static analysers.

The microservices development was done with the editor Visual Studio Code. It is a source-code editor made by Microsoft which includes support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git. For better interactions with the database, the MongoDB Compass was used. It is an interactive tool for querying, optimising, and analysing database data.

Finally, Git was used for version control in the end-device, cloud, and scripts. It is a free and open-source distributed version control system designed to handle small and substantial projects quickly and efficiently.

Figure 6.1: Tools and programming languages.

During the development phase, it was implemented the modules designed previously and development helpers. The following sections will be present its implementation.

6.1 Firmware Over-The-Air Delta

In the section will be present the Firmware Over-The-Air Delta implementation. It consists of implementing the Array Conversion, Differencing Algorithm, and Package Conversion stages depicted in Figure 5.4.

The Delta implementation stage is embedded in the OTA class. This class receives through its constructor the old firmware version, new firmware version, MCU model, the max payload and header length as represented in Code 6.1.

```python
class OTA:
    def __init__(self, old_version, new_version, old_addr, new_addr,
                 mcu, max_payload=512, header_len=11):
        ...
```

Code 6.1: OTA class constructor.

The OTA class has the function generate, represented in Code 6.2. This function starts by calling the firmware array conversion. Then it declares the DeltaGen class, which is responsible for implementing the differencing algorithm. After generating the delta through the DeltaGen class, the new firmware checksum is calculated, and the package conversion converts the delta commands to board compatible packages. The package conversion returns a commands delta dictionary composed of multiple delta datagrams. It finishes by saving the delta dictionary into the Delta collection in the database.

```python
def generate(self):
    # Array Conversion Stage
    self.old_firmware_array = self.array_conversion(self.app1_name,
                                                     self.old_addr)
    self.new_firmware_array = self.array_conversion(self.app2_name,
                                                     self.new_addr)

    if self.old_firmware_array is False or
       self.new_firmware_array is False:
        return False

    # Differencing Algorithm Stage
    self.deltagen = DeltaGen(self.old_firmware_array,
                             self.new_firmware_array)
    if not self.deltagen.generate():
        return False

    self.checksum = self._calculate_checksum(self.new_firmware_array)

    # Package conversion stage
    delta_cmd_packages_dict = self.package_conversion()

    self.save(delta_cmd_packages_dict)

    return True
```

Code 6.2: OTA class generate function.

The Array Conversion function, represented in Code 6.3, reads from the Hex or Bin file according to the received format and loads into byte arrays. It uses the python *intelhex* package to parse the firmware file and check if it has gaps. If it does not, it returns the respective firmware array.

```python
def array_conversion(self, path, start_addr):
    file_path = os.path.join(dirname, path)

    ih = IntelHex()
    ih.fromfile(file_path, format = 'hex' if path[-3:] == 'hex' else 'bin')

    has_gap, last_addr = self.has_gaps(ih.todict(), start_addr)

    if has_gap:
        return False

    return ih.tobinarray(start=start_addr, size=last_addr-start_addr)
```

Code 6.3: Array Conversion.

The old and new firmware arrays are fed into the DeltaGen class. Its constructor adds an offset and joins the arrays forming the array T, as explained in Section 5.2. The delta generation is done through the function generate of DeltaGen. This function, depicted in Code 6.4, generates the delta commands by calling all DeltaGen algorithm steps. It finishes by generating the new firmware through the old and the delta commands. If the firmware generated through delta matches the new firmware, it returns the delta commands. If not, it returns *False*. The DeltaGen most important functions will be explained next.

```python
def generate(self):
    self.generate_suffix_array()

    self.generate_rank_array()

    self.generate_height_array()

    self.generate_raw_delta(add_bytes=3, cpy_bytes=7)

    self.merge_adds()

    self.generate_new_with_delta()

    if not self.is_equal():
        return False

    self.remove_delta_offset()

    return True
```

Code 6.4: DeltaGen generate function.

The Suffix Array (SA) was implemented using the python library *pydivsufsort*, as illustrated in Code 6.5 because it is a fast and optimized implementation of the SA.

```python
    def generate_suffix_array(self):
        self.SA = divsufsort(self.T)
        return
```

Code 6.5: Optimized suffix array generation.

The Rank Array (RA) is calculated from Expression 5.4 as Code 6.6. It assigns the index (i) to the position of the SA value in the Rank Array.

```python
def generate_rank_array(self):
    self.RA = np.empty(len(self.SA), dtype=np.uint32)

    for i in range(len(self.SA)):
        self.RA[self.SA[i]] = i
    return
```

Code 6.6: Rank Array (RA) generation function.

The Height Array (HA) construction is in Code 6.7 it uses the Suffix Array and Height Array and forms the height array in time complexity of O($n \log_{10}(n)$) as proposed in [47].

```python
def generate_height_array(self):
    self.HA = np.empty(len(self.SA), dtype=np.uint32)
    k = 0

    for i in range(len(self.SA)):
        if self.RA[i] == 0:
            self.HA[0] = 0
        else:
            j = self.SA[self.RA[i] - 1]
            while self.T[i + k] == self.T[j + k]:
                k += 1

            self.HA[self.RA[i]] = k

            if k > 0:
                k -= 1
    return
```

Code 6.7: Height Array (HA) generation function.

The HA is used in the LCP function. This function receives an interval and returns the minimum LCP value. The offset of one in the interval is due to the HA values starting in position one. For example, the LCP value of comparing the first SA and second is stored in position one of HA.

```python
def LCP(self, i, j):
    return min(self.HA[i+1:j+1])
```

Code 6.8: Longest Common Prefix (LCP) calculation.

The SA and HA arrays and the LCP function are used in generating the raw delta function, depicted in Code 6.9. This function iterates through the new firmware and finds the common bytes between the new and old firmware. If there are bytes in common, it adds a COPY command into the delta. If not, it appends an ADD command. In the end, it appends the END command to signal the delta end.

```python
def generate_raw_delta(self, add_bytes, cpy_bytes):
    self.delta =  list()

    i = 0
    while i < len(self.new):
        (old_index, new_i) = self.findCommonBytes(i)

        if old_index != None:
            if new_i - i > cpy_bytes - add_bytes:
                self.delta.append({'op':'CPY',
                                   'len': new_i - i,
                                   'start_index': old_index
                                })
                i = new_i
                continue

        self.delta.append({'op':'ADD',
                           'len': new_i-i,
                           'data': [self.new[i]] if new_i - i == 1 else
                                   self.new[i:(new_i-i)+i]
                        })
        i = new_i

    self.delta.append({'op':'END',
                       'len': i,
                    })
    return
```

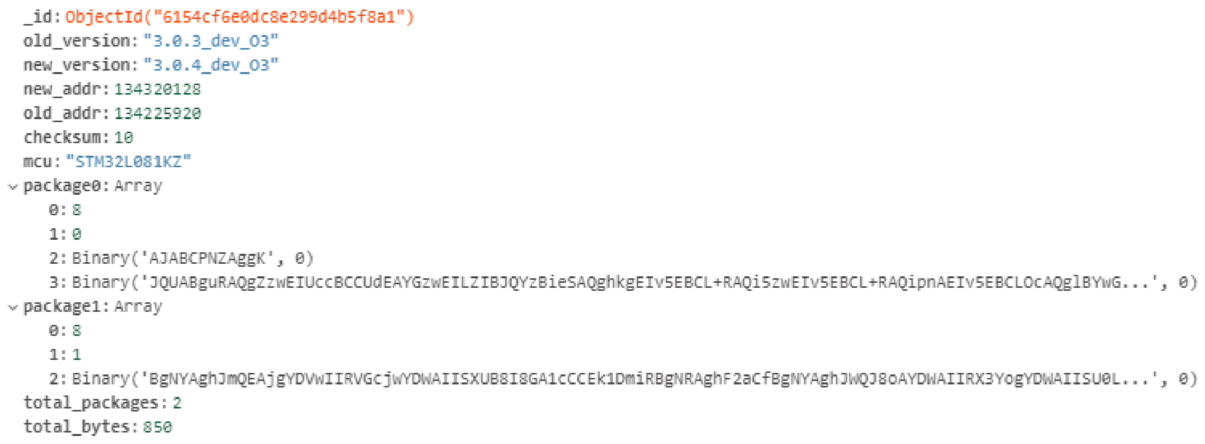
Code 6.9: Raw delta generation function.

The last stage of Figure 5.4 is the package conversion. This stage is implemented recurring to the function represented in Code 6.10. It starts by calling the `convert_delta_to_small_datagrams`, which receives the delta, the maximum payload and the maximum payload with the header (max payload minus the header length) and returns the delta converted into board compatible arrays according to the package limit. With all the delta byte arrays, it is generated the Link4S commands. Therefore, the package conversion function iterates through all the small datagrams adding the command opcode, package index, and the respective delta forming the Link4S command package. This package is saved to the `board_delta_packages` dictionary. Notice that in the first package it is added the OTA header.

```
def package_conversion(self):
        board_delta_packages = dict()
        small_datagrams_cmds = convert_delta_to_small_datagrams(
                                        self.deltagen.delta,
                                        self.max_payload,
                                        self.max_payload-self.header_len)

        for ota_datagram in small_datagrams_cmds:
            datagram.append(self._getOpcodes('otaUpdate'))
            datagram.append(pckg_i)

            if pckg_i == 0:
                # Add OTA header
                ...
                datagram.append(ota_header)

            datagram.append(ota_datagram)

            board_delta_packages[f"package"{pckg_i}] = datagram.copy()
            pckg_i += 1

        board_delta_packages["total_packages"] = pckg_i
        board_delta_packages["total_bytes"] = total_bytes

        return board_delta_packages
```

Code 6.10: Delta package conversion.

Finally, when all the commands are ready, it adds to the board delta packages dictionary the total bytes and number of packages and returns the board delta packages dictionary. This dictionary will be extended with delta configurations and saved into the database resulting in an object represented in Figure 6.2.



```
  _id: ObjectId("6154cf6e0dc8e299d4b5f8a1")
  old_version: "3.0.3_dev_03"
  new_version: "3.0.4_dev_03"
  new_addr: 134320128
  old_addr: 134225920
  checksum: 10
  mcu: "STM32L081KZ"
v package0: Array
     0: 8
     1: 0
     2: Binary('AJABCPNZAggK', 0)
     3: Binary('JQUABguRAQgZzwEIUccBCCUdEAYGzwEILZIBJQYzBieSAQghkgEIv5EBCL+RAQi5zwEIv5EBCL+RAQipnAEIv5EBCLOcAQglBYwG...', 0)
v package1: Array
     0: 8
     1: 1
     2: Binary('BgNYAghJmQEAjgYDVwIIRVGcjwYDWAIISXUB8I8GA1cCCEk1DmiRBgNRAghF2aCfBgNYAghJWQJ8oAYDWAIIRX3YogYDWAIISU0L...', 0)
  total_packages: 2
  total_bytes: 850
```

Figure 6.2: Example of delta in Delta collection.

The database object is composed of the MCU model, old and new versions names and addresses, the delta checksum, and the respective packages. The delta example of Figure 6.2 has a total of two packages. Notice that both packages have the command opcode 8 in the first position, and the first package has the OTA header in position two and the first delta partition in position three.

This section explained how the delta generation occurs from the board firmware files to the delta database collection. Section 6.3.2 shows how the database is queried, and a command is sent to the end-device. The following sections will illustrate the end-device and cloud implementation.

6.2 End-device

The end-device application was implemented according to the design, and because it is primarily a translation of the flowcharts into code, it is presented in Appendix A.1. This section will show the peculiar implementations, such as the Azure RTOS ThreadX low-power mode patch, the bootloader, and several scripts made to help the development. It will end by presenting the code unit tests, coverage, static analysers and the resulting application flash and ram size.

6.2.1 Azure RTOS ThreadX Low-power mode

The Azure RTOS ThreadX Low-power mode patch was performed by implementing a function extension called `_tx_low_power_mode`. This function is called when the ThreadX kernel enters in waiting for events state as represented in line 9 of ThreadX scheduler assembly Code 6.11. Before the function call, it is required to push the registers `R0` to `R3` because the kernel uses its values.

```
 1| __tx_ts_wait
 2|     CPSID    i
 3|     LDR      r1, [r2]
 4|     STR      r1, [r0]
 5|     CMP      r1, #0
 6|     BNE      __tx_ts_ready
 7|     IF :DEF:TX_ENABLE_LOW_POWER_MODE
 8|     PUSH     {r0-r3}                ; Save of r0 to r3 registers
 9|     BL _tx_low_power_mode           ; Jumps to the function of low-power mode
10|     POP     {r0-r3}                 ; Pop of r0 to r3 registers
11|     ENDIF
12| __tx_ts_ISB
13|     CPSIE    i
14|     B        __tx_ts_wait
```

Code 6.11: ThreadX kernel waiting stage.

This mode uses three user functions `configPreSleepMode`, `sleepMode`, and `configPosSleepMode` that are implemented using the weak keyword. This keyword allows the user to reimplement these functions taking advantage of the MCU low-power modes.

The `configPreSleepMode` receives the time in ticks that the kernel can sleep. Its purpose is to turn off the kernel tick source, set a wakeup mechanism and turn off peripherals that do not require running while sleeping. The `sleepMode` default implementation calls the `WFI` and `ISB` assembly instructions. The `WFI` puts the MCU cortex sleeping and waiting for an interrupt to be wake up. The `ISB` ensures that the pipeline is flushed after waking up.

The `configPosSleepMode` is called after the MCU is awake. This function aims to return the time elapsed while sleeping in ticks and re-enable the peripherals turned off in the pre-config function.

These functions allow the low-power implementation to be ported to others MCUs and only is required these board specific functions reimplementation.

```
    __weak ULONG configPreSleepMode(ULONG sleep_time_ul){
        if(sleep_time_ul <= (TX_TIMER_ENTRIES<<1) ){
            return (ULONG)(~0);
        }
        return 0;
    }
    __weak VOID sleepMode(VOID){
        __asm volatile ( "WFI" ); /*Wait for interrupt*/
        __asm volatile ( "ISB" ); /*Ensure pipeline is flushed*/
        return;
    }
    __weak ULONG configPosSleepMode(VOID){
        return 0;
    }
```

Code 6.12: User low-power functions.

The low-power mode kernel extension starts by finding the time of the next timer to expire by iterating through the timers list, which has all the current ThreadX active timers as illustrated in Code 6.13.

```
    tx_timer_current_pps =  _tx_timer_list_start;

    do{
        /* Iterate for all the current ThreadX active timers to find
           minimum time */
    }while(tx_timer_current_pps != _tx_timer_list_end);

    /* Setup the sleep time with the minimum remaining tick*/
    sleep_time_ticks_ul = min_remaining_ul;
```

Code 6.13: Finding the next timer to expire time.

After the next timer expiration is found, it is checked if it is greater than the timer list array as represented in Code 6.14. If it is, the `configPreSleepMode` function is called with the time in ticks of the next timer to expire. The time to expire is decremented to wake up one tick before the timer expiration. If it enters low-power mode, it calls the `sleepMode` function entering in the MCU low-power state. When the MCU wakes up, it calls the `configPosSleepMode` function, which returns the time passed in ticks. After getting the time in sleeping, the remaining ticks of all the active timers are refreshed.

```
if(sleep_time_ticks_ul > TX_TIMER_ENTRIES){
    /* Wake up 1 tick before*/
    sleep_time_ticks_ul -= 1;

    /* Configuration pre sleep mode, disable ticks */
    before_sleep_time_ul = configPreSleepMode(sleep_time_ticks_ul);

    if(before_sleep_time_ul != ((ULONG)~0)){
        /* Enter sleep mode*/
        sleepMode();

        /* Configuration pos sleep mode, enable ticks*/
        after_sleep_time_ul = configPosSleepMode();

        /*Exited sleep mode, Using min_remaining_ticks variable
          to have the result of ticks passed*/
```

```
            min_remaining_ul = after_sleep_time_ul - before_sleep_time_ul;

                /*** Refresh remaining ticks in the timers structs ***/
        }
```

Code 6.14: Low-power user call functions.

The refresh remaining ticks in the timers structs iterate the timer entry wrap list removing the ticks passed from the active timers. Then it reorganizes all timers entries because of the new times. It uses an auxiliary timer list and inserts the timers with the new remaining values into the auxiliary list. After inserting all the timers, the auxiliary list is copied to the timer list and set the current timer pointer pointing to the start of the timer entry wrap. Finally, the timer system clock variable is refreshed with the time passed. This variable has the tick count from the program begin.

This implementation was presented to the Azure RTOS ThreadX team, but they were also working on this feature and release it further testing within the scope of this project before the official release to the public. Their implementation is similar, but they separated the `_tx_low_power_mode` in two functions: `tx_low_power_enter` and `tx_low_power_exit` as represented in Figure 6.3.

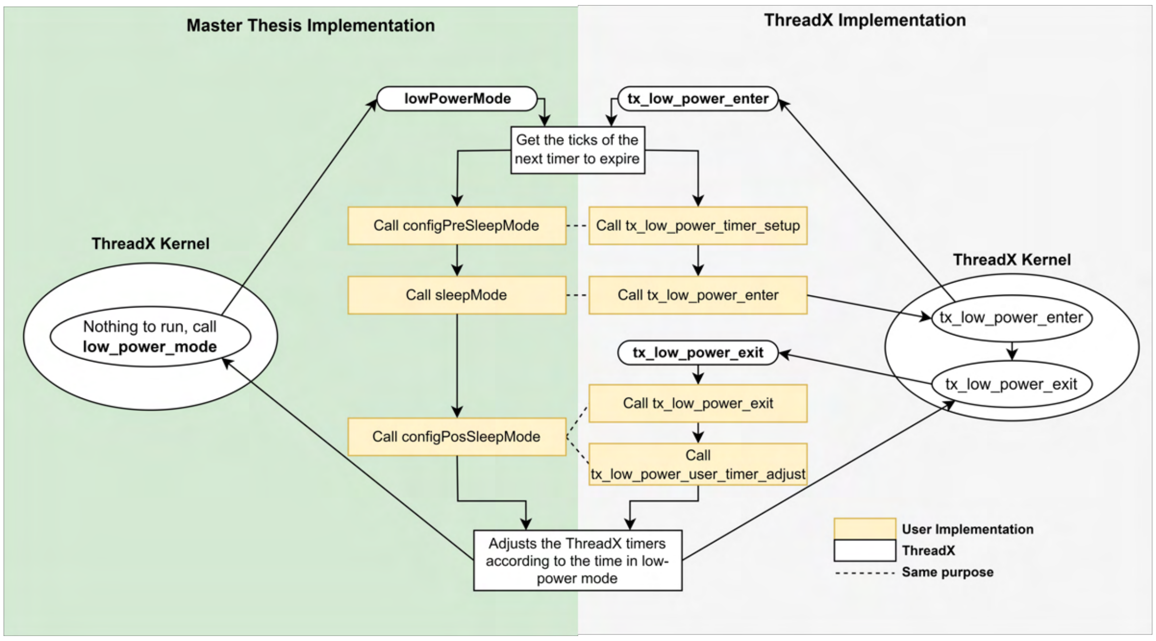


Figure 6.3: Low-power mode implementation comparison.

The `tx_low_power_enter` is responsible for searching the next active timer to expire, calling the user implementation of the set up of the MCU wakeup mechanism and entering in low power mode. The timer setup is done by `TX_LOW_POWER_TIMER_SETUP` and the low-power MCU entering through `TX_LOW_POWER_USER_ENTER` macros. The user can reimplement these macros to call functions. These functions are similar to `configPreSleepMode` and `sleepMode`.

When the MCU wakes up, it exits the tx_low_power_enter and enters the tx_low_power_exit. This function calls the TX_LOW_POWER_USER_EXIT and TX_LOW_POWER_USER_TIMER_ADJUST. They separated the configPosSleepMode into these two macros. The TX_LOW_POWER_USER_EXIT purpose is to re-enable peripherals that were disabled when entering the low-power mode. The TX_LOW_POWER_USER_TIMER_ADJUST returns the time passed in ticks while in sleep mode. Then the ThreadX implementation refreshes all the active timers.

Since the Azure RTOS ThreadX version is tested and well refined, it was used to enable the RTOS low-power mode. Therefore, the user functions were implemented to the Link4S board. The tx_low_power_timer_setup is represented in Code 6.15. As described above, this function receives the sleep time in ticks and starts by checking if the LPUART function is transmitting and if the sleep time is greater than the time minimum required to go to sleep. If it is, it needs to set up a timer to wake up the MCU. The LPTIM is used when the sleep time is lower than one second because the RTC does not have enough resolution. Moreover, the RTC is used for values greater than the LPTIM limit. The use of the LPTIM allows the system to enter in the lowest power consumption for short times, for example, while waiting for modem communications.

```
void tx_low_power_timer_setup(ULONG sleep_time_ticks_ul){
    if( lpuart_tx_bsem.tx_semaphore_count != 0 &&
        sleep_time_ticks_ul > MIN_TIME_TO_GO_SLEEP){
        go_to_sleep_u8 = 1;

        /* Disable systick */
        ...

        /* Read RTC time */

        if(sleep_time_ticks_ul < MS_TO_TICKS(MAX_LPTIM_TIME_MS)){
            /*Set up LPTIM*/
            ...
            return;
        }
        used_lptim_u8 = 0;

        /* Set RTC wake up */
        ...
        return;
    }
    go_to_sleep_u8 = 0;
    return;
    }
```

Code 6.15: Azure RTOS ThreadX user low power timer setup function.

6.2.2 Bootloader

The bootloader is the code that runs before the main application. As represented in Figure 5.46, its location is in the flash start. Therefore, the bootloader starts in address 0x08000000 and has the size of

0x1F80 bytes (0x2000 minus the configuration page) as represented in its scatter file in Code 6.16.

```
    LR_IROM1 0x08000000 0x1F80  {       ; load region size_region
        ER_IROM1 0x08000000 0x1F80  {   ; load address = execution address
            *.o (RESET, +First)
            *(InRoot$$Sections)
            *(+RO)
        }
        RW_IRAM1 0x20000000 0x00005000  {  ; RW data
            .ANY (+RW +ZI)
        }
    }
```

Code 6.16: Bootloader scatter file.

The bootloader application calls the bootloader initialize function depicted in Code 6.17. This function read all bootloader's configuration variables. Checks if the magic key has been written. If it has, it resets the magic key and peripherals, set the MCU vector table to the new application address and jumps to the new application address. If it has not and the current application address is different from zero, it sets the vector table to the currently running application and jumps to it. If it equals 0, it sets the vector table to the application one start address (0x0802000).

```c
void bootloaderInit_v(void){
    uint32_t new_app_addr_u32;
    uint32_t current_app_addr_u32;
    uint32_t magic_key_u32;

    unlockFlash_v();
    new_app_addr_u32 = *(volatile uint32_t*)(NEW_APP_ADDR);
    current_app_addr_u32 = *(volatile uint32_t*)(CURRENT_APP_ADDR);
    magic_key_u32 = *(volatile uint32_t*)(MAGIC_KEY_ADDR);
    lockFlash_v();

    if(magic_key_u32 == MAGIC_KEY){
        /* Jump to New App */
        resetMagicKey_v(new_app_addr_u32);

        resetPeripherals_v();

        SCB->VTOR = new_app_addr_u32;
        boot_jump(new_app_addr_u32);
    }
    if(current_app_addr_u32 != 0){
        /* Jump to the current app */
        SCB->VTOR = current_app_addr_u32;
        boot_jump(current_app_addr_u32);
    }
    else{
        /* Jump to default first app */
        SCB->VTOR = APP1_START;
        boot_jump(APP1_START);
    }
}
```

Code 6.17: Bootloader initialize function.

The application jump is performed by the assembly function represented in Code 6.18. It loads the first function parameter (application address) to R1, moves it to the stack pointer, then loads into R0 the program counter of the new application and branches to it. Notice that in the Cortex M0+ architecture, the stack pointer and the program counter are located in the firmware's first and second addresses, respectively.

```
    EXPORT  boot_jump
boot_jump
   LDR R1, [R0]
   MOV SP, R1               ;Load new stack pointer address
   LDR R0, [R0, #4]         ;Load new program counter address
   BX R0
```

Code 6.18: Bootloader jump function.

As mentioned before, the application implementation source code was not presented, and only the most exotic implementations were shown. The following section reveals the helper source developed, which helped with the software's implementation and testing.

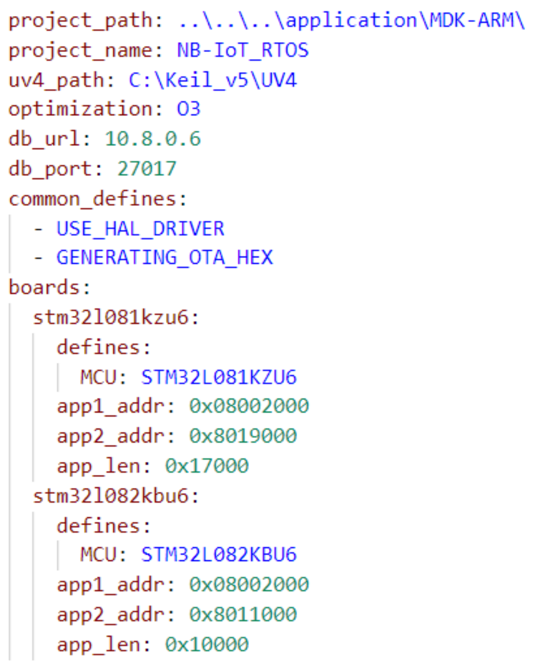
6.2.3 Helper Scripts

This section will present helper scripts that were used in the development phase. These scripts are used to compile the application for multiple board devices, generate the encrypted cloud header used in compile-time, and a System Trace EEPROM parser in a human-readable manner.

Compilation Script

Due to the component shortage, two different MCUs were used during the development of the Link4S solution. Consequently, these MCUs models have different flash memories sizes and layouts. Hence, to generate the code and then the delta efficiently, the compilation script was developed. This script compiles the application changing it according to a configuration file saving the generated binaries in the database. After having all board binaries, it generates the delta of the current version with one given by the user.

The configuration file of the compilation script is depicted in Figure 6.4. It comprises the project and Keil paths, the database connection configuration, and common defines used in all boards. It has a list of boards with the "stm32l081kzu6" and "stm32l082kbu6" configurations. According to the board, there is specific defines and OTA configurations.



```yaml
project_path: ..\..\..\application\MDK-ARM\
project_name: NB-IoT_RTOS
uv4_path: C:\Keil_v5\UV4
optimization: O3
db_url: 10.8.0.6
db_port: 27017
common_defines:
  - USE_HAL_DRIVER
  - GENERATING_OTA_HEX
boards:
  stm32l081kzu6:
    defines:
      MCU: STM32L081KZU6
    app1_addr: 0x08002000
    app2_addr: 0x8019000
    app_len: 0x17000
  stm32l082kbu6:
    defines:
      MCU: STM32L082KBU6
    app1_addr: 0x08002000
    app2_addr: 0x8011000
    app_len: 0x10000
```

Figure 6.4: Compile script configuration.

The script uses the class Project that changes the Keil project file and the MCU's scatter file for each board. The constructor within the class Project (represented in Code 6.19) starts by reading the YAML configuration file, the current Keil project, and backs it up; finds the software version in the application codebase and the build type. Afterwards, it sets the configuration file optimization for the Keil project. Finally, it tries to initialize the Link4S database, and if it succeeds, it uses it. The project backup is performed to restore to the initial state at the end of the script execution.

```python
class Project:
    def __init__(self):
        self.config = yaml.load(path.join(self.dirname, 'config.yaml'))
        self.boards = self.config['boards']

        self.proj = self.readKeilProject()
        self.proj_backup = self.proj

        self.find_software_version()
        self.find_build_type()
        self.full_version = \
         f"{self.version}_{self.build_type}_{self.config['optimization']}"

        self.setOptimization(self.config['optimization'])

        try:
           db.init((self.config['db_url'],self.config['db_port']),"NB-IoT")
           self.use_database = True
        except:
           self.use_database = False
```

Code 6.19: Project compile constructor.

All boards file generation is performed by the generate function represented in Code 6.20. It starts by deleting the firmware in the destination folder and setting the project scatter file path to the one used in the script. Then, it generates the firmware for both application positions by calling the compile function. In the end, if the database is being used, it saves the firmware generated into the database.

```python
def generate(self):
    self.delete_bin_folder_content()
    self.setScatterFilePath()

    for board in self.boards:
        start_addr = self.boards[board]['app1_addr']
        app_len = self.boards[board]['app_len']
        self.compile(board, start_addr, app_len)

        start_addr = self.boards[board]['app2_addr']
        self.compile(board, start_addr, app_len)

    if self.use_database:
        self.save_to_database()
    return

def compile(self, board, start_addr, app_len):
    self.generate_scatter_file(start_addr, app_len)
    self.setDefines(self.boards[board]['defines'])
    self.saveKeilProject()

    system(f"{path.join(self.config['uv4_path'],'UV4.exe')} -b {self.
        proj_abspath}")

    self.copy_bin_hex_files()
    self.rename_files(f"{self.full_version}_{hex(start_addr)}_{'MCU'}")
    return
```

Code 6.20: Project generate and compile functions.

The compile function generates the MCU's specific scatter file and sets its defines. Next, it calls the Keil compilation command-line command. After the build, the generated files are copied and renamed according to the MCU's configuration. The delta generation is possible by joining above class with the DeltaGen algorithm explained in Section 6.1.

Static Encryption Header Generation

A script was created to generate the static encryption cloud headers arrays used in compile-time. The user sets the microservice opcode, AES key, IV parameters, and generates the cloud header array and encrypts it, finalizing printing the respective C header array. This script's main code is represented in Code 6.21. The Crypto class has the AES and RSA encryption functions that wrap the python Crypto package.

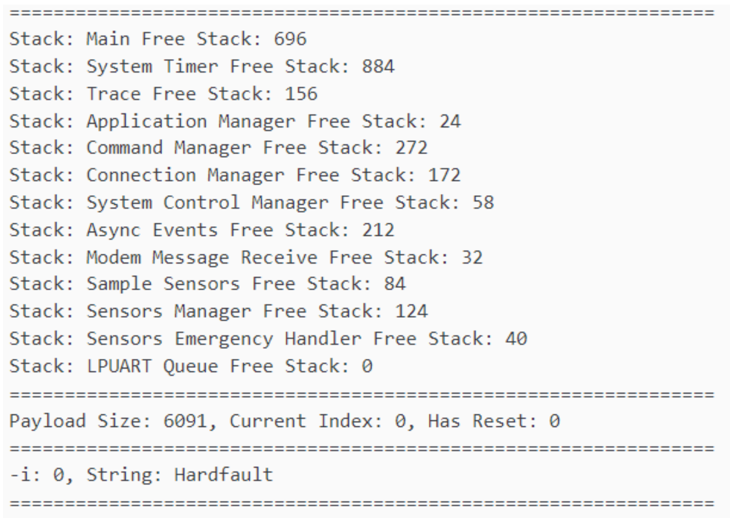
```python
def main():
    crypto = Crypto()

    microservice = "9b41650f8148c71a"
    aes_key = bytearray([0x85,0x89,0x44,...])
    aes_iv =  bytearray([0x4A,0xB8,0x48,...])

    cloud_header = [microservice, aes_key, aes_iv]

    cloud_header = msgpack.packb(cloud_header)

    encrypted_data = crypto.rsa_encrypt(cloud_header)

    print(convert_to_c_declaration(encrypted_data))
```
Code 6.21: Static encryption cloud header generation script.

The message pack is used to pack the cloud header to be as expected in the cloud. The result is a C array within the RSA encrypted header for the respective microservice, AES key, and IV given, as illustrated in Code 6.22.

```
    {0xb, 0x11, 0x1b, 0xf8, 0x19, 0xb7, 0x7a, ...};
```
Code 6.22: C array header generation output example.

System Trace

The system trace is a script used for easier testing because it reads the EEPROM in an Intel Hex format file and parses it. The script parses the free stack size values, the thread's name that overflows, and finally, the log space. The script output of the EEPROM shown in a human-readable manner is illustrated in Figure 6.5.



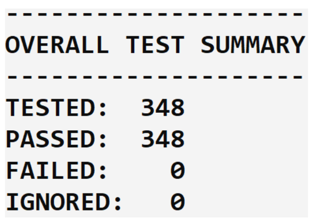
```
================================================================
Stack: Main Free Stack: 696
Stack: System Timer Free Stack: 884
Stack: Trace Free Stack: 156
Stack: Application Manager Free Stack: 24
Stack: Command Manager Free Stack: 272
Stack: Connection Manager Free Stack: 172
Stack: System Control Manager Free Stack: 58
Stack: Async Events Free Stack: 212
Stack: Modem Message Receive Free Stack: 32
Stack: Sample Sensors Free Stack: 84
Stack: Sensors Manager Free Stack: 124
Stack: Sensors Emergency Handler Free Stack: 40
Stack: LPUART Queue Free Stack: 0
================================================================
Payload Size: 6091, Current Index: 0, Has Reset: 0
================================================================
-i: 0, String: Hardfault
================================================================
```

Figure 6.5: Output of system trace python script.

6.2.4 Testing

The modules were tested using the Ceedling framework and were used the static analyzer Cppcheck. The board logs data to the EEPROM when running using the Trace module using the System Trace script. This section will be started by showing the test summary and two unit tests as an example, and finally, the unit tests code coverage and the results of the Cppcheck.

The test summary is represented in Figure 6.6. It performed 348 unit tests in the code base.



```
--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:   348
PASSED:   348
FAILED:     0
IGNORED:    0
```

Figure 6.6: Unit tests summary.

Code 6.23 shows one test of the System Control function `fitDatetimeInTime` when the seconds are equal to 60, and it is 28 of February in a not leap year. The expected results is a datetime with time 00:00:00 and date 1/03/2021 as the test asserts checks.

```c
void test_feb_not_leap_year_day29_to_1(void){
    datetime_st datetime = {
        .hour_u8 = 23,
        .min_u8 = 59,
        .sec_u8 = 60,
        .wday_u8 = 1,
        .day_u8 = 28,
        .mon_u8 = 2,
        .year_u8 = 21
        };

    _fitDatetimeInTime_v(&datetime);

    TEST_ASSERT_EQUAL_UINT8(0, datetime.hour_u8);
    TEST_ASSERT_EQUAL_UINT8(0, datetime.min_u8);
    TEST_ASSERT_EQUAL_UINT8(0, datetime.sec_u8);
    TEST_ASSERT_EQUAL_UINT8(1, datetime.day_u8);
    TEST_ASSERT_EQUAL_UINT8(3, datetime.mon_u8);
    TEST_ASSERT_EQUAL_UINT8(21, datetime.year_u8);
}
```

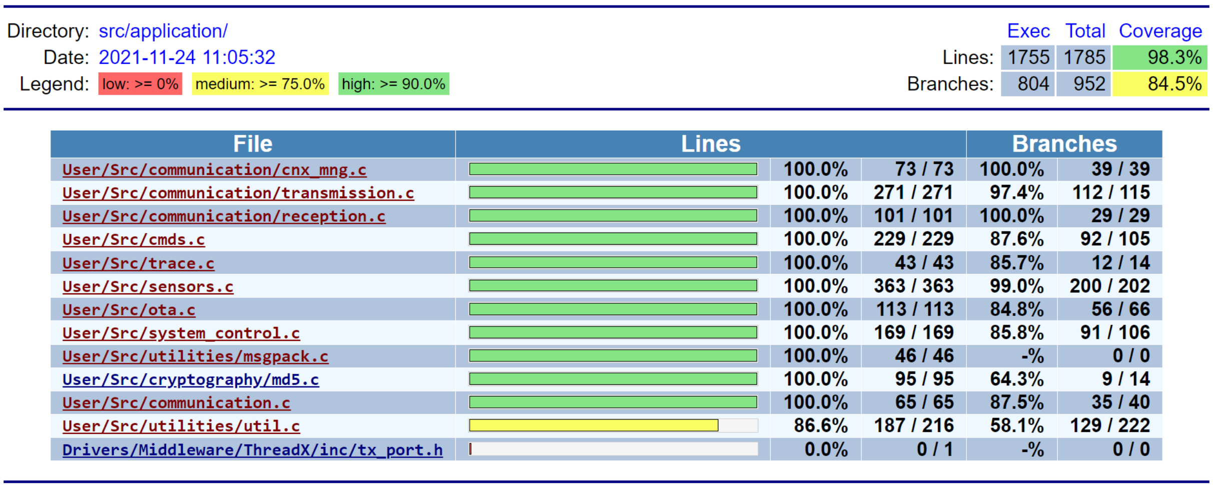
Code 6.23: Unit test of the fitDatetimeInTime function.

The previous test does not require function mocks since there are no external dependencies. The unit test in Code 6.24 uses the `eraseFlash`, `flashWord`, and ThreadX event flag set functions. Therefore, these functions are mocked, setting what they are expected to receive as parameters and its return value. Finally, when the `endOTAUpdateState` function is called, it asserts its return value and the function calls.

```c
void test_flash_in_app1(void)
{
    eraseFlash_e_ExpectAndReturn(CONFIGS_PAGE_ADDR, 1 , success);

    flashWord_e_ExpectAndReturn(BOOTLOADER_NEW_APP_ADDR, OTA_APP2_ADDR,
                                success);

    flashWord_e_ExpectAndReturn(BOOTLOADER_MAGIC_KEY_ADDR, OTA_MAGIC_KEY,
                                success);

    _txe_event_flags_set_IgnoreAndReturn(TX_SUCCESS);

    TEST_ASSERT_EQUAL_UINT8(S_WAIT_FOR_RESTART, endOTAUpdateState(1,NULL,2)
    );
}
```

Code 6.24: Unit test for the OTA end state.

The unit test coverage result is illustrated in Figure 6.7. The core modules have 100% coverage resulting in cleaner code with fewer bugs.

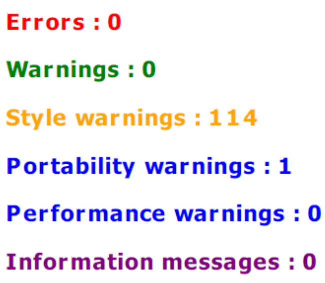


Directory:	src/application/		Exec	Total	Coverage
Date:	2021-11-24 11:05:32	Lines:	1755	1785	98.3%
Legend:	low: >= 0% medium: >= 75.0% high: >= 90.0%	Branches:	804	952	84.5%

File	Lines			Branches	
User/Src/communication/cnx_mng.c	🟩	100.0%	73 / 73	100.0%	39 / 39
User/Src/communication/transmission.c	🟩	100.0%	271 / 271	97.4%	112 / 115
User/Src/communication/reception.c	🟩	100.0%	101 / 101	100.0%	29 / 29
User/Src/cmds.c	🟩	100.0%	229 / 229	87.6%	92 / 105
User/Src/trace.c	🟩	100.0%	43 / 43	85.7%	12 / 14
User/Src/sensors.c	🟩	100.0%	363 / 363	99.0%	200 / 202
User/Src/ota.c	🟩	100.0%	113 / 113	84.8%	56 / 66
User/Src/system_control.c	🟩	100.0%	169 / 169	85.8%	91 / 106
User/Src/utilities/msgpack.c	🟩	100.0%	46 / 46	-%	0 / 0
User/Src/cryptography/md5.c	🟩	100.0%	95 / 95	64.3%	9 / 14
User/Src/communication.c	🟩	100.0%	65 / 65	87.5%	35 / 40
User/Src/utilities/util.c	🟨	86.6%	187 / 216	58.1%	129 / 222
Drivers/Middleware/ThreadX/inc/tx_port.h	🟥	0.0%	0 / 1	-%	0 / 0

Figure 6.7: Unit tests coverage report.

It was used the Cppcheck with clang-tidy to static analyze the code. The output is represented in Figure 6.8. There are no critical errors and warnings, only style warnings and portability. The style warnings are not critical, and the portability is for the different operating systems like Windows and Linux.



```
Errors : 0
Warnings : 0
Style warnings : 114
Portability warnings : 1
Performance warnings : 0
Information messages : 0
```

Figure 6.8: Cppcheck report results.

6.2.5 Code Size

The application code size varies according to its compile-time configurations, as represented in Table 6.1. The default application has enabled the accelerometer, temperature, and humidity sensors with 204 samples' arrays each. Also, it has static encryption enabled. Table 6.1 has the number of bytes occupied in the flash and RAM. The flash memory comprises the code and Read-only (RO) memory.

Table 6.1: Application size metrics compiled with O3.

Application	**Code (bytes)**	**RO-data (bytes)**	**FLASH (bytes)**	**RAM (bytes)**
No Encryption	51172	388	51560	14 452
Run-time encryption	59100	7572	66672	19 740
Static-encryption	53004	868	53872	14708

The application without encryption has the smallest flash and RAM footprint with 51560 and 14452, respectively. The addition of the run-time encryption increases the footprint due to the RSA algorithm code and RO memory. Also, it requires a buffer size of 4096 bytes to perform the RSA, which is a big chunk of the memory RAM. Moreover, the encryption function uses approximately 1000 bytes to 256 without or with static encryption. As a result, the run-time encryption has 66672 and 19740 bytes of flash and ram, respectively. The static encryption footprint is close to the no encryption application, but it has the AES algorithm code and RAM structures. Also, the encrypted headers are saved in the flash RO memory. Therefore, it is bigger than the no encryption application. The resulting memory footprint is 53872 and 14708 bytes of flash and RAM, respectively.

6.3 Cloud

The cloud comprises several messages parse microservices, the board's command response, and the commands module. This section will present the microservices using the Blackwing framework, the command response, the commands, and helper scripts.

6.3.1 Microservices

The Blackwing framework server receives the message and forwards it to the respective microservice. The microservice consists of a handler that receives the message, which in Blackwing this message is called "letter". This handler code is represented in 6.25. In the configuration microservice, this function starts by initializing the NB-IoT database. Then it calls the parse function of the config class. After parsing the message, it calls the response function with the respective database handle, board id and the microservice name. The response function will be explained in Section 6.3.2, but it is responsible for querying the available commands to send back to the board.

```
class ParseConfigHandler(BlackWingHandler):
    def attendRequest(self):
            db.init(database_credentials, "NB-IoT")
            config.parse(self.letter)
            self.response = response(db, config.board_id, 'Configuration')
        except Exception as e:
            self.response = None
```

Code 6.25: Blackwing config microservice handler.

The parse configuration function was implemented using the class Config. This class has the parse function illustrated in Code 6.27. This function receives the payload from the board similar to the array represented in Code 6.26 and then pairs the received message. The pair joins the string identifier with the following values, as an example, the identifier 'ID' with the number "268031902005789", and the identifier "ALARMS" with the array of values "1,1,0,0".

```
['P', 1, 1, 'ID', 268031902005789, 'TIME', 1635259412,
 'SWVER', '3.0.19_dev_03', 'OTA_APP1', 134225920, 'OTA_APP2', 134320128,
 'OTA_NOW', 134225920, 'MCU', 'STM32L081KZ', 'MCONF', '...',
 'MP_VCONF', b'...', ..., 'ALARMS', 1, 1, 0, 0]
```

Code 6.26: Example of the configuration message array.

After pairing the payload, it queries the board configuration from the database, and each pair is post-processed. The post-processing depends on the special identifier characteristics. For example, the board id is renamed from "ID" to "board_id'" to prettier database insertion. Other identifiers required post parse, like alarms, the modem configuration, and identifiers that start with "MP_" or "OTA_". The "MP_" means that the message is packed with the message pack format and needs to be unpacked as represented in line 21 of Code 6.27.

With all the pairs post-processed, the current system timestamp is appended and updated the configuration if it already exists in the database or inserts a new one.

```python
 1 class Config:
 2     board_id = None
 3 
 4     @classmethod
 5     def parse(cls, payload):
 6         pair_list = cls._pairReceivedMessage(payload)
 7 
 8         # Get board id configuration
 9         ...
10 
11         for pair in pair_list[:]:
12             if pair[0] == 'ID':
13                 pair_list[i] = ('board_id' ,pair[1])
14                 cls.board_id = pair[1]
15             elif pair[0] == 'TIME':
16                 pair_list[i] = ('board_timestamp' ,pair[1])
17             elif pair[0] == 'MCONF':
18                 ...
19             ...
20             elif 'MP_' in pair[0]:
21                 pair_list[i]=(pair[0][3:].lower(),msgpack.unpackb(pair[1]))
22             else:
23                 pair_list[i] = (pair[0].lower() , pair[1])
24             i += 1
25 
26         pair_list.append(('system_timestamp', system_time_unix))
27 
28         # Save in the database, if already exists overwrites
29         db.update_one("Configurations", {"board_id": cls.board_id},
30                       payload_dict)
```

Code 6.27: Config message parse function.

The other microservices are similar to the configuration. However, they can have different post identifiers processing, and instead of updating the configuration database, new data is inserted into the respective database collection (emergency, regular, and command response).

6.3.2 Command Response

The command response function is responsible for querying the available commands to the respective board, prioritizing the OTA update command as depicted in Code 6.28.

```python
def response(db: db, board_id: int, microservice: str):
    # Find OTA command
    cmd = db.find_one('Commands', ...)

    if cmd:
        board_config = db.find_one("Configurations", ...)

        delta = db.find_one('Deltas', ...)
        if delta != None:
            package = delta[f"package"{board_config['ota']['pckg_i']}]
            datagram_packed = msgpack.packb(package)
        else:
            cmd = False

    else:
        # No OTA update, lets check for others commands
        cmd = db.find_one('Commands', ...)
        if cmd:
            datagram_packed = msgpack.packb(cmd['datagram'])
            db.delete_one('Commands', {"_id": cmd['_id']})

    if cmd:
        if cmd['md5']:
            # Adds MD5
            ...
        return cmd_datagram

    return None
```

Code 6.28: Command response function.

This function receives the MongoDB database pointer, the board id and the respective microservice name that is calling it. It starts by finding an OTA command in the Commands collection. If there is, it gets the board configuration and retrieves the respective delta from the Deltas database. If there is no OTA command, it queries other commands to the respective board. If it is found, it is deleted from the database. Finally, according to the command, if the MD5 is enabled, the MD5 hash is added and returns the command datagram. If there is not any command, `None` will be returned.

6.3.3 Commands

A command is formed using the Commands class. This class presents all the end-device set and get commands available—the command structure changes according to the command. For example, the set and get of the alarms and the auxiliary structures to configure the sensors' variables will be presented.

The set commands functions follow a similar structure. They start by appending the command opcode and the specific command data to the datagram array. Code 6.29 represents the set alarms command function. It starts by appending the command opcode and a byte array with the time interval mode and the respective alarms. In the array's last position is appended the number of the alarms. Finally, the `insertCommand` function is called. This function receives the board id, command datagram and description and inserts them into the Commands database.

```
def setAlarms(self, board_id:int, interval_mode:bool,
                hours:list, min:list, sec:list):
    datagram = list()
    datagram.append(self._getOpcodes('setAlarms'))

    alarms_len = max(len(hours), len(min), len(sec))

    if interval_mode:
        datagram.append(bytearray([1]))
    else:
        datagram.append(bytearray([0]))

    for i in range(alarms_len):
        datagram[1] += bytearray([hours[i], min[i], sec[i]])

    datagram.append(alarms_len)

    description = f'Set Interval Mode: '{interval_mode}
                  ', Hours: '{hours}', Min: '{min}', Sec '{sec}

    return self._insertCommand(board_id, datagram, description )
```

Code 6.29: Set alarms command function.

The get alarms function is illustrated in Code 6.30. This command only has the command get opcode, which is the `setAlarms` opcode with the MSB bit set.

```
def getAlarms(self, board_id:int):
    datagram = list()

    datagram.append(self._getOpcodes('setAlarms') | 0x8000)

    return self._insertCommand(board_id, datagram, f'Get Alarms')
```

Code 6.30: Get alarms command function.

Moreover, the VarID and SensorVariableConfig data classes were used to facilitate the variable configuration. The VarID identifies the variable, and the SensorVariableConfig abstract the variable configurations, having all the available variable's parameters that can be changed. These data classes are fed to the set and get functions involving sensors' variables.

The commands module provides an interface to control the end-device. It enables changing its operation during its lifetime and even altering its running software through OTA updates.

The following sections will present helper scripts developed to enchant the development phase. These scripts are the Check Lost Packages and Regular Transmission Data Exportation, which will be nextly shown.

6.3.4 Check Lost Packages

In order to follow the board's regular transmissions are sent successfully, the check lost packages script was developed. This script receives the board id and a time margin and periodically checks if the database has received the regular packages. If not, it signals to a log file. This script is illustrated in Figure 6.31. It starts by setting the process name according to the board id's last seven digits and entering in syncing mode. The sync mode periodically queries the database for new board configuration or regulars. If any is found, it returns the system receive timestamp. After syncing with the board, it gets the board configuration to get its transmission interval. This script only works with the board operating in the interval mode. Next, it sleeps for the regular interval, and when it wakes up, it enters an infinity loop where it searches for regulars in a time window according to the current time and the margin time. If no regular is found, it re-enters in the syncing mode.

```python
def main()
  # Parse args received
  ...

  setproctitle.setproctitle(f'NB-{str(board_id)[-7:]}')

  timestamp = sync_mode(board_id)

  board_config = db.find_one("Configurations", {"board_id":board_id})
  interval_sec = board_config['alarms']

  sleep((timestamp+interval_sec) - m_time.now() + margin_sec)

  while True:
    results=db.find(
              db_collection, (
                { "board_id": board_id ,
                  "$and" :
                  [{
                    "board_timestamp" : {"$gte":m_time.now()-2*margin_sec}
                  },
                  {
                    "board_timestamp" : {"$lte":m_time.now()+3600}
                  }]
                  },)
              )

      if results.count() == 0:
          # Package missed
          timestamp = sync_mode(board_id)
      else:
          timestamp = results[0]['system_timestamp']

      board_config = db.find_one("Configurations", {"board_id":board_id})
      sleep(timestamp + board_config['alarms'] + margin_sec - m_time.now())
```

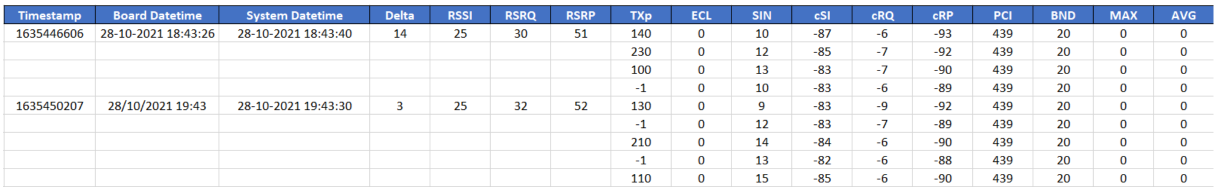
Code 6.31: Check lost packages script main function.

6.3.5 Regular Transmissions Data Exportation

Previously have been shown the check lost package script that helps monitor the board regular's transmissions. However, the regular transmissions data exportation script is used for the data required to be read from the database to be post-processed. This script parses the received regular database query and rearranges it into an excel file.

Since the same transmission can be divided into multiple parts, it starts by joining the regular transmission with the same board timestamp and finding all variables. If it has more than one, it starts by sorting the packages according to the package part and then joins all the samples. In both cases of having one or more than one, it identifies the variables, and if there is a new variable column is added.

As an example, a signal quality test run was made. Its sampled variables (TXp, ECL, among others) were exported, as is illustrated in Figure 6.9.



Timestamp	Board Datetime	System Datetime	Delta	RSSI	RSRQ	RSRP	TXp	ECL	SIN	cSI	cRQ	cRP	PCI	BND	MAX	AVG
1635446606	28-10-2021 18:43:26	28-10-2021 18:43:40	14	25	30	51	140	0	10	-87	-6	-93	439	20	0	0
							230	0	12	-85	-7	-92	439	20	0	0
							100	0	13	-83	-7	-90	439	20	0	0
							-1	0	10	-83	-6	-89	439	20	0	0
1635450207	28/10/2021 19:43	28-10-2021 19:43:30	3	25	32	52	130	0	9	-83	-9	-92	439	20	0	0
							-1	0	12	-83	-7	-89	439	20	0	0
							210	0	14	-84	-6	-90	439	20	0	0
							-1	0	13	-82	-6	-88	439	20	0	0
							110	0	15	-85	-6	-90	439	20	0	0

Figure 6.9: Exported data example.

Chapter 7: Results

This chapter shows the results of the power consumption of several application implementations. These implementations include the RTOS, run-time and static encryption, command response, emergencies and OTA updates.

The application uses three sensors (accelerometer, luminosity, and temperature/humidity). The tests implied an accelerometer sample rate of 5 seconds and the remaining sensors to 10 seconds. All sensors' variables except humidity were configured as simple and the humidity as average. The time interval between data communications to the cloud depends on the measure type. Then, to estimate the total current consumption, real application scenarios are presented.

Current consumption measurements have been made in two parts: the microcontroller and the sensors, the modem and the ATECC608A IC since the Printed Circuit Board (PCB) developed in the Link4S project supports it.

The setup used to measure the power consumption is represented in Figure 7.1. It was composed of the precision Source/Measure Unit (SMU) (Keysight B2901A [90]), an oscilloscope (Tektronix MDO3012 [91]) to record MCU GPIO toggles from the application, and the power supply MP710067. The software OpenChoiceDesktop from Tektronix was used to get the oscilloscope data. The Keysigh B2900 Quick IV Measurement Software was used to measure the end-device power consumption and Matlab to post-process the measured data, plot, and calculate the average current consumption.

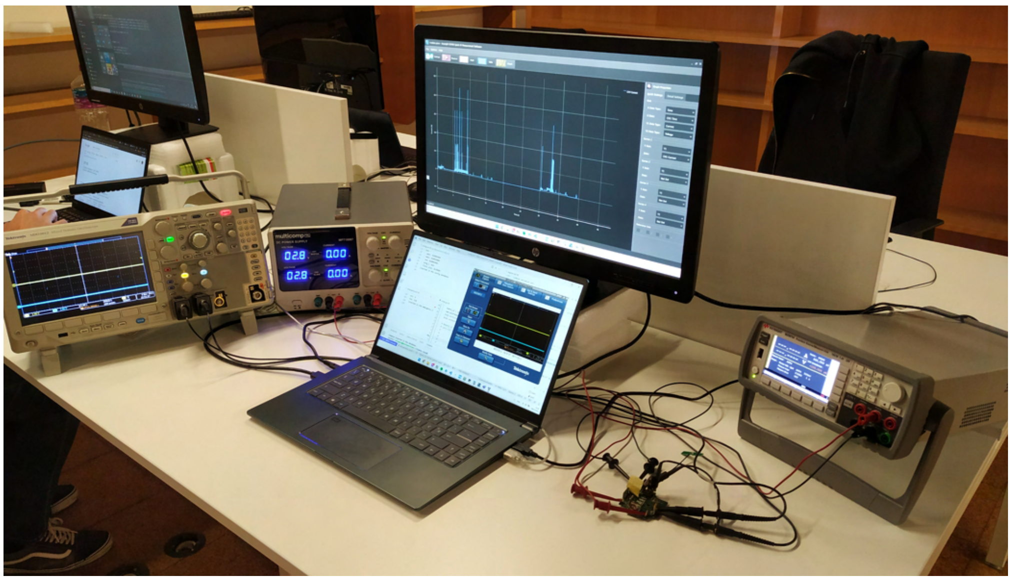


Figure 7.1: Measurements setup

7.1 Modular Power Consumption

Figure 7.2 represents the application MCU power consumption with a send interval of 30 seconds. In dashed orange line is the sensors' initialization and from the beginning to the first blue dashed line is the modem's initialization.

The second blue dashed line represents the transmission of the configuration message. Between the blue dashed lines is, in the beginning, the DateTime request from the system control that has the MCU entering in low-power mode waiting for the modem DateTime request and in the continuous power consumption the configuration datagram setup and transmission.

The pink dashed lines represent the sensors sample every 5 seconds, and between green dashed lines is the regular transmission. The regular transmission wake up time is 797 milliseconds.

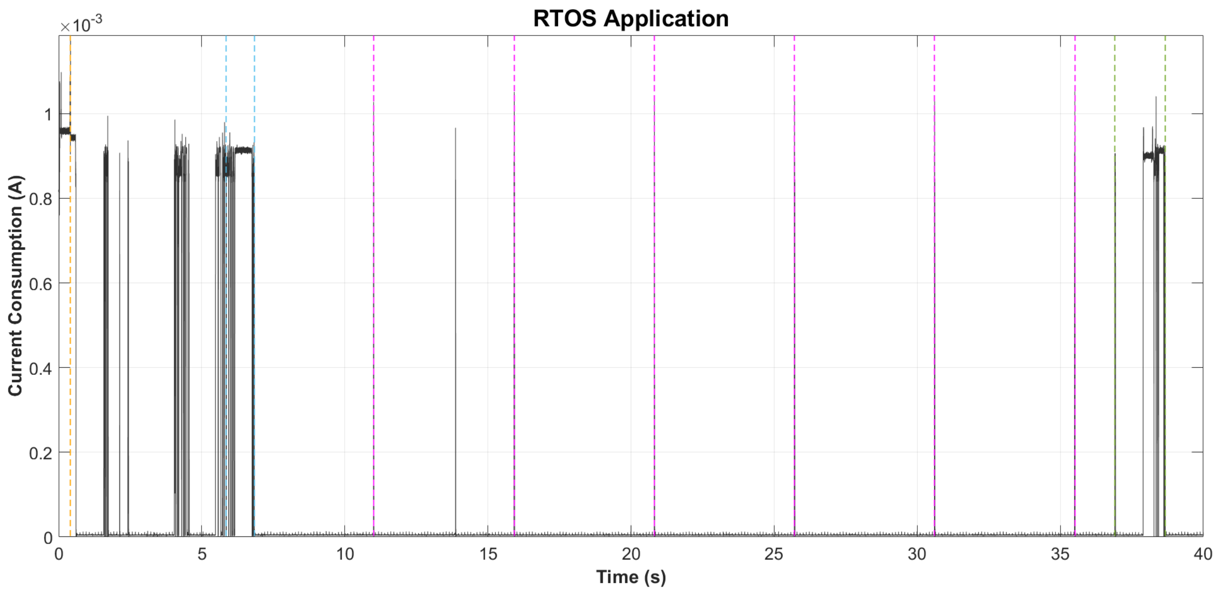


Figure 7.2: RTOS application power consumption.

The current spike at second 13.86 occurs when the modem enters its lower power consumption state (PSM). It occurs 6 seconds after the last transmission. Also, the MCU enters in low-power mode to save energy in the modem communications. Moreover, it can be seen in its initialization when it waits for the modem's network connection, server ping, and socket opening.

The following sections use the same 40 seconds measurement run but add the encryption using the same dashed lines colour meaning.

7.1.1 Encryption

This subsection will shown the run-time, static encryption and compare the ATECC608A IC and MCU engines AES encryption.

Run-Time Encryption

The run-time encryption adds the header and payload encryption using RSA and AES, respectively. The difference from the previous measurement run is that the configuration and regular transmissions take longer due to the encryption, as represented in Figure 7.3. The application with the previous run configuration increased the regular transmission time from 797 to 2499 milliseconds.

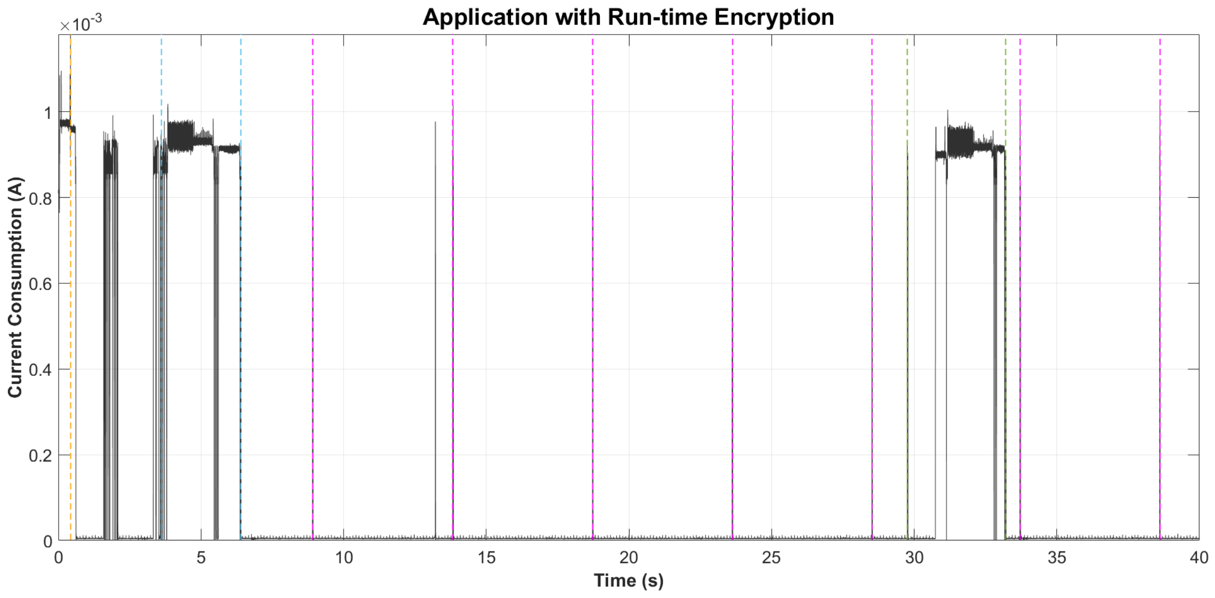


Figure 7.3: Application with run-time encryption power consumption.

Figure 7.4 represents the measurement of a regular transmission of 1024 bytes using run-time encryption. The RSA encryption occurs between the green dashed lines, and between the last green and blue dashed lines is the AES payload encryption.

In a 1024 bytes regular transmission, the RSA is a significant part of the transmission taking 1.59 seconds to encrypt the cloud header of 48 bytes, resulting in an encrypted header of 128 bytes long. The cloud header uses one more byte to be identified, remaining 895 bytes message payload. This message is encrypted using the AES and takes 22 milliseconds. As a result, the RSA takes 45 % of the 3.55 seconds transmission.

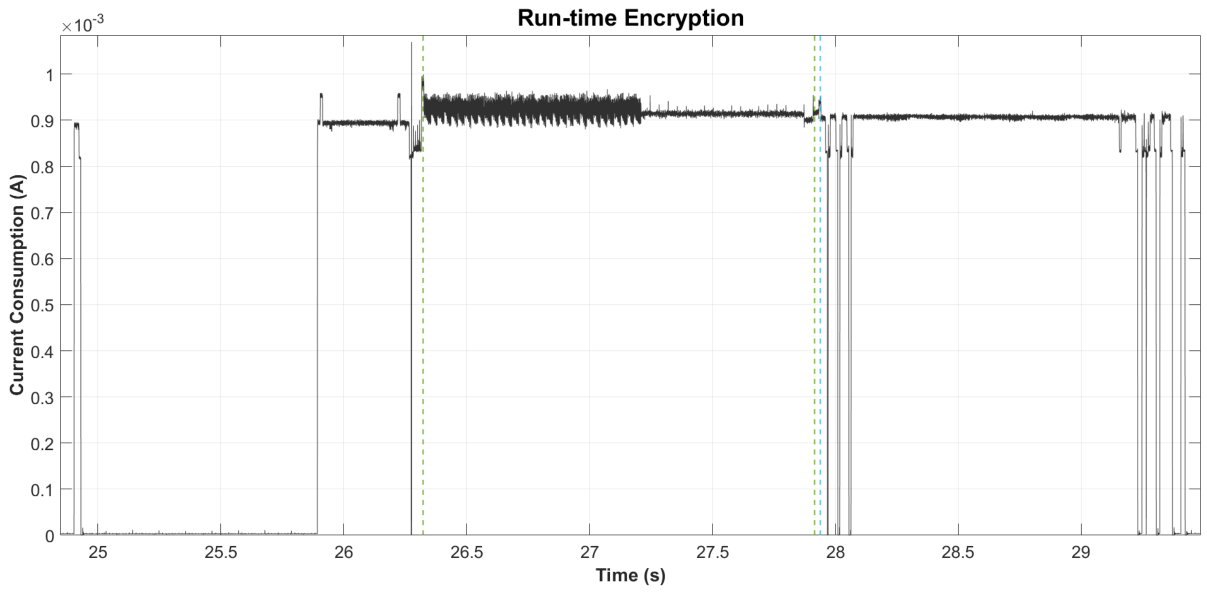


Figure 7.4: Regular transmission of 1024 bytes using run-time encryption.

Static Encryption

When the AES key does not change, the cloud header is constant. Since in the application it is constant, was added static encryption. The static encryption has in the flash memory hardcoded the cloud header, and only the payload AES encryption is performed.

The application power consumption in a 40-second run using static encryption is illustrated in Figure 7.5. It is similar to the one without encryption but increases the regular transmission time from 797 to 918 milliseconds. Compared to the run-time encryption, it reduces the time from 2499 to 918 milliseconds.

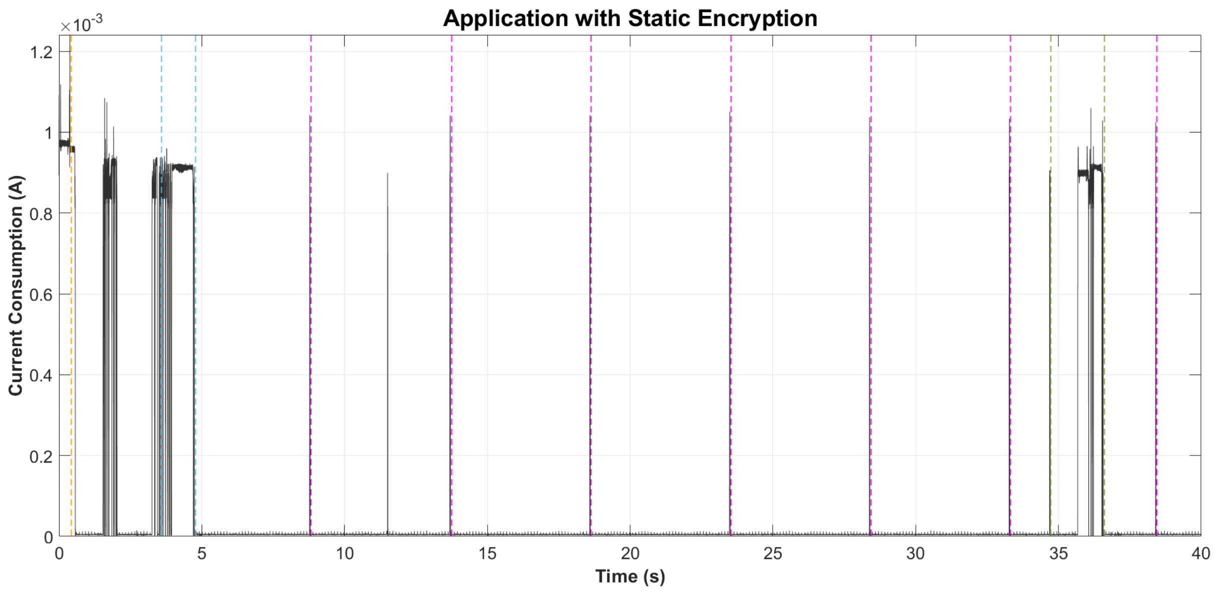


Figure 7.5: Application with static RSA encryption power consumption.

Figure 7.6 represents a 1024 bytes regular transmission measurement, where between the green and blue line, the AES encryption is represented. The AES power consumption is similar to the run-time. As a result, the regular transmission of 1024 bytes still has both encryptions, but the computing time was reduced to 2,21 seconds.

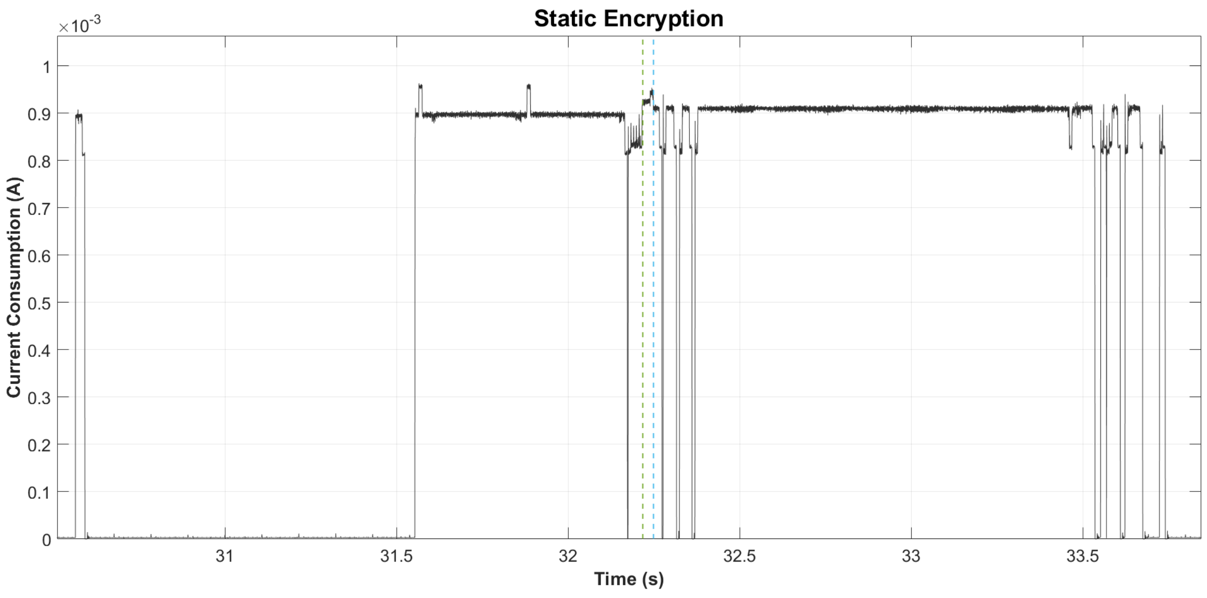


Figure 7.6: Static encryption.

This AES encryption is using the MCU encryption engines. The next subsection will be compared with the ATECC608A IC.

ATECC608A vs MCU AES Encryption

Figure 7.7 illustrates the current consumption of the MCU encryption on the left, between dashed orange lines; on the right, between purple dashed lines, the ATECC608A IC initialization and, between blue dashed lines, the encryption. Both encryptions are of 136 bytes. In the MCU, it took $1.76\ ms$ with an average current consumption of $934.2\ \mu A$, in contrast, the ATECC608A took $76.72\ ms$ due to the I2C communications overhead with an average current consumption of $1.02\ mA$. The ATECC608A took $236\ ms$ to initiate with an average power consumption of $1.34\ mA$.

In terms of current consumption, the AES MCU encryption engine consumes less and is faster because it does not have the I2C communication overhead. Also, it adds security, removing the data transmission in the I2C line.

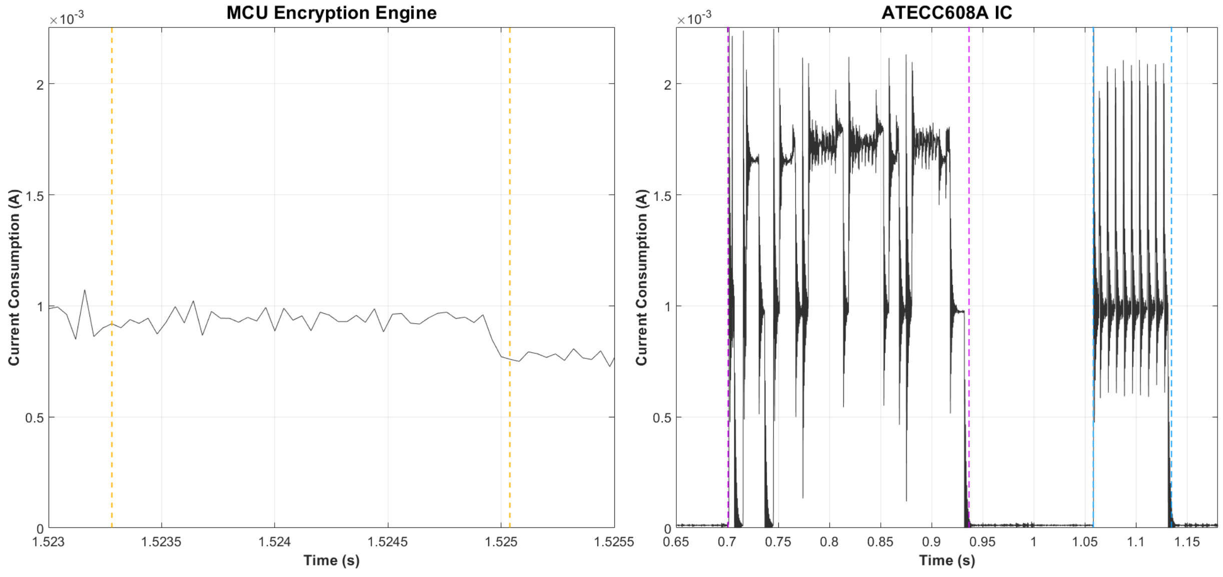


Figure 7.7: ATEC vs MCU encryption engine comparison.

7.1.2 Full System

The full system power consumption includes the MCU, sensors and modem power consumption, as represented in Figure 7.8. The regular transmission interval was set to 20 seconds to the modem power consumption fit in the 40 seconds run. From the figure can be noticed modem current spikes while initializing the modem; the DateTime request and the configuration transmission through the modem transmission current spikes. Nextly, the modem enters a stage that can receive data from the cloud and then enters in PSM, saving energy.

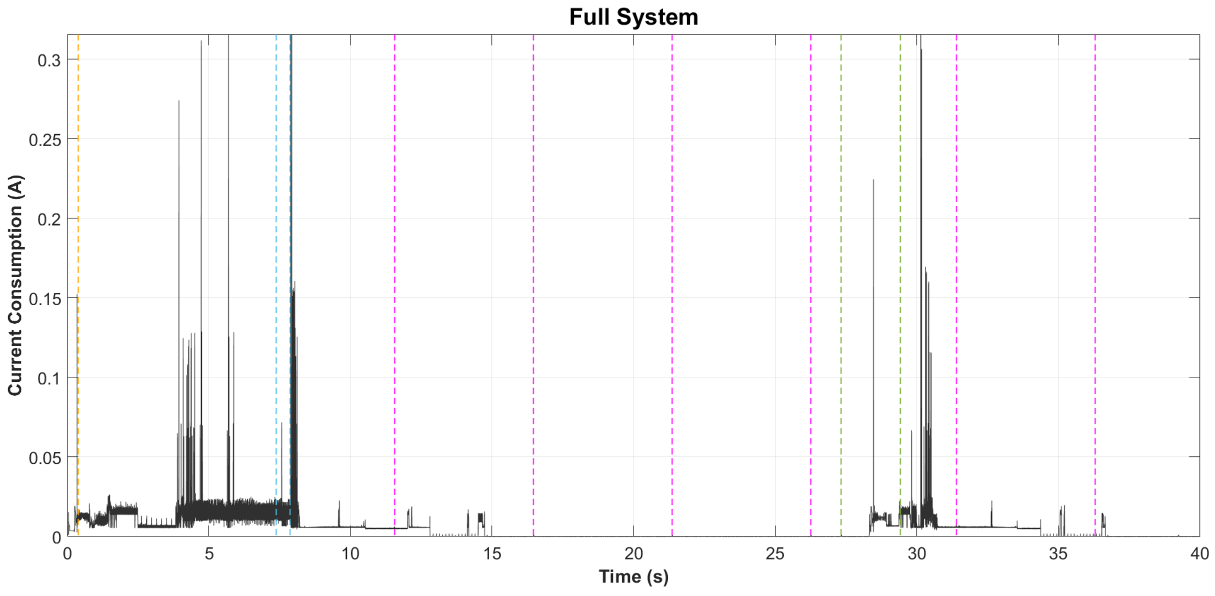


Figure 7.8: Full application power consumption.

In the regular transmission, the modem transmission current spikes can be seen. Notice that the MCU and sensors power consumption go unnoticed with the modem.

The typical application operation was described, from now on will be presented the emergencies, the command response, and the OTA results.

7.1.3 Emergencies

In order to exemplify how emergencies affect power consumption, Figure 7.9 presents, between blue and orange dashed lines, the trigger of the light and accelerometer emergencies, respectively.

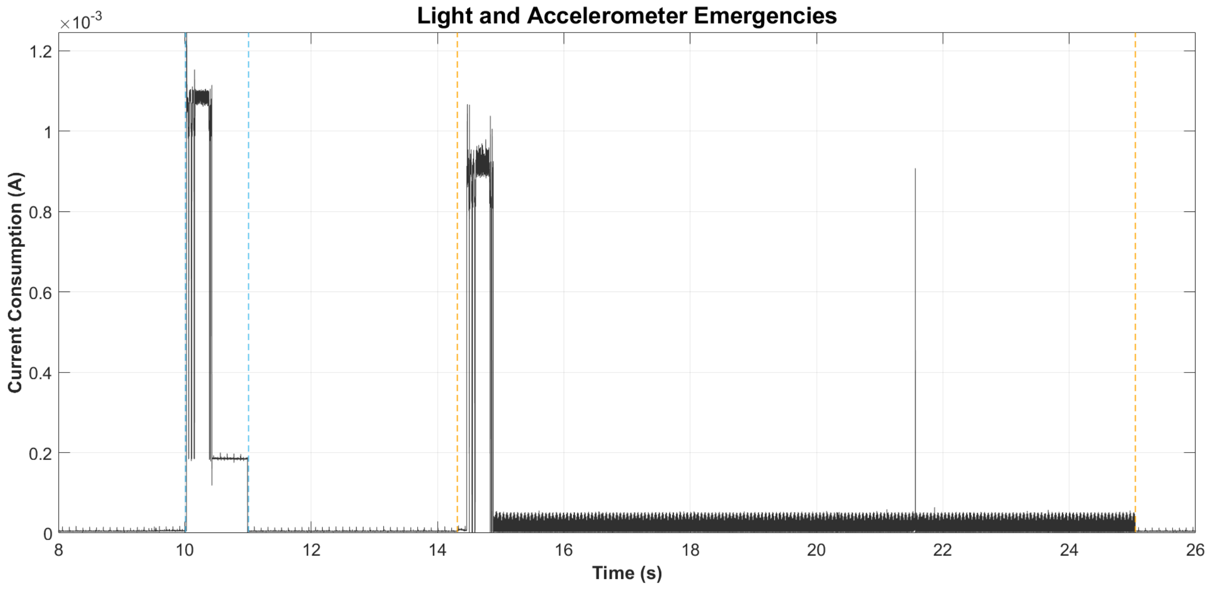


Figure 7.9: Light and accelerometer emergencies power consumption.

Both sensors signal the MCU that an emergency has been triggered, and it sends an emergency to the cloud.

When a light emergency is triggered, the sensor stays awake for 966 milliseconds and re-enters low-power mode. On the other hand, the accelerometer sensor, when triggered, returns to the low-power mode after 10 seconds without activity. As previously explained, the spike in the second 21.56 is due to the modem entering in PSM.

The temperature emergency is presented in Figure 7.10. The sensor wakes up at second 10.89, and then as the light and accelerometer, it wakes up the MCU and transmits to the cloud. The current spike in the second 25 is from the modem entering PSM.

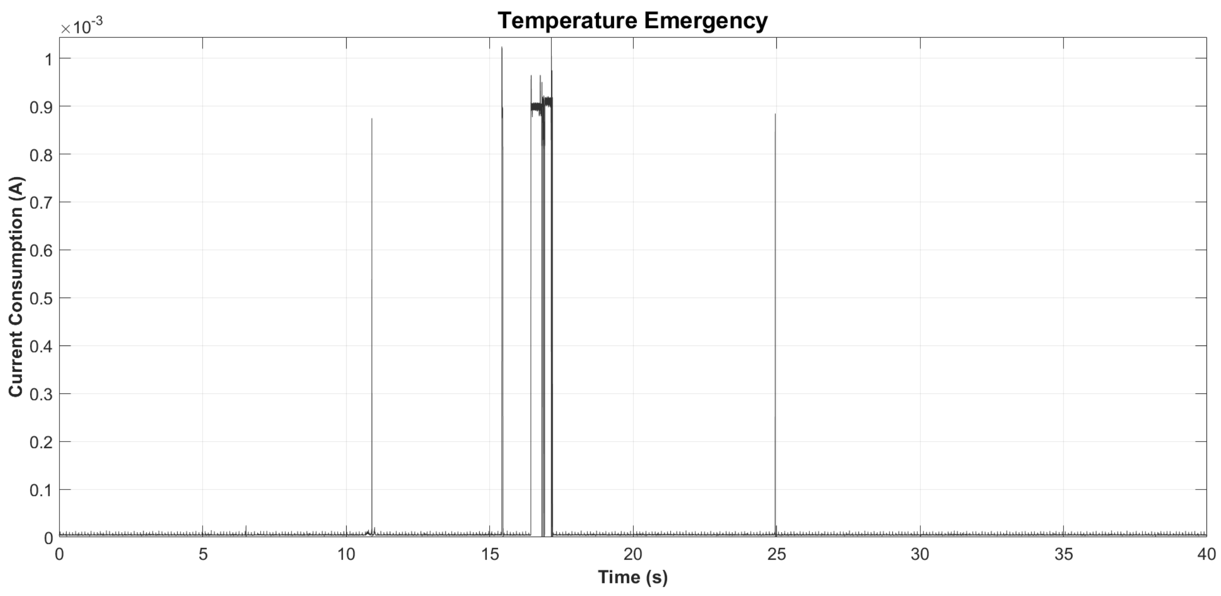
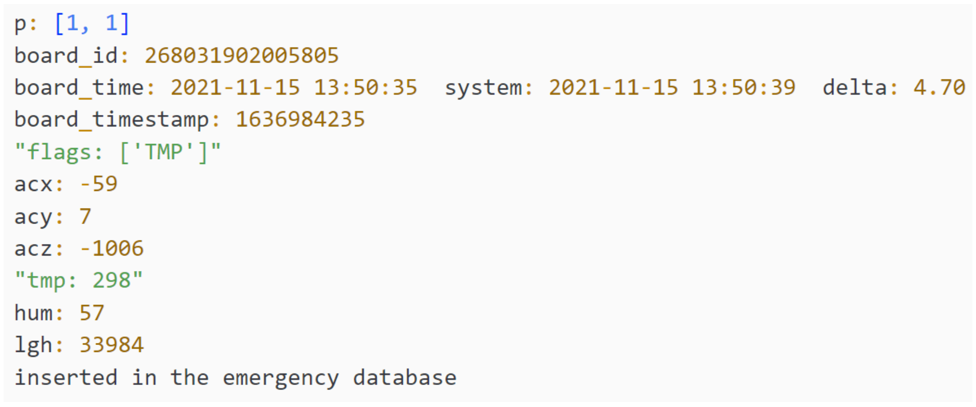


Figure 7.10: Temperature emergency power consumption.

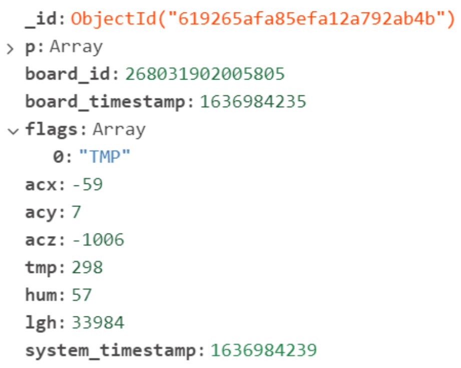
When an emergency is triggered, it is sent to the cloud server, and it is saved to the Emergency collection in the NB-IoT database. The server log for the temperature emergency in Figure 7.10 is illustrated in Figure 7.11. The temperature limit was set to 28 degrees celsius, and as can be seen, highlighted as green, the sensors' samples show a temperature of 29.8 with the triggered flag TMP.



```
p: [1, 1]
board_id: 268031902005805
board_time: 2021-11-15 13:50:35  system: 2021-11-15 13:50:39  delta: 4.70
board_timestamp: 1636984235
"flags: ['TMP']"
acx: -59
acy: 7
acz: -1006
"tmp: 298"
hum: 57
lgh: 33984
inserted in the emergency database
```

Figure 7.11: Temperature emergency server log.

The database saves the sensors samples, the respective triggered flags, and the system timestamp, as represented in Figure 7.12.



```
_id: ObjectId("619265afa85efa12a792ab4b")
> p: Array
board_id: 268031902005805
board_timestamp: 1636984235
v flags: Array
    0: "TMP"
acx: -59
acy: 7
acz: -1006
tmp: 298
hum: 57
lgh: 33984
system_timestamp: 1636984239
```

Figure 7.12: Temperature emergency database element.

7.1.4 Command Response

As an example of the command response, the default initial regular transmission interval 10 seconds. Then, when it sends the regular transmission, the command is sent to the MCU as represented between orange dashed lines in Figure 7.13. The command received sets the regular transmission to 15 seconds. Therefore, the regular transmission between green dashed lines is seen after 15 seconds of the command parse.

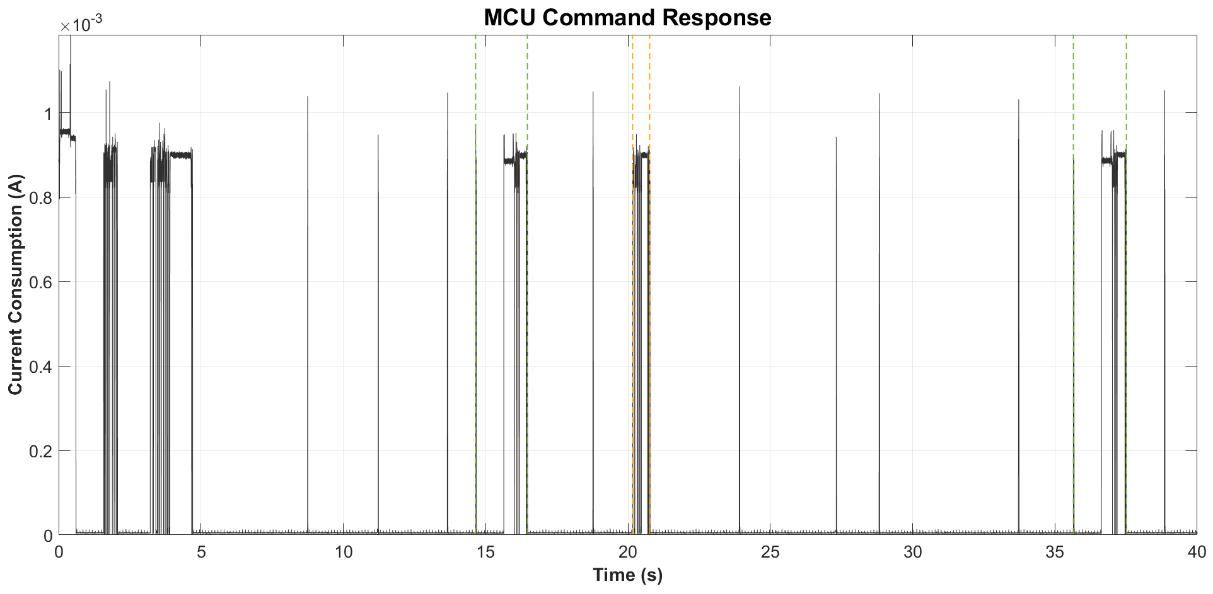


Figure 7.13: MCU set interval command power consumption.

The modem power consumption to the command response is represented in Figure 7.14. The first green dashed line illustrates when the regular transmission occurs, and the command response reception and transmission are seen in the orange dashed line. When a command is received, the MCU sends a command response. Therefore, it is seen power consumption similar to regular transmission. Moreover, the second green dashed line represents the second regular transmission power consumption.

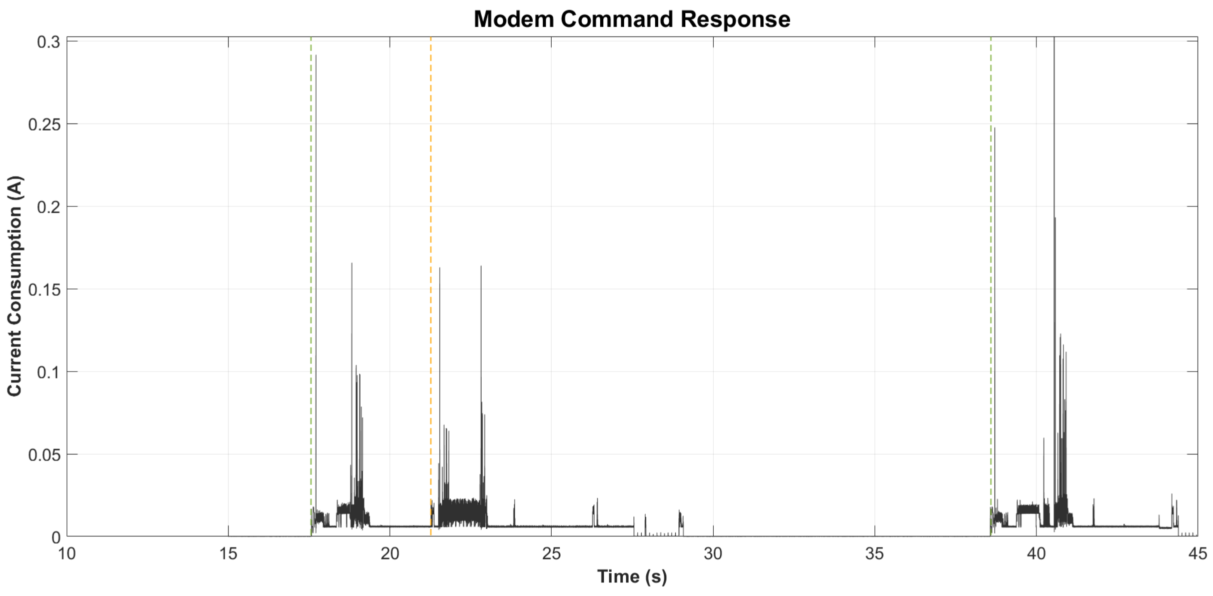
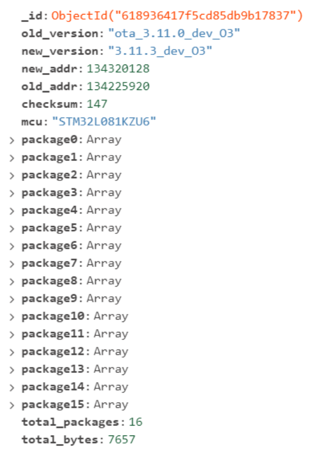


Figure 7.14: Modem set interval command power consumption.

7.1.5 OTA

In order to exemplify an OTA real scenario, the main application was modified, resulting in the application with software version "ota_3.11.0". This modified version has the regular transmission disabled after the first transmission, the sampling disabled, and the transmission interval set to 20 seconds.

The OTA update will upgrade the firmware from the adapted version to "3.11.3", which differences are represented in Appendix A.2. To summarize, the modem UART messages parse thread was fully reimplemented getting optimized; minor tweaks were made to fix bugs or typos; re-enables what was disabled in the OTA version; changes the version string. The delta commands datagram resulted in a total of 7657 bytes that represent 16 modem packages, as represented in Figure 7.15. The new software size is 52644 bytes, which were reduced to a delta of 7657 bytes.



```
  _id: ObjectId("618936417f5cd85db9b17837")
  old_version: "ota_3.11.0_dev_03"
  new_version: "3.11.3_dev_03"
  new_addr: 134320128
  old_addr: 134225920
  checksum: 147
  mcu: "STM32L081KZU6"
> package0: Array
> package1: Array
> package2: Array
> package3: Array
> package4: Array
> package5: Array
> package6: Array
> package7: Array
> package8: Array
> package9: Array
> package10: Array
> package11: Array
> package12: Array
> package13: Array
> package14: Array
> package15: Array
  total_packages: 16
  total_bytes: 7657
```

Figure 7.15: OTA delta in database.

Figure 7.16 illustrates all the MCU OTA power consumption. The MCU starts by initializing and, after 20 seconds, sends the first and only regular transmission highlighted between green dashed lines. When the cloud receives it, it sends the first OTA command, which MCU handle is in orange dashed lines. Then the board enters in a loop of handling the OTA command received and, when responding to the cloud, receives the next OTA package. Notice that the first handle is more extensive since it erases the new firmware memory and handles the first delta received. The higher current consumption is due to performing read and write flash operations.

Moreover, from the figure, the 16 packages can be seen. According to the copy command, the actuation length depends on how many bytes are copied from the old firmware version flash. When the device receives all the packages, it realizes the checksum and restarts running the new application.

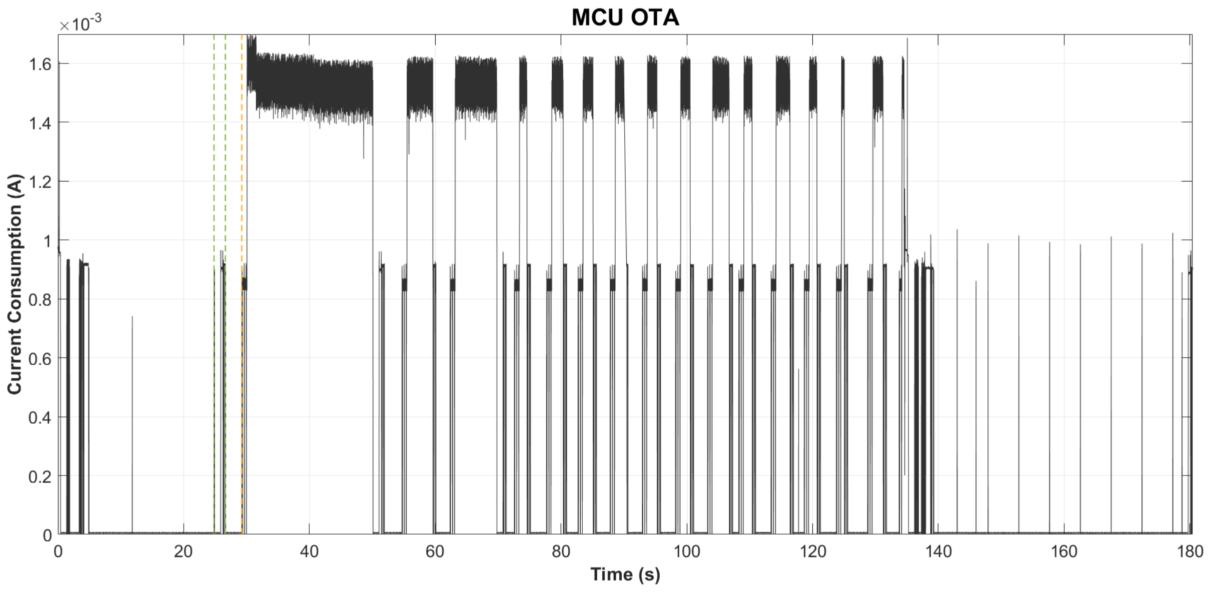


Figure 7.16: OTA upgrade power consumption.

Figure 7.17 illustrates the figure above zoomed-in at second 91 for better understanding. The command handling starts in the first blue dashed line, and between the green orange dashed lines is the OTA module writes and reads from and into the flash, respectively. Finally, from the second orange dashed line to the second blue one, it is the command response datagram preparation and transmission to the cloud.

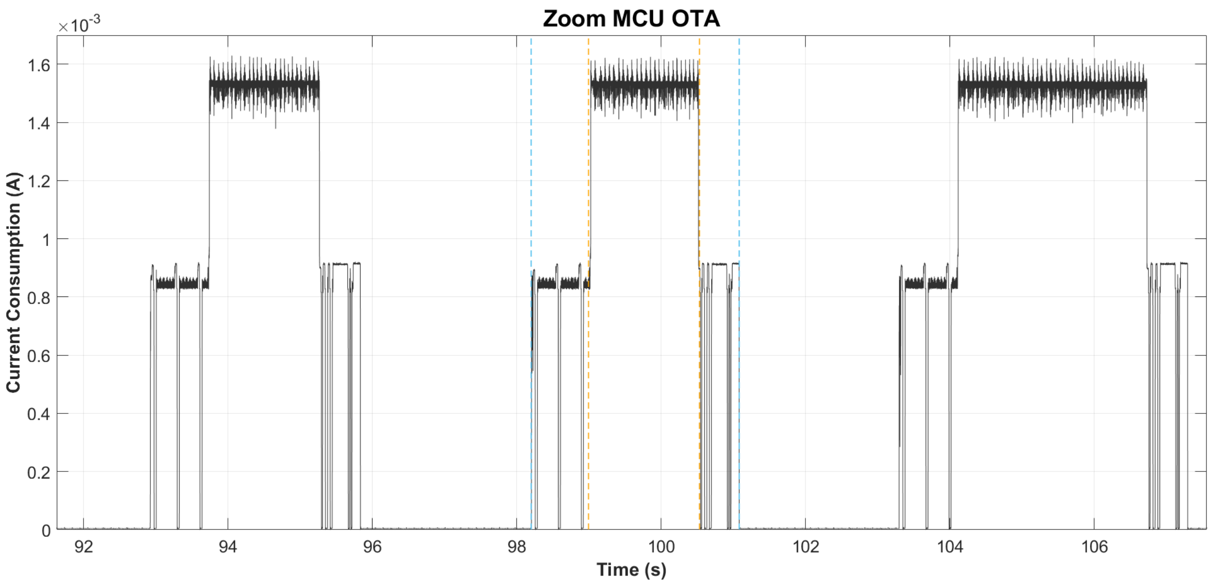


Figure 7.17: OTA upgrade package zoomed power consumption.

When all the packages are received and the new firmware is reconstructed successfully, the MCU restarts to the new application received. Figure 7.18 results in zooming at second 134 the Figure 7.16. It is after the MCU restart to run the new application received. It started by initializing the modem and sampling the sensors as explained previous, every 5 seconds, and after 40 seconds (default-test value) of the modem initialization, it sends a regular.

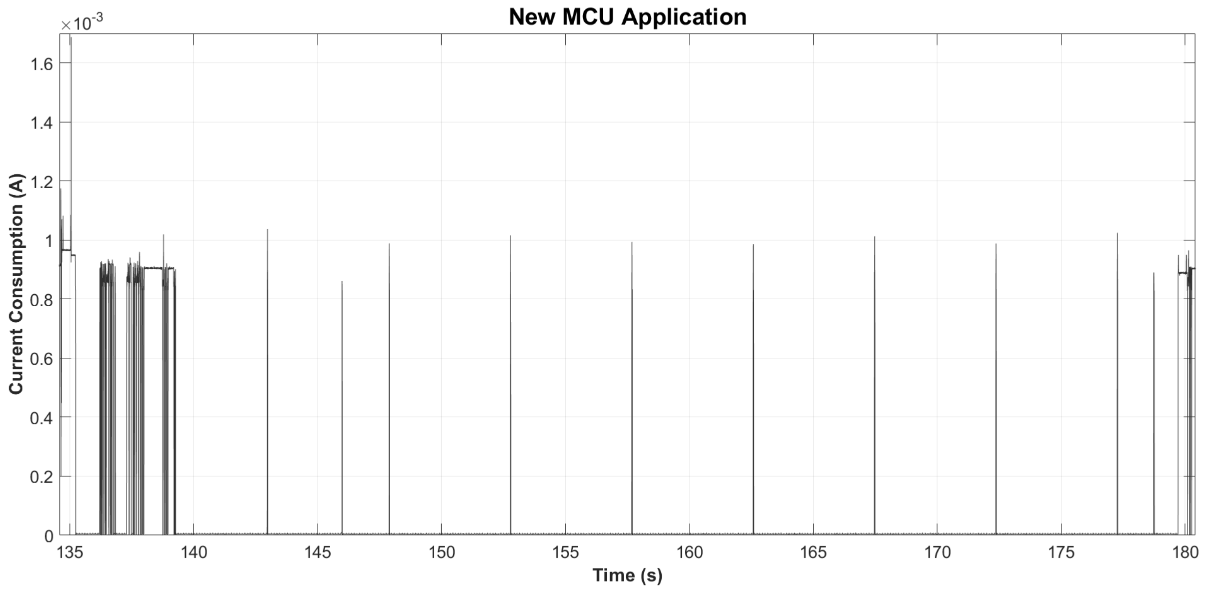


Figure 7.18: OTA upgrade new application power consumption.

The modem power consumption of the OTA update is illustrated in Figure 7.19. It starts with the modem initialization as previously seen, and then when the regular is sent highlighted from the green dashed lines. While the MCU handles the first OTA package, the modem enters in PSM, but when the command response is sent for the first command, it enters in a loop of sending and receiving. The last part of the modem power consumption is the MCU restart, system initialization and configuration transmission.

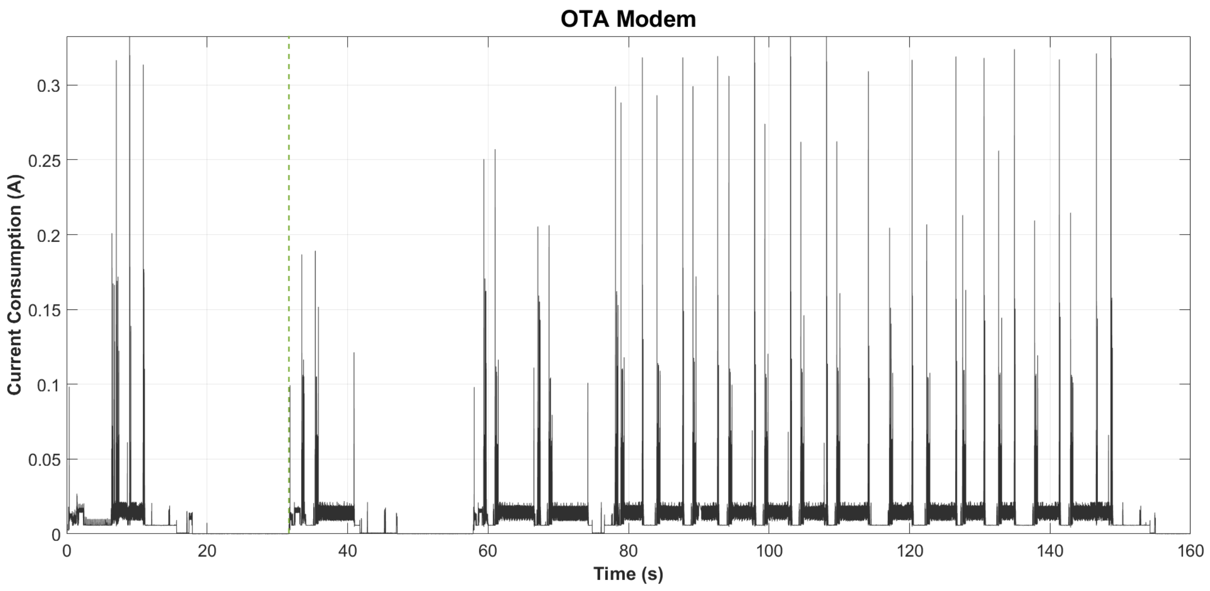
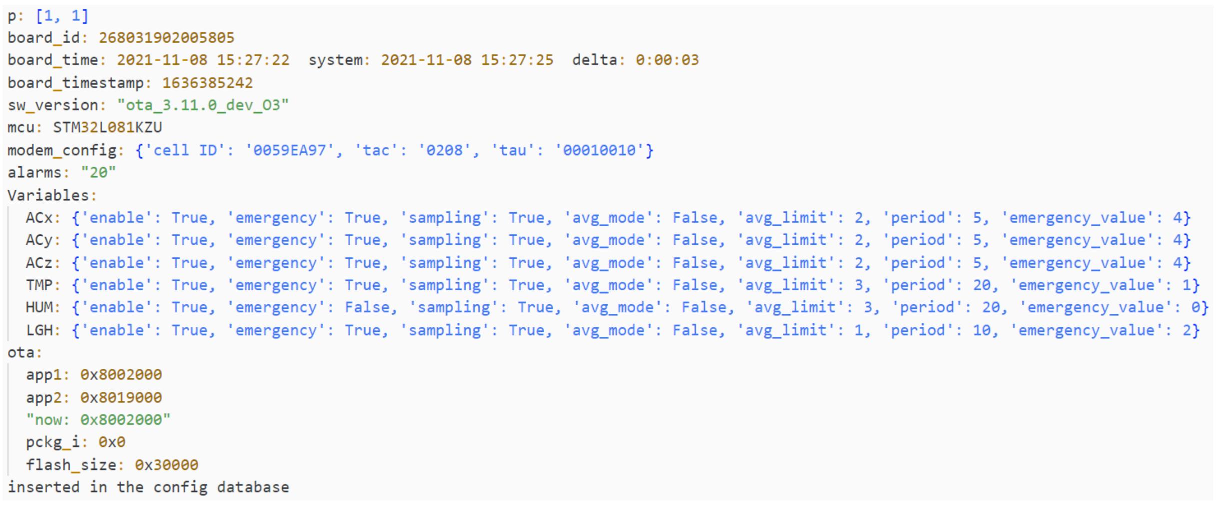


Figure 7.19: Modem OTA transmissions power consumption.

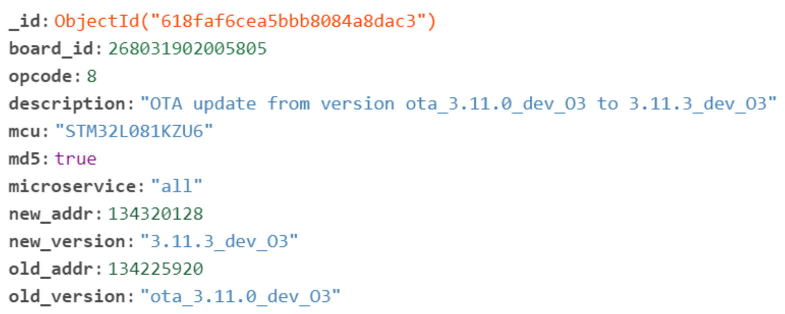
The OTA's end-device side was explained, now will be shown the cloud and database operations. The device started by sending the configuration represented in Figure 7.20. From this figure is highlighted in green the "ota_3.11.0_dev_O3" software version and in the OTA section the address for the current application running (now) "0x8002000".



```
p: [1, 1]
board_id: 268031902005805
board_time: 2021-11-08 15:27:22  system: 2021-11-08 15:27:25  delta: 0:00:03
board_timestamp: 1636385242
sw_version: "ota_3.11.0_dev_O3"
mcu: STM32L081KZU
modem_config: {'cell ID': '0059EA97', 'tac': '0208', 'tau': '00010010'}
alarms: "20"
Variables:
  ACx: {'enable': True, 'emergency': True, 'sampling': True, 'avg_mode': False, 'avg_limit': 2, 'period': 5, 'emergency_value': 4}
  ACy: {'enable': True, 'emergency': True, 'sampling': True, 'avg_mode': False, 'avg_limit': 2, 'period': 5, 'emergency_value': 4}
  ACz: {'enable': True, 'emergency': True, 'sampling': True, 'avg_mode': False, 'avg_limit': 2, 'period': 5, 'emergency_value': 4}
  TMP: {'enable': True, 'emergency': True, 'sampling': True, 'avg_mode': False, 'avg_limit': 3, 'period': 20, 'emergency_value': 1}
  HUM: {'enable': True, 'emergency': False, 'sampling': True, 'avg_mode': False, 'avg_limit': 3, 'period': 20, 'emergency_value': 0}
  LGH: {'enable': True, 'emergency': True, 'sampling': True, 'avg_mode': False, 'avg_limit': 1, 'period': 10, 'emergency_value': 2}
ota:
  app1: 0x8002000
  app2: 0x8019000
  "now: 0x8002000"
  pckg_i: 0x0
  flash_size: 0x30000
inserted in the config database
```

Figure 7.20: OTA old configuration server message.

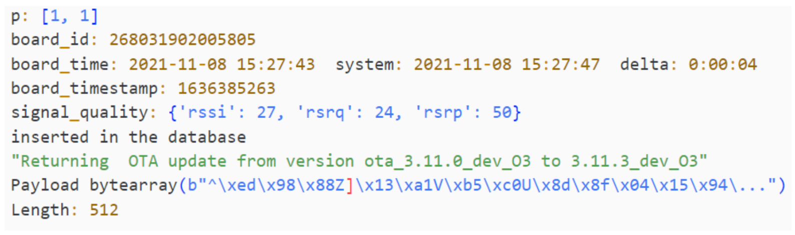
Afterwards, the OTA command to update to the version "3.11.3_dev_O3" is executed and inserted into the database, as illustrated in Figure 7.21. In the database is the command configuration with the MD5 enabled, the old and new application address, and the respective MCU, among others.



```
_id: ObjectId("618faf6cea5bbb8084a8dac3")
board_id: 268031902005805
opcode: 8
description: "OTA update from version ota_3.11.0_dev_03 to 3.11.3_dev_03"
mcu: "STM32L081KZU6"
md5: true
microservice: "all"
new_addr: 134320128
new_version: "3.11.3_dev_03"
old_addr: 134225920
old_version: "ota_3.11.0_dev_03"
```

Figure 7.21: OTA database command.

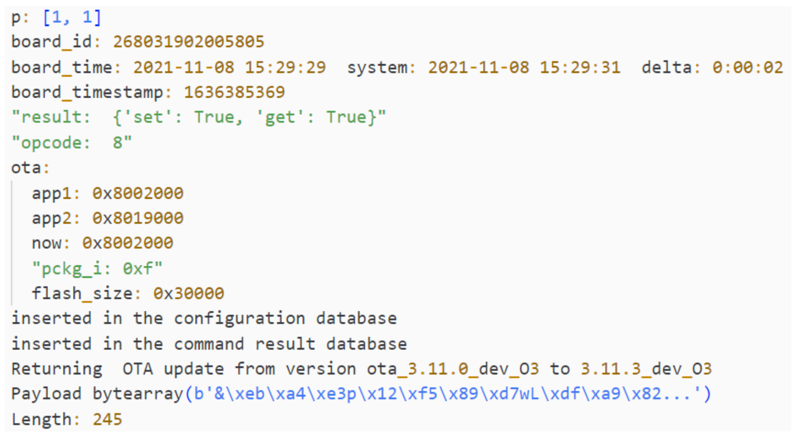
The command inserted in the database waits for the end-device communication with the cloud. Figure 7.22 shows the regular transmission where the OTA command is sent to the end-device, which description is highlighted as green.



```
p: [1, 1]
board_id: 268031902005805
board_time: 2021-11-08 15:27:43  system: 2021-11-08 15:27:47  delta: 0:00:04
board_timestamp: 1636385263
signal_quality: {'rssi': 27, 'rsrq': 24, 'rsrp': 50}
inserted in the database
"Returning  OTA update from version ota_3.11.0_dev_03 to 3.11.3_dev_03"
Payload bytearray(b"^\xed\x98\x88Z]\x13\xa1V\xb5\xc0U\x8d\x8f\x04\x15\x94\...")
Length: 512
```

Figure 7.22: OTA old regular server message.

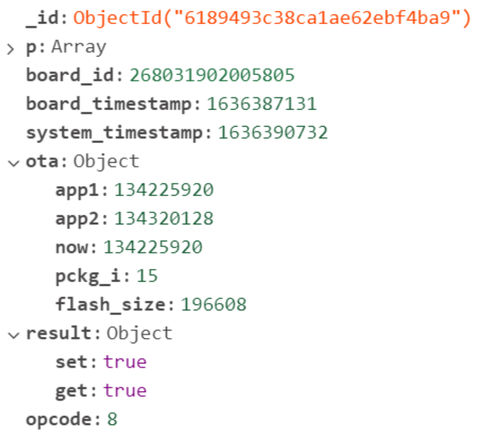
Afterwards, the end-device receives the command and answers to the command response microservice. When received in the cloud, it is sent back to the end-device the OTA next package. Figure 7.23 is the last command response. Since it has 16 packages, the last command response is the 15 package because the board receives the 16 packages and restarts being the command response the new software configuration.



```
p: [1, 1]
board_id: 268031902005805
board_time: 2021-11-08 15:29:29  system: 2021-11-08 15:29:31  delta: 0:00:02
board_timestamp: 1636385369
"result:  {'set': True, 'get': True}"
"opcode:  8"
ota:
  app1: 0x8002000
  app2: 0x8019000
  now: 0x8002000
  "pckg_i: 0xf"
  flash_size: 0x30000
inserted in the configuration database
inserted in the command result database
Returning  OTA update from version ota_3.11.0_dev_03 to 3.11.3_dev_03
Payload bytearray(b'&\xeb\xa4\xe3p\x12\xf5\x89\xd7wL\xdf\xa9\x82...')
Length: 245
```

Figure 7.23: OTA last command response server message.

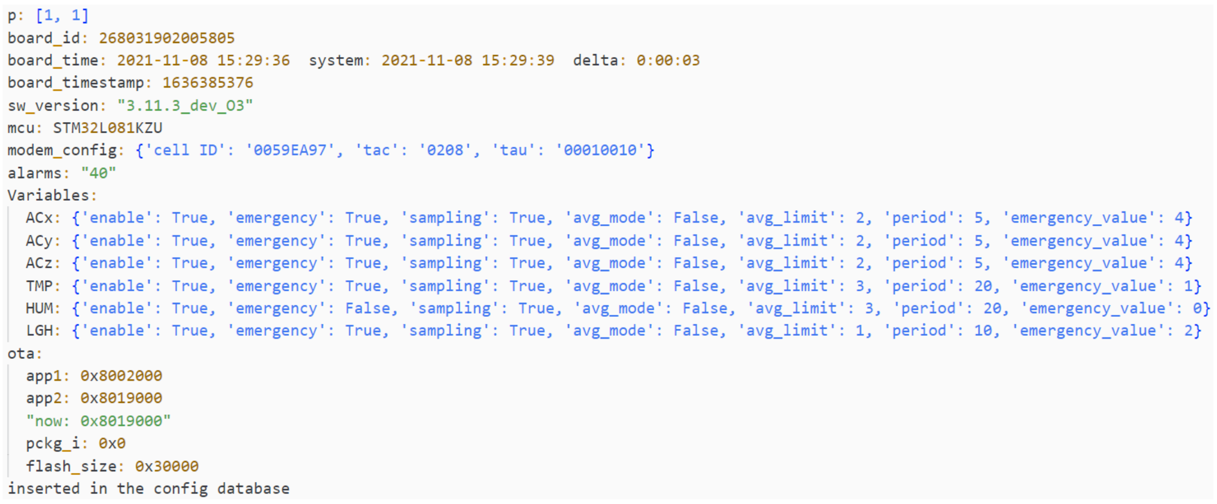
The command response is saved in the database, as represented in Figure 7.24. This response has the respective timestamps, OTA configuration, command result and opcode. In this case, the set and get command were both successful.



```
  _id: ObjectId("6189493c38ca1ae62ebf4ba9")
> p: Array
  board_id: 268031902005805
  board_timestamp: 1636387131
  system_timestamp: 1636390732
v ota: Object
    app1: 134225920
    app2: 134320128
    now: 134225920
    pckg_i: 15
    flash_size: 196608
v result: Object
    set: true
    get: true
  opcode: 8
```

Figure 7.24: OTA last command response in the database.

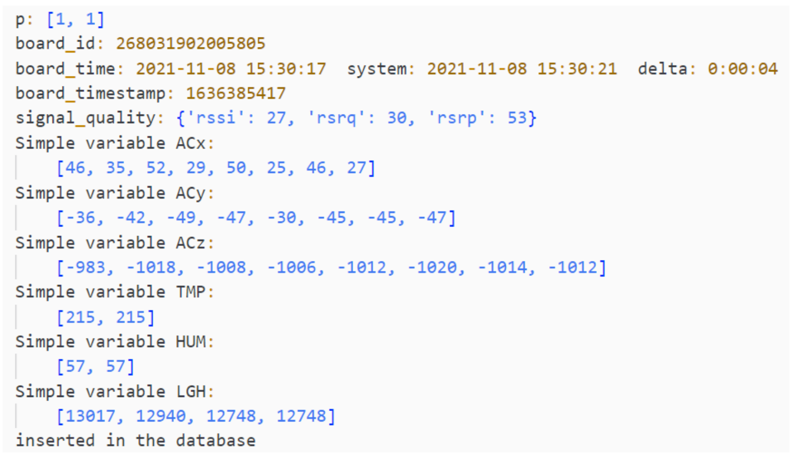
Since the board received all the packages and the checksum matched, it restarted as explained previously. Therefore, it sends the configuration running the new firmware, as illustrated in Figure 7.25. This figure highlighted in green is the new software version “3.11.3_dev_O3” with the current application running address “0x8019000”, the second application area. When a configuration is received and the software version does not match the sent in the previous board’s configuration, the OTA command is deleted from the database.



```
p: [1, 1]
board_id: 268031902005805
board_time: 2021-11-08 15:29:36  system: 2021-11-08 15:29:39  delta: 0:00:03
board_timestamp: 1636385376
sw_version: "3.11.3_dev_O3"
mcu: STM32L081KZU
modem_config: {'cell ID': '0059EA97', 'tac': '0208', 'tau': '00010010'}
alarms: "40"
Variables:
  ACx: {'enable': True, 'emergency': True, 'sampling': True, 'avg_mode': False, 'avg_limit': 2, 'period': 5, 'emergency_value': 4}
  ACy: {'enable': True, 'emergency': True, 'sampling': True, 'avg_mode': False, 'avg_limit': 2, 'period': 5, 'emergency_value': 4}
  ACz: {'enable': True, 'emergency': True, 'sampling': True, 'avg_mode': False, 'avg_limit': 2, 'period': 5, 'emergency_value': 4}
  TMP: {'enable': True, 'emergency': True, 'sampling': True, 'avg_mode': False, 'avg_limit': 3, 'period': 20, 'emergency_value': 1}
  HUM: {'enable': True, 'emergency': False, 'sampling': True, 'avg_mode': False, 'avg_limit': 3, 'period': 20, 'emergency_value': 0}
  LGH: {'enable': True, 'emergency': True, 'sampling': True, 'avg_mode': False, 'avg_limit': 1, 'period': 10, 'emergency_value': 2}
ota:
  app1: 0x8002000
  app2: 0x8019000
  "now: 0x8019000"
  pckg_i: 0x0
  flash_size: 0x30000
inserted in the config database
```

Figure 7.25: OTA new configuration server message.

From the configuration can be seen that the send interval is 40 seconds. Therefore, after 40 seconds, the new firmware sends the regular transmission with the sensors’ samples, as illustrated in Figure 7.26.



```
p: [1, 1]
board_id: 268031902005805
board_time: 2021-11-08 15:30:17  system: 2021-11-08 15:30:21  delta: 0:00:04
board_timestamp: 1636385417
signal_quality: {'rssi': 27, 'rsrq': 30, 'rsrp': 53}
Simple variable ACx:
    [46, 35, 52, 29, 50, 25, 46, 27]
Simple variable ACy:
    [-36, -42, -49, -47, -30, -45, -45, -47]
Simple variable ACz:
    [-983, -1018, -1008, -1006, -1012, -1020, -1014, -1012]
Simple variable TMP:
    [215, 215]
Simple variable HUM:
    [57, 57]
Simple variable LGH:
    [13017, 12940, 12748, 12748]
inserted in the database
```

Figure 7.26: OTA new regular server message.

The following section presents the application real-scenario power consumption and the memory occupied by the software with and without encryption.

7.2 Power Consumption Estimation

It is required to estimate how many Milliamps Hours (mAh) per day the system consumes to estimate the application power consumption. Therefore, it can be expressed as the sum of the average power consumption of the modem, the MCU & sensors, and the battery leakage as represented in Equation 7.1. Since the average power consumption depends on the application configuration, the math to calculate it will be generic. At the end of the section, will be presented application examples with the respective estimated power consumption and lifetime.

$$Avg_{total} = Avg_{modem} + Avg_{mcu\&sensors} + Avg_{leakage} \tag{7.1}$$

The modem average power consumption per day is given by the number of regular, emergency and command response transmissions consumption and the modem sleep. Resulting in:

$$Avg_{modem} = Avg_{regular} + Avg_{emegencies} + Avg_{cmd\_response} + Avg_{sleep} \tag{7.2}$$

In each of them is present an average current consumption and its duration. Afterwards, the resulting average current consumption is converted to an hour according to the number of times it occurs and the remaining time the modem is sleeping. The modem power consumption is the sum of each type of power consumption per day which is given by the average current consumption times the time per hour as illustrated in Table 7.1.

Table 7.1: Modem average current consumption.

Current Type	Average Current (mA)	Time (s)	Per Hour (h)	mAh/day per day
Regular Transmission	7,0242	9,296	$\frac{n_{regulars}*9,296}{3600}$	1 = (Average Current * Per Hour)
Emergencies Transmission	6,0273	7,7908	$\frac{n_{emergencies}*7,7908}{3600}$	2 = "
Command Response Transmission	7,189	7,8234	$\frac{n_{cmd\_responses}*7,8234}{3600}$	3 = "
Sleep	0,00395	-	Remaining time	4 = "
			Avg_{modem}	1 + 2 + 3 + 4

Table 7.2 represents the power consumption and time of the MCU and sensors in different moments. It was measured the 41 samples of each sensor and was made an average. Also, the MCU and sensors calculation is similar to the modem, but it has the sensors sampling and the accelerometer, light, and temperature emergencies. The sleep joins the MCU and sensors continuous sample that enables the emergencies.

Table 7.2: Sensors and MCU average current consumption.

Current Type	Average Current (mA)	Time(s)	Per hour (h)	mA/h per day
Sensors Sampling	0.872	0.0187	$\frac{n_{samples}*0.0187}{3600}$	1 = (Average Current * Per hour)
Regular Transmission	0.898	2.21	$\frac{n_{regulars}*2.21}{3600}$	2 = "
Command Response Transmission	0.871	0.61	$\frac{n_{cmd\_response}*0.61}{3600}$	3 = "
Accelerometer Emergencies	0.0388	10.74	$\frac{n_{acc\_emergencies}*10.74}{3600}$	4 = "
Light Emergencies	0.504	1.01	$\frac{n_{lght\_emergencies}*1.01}{3600}$	5 = "
Temperature Emergencies	0,63	1.1336	$\frac{n_{temp\_emergencies}*1.1336}{3600}$	6 = "
Sleep	0.003484	-	Remaining time	7 = "
			$Avg_{sensors\&mcu}$	1 + 2 + 3 + 4 + 5 + 6 + 7

The battery used is the Saft LM 17500 [84], which has a leakage of 1 %. The battery average leakage current is given by the battery capacity (C) in mAh, $I_L\%$ is the self-leakage percentage per year. The average leakage current consumption is represented in Equation 7.3. With the battery average leakage current consumption, the average power consumption results in **0.082192** mAh/day.

$$I_{leakage} = \frac{C \times I_L\%}{24 \times 365} \tag{7.3}$$

The OTA shown in Figures 7.16 and 7.19 power consumption is represented in Table 7.3. Since it was using the power consumption daily, the OTA is very asynchronous. Therefore, the used mAh from the OTA update will be calculated and substrated from the battery capacity.

Table 7.3: “ota_3.11.0_dev_O3” to “3.11.3_dev_O3” OTA power consumption.

Current Type	Average Current (mA)	Time (s)
MCU	1,347	70,51
Modem	12,896	112,16

The real capacity that will be used to estimate is given by Formula 7.4. It results in the battery capacity minus the power consumption of the MCU and modem in an hour multiplied by the number of OTAs using the values of Table 7.3.

$$C_{real} = C_{battery} - n_{OTAs} * \frac{(I_{MCU} \times T_{MCU} + I_{modem} \times T_{modem})}{3600} \tag{7.4}$$

With the system's daily average power consumption is possible to calculate the number of days that the system can run, dividing the real battery capacity by the daily average as represented in Formula 7.5.

$$days = \frac{C_{real}}{Avg_{daily}} \tag{7.5}$$

As an example real application use case, the temperature, humidity and light sensor will do 72 samples, each having 216 samples between transmission filling the datagram array in 6 hours. The accelerometer sensors will be still enabled to signal emergencies and will be performed ten OTAs. Per day 864 samples, four regular transmissions, 0.33 command response, 0.33 accelerometer, 0.33 light, and 0,16 temperature emergencies — results in a total of five temperature emergencies and ten accelerometer and light emergencies per month. In the lifetime, also having ten OTAs. Using the static encryption the Avg_{daily} equals **0.353** mAh/day resulting in 8477 days, **23** years. If run-time encryption is used instead of static encryption, the expected lifetime is reduced to 8442 days, 35 days less from static encryption. The system components weight on power consumption is represented in Figure 7.27. If it had ideal conditions where leakage does not exist, the expected operation time is **30** years.

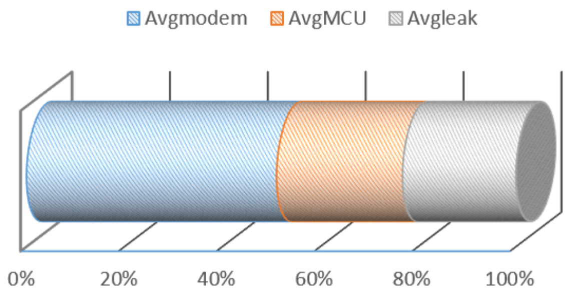


Figure 7.27: System components weight on power consumption with example 1.

Using as an example the proposed in [1], where it has 60 samples and two transmissions per day. It is expected without battery leakage a total of **36** years which is reduced to **26** due to the discharge rate of less than 1% per year.

Chapter 8: Conclusion

The increasing use of IoT devices for environmental monitoring or data acquisition to feed the machine-learning models has never been so important. Battery-powered devices with a 10-years lifetime are great to add to the current systems since they can acquire data without human intervention for several years.

The NB-IoT is one of the LPWAN technologies that allow reaching this lifetime. Since the NB-IoT uses the existing LTE service, it can seamless connect to network even in location with poor LTE coverage, such as, remote and underground locations. Therefore, human intervention or maintaining is costly when the end-devices are deployed. As a result, the software must work during the the battery life without physical-human intervention. Moreover, the ability to change the end-device software through Over-The-Air updates is nowadays mandatory. However, to reduce the end-device power consumption, adding functionalities, and fixing application bugs by unit testing and static analysers in the development phase are helpful to catch bugs earlier.

The end-devices have an upper cyber layer (cloud) that receives their data and can be used to control them. Therefore, the increasing number of devices requires scalable technology, such as the microservices architecture. This architecture allows easier architectural cloud changes letting continuous development and device control.

Regarding the work developed on this dissertation, one of the objectives was to re-design a bare-metal solution to an RTOS based one. In the re-design, new features were added like the ability to change run-time board parameters, such as sampling period and send interval through commands, a new sensor support architecture to optimise the memory use, a generic encrypted communication architecture and application error recovery.

The RSA and AES encryption were added in the communication layer resulting in end-to-end secure communications. The communications were secured using encryption at run-time and statically (only the AES is performed in run-time). When compared, it was possible to conclude that the run-time encryption increased the RAM and flash memory usage from 14452 and 51560 bytes to 19740 and 66672 bytes, respectively. On the other hand, the static encryption only increases to 53872 and 14708 bytes. Implementing the AES encryption throughout the ATECC608A IC and the MCU encryption engines were also compared. The latter was chosen because it has been proven faster and decreases the device's Bill of materials (BOM).

The modules were tested and validated in the development phase using unit tests, static analysis, and running the device in harsh signal quality conditions to expose bugs. To help the development, several helper scripts were made to simplify run-time debug and post transmission analysis. Since the development is continuous and in development, it is possible to catch several bugs, but the boards can still be deployed with software bugs. In order to be able to fix these bugs, Over-The-Air updates were added to the end-device.

The OTA updates were added by doing incremental updates, resulting in the board receiving a delta with commands to reconstruct the new firmware through the old one, receiving only the new things that are not common between both firmwares, resulting in OTA updates with reduced size. In order to do that it was implemented the DeltaGen differencing algorithm. This algorithm generates the delta using suffix arrays and having a byte-level comparison between firmwares.

All these features culminated into an end-device with a life expectancy using the use case of 864 samples per day, with four transmissions with an interval of 6 hours, receiving up to ten commands, accelerometer and light emergencies, five temperature emergencies with ten OTA updates. The board lifetime is expected to be **23 years** (taking into account the battery leakage). New features were added to the software, but its lifetime was extended from 17 years to 23 years compared to the first software version.

Alongside the end-device, cloud services were also developed that allow full control of all the devices. It resulted in a microservice architectural server with microservices responsible for parsing the specific board's messages. These microservices are responsible for parsing the messages and saving them into the database. A command module was developed that enables commands to be added to the database. After the microservices finishes parsing the board's message, these commands are queried by its board unique identification number. These commands enable changing the device behaviour in run-time and also perform OTA updates.

As a result, a well-designed, tested and validated end-device software application was developed that is expected to run for **23 years**. This device has a cloud to receive data, interact with the user and control the devices. Also, throughout the database it is possible to read the end-devices data and change the end-device at run-time with the commands module developed. If the application paradigm changes or bugs are found, it is possible to update Over-The-Air the end-device.

Furthermore, a full deployment of 100 units is expected until June. Up to now, 20 units have been deployed, and the results obtained proved that this dissertation reached its goal and objectives.

8.1 Future Work

The unit tests and version control could be extended and implemented in a remote server automatic test when new software commits are added, checking if new code additions can break the old code, adding Continuous Integration (CI) to the Link4S project.

The sensors storage architecture has been optimised, but the sampling period could be improved to utilise the RTOS timers. It can result in more memory (RAM) using the timers for each variable, but the sampling flexibility would increase.

Also, the HAL Wrapper is currently implemented using the STMicroelectronics' HAL, which can add more overhead. It could be changed to the STMicroelectronics' Low Level (LL), which is more MCU specific and closer to the MCU hardware.

A high-level application, such as a dashboard, could be developed to more accessible data view and device control using the commands module and database.

References

[1] S. Paiva, “Software/hardware co-design for nb-iot low-power applications: Consumption and performance analysis,” Master’s thesis, University of Minho, 2020.

[2] Netmanias, “LTE Security II: NAS and AS Security,” 2021. [Online; accessed 23-April-2021].

[3] M. . Rouse, “internet of things (iot),” 2020. [Online; accessed 26-November-2020].

[4] E. Brown, “21 open source projects for iot,” 2016. [Online; accessed 26-November-2020].

[5] ITU, “Internet of things global standards initiative,” 2015. [Online; accessed 26-November-2020].

[6] DrewAHendricks, “The trouble with the internet of things,” 2015. [Online; accessed 26-November-2020].

[7] J. Höller, V. Tsiatsis, C. Mulligan, S. Karnouskos, S. Avesand, and D. Boyle. Oxford: Academic Press, 2014.

[8] M. H. Miraz, M. Ali, P. S. Excell, and R. Picking, “A review on internet of things (iot), internet of everything (ioe) and internet of nano things (iont),” in *2015 Internet Technologies and Applications (ITA)*, pp. 219–224, 2015.

[9] M. Vega, “Internet of things statistics, facts & predictions,” 2020. [Online; accessed 27-November-2020].

[10] S. Paiva, S. Branco, and J. Cabral, “Design and power consumption analysis of a nb-iot end device for monitoring applications,” in *IECON 2020 The 46th Annual Conference of the IEEE Industrial Electronics Society*, pp. 2175–2182, 2020.

[11] S. Pinto, J. Cabral, and T. Gomes, “We-care: An iot-based health care system for elderly people,” in *2017 IEEE International Conference on Industrial Technology (ICIT)*, pp. 1378–1383, 2017.

[12] K. Gill, S. Yang, F. Yao, and X. Lu, “A zigbee-based home automation system,” *IEEE Transactions on Consumer Electronics*, vol. 55, no. 2, pp. 422–430, 2009.

[13] M. T. Lazarescu, "Design of a wsn platform for long-term environmental monitoring for iot applications," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 3, no. 1, pp. 45–54, 2013.

[14] S. K. Routray and S. Anand, "Narrowband iot for healthcare," in *2017 International Conference on Information Communication and Embedded Systems (ICICES)*, pp. 1–4, 2017.

[15] P. Chatterjee, L. J. Cymberknop, and R. L. Armentano, "Iot-based decision support system for intelligent healthcare — applied to cardiovascular diseases," in *2017 7th International Conference on Communication Systems and Network Technologies (CSNT)*, pp. 362–366, 2017.

[16] O. B. Mora, R. Rivera, V. M. Larios, J. R. Beltrán-Ramírez, R. Maciel, and A. Ochoa, "A use case in cybersecurity based in blockchain to deal with the security and privacy of citizens and smart cities cyberinfrastructures," in *2018 IEEE International Smart Cities Conference (ISC2)*, pp. 1–4, 2018.

[17] B. K. Barman, S. N. Yadav, S. Kumar, and S. Gope, "Iot based smart energy meter for efficient energy utilization in smart grid," in *2018 2nd International Conference on Power, Energy and Environment: Towards Smart Technology (ICEPE)*, pp. 1–5, 2018.

[18] R. A. Kjellby, L. R. Cenkeramaddi, A. Frøytlog, B. B. Lozano, J. Soumya, and M. Bhange, "Long-range self-powered iot devices for agriculture aquaponics based on multi-hop topology," in *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, pp. 545–549, 2019.

[19] C. Yi, H. Chen, and Y. Chen, "A smart meter design implemented with iot technology," in *2018 International Symposium on Computer, Consumer and Control (IS3C)*, pp. 360–363, 2018.

[20] X. Deng, Y. Jiang, L. T. Yang, L. Yi, J. Chen, Y. Liu, and X. Li, "Learning-automata-based confident information coverage barriers for smart ocean internet of things," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 9919–9929, 2020.

[21] R. Chow, "The last mile for iot privacy," *IEEE Security Privacy*, vol. 15, no. 6, pp. 73–76, 2017.

[22] L. Xing, "Reliability in internet of things: Current status and future perspectives," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 6704–6721, 2020.

[23] J. K. Reena and R. Parameswari, "A smart health care monitor system in iot based human activities of daily living: A review," in *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, pp. 446–448, 2019.

[24] M. Borges, S. Paiva, A. Santos, B. Gaspar, and J. Cabral, “Azure rtos threadx design for low-end nb-iot device,” in *2020 2nd International Conference on Societal Automation (SA)*, pp. 1–8, IEEE, 2021.

[25] A. L. (2017), “Iot cellular networks,” 2017. [Online; accessed 30-May-2021].

[26] BehrTech, “6 leading types of iot wireless tech and their best use cases,” 2021. [Online; accessed 30-May-2021].

[27] B. Nath, F. Reynolds, and R. Want, “Rfid technology and applications,” *IEEE Pervasive Computing*, vol. 5, no. 1, pp. 22–24, 2006.

[28] Y. Song, J. Lin, M. Tang, and S. Dong, “An internet of energy things based on wireless lpwan,” *Engineering*, vol. 3, no. 4, pp. 460–466, 2017.

[29] M. Centenaro, L. Vangelista, A. Zanella, and M. Zorzi, “Long-range communications in unlicensed bands: The rising stars in the iot and smart city scenarios,” *IEEE Wireless Communications*, vol. 23, no. 5, pp. 60–67, 2016.

[30] D. Patel and M. Won, “Experimental study on low power wide area networks (lpwan) for mobile internet of things,” in *2017 IEEE 85th Vehicular Technology Conference (VTC Spring)*, pp. 1–5, IEEE, 2017.

[31] U. Raza, P. Kulkarni, and M. Sooriyabandara, “Low power wide area networks: An overview,” *IEEE Communications Surveys & Tutorials*, vol. 19, no. 2, pp. 855–873, 2017.

[32] K. Mekki, E. Bajic, F. Chaxel, and F. Meyer, “Overview of cellular lpwan technologies for iot deployment: Sigfox, lorawan, and nb-iot,” in *2018 ieee international conference on pervasive computing and communications workshops (percom workshops)*, pp. 197–202, IEEE, 2018.

[33] L. Vangelista, A. Zanella, and M. Zorzi, “Long-range iot technologies: The dawn of lora™,” in *Future access enablers of ubiquitous and intelligent infrastructures*, pp. 51–58, Springer, 2015.

[34] Y.-P. E. Wang, X. Lin, A. Adhikary, A. Grovlen, Y. Sui, Y. Blankenship, J. Bergman, and H. S. Razaghi, “A primer on 3gpp narrowband internet of things,” *IEEE Communications Magazine*, vol. 55, no. 3, pp. 117–123, 2017.

[35] K. Kerliu, A. Ross, G. Tao, Z. Yun, Z. Shi, S. Han, and S. Zhou, "Secure over-the-air firmware updates for sensor networks," in *2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems Workshops (MASSW)*, pp. 97–100, 2019.

[36] Thales, "Is your fota solution efficient enough for lpwan?," 2015. [Online; accessed 2-January-2021].

[37] Y. Wee and T. Kim, "A new code compression method for fota," *IEEE Transactions on Consumer Electronics*, vol. 56, no. 4, pp. 2350–2354, 2010.

[38] R. K. Panta, S. Bagchi, and S. P. Midkiff, "Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation," in *Proc. of USENIX Annual Technical Conference*, p. 65, 2009.

[39] O. Kachman, "Effective multiplatform firmware update process for embedded low-power devices," 2018.

[40] D. G. Korn and K.-P. V. A. G. Differencing, "Compression data format," tech. rep., Technical Report HA1630000-021899-02TM, AT&T Labs-Research, 1999.

[41] K. Arakadakis, P. Charalampidis, A. Makrogiannakis, and A. Fragkiadakis, "Firmware over-the-air programming techniques for iot networks-a survey," *ACM Computing Surveys (CSUR)*, vol. 54, no. 9, pp. 1–36, 2021.

[42] J. Koshy and R. Pandey, "Remote incremental linking for energy-efficient reprogramming of sensor networks," in *Proceeedings of the Second European Workshop on Wireless Sensor Networks, 2005.*, pp. 354–365, IEEE, 2005.

[43] W. Dong, Y. Liu, C. Chen, J. Bu, C. Huang, and Z. Zhao, "R2: Incremental reprogramming using relocatable code in networked embedded systems," *IEEE Transactions on Computers*, vol. 62, no. 9, pp. 1837–1849, 2013.

[44] R. K. Panta and S. Bagchi, "Hermes: Fast and energy efficient incremental code updates for wireless sensor networks," in *IEEE INFOCOM 2009*, pp. 639–647, 2009.

[45] J. Jeong, "Node-level representation and system support for network programming," *University of California, Berkeley*, 2003.

[46] A. Tridgell *et al.*, "Efficient algorithms for sorting and synchronization," 1999.

[47] B. Mo, W. Dong, C. Chen, J. Bu, and Q. Wang, "An efficient differencing algorithm based on suffix array for reprogramming wireless sensor networks," in *2012 IEEE International Conference on Communications (ICC)*, pp. 773–777, IEEE, 2012.

[48] W. Dong, B. Mo, C. Huang, Y. Liu, and C. Chen, "R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems," in *2013 Proceedings IEEE INFOCOM*, pp. 315–319, IEEE, 2013.

[49] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders, "Better external memory suffix array construction," *Journal of Experimental Algorithmics (JEA)*, vol. 12, pp. 1–24, 2008.

[50] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli, "Secure firmware updates for constrained iot devices using open standards: A reality check," *IEEE Access*, vol. 7, pp. 71907–71920, 2019.

[51] P. Hambarde, R. Varma, and S. Jha, "The survey of real time operating system: Rtos," in *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*, pp. 34–39, 2014.

[52] J. W. Valvano, *Embedded Systems: Real-Time Operating Systems for Arm Cortex M Microcontrollers*. Texas: CreateSpace Independent Publishing Platform; 2nd ed. edition, 2017.

[53] H. J. Park, D. Woo, S. Kim, and P. Mah, "Multi-level ultra low-power mode support mechanisms for wearable device," in *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 171–176, 2016.

[54] P. Kumar and M. Srivastava, "Predictive strategies for low-power rtos scheduling," in *Proceedings 2000 International Conference on Computer Design*, pp. 343–348, 2000.

[55] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[56] C. Sabri, L. Kriaa, and S. L. Azzouz, "Comparison of iot constrained devices operating systems: A survey," in *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, pp. 369–375, 2017.

[57] Eclipse, "2020 IoT Developer Survey Key Findings," 2021. [Online; accessed 30-November-2021].

[58] V. Research, "IoT & Embedded Operating Systems," 2021. [Online; accessed 30-November-2021].

[59] FreeRTOS, "FreeRTOS," 2021. [Online; accessed 30-November-2021].

[60] A. Musaddiq, Y. B. Zikria, O. Hahm, H. Yu, A. K. Bashir, and S. W. Kim, "A survey on resource management in iot operating systems," *IEEE Access*, vol. 6, pp. 8459–8482, 2018.

[61] M. Simonović and L. Saranovac, "Power management implementation in freertos on lm3s3748," *Serbian Journal of Electrical Engineering*, vol. 10, no. 1, pp. 199–208, 2013.

[62] Zephyr, "Zephyr Project," 2021. [Online; accessed 30-November-2021].

[63] Microsoft, "Azure RTOS ThreadX," 2021. [Online; accessed 12-February-2021].

[64] Microsoft, "Azure RTOS ThreadX documentation," 2021. [Online; accessed 15-June-2021].

[65] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[66] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Migrating to cloud-native architectures using microservices: an experience report," in *European Conference on Service-Oriented and Cloud Computing*, pp. 201–215, Springer, 2015.

[67] D. R. Stinson, *Cryptography: theory and practice*. Chapman and Hall/CRC, 2005.

[68] M. F. Mushtaq, S. Jamel, A. H. Disina, Z. A. Pindar, N. S. A. Shakir, and M. M. Deris, "A survey on the cryptographic encryption algorithms," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 11, pp. 333–344, 2017.

[69] G. Singh, "A study of encryption algorithms (rsa, des, 3des and aes) for information security," *International Journal of Computer Applications*, vol. 67, no. 19, 2013.

[70] F. Pub, "Data encryption standard (des)," *FIPS PUB*, pp. 46–3, 1999.

[71] D. Coppersmith, D. B. Johnson, and S. M. Matyas, "A proposed mode for triple-des encryption," *IBM Journal of Research and Development*, vol. 40, no. 2, pp. 253–262, 1996.

[72] T. Nie and T. Zhang, "A study of des and blowfish encryption algorithm," in *TENCON 2009 - 2009 IEEE Region 10 Conference*, pp. 1–4, 2009.

[73] S. Jamel, M. M. Deris, I. T. R. Yanto, and T. Herawan, “The hybrid cubes encryption algorithm (hisea),” in *Advances in Wireless, Mobile Networks and Applications* (S. S. Al-Majeed, C.-L. Hu, and D. Nagamalai, eds.), (Berlin, Heidelberg), pp. 191–200, Springer Berlin Heidelberg, 2011.

[74] P. Jindal and B. Singh, “Rc4 encryption-a literature survey,” *Procedia Computer Science*, vol. 46, pp. 697–705, 2015. Proceedings of the International Conference on Information and Communication Technologies, ICICT 2014, 3-5 December 2014 at Bolgatty Palace & Island Resort, Kochi, India.

[75] T. Hardjono and L. R. Dondeti, *Security in Wireless LANS and MANS (Artech House Computer Security)*. Artech House, Inc., 2005.

[76] B. Girgenti, P. Perazzo, C. Vallati, F. Righetti, G. Dini, and G. Anastasi, “On the feasibility of attribute-based encryption on constrained iot devices for smart systems,” in *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*, pp. 225–232, 2019.

[77] A. Hameed and A. Alomary, “Security issues in iot: A survey,” in *2019 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT)*, pp. 1–5, 2019.

[78] T. Instruments, “HDC2080 Ultra-low-power digital humidity and temperature sensor ,” 2021. [Online; accessed 23-June-2021].

[79] Bosch, “Acceleration sensor BMA400 sensor,” 2021. [Online; accessed 23-June-2021].

[80] T. Instruments, “OPT3002 Light-to-digital sensor,” 2021. [Online; accessed 23-June-2021].

[81] Microchip, “ATECC608A, Network and Accessories secure authentication,” 2021. [Online; accessed 23-June-2021].

[82] Quectel, “LTE BC66 NB-IoT,” 2021. [Online; accessed 23-June-2021].

[83] S. Eletronics, “Ultra-low-power Arm Cortex-M0+ MCU with 192-Kbytes of Flash memory, 32 MHz CPU, AES,” 2021. [Online; accessed 23-June-2021].

[84] Saft, “Saft LM17500 Battery ,” 2021. [Online; accessed 21-April-2021].

[85] T. Instruments, “TPS7A0218PDBVR, 200-mA, nanopower-IQ (25 nA), low-dropout (LDO) voltage regulator with enable,” 2021. [Online; accessed 23-June-2021].

[86] NOS, “NOS,” 2021. [Online; accessed 21-April-2021].

[87] D. Andrews, I. Bate, T. Nolte, C. M. O. Pérez, and S. M. Petters, “Impact of embedded systems evolution on rtos use and design,” in *Proceedings of the 1st International Workshop Operating System Platforms for Embedded Real-Time Applications (OSPERT’05) in conjunction with the 17th Euromicro International Conference on Real-Time Systems (ECRTS’05)*, pp. 13–19, 2005.

[88] Wikipedia contributors, “Intel hex — Wikipedia, the free encyclopedia.” `https://en.wikipedia.org/w/index.php?title=Intel_HEX&oldid=1037499062`, 2021. [Online; accessed 20-September-2021].

[89] Wikipedia contributors, “Md5 — Wikipedia, the free encyclopedia.” `https://en.wikipedia.org/w/index.php?title=MD5&oldid=1038874008`, 2021. [Online; accessed 15-August-2021].

[90] Keysight, “B2901A Precision Source/Measure Unit, 1 ch, 100 fA, 210 V, 3 A DC/10.5 A Pulse,” 2021. [Online; accessed 10-November-2021].

[91] Tektronix, “MDO3000 Mixed Domain Oscilloscope,” 2021. [Online; accessed 10-November-2021].

Appendix A: Appendix

A.1 End-device Implementation

A.1.1 System Control

The system control is responsible for the MCU power modes and the RTC peripheral. This section will show the system control thread implementation, the struct and the set function of the alarms.

The system control thread is represented in Code A.1. This thread can be enabled and disabled through the system control event flag group. When enabled, it reads the current RTC and alarm time and calculates the time interval to the next alarm. Then, it uses the getDateTime_pf to get the current date-time. If it is successful, the system control set date-time function is called. With the RTC date-time updated, re-enables the next alarm and signals RTC update. Finally, this thread sleeps and will repeat the process when awake.

```c
while(1){
    tx_event_flags_get(&sc_eventflags, sc_enable_rtc_update, TX_AND,
                       &unused_var, TX_WAIT_FOREVER);

    /* Read RTC current time */
    ...

    /* Read Alarm time */
    ...

    /* Time interval between the current time and next alarm */
    ...

    if(getDateTime_pf(&datetime_s) != success){
        /* Get datetime error Error */
        signalAppError_v(getdatetime);
    }
    else{
        _fitDateTimeInTime_v(&datetime_s);

        if(systemControlSetCurrentDateTime_e(&datetime_s) == success){
            if(alarm_interval_s_s32 > 0){
                /* Set again the next alarm */
            }

            tx_event_flags_set(&sc_eventflags, sc_rtc_updated_ef, TX_OR);
            return;
        }
    }
    tx_thread_sleep(datetime_interval_tick_u32);
}
```

Code A.1: System control refresh datetime thread.

The alarms were added to the system control. These alarms are stored in the struct represented in

Code A.2. This struct contains if the alarm is in time interval mode and the respective parameters of an alarm (hour, minutes, and seconds). The index indicates the next alarm to be activated.

```c
    typedef struct alarm{
        uint8_t time_interval_u8;
        uint8_t hour_au8[MAX_ALARMS];
        uint8_t min_au8[MAX_ALARMS];
        uint8_t sec_au8[MAX_ALARMS];
        uint8_t index_u8;
    }alarm_st;
```

Code A.2: System Control alarms struct.

The alarms struct is filled in compile-time with the time interval enabled and default transmission interval. It can be set in run-time using the function represented in Code A.3. It receives a configuration byte array and the number of alarms in the configuration array. The first position in the array has the transmission mode, and the remaining are alarms. This function starts by doing alarms validation, setting the transmission mode, and saving the new alarms received into the structure A.2. After saving the alarms, the next alarm index needs to be set. Therefore, the index of the next alarm is synced according to the current time. If something fails, it is returned failure; if not, it is returned success.

```c
status_et systemControlSetAlarms_e(uint8_t*config_pu8,uint8_t n_alarms_u8){
    /* Alarms validation */
    ...

    alarm_s.time_interval_u8 = 0;
    if (config_pu8[0] >= 1){
        alarm_s.time_interval_u8 = 1;
    }

    /* Save the new alarms */
    ...

    /* Sync the next alarm index with the current time*/
    ...

    return success;
}
```

Code A.3: System control set alarms function.

Finally, the system control has the `systemControlInitialize` function. This function sets the MCU low-power settings, the LPUART sleep configuration, the standby and alarms default values. The MCU low-power configurations include setting the main internal regular output voltage to the minimum scale, enabling the ultra-low-power mode, and enabling the MCU to fast wake up from ultra-low-power mode. The LPUART configurations enable the peripheral to wake up the MCU when receiving data.

A.1.2 Sensors

The sensors module comprises three threads: Sensors Manager, Sensors Emergency Handler, and Sample Sensors. This subsection will present the implementation of each thread and, in the final, will be shown the sensor and variable structs with an example function to alter and read them.

Code A.4 represents the sensors initialization thread. If the sensors "init" event flag is set, this thread clears the sensors' initialized event flag and starts by calling the sensors' initialization functions.

This function iterate through all the sensors and confirms if they are alive. If any sensor is functioning, the sensor manager thread configures them and signals through the sensors' event flag that the sensors were initialized. If there are no sensors and the sensor initialize event flag is set, it will sleep a defined compilation time and try again. Both the Emergency Handler and Sample Sensors thread waits for the sensors initialized event flag to be set to start its operation.

```
tx_event_flags_get(&s_eventflags, s_init_sensors_ef,
                    TX_AND_CLEAR, &unused_var, TX_WAIT_FOREVER);

tx_event_flags_get(&s_eventflags, s_sensors_inited_ef,
                    TX_OR_CLEAR, &unused_var, TX_NO_WAIT);
initializeSensors_v();

if(getSensorsEnabled_u8(NULL) > 0){
    configSensors_v();

    /* Sensors initialized */
    tx_event_flags_set(&s_eventflags, s_sensors_inited_ef, TX_OR);
    continue;
}
tx_thread_sleep(S_TO_TICKS(NO_SENSORS_TRY_SLEEP_S));
```
Code A.4: Sensors Manager thread entry loop.

The Emergency Manager thread, represented in Code A.5, waits for the hardware event flags to signal a sensor's external interrupt. When an interrupt is triggered, it gets the sensors and emergency binary semaphores and then iterates through all the sensors finding the emergency that has been triggered. When it is found, it is called the interrupt handler and the emergency is added to the `emergencies_triggered_u8` if the sensor variable interrupt is enabled. If any emergency was triggered, the application emergency event flag is set, signalling that an emergency has occurred.

```
/* Waits for an emergency */
tx_event_flags_get(&hw_eventflags, sensors_int_cu16, TX_OR_CLEAR,
                   &interrupts_flags_ul, TX_WAIT_FOREVER);

/* Waits until the sensors are initialized*/
...

/* Get sensors and emergency binary semaphores */
...

for(i_u8=0; i_u8 < n_sensors_u8; i_u8++){
    /* Find sensor interrupt */
    ...
    /* Call interrupt handler */
    sensors_aps[i_u8]->interruptHandler_pf(NULL);

    /* if interrupt is enabled*/
    emergencies_triggered_u8 |= \
        (interrupts_flags_ul & sensors_aps[i_u8]->int_mask_u8);
}
if(interrupts_flags_ul & emergencies_triggered_u8){
    /* Alerts to a emergency received */
    tx_event_flags_set(&a_eventflags, a_emergency_ef, TX_OR);
}
/* Put sensors and emergency binary semaphores */
...
```

Code A.5: Sensors Emergency Handler thread entry loop.

Code A.6 illustrates the Sample Sensors thread entry loop. This thread iterates through all the sensors, and if the variable sample time is reached, it samples the variable and saves the sampled value according to the sensor's sample mode (simple or average). The sensors sample have been joined into this thread, and the sample time has a common multiplier and according to the variable period is calculated the counts variable. The variable sample time is given by variable counter equals the variable counts.

```
tx_thread_sleep(min_sampling_time_u32);

/* Waits until the sensors are initialized and check if the
    the sampling flag is set */
...

tx_bin_semaphore_get(&sensors_aps_bsem, TX_WAIT_FOREVER);

for(i_u8=0; i_u8<n_sensors_u8; i_u8++){
    if(/* Sensor active */){
        for(j_u8=0; j_u8<sensors_aps[i_u8]->n_variables_u8; j_u8++){
            if(/* variable and sampling enable */ &&
               /* variable counter == variable counts */){
                if (!sampled_u8){
                    status_u8=sensors_aps[i_u8]->readSensor_pf(data_as32);
                    if(status_u8 == OK){
                        sampled_u8 = 1;
                        j_u8 = 0;
                    }
                }
                if (status_u8 == OK){
                    if(/* Variable in average mode? */){
                        /* Save average sample */
                        ...
                    }
                    else{
                        /* Save simple sample*/
                        ...
                        }
    ...
}
tx_bin_semaphore_put(&sensors_aps_bsem);
```

Code A.6: Sample Sensors thread entry loop.

In the sensors module, each sensor is described by its sensor struct. This struct, represented in Code A.8, is composed of the model string, the sensors' variables, active state, interrupt mask, and the pointers to functions to initialize, configure, read and handle the interrupt. This module serves as the sensors module interface.

```
typedef struct sensor{
    uint8_t model_au8[S_MODEL_STR_SIZE];
```

```c

    variable_st* variables_aps[MAX_VARIABLES];
    uint8_t n_variables_u8;
    uint8_t active_u8;
    uint8_t int_mask_u8;

    uint8_t (*initSensor_pf)();
    void (*configSensor_pf)(char*, uint8_t);
    uint8_t (*readSensor_pf)(int32_t*);
    void (*interruptHandler_pf)(uint8_t);
} sensor_st;
```

Code A.7: Sensor struct.

The sensor is composed of its physical variables. These variables struct are illustrated in Code A.8. A variable is composed of the variable's name, units, status, samples, period, counter, counts, and the latest sample. The status variable enables the configuration of the emergency, sample mode, and variable status.

```c
typedef struct variable{
    uint8_t name_au8[FIXED_SENSOR_NAME_LENGTH];
    uint8_t units_au8[8];
    status_st status_s;

    samples_st samples_s;

    uint16_t period_u16;
    uint8_t counter_u8;
    uint8_t counts_u8;

    uint32_t latest_sample_u32;
} variable_st;
```

Code A.8: Sensors' variable struct.

As an example of altering the sensor behaviour, changing the period can be performed by using the function represented in Code A.9. This function receives the sensor and variable index with the respective new period. If it is a valid index and period, it sets the variable with the new period.

```c
status_et setSensorVariablePeriod_e(uint8_t sensor_u8, uint8_t variable_u8,
     uint16_t new_period_u16){
    tx_bin_semaphore_get(&sensors_aps_bsem, TX_WAIT_FOREVER);
    if (sensor_u8 >= n_sensors_u8 ||
        variable_u8 >= sensors_aps[sensor_u8]->n_variables_u8 ||
        new_period_u16 == 0 ){
        tx_bin_semaphore_put(&sensors_aps_bsem);
        return failure;
    }
    sensors_aps[sensor_u8]->variables_aps[variable_u8]->period_u16 = \
                                                         new_period_u16;
    tx_bin_semaphore_put(&sensors_aps_bsem);
    return success;
}
```

Code A.9: Set sensor variable period functions.

On the other hand, if the objective is to get the variable's periods. It is possible recurring the Function A.10. This function receives a pointer to a 16-bits array and fills it with the variable's periods, returning the number of periods written to the array.

```c
uint8_t getSensorsPeriods_u8(uint16_t *periods_pu16){
    tx_bin_semaphore_get(&sensors_aps_bsem, TX_WAIT_FOREVER);
    for(i_u8=0; i_u8 < variable_index_u8; i_u8++){
        periods_pu16[i_u8] = variables_data_aps[i_u8]->period_u16;
    }
    tx_bin_semaphore_put(&sensors_aps_bsem);

    return i_u8;
}
```

Code A.10: Get sensor's variables period functions.

There are more functions to set the variable status and its configuration. Therefore, the sensor's configuration and behaviour can be set and get in run-time with these functions.

A.1.3 Cryptography

The cryptography module has the run-time RSA algorithm, AES, and MD5 with the STMicroelectronics Encryption library. Also, the AES implementation using the encryption chip ATECC608A and the MD5 implementation adapted from an open-source project. These implementations allow in compile-time, through macros, to choose the algorithm implementation. The implementations will be compared in the results section.

The code is presented in Listing A.3.1. It provides initialization, encryption and decryption functions. The MD5 has two implementations to decrease the code size when the STMicroelectronics encryption libraries are not used.

The ATECC608A module configures all module registers and locks them. This configuration is made for the module to sleep 1.3 seconds after the last use. An API that initializes the module and provides encryption and decryption functions was exposed.

A.1.4 Communication

The communication is divided into four modules: transmission, reception, asynchronous events, and the connection manager as designed in Section 5.3.7.

In this module is shared between the transmission and reception a 1024 bytes buffer and the datagram structure represented in Code A.11. This structure is comprises a pointer to the microservice, an array of datagram objects, and the respective array object number. A datagram object is composed of a pointer to the id, data, and binary semaphore and its respective length and type.

```c
typedef struct datagramobject{
    const char* id_pcc;
    const void* data_pv;
    TX_SEMAPHORE *bsem_p;
    uint16_t len_u16;
    uint8_t type_u8;
} datagramobject_st;

typedef struct datagram{
    const uint8_t* microservice_pu8;
    datagramobject_st objects_as[C_DATAGRAM_OBJ_SIZE];
    uint16_t n_objects_u16;
} datagram_st;
```

Code A.11: Datagram and datagram objects structures.

The following subsections will explain how the respective modules use these structures.

Transmission

The transmission has an API that allows the user to transmit data to the cloud efficiently. This interface starts with a function to set up a new transmission as represented in Code A.12. It starts by getting the shared datagram buffer and then clears the microservice and the number of objects in the datagram.

```c
void communicationSetupNewTransmission_v(void){
    tx_bin_semaphore_get(&_datagram_bsem, TX_WAIT_FOREVER);

    _datagram_s.microservice_pu8 = NULL;
    _datagram_s.n_objects_u16 = 0;

    return;
}
```

Code A.12: Transmission setup new transmission function.

The microservice can be set using the `communicationSetupMicroservice` function represented in Code A.13.

```c
void communicationSetupMicroservice_v(const uint8_t *microservice_pu8){
    _datagram_s.microservice_pu8 = microservice_pu8;
    return;
}
```

Code A.13: Transmission microservice setup.

In order to add objects to the datagram, it can be done by appending into the array or inserting in specific array position with the functions as illustrated in Code A.14. The append function set the received values into the last datagram object position, and the insert also receives an index in the parameter the array set location.

```c
void communicationAppendDatagramObject_v(void* data_pv, uint8_t type_u8,
                                          const char *id_pcc,
                                          uint16_t len_u16,
                                          TX_SEMAPHORE *ptr_pbsem){
    _datagram_s.objects_as[_datagram_s.n_objects_u16].data_pv = data_pv;
    _datagram_s.objects_as[_datagram_s.n_objects_u16].type_u8 = type_u8;
    _datagram_s.objects_as[_datagram_s.n_objects_u16].len_u16 = len_u16;
    _datagram_s.objects_as[_datagram_s.n_objects_u16].bsem_p = ptr_pbsem;
    _datagram_s.objects_as[_datagram_s.n_objects_u16].id_pcc = id_pcc;

    _datagram_s.n_objects_u16 =  \
                    (_datagram_s.n_objects_u16 + 1) % C_DATAGRAM_OBJ_SIZE;

    return;
}

void communicationInsertDatagramObject_v(uint8_t index_u8, void* data_pv,
                                          uint8_t type_u8,
                                          const char *id_pcc,
                                          uint16_t len_u16,
                                          TX_SEMAPHORE *ptr_pbsem){
    if(index_u8 < C_DATAGRAM_OBJ_SIZE){
        _datagram_s.objects_as[index_u8].data_pv = data_pv;
        _datagram_s.objects_as[index_u8].type_u8 = type_u8;
        _datagram_s.objects_as[index_u8].len_u16 = len_u16;
        _datagram_s.objects_as[index_u8].bsem_p = ptr_pbsem;
        _datagram_s.objects_as[index_u8].id_pcc = id_pcc;

        _datagram_s.n_objects_u16++;
    }
    return;
}
```

Code A.14: Transmission append and insert datagram object functions.

When the application finalizes inserting and appending the data that it wants to transmit, it calls the function `communicationSendDatagram`. This function receives the index, used to repeat the objects from the beginning to the index received if the datagram exceeds the modem maximum of 1024 bytes. Then it starts by filling the cloud header.

The cloud header adds the microservice if it exists (different from NULL). If the encryption is enabled, it can be performed in run-time or static. Since the cloud header does not change, it can be used static headers already encrypted, and this way, the microcontroller does not require the overhead of encrypting the header with the RSA. When the static encryption is enabled, the microservice pointer has a 128-bytes string of the header, and it is copied. If it is using the run-time encryption, it adds the microservice opcode, the AES and IV and encrypts it using the RSA. If no encryption is used and it has a microservice, it is added.

After filling the encryption header, the transmission enters the conversion to message pack the objects and insert them into the datagram using the `insertObject` function. This function returns if it fails, the datagram is full or not full. When the datagram is full, it saves the respective auxiliary variables, fills the array header, and is encrypted and sent into the cloud. After transmitting, it repeats the number of objects received as a parameter from the beginning of the object array. When it adds these objects, it returns to the object that was not inserted because the datagram was full.

When there are no more objects to insert into the datagram, the last datagram is sent, and the function returns success. If something fails, it returns failure.

```
status_et communicationSendDatagram_e(uint8_t repeat_until_object_i_u8){
    ...
    payload_start_u8 = fillCloudHeader_siu8();

    trans_s.msg_len_u16 = payload_start_u8 + 9; /* 9 is the header offset
   */
    start_msg_len_u8 = trans_s.msg_len_u16;

    do{
        if(repeat_until_object_i_u8 > 0){
            /* Reset to the first array object */
            ...
        }
        for(; obj_i_u16 < _datagram_s.n_objects_u16; obj_i_u16++){
            /* Gets binary semaphore if != NULL*/
            switch(_insertObject_ie(&obj_i_u16, &trans_s)){
                case FAILURE_TRANSMISSION:
                    /* Puts binary semaphore if != NULL */
                    return failure;
                case FULL:
                    total_parts_u8++;
                    datagram_full_u8 = 1;
                    /* Backup array index variables*/
                default:
                    break;
            }
            /* Puts binary semaphore if != NULL */
            if(obj_i_u16 + 1 == repeat_until_object_i_u8){
                /* Already repeated the objects, returns to the break
                   object */
                ...
            }
            if(datagram_full_u8){
                break;
            }
        }
        /* Set the parts and total parts and array beginning */

        /* Encrypt using AES */

        /* Sends the datagram */
    }while(part_u8 != total_parts_u8);
    /* Returns transmission success or failure */
}
```

The parts and total parts variables are used to indicate the current datagram part and when the current part equals the total parts indicates that the datagram was all sent. After the transmission, this module offers the `communicationEndTransmission_v`, releasing the datagram binary semaphore.

Reception

The reception does the opposite of the transmission. It receives the data in message pack format and converts it into datagram objects. Also, it has a generic interface as the transmission.

The `communicationSetupNewReception_v` gets the datagram binary semaphore, and its release can be performed with the function `communicationEndReception_v`. The reception core function is the `communicationReceiveDatagram_e` which its structure is represented in Code A.15.

```
status_et communicationReceiveDatagram_e(uint16_t len_u16){
    /* Data received decryption*/
    ...

    /* MD5 verification */
    ...

    /* Number of elements in array conversion*/
    ...

    /* Parse message pack payload */
    for(object_i_u8 = 0; object_i_u8 < _datagram_s.n_objects_u16;
   object_i_u8++){
        switch(_buffer_au8[index_u8]){
            case MP_BIN8:{ /*Parse BIN8*/
                ...
            }
            case MP_BIN16:{ /*Parse BIN16 */
                ...
            }
            case MP_I8:{
                *free_memory_pu8 =  _buffer_au8[index_u8+1];
                _insertDatagramObject_v(free_memory_pu8, 1, INT8_T,
                                        object_i_u8);

                free_memory_pu8 += 1;
                index_u8 += 2;
                break;
            }
            case MP_U8:{ /* Parse UIN8_T */
                ...
            }
            case MP_I16:{ /* Parse INT16_T */
                ...
            }
            case MP_U16:{ /* Parse UINT16_T */
                ...
            }
            case MP_I32:{ /* Parse INT32_T */
                ...
            }
            case MP_U32:{ /* Parse UINT32_T */
                ...
            }
            case MP_I64:{ /* Parse INT64_T */
                ...
            }
            case MP_U64:{ /* Parse UINT64_T */
                ...
            }
        }
    }
    return success;
}
```

Code A.15: Communication receive datagram function.

This function starts by decrypting the data received if the encryption is enabled. Then, if the MD5 is enabled, generate the hash key and compares it with the received. If it does not match, it returns failure. After the integrity verification, it parses the number of objects in the array's header.

If the number of objects does not exceed what the device can receive, it enters the message pack payload loop parse. In this loop, it converts the message pack to datagram objects. In the listing above, it is represented the conversion of an integer of 8 bits. It starts by saving the data received in the buffer to free memory and calls the auxiliary function that inserts as datagram object the parameters received.

It is used the `insertDatagramObject_v` represented in Code A.16 as an auxiliary function. It receives the object index with the void pointer to the data, its length and type, inserting it into the datagram object array.

```
static inline void _insertDatagramObject_v(void* data_pv, uint16_t len_u16,
                                            valuetype_et type_e,
                                            uint8_t object_i_u8){
    _datagram_s.objects_as[object_i_u8].type_u8 = type_e;
    _datagram_s.objects_as[object_i_u8].len_u16 = len_u16;
    _datagram_s.objects_as[object_i_u8].data_pv = data_pv;

    return;
}
```

Code A.16: Insert datagram object auxiliary function.

In order to facilitate the use outside the module, the reception has the function `communicationGet-ReceivedSize_u16` that returns the number of objects in the datagram object array.

Asynchronous Events

The asynchronous events thread is responsible for handling the modem's asynchronous events. These are data received from the cloud and if the cloud closes the modem socket. This thread implementation is depicted in Code A.17, it waits for asynchronous signals from the modem event flags, and according to the asynchronous event, it reacts to it. If data was received from the cloud, it set up a new reception, gets the data from the cloud and parses it with the `communicationReceiveDatagram`, explained previously in this section. If data was received and parsed successfully, the cloud data communication event flag is set.

When the socket closed URC occurs, it is closed the respective socket.

```
tx_event_flags_get(&m_eventflags, urc_ef, TX_OR_CLEAR,
                    &actual_flags_ul, TX_WAIT_FOREVER);

if(actual_flags_ul & urc_ef){
    communicationSetupNewReception_v();
    tx_bin_semaphore_get(&modem_bsem, TX_WAIT_FOREVER);
    if(modemGetCloudData_e(_buffer_au8, &aux_u16, C_BUFFER_SIZE)==success){
        if(communicationReceiveDatagram_e(aux_u16) == success){
            tx_event_flags_set(&c_eventflags, c_cloud_data_ef, TX_OR);
        }
    }
    tx_bin_semaphore_put(&modem_bsem);
    communicationEndReception_v();
}
else if(isSocketClosedURC_e(UINT8_PTR(&aux_u16)) == success){
    tx_bin_semaphore_get(&modem_bsem, TX_WAIT_FOREVER);
    modemCloseSocket_e(aux_u16);
    tx_bin_semaphore_put(&modem_bsem);
}
```

Code A.17: Asynchronous thread.

Connection Manager

The connection manager thread is handles the modem configurations, network, and server connection. This thread is implemented using a state machine, depicted in Figure 5.31, it was implemented using an array of pointers as illustrated in Code A.18. It comprises the initialization, configuration, check network connection, server connection, socket, and wait for error states. This section will present how a communication error is signal and show the configuration state as an example.

```
machineState_fp* errorRecoveryMachineState_apf[S_MACHINE_LEN] = {
                                               initializationState,
                                               configState,
                                               checkNetworkConnectionState,
                                               serverConnectionState,
                                               socketState,
                                               waitForErrorState
                                           };
```

Code A.18: Connection handler state machine pointer to functions array.

The communication errors possible are sending and getting clock errors. They are used to signal the communication error through the `signalCommunicationError` function. This function, represented in Code A.19, starts by getting the error handling binary semaphore and alerts the error received. Then, it waits for the error correction and frees the error handling binary semaphore.

```
void signalCommunicationError_v(communicationEventFlags_et error_flag_e){
    tx_bin_semaphore_get(&_error_handling_bsem, TX_WAIT_FOREVER);

    tx_event_flags_set(&c_eventflags, error_flag_e, TX_OR);

    /* Waits for the correction of the error*/
    tx_event_flags_get(&c_eventflags, error_ok_ef,
                       TX_OR_CLEAR, &unused_var TX_WAIT_FOREVER);

    tx_bin_semaphore_put(&_error_handling_bsem);
}
```

Code A.19: Signal communication error function.

The `waitForErrorState` function waits for an error event flag to be set. Then, according to the error, it jumps to the respective error handler state. If the error was the transmission, it jumps to the `socketState`. On the other hand, if the error was getting the clock, the problem could be with the modem connection to the network. Therefore, it jumps to the `checkNetworkConnectionState`.

All the states receive the last state and try variables. Code A.20 presents the configuration state. This state configures the modem and waits for the modem network connection. If it is unsuccessful, it checks if the last state executed was the previous state (`initializationState`) or the current state (`configState`). If it was, it enters in `sleepAndChangeStateMachineState`. This state receives the tries, previous and next state, and if the tries exceed a compilation time defined limit, it returns to the state before the current running or the current state.

```
nextstate_t configState(uint8_t last_state_u8,  uint8_t* tries_pu8){
    uint8_t next_state_u8;

    if(configModem_e() != success ||
       waitForModemNetworkConnection_e() != success){

        next_state_u8 = S_INIT;
        if(last_state_u8 == S_INIT || last_state_u8 == S_CONFIG){
            next_state_u8 = sleepAndChangeStateMachineState_siu8(...);
        }
        return next_state_u8;
    }

    *tries_pu8 = 0;
    return S_NETWORK_CNX;
}
```

Code A.20: Configuration connection manager state.

All the states follow the same structure as the `configState`, but they can apply an error handling mechanism. For example, the `socketState` changes the socket if it is not successful.

In the `initializationState`, if the modem fails, the `recoverModem` is called to try to recover it. If it can not recover, the board enters standby mode.

A.1.5 Commands

This section presents the commands manager thread implementation followed by an example of the set and gets alarms command.

The commands format mentioned in Design 5.3.8 and represented in Figure 5.32 has in the first position the command opcode and then the respective command values.

The commands manager thread implementation is divided into five stages. The first stage, represented in Code A.21, waits for a signal for the communication event flag. This flag indicates that the modem has received, parsed data and is waiting to be handled by the upper layer. After receiving the signal, the commands thread locks the shared memory binary semaphore and gets the received size.

```
    tx_event_flags_get(&c_eventflags, c_cloud_data_ef, TX_AND_CLEAR,
                       &unused_var, TX_WAIT_FOREVER);

    tx_bin_semaphore_get(&sharedata_bsem, TX_WAIT_FOREVER);

    rcv_size_u8 = communicationGetReceivedSize_u16();
```

Code A.21: Data reception event flag wait.

If the received size is equal to or greater than one, it gets the datagram object containing the command opcode and cast it from the void pointer. If it is not a valid type, the set status is set as an invalid type. On the other hand, if the receive size is 0, the shared memory binary semaphore is put, and it returns to stage one.

```
    if(rcv_size_u8 >= 1){
        /* Valid Command Size */
        communicationGetDatagramObject_e(0, &data_pv, &type_u8, &len_u16);

        switch (type_u8){
            case UINT8_T:{
                opcode_u16 = *UINT8_PTR(data_pv); break;
            }
            case UINT16_T:{
                opcode_u16 = *UINT16_PTR(data_pv); break;
            }
            default:{
                set_status_e = invalid_type;
            }
        }
        ...
    }
    else{
        /* Invalid Command */
        tx_bin_semaphore_put(&sharedata_bsem);
        continue;
    }
```

Code A.22: Command opcode parse command opcode.

If the opcode type is valid, the set and get functions are called in a set command, and only the get function is in the get command. All the set and get commands have the same structure, receiving the received size as the parameter, and at the end of this section, a set and get example will be presented. If it is not a valid type, it releases the shared memory binary semaphore and returns to stage one. If a valid command was received before the get function call, it starts a new communication transmission.

```
        if(set_status_e != invalid_type){
            if(opcode_u16 > CMDS_LEN - 1){
                /*Invalid Set Command OPCode*/
                set_status_e = invalid_cmd;
            }
            else{
                /* Call Set Command Function*/
                set_status_e = set_commands_afp[opcode_u16](rcv_size_u8);
            }
        }

        if(set_status_e == invalid_type){
            /*Invalid Type Response*/
            tx_bin_semaphore_put(&sharedata_bsem);
            continue;
        }

        communicationSetupNewTransmission_v();
        communicationSetupMicroservice_v(cmd_response_microservice);

        if((opcode_u16 & GET_CMD_MASK) < CMDS_LEN){
            /* Valid Command*/
            get_status_e = \
                get_commands_afp[opcode_u16 & GET_CMD_MASK](rcv_size_u8);
        }
```

Code A.23: Command set and get function call.

As an example of a set and get command, Code A.24 represents the set and get alarms command functions. The set alarms command receives an array with the alarms and the number of alarms received. These variables are supplied into the system control set alarms function. Finally, since the transmission alarms have changed, it activates the transmission next alarm and returns success. If something fails, it returns failure.

The get alarms command uses the shared memory data to save the query alarms from the system control, appending a new alarms object into the communication.

The remaining set and get functions are similar. According to the command, they get the command received data and set and get the respective command purpose.

In case of a command that only has or a set or get implementation and it is tried to execute, it will be returned failure. For example, if it is tried to set the software version.

```c
status_et _setAlarmsCMD_e(uint16_t rcv_i_objs_u16){
    void* config_array_pv, *n_alarms_pv;
    uint8_t type_u8;
    uint16_t len_u16;
    status_et status_e;

    communicationGetDatagramObject_e(1, &config_array_pv, &type_u8,
                                     &len_u16);

    if(type_u8 != UARR8_T){
        return failure;
    }

    status_e = communicationGetDatagramObject_e(2, &n_alarms_pv, &type_u8,
                                                &len_u16);

    if(status_e != success || type_u8 != UINT8_T){
        return failure;
    }

    if(systemControlSetAlarms_e(config_array_pv,
                               (*UINT8_PTR(n_alarms_pv))) != success){
        return failure;
    }
    systemControlActivateNextAlarm_e();
    return success;
}

status_et _getAlarmsCMD_e(uint16_t rcv_i_objs_u16){
    uint8_t *current_addr_pu8 = UINT8_PTR(&sharedata_u);

    *UINT8_PTR(current_addr_pu8) = systemControlGetAlarms_u8(
                                            current_addr_pu8+1);

    communicationAppendDatagramObject_v(current_addr_pu8 + 1, UARR8_T,
                                        "ALARMS", *current_addr_pu8, NULL);
    return success;
}
```

Code A.24: Set and get alarm command.

After the respective command set and get execution, is performed a final communication datagram object append of the command result, opcode, board id, and current timestamp as represented in Code

A.25.

```c
    /* Valid Command Size */
    appendIDTimestampObject_v();

    if(set_status_e != invalid_type){
        /* Valid Command */
        cmd_result_au8[SET_RESULT_POS] = MP_FALSE;
        cmd_result_au8[GET_RESULT_POS] = MP_FALSE;
        if(set_status_e == success){
            cmd_result_au8[SET_RESULT_POS] = MP_TRUE;
        }
        if(get_status_e == success){
            cmd_result_au8[GET_RESULT_POS] = MP_TRUE;
        }
        communicationAppendDatagramObject_v(VOID_PTR(cmd_result_au8),
                                            BIN8_T, "MP_R_SG", 3, NULL);

        communicationAppendDatagramObject_v(data_pv, type_u8, "OPCODE",
                                            1, NULL);
    }
```

Code A.25: Command result, opcode, board ID, and timestamp append.

The final stage of the commands is the transmission of the command response. The command manager thread signals the application manager thread that a command response can be sent and waits for the application manager to answer that the command response has been sent. When the command response is sent, the command manager frees the shared memory binary semaphore as represented in Code A.26.

```c
    tx_event_flags_set(&a_eventflags, a_cmd_response_ef, TX_OR);

    tx_event_flags_get(&a_eventflags, a_cmd_response_sent_ef, TX_AND_CLEAR,
                       &unused_var, TX_WAIT_FOREVER);
    communicationEndTransmission_v();

    tx_bin_semaphore_put(&sharedata_bsem);
```

Code A.26: Command response transmission.

A command addition can be performed by adding the set and get function pointers into the respective command array, increasing the command's opcode with a maximum of 32767 (15 bits value).

A.1.6 Trace

The trace module is helpful to trace the application. When enabled, it periodically traces the threads (including the main thread) and queue stacks, including the thread's stack overflows.

The trace thread code is represented in A.27. It starts by sleeping, enabling the user to get the last trace EEPROM values. After this interval, it initializes the trace EEPROM and enters in the calculate threads and queues stack free space loop.

```c
    void traceThreadEntry_v(unsigned long thread_input_ul){
        tx_thread_sleep(S_TO_TICKS(STARTUP_DELAY_S));

        _initializeTraceEEPROM_iv();

        while(1){
            /* Calculate threads and queues stack free space */
            _traceThreadsQueuesStackFreeSpace_v();

            tx_thread_sleep(S_TO_TICKS(TRACE_DELAY_S));
        }
    }
```

Code A.27: Trace thread entry.

The function responsible for trace the threads and queues stacks are represented in Code A.28. This function calls the functions responsible for calculating the respective free thread and queue stack space. There are static const arrays of pointers to the stacks and the respective sizes as an auxiliary. The free space is saved in the EEPROM using the writeEEPROM function. The main thread pointer and size are queried in run-time. Therefore it is not presented in the static const arrays. As a result, its calculation is done last.

```c
static inline void _traceThreadsQueuesStackFreeSpace_v(void){
    uint16_t i_u16;
    uint16_t free_space_u16;

    for (i_u16 = 1; i_u16 < THREADS_N; i_u16++){
        free_space_u16 = _calculateThreadFreeSpace_iu16(
                                    thread_stacks_apu8[i_u16],
                                    thread_stacks_size_au8[i_u16]);

        writeEEPROM_e(STACK_FREE_SPACE_BEGIN+(i_u16<<1),
                      UINT8_PTR(&free_space_u16), 2);
    }

    for (i_u16 = 0; i_u16 < QUEUE_N; i_u16++){
        free_space_u16 = _calculateQueueFreeSpace_iu16(
                                    queue_stacks_apu8[i_u16],
                                    queue_stacks_size_au8[i_u16]);

        /* Write eeprom queue free space */
    }

    free_space_u16 = _calculateThreadFreeSpace_iu16(
                                    main_thread_stack_ppu8,
                                    main_thread_stack_size_u32);
    /* Write eeprom main thread free space */
}
```

Code A.28: Trace threads and queue stack free space calculation function.

The stack free space calculation varies if it is a thread or a queue because the thread stack is used from the end of the array to the begin and the queue is the reverse. The stacks are filled with the value “0xEF” in the program startup. Therefore, calculating the free space is counting the intact “0xEF”. As illustrated in Code A.29, it is received the pointer to the thread or queue stack and the respective size. If it is a thread, it counters the free space from the beginning. If it is a queue, it counts from the stack end.

```c
static inline uint16_t _calculateThreadFreeSpace_iu16(
                                                  uint8_t *thread_pu8,
                                                  uint16_t thread_size_u16){
    uint16_t free_space_counter_u16 = 0;

    while(*(thread_pu8 + free_space_counter_u16) == 0xEF &&
          free_space_counter_u16 < thread_size_u16){
        ++free_space_counter_u16;
    }
    return free_space_counter_u16;
}

static inline uint16_t _calculateQueueFreeSpace_iu16(
                                                  uint8_t *queue_pu8,
                                                  uint16_t queue_size_u16){
    uint16_t free_space_counter_u16 = queue_size_u16 - 1;

    while(*(queue_pu8 + free_space_counter_u16) == 0xEF &&
          free_space_counter_u16 > 0){
        --free_space_counter_u16;
    }
    return queue_size_u16 - free_space_counter_u16 - 1;
}
```

Code A.29: Calculate thread and queue free space functions.

Moreover, the trace module has an available log to EEPROM macros that can be useful to trace, for example, MCU hard faults.

When the trace is enabled, the ThreadX stack checking feature is enabled. This feature checks if the thread stack overflows, and if it has, it calls a callback. The callback is implemented in the trace module, as represented in Figure A.30. It receives the respective stack overflowed thread pointer, saves the thread name into the EEPROM, traces the threads and queues stack free space and puts the system in the lowest power mode.

```c
void stack_error_handler(TX_THREAD *thread_ptr){
    _traceThreadsQueuesStackFreeSpace_v();

    writeEEPROM_e(TX_STACK_CHECKING_BEGIN_ADDR,
                  UINT8_PTR(thread_ptr->tx_thread_name),
                  TX_STACK_CHECKING_SIZE);

    uint8_t aux_u8 = '\0';
    writeEEPROM_e(TX_STACK_CHECKING_END_ADDR, UINT8_PTR(&aux_u8), 1);

    systemControlEnterStandby_v();
}
```

Code A.30: Trace implementation of ThreadX stack error handler function.

A python script was implemented that reads the exported EEPROM memory and shows it in a human-readable manner, as explained in Section 6.2.3.

A.1.7 OTA

In order to perform an Over-The-Air (OTA) update, the end-device receives the delta and handles it.

The OTA interface is the `OTA_e` and `getOTA_u32` functions. These functions are represented in Code A.31. The `OTA_e` is responsible for the OTA state machine implementation. It takes the OTA package index, delta, and delta length received and calls the respective OTA state responsible for handling it. Since some OTA states are sequential, for example, if the Begin state is successful, it enters the New App Flash Erase State. This function handles the state machine logic. If any state fails, it sets the machine state to the Begin state. The `getOTA_u32` function returns the expected OTA package index.

```c
status_et OTA_e(uint16_t ota_pckg_index_u16, uint8_t* delta_pu8,
                uint16_t delta_len_u16){
    while(1){
        ota_state_u8=otaMachineState_apf[ota_state_u8](ota_pckg_index_u16,
                                                       delta_pu8,
                                                       delta_len_u16
                                                    );

        switch(ota_state_u8){
            case S_FAILURE:{
                /* Set the initial state and return failure */
                ota_state_u8 = S_OTA_BEGIN;
                return failure;
            }
            case S_NEW_APP_FLASH_ERASE:
            case S_NEW_APP_CHECKSUM:
            case S_END_OTA:
                continue;
            default:
                return success;
        }
    }
}
```

Code A.31: OTA module interface functions.

The OTA states were implemented using an array of pointers to functions. These pointers to functions receive the OTA package index, a pointer to delta array, and delta length. The Begin state case, receives the OTA header instead of the delta. The rest of this section will show the implementation of the several OTA states.

Code A.32 is the OTA Begin state, it starts by checking if the OTA package is 0 and the header length is valid. If it is valid, it parses the received header. Then it validates that the new application address is located in the other application area. If the header received is valid, it sets the new application current address variable and resets the word window index, returning the New App Flash Erase state.

```c
static nextstate_t otaBeginState(uint16_t ota_pckg_index_u16,
                                 uint8_t* header_pu8,
                                 uint16_t header_len_u16){

    if(ota_pckg_index_u16 != 0 || header_len_u16 < 9){
        return S_FAILURE;
    }
    _new_app_begin_addr_vu32 = ARR_TO_INT32(header_pu8, 0);
    _new_app_end_addr_vu32 = ARR_TO_INT32(header_pu8, 4);
    _new_app_checksum_vu8 = header_pu8[8];
    _pckg_index_vu32 = ota_pckg_index_u16;

    if(_new_app_begin_addr_vu32 == TO_UINT32(current_app_addr_cpv)){
        /* Not valid address since it is the current app address*/
        return S_FAILURE;
    }

    if(_new_app_begin_addr_vu32 != OTA_APP1_ADDR &&
       _new_app_begin_addr_vu32 != OTA_APP2_ADDR){
        /* Begin address is invalid */
        return S_FAILURE;
    }

    /* Valid begin address */
    _new_app_current_addr_vu32 = _new_app_begin_addr_vu32;
    _word_window_i_u8 = 0;

    return S_NEW_APP_FLASH_ERASE;
}
```

Code A.32: OTA begin state.

The new application erases state, represented in Code A.33, starts by calculating the number of pages occupied by the new firmware. If the number of pages exceeds the maximum number per application, it erases the new application space and returns the new application flash program state.

```
static nextstate_t newAppFlashEraseState(...){
    uint32_t pages_to_erase_u32=(_new_app_end_addr_vu32-
                                 _new_app_begin_addr_vu32)/
                                 OTA_FLASH_PAGE_SIZE;

    if((_new_app_end_addr_vu32 - _new_app_begin_addr_vu32) %
        OTA_FLASH_PAGE_SIZE){
        pages_to_erase_u32++;
    }

    if(pages_to_erase_u32 > OTA_MAX_PAGES){
        return S_FAILURE;
    }

    if(TO_UINT32(current_app_addr_cpv) == OTA_APP1_ADDR){
        /* We are in the APP1 and need to delete the APP2 space*/
        unlockFlash_v();
        eraseFlash_e(OTA_APP2_ADDR, pages_to_erase_u32);
        lockFlash_v();

        return S_NEW_APP_FLASH_PROGRAM;
    }

    /* We are in the APP2 and need to delete the APP1 space*/
    unlockFlash_v();
    eraseFlash_e(OTA_APP1_ADDR, pages_to_erase_u32);
    lockFlash_v();

    return S_NEW_APP_FLASH_PROGRAM;
}
```

Code A.33: OTA new application erase state.

The new application flash program is the core state of the OTA update. Since it is here where the delta commands are parsed and executed into the flash. Therefore, if it receives the expected package, it iterates through the delta commands and parses them with the switch case of line 14 in Code A.34.

According to the command, the behaviour is different, but they have all one thing in common, the data insertion into the flash. Since the flash needs to be written aligned, the _word_window_i_u8 index of the four bytes array _word_window_au8 purpose is to accumulate bytes read from to when it has 4 bytes be written to the flash.

The COPY command copies from the flash starting on the address received, appending to the word window array. On the other hand, the ADD command appends the data received. In both cases, when the word window has four bytes, they are written into the new application area. The insertion in the new application area is sequential and incremented according to the word window array writes.

Finally, when an END command is received, the word window array is padded with zeros and written into the flash, returning the checksum as the next state. If all delta commands are parsed, and an END command was not found, it returns the current state as the next state. This state is depicted in Code A.34.

```c
static nextstate_t newAppFlashProgramState(uint16_t ota_pckg_index_u16,
                                           uint8_t* delta_pu8,
                                           uint16_t delta_len_u16){
    uint16_t delta_i_u16 = 0;
    otat32bit_st read_word_s;

    if(ota_pckg_index_u16 != _pckg_index_vu32){
        return S_NEW_APP_FLASH_PROGRAM;
    }

    unlockFlash_v();

    do{
        switch (OPCODE_MASK(delta_pu8[delta_i_u16])){
            case OTA_CPY_OPCODE:{
                /* Handle unfinished word*/
                uint16_t bytes_to_copy_u16;
                SWITCH_BYTES_OCCUPIED(
                                      bytes_to_copy_u16, delta_pu8,
                                      delta_i_u16 + 1,
                                      LEN_BYTES_LEN(delta_pu8[delta_i_u16])
                                     );

                uint32_t cpy_addr_u32;
                SWITCH_BYTES_OCCUPIED(
                     cpy_addr_u32, delta_pu8,
                     delta_i_u16+LEN_BYTES_LEN(delta_pu8[delta_i_u16])+1,
                     CPY_BYTES_LEN(delta_pu8[delta_i_u16])
                    );
                cpy_addr_u32 += TO_UINT32(current_app_addr_cpv);

                for(uint16_t cpy_i_u16=0; cpy_i_u16 < bytes_to_copy_u16;){

                    /* Align copy address*/
                    uint8_t alignment_offset_u8 =  cpy_addr_u32 % 4;

                    uint8_t to_read_u8;
                    if(bytes_to_copy_u16 - cpy_i_u16 > 4){
                        to_read_u8 = 4;
                    }
                    else{
                        to_read_u8 = bytes_to_copy_u16 - cpy_i_u16;
                    }

                    /* Read the word*/
                    uint32_t cpy_addr_u32=cpy_addr_u32-alignment_offset_u8;
                    read_word_s.value=readWordFromFlash_u32(cpy_addr_u32);
                    cpy_addr_u32 += to_read_u8 - alignment_offset_u8;

                    /* Insert to the word window */
                    uint8_t j_u8 = 0;
                    for(uint8_t i_u8=4-(to_read_u8-alignment_offset_u8);
                        i_u8 < 4; i_u8++){
                        _word_window_au8[_word_window_i_u8++] = \
                          read_word_s.value_b[alignment_offset_u8+j_u8++];

                        if(_word_window_i_u8 == 4){
                            /* Window complete, lets flash word */
                            _word_window_i_u8 = 0;
                            flashWord_e(_new_app_current_addr_vu32,
                                        ARR_TO_INT32(_word_window_au8, 0));
                            _new_app_current_addr_vu32 += 4;
                        }
                    }
                    cpy_i_u16 += 4 - alignment_offset_u8;
                }

                delta_i_u16 += 1 + LEN_BYTES_LEN(delta_pu8[delta_i_u16]) +
                               CPY_BYTES_LEN(delta_pu8[delta_i_u16]);
                break;
            }
            case OTA_ADD_OPCODE:{
                /* Handle unfinished word*/
                uint16_t bytes_to_add_u16;
                SWITCH_BYTES_OCCUPIED(
                                    bytes_to_add_u16,
                                    delta_pu8, delta_i_u16 + 1,
                                    LEN_BYTES_LEN(delta_pu8[delta_i_u16])
                                   );

                for(uint16_t add_i_u16 = 0; add_i_u16 < bytes_to_add_u16;
                    add_i_u16++){
                    /* Insert to the word window */
                    _word_window_au8[_word_window_i_u8++] =delta_pu8[
                       delta_i_u16+1+LEN_BYTES_LEN(delta_pu8[delta_i_u16])+
                       add_i_u16
                      ];

                    if(_word_window_i_u8 == 4){
                        /* Window complete, lets flash word */
                        _word_window_i_u8 = 0;
                        flashWord_e(_new_app_current_addr_vu32,
                                    ARR_TO_INT32(_word_window_au8, 0));
                        _new_app_current_addr_vu32 += 4;
                    }
                }

                delta_i_u16 += 1 + LEN_BYTES_LEN(delta_pu8[delta_i_u16]) +
                               bytes_to_add_u16;
                break;
            }
            case OTA_END_OPCODE:{
                /* In case of window are not empty, pad it
                   with zeros and flash the rest */
                if(_word_window_i_u8 != 0){
                    while(_word_window_i_u8 < 4){
                        _word_window_au8[_word_window_i_u8++] = 0;

```

```
                            if(_word_window_i_u8 == 4){
                                /* Window complete, lets flash word */
                                flashWord_e(_new_app_current_addr_vu32,
                                            ARR_TO_INT32(_word_window_au8, 0));
                                _new_app_current_addr_vu32 += 4;
                            }
                        }
                    }
                    _pckg_index_vu32 = ota_pckg_index_u16 + 1;

                    return S_NEW_APP_CHECKSUM;
                }
                default:{
                    lockFlash_v();
                    return S_FAILURE;
                }
            }
    } while (delta_i_u16 < delta_len_u16);

    lockFlash_v();

    _pckg_index_vu32 = ota_pckg_index_u16 + 1;

    /* Continue flashing */
    return S_NEW_APP_FLASH_PROGRAM;
}
```

Code A.34: OTA new application program state.

When the delta reception is finalized, the OTA's next state is the New Application Checksum State, which implementation is as illustrated in Figure A.35. This state iterates through the new application space summing all the bytes. After having the sum, it is done the mod of 256. The result is compared with the checksum received in the OTA header. If the checksum equals, the new application was written successfully, and the current state machine state is changed to the end state to end the update.

```
static nextstate_t newAppChecksumState(...){
    uint32_t sum_u32 = 0;
    uint32_t read_word_u32;

    unlockFlash_v();
    for(uint32_t i_u32 = 0;
        i_u32 + _new_app_begin_addr_vu32 < _new_app_end_addr_vu32;
        i_u32 += 4){
        read_word_u32 = readWordFromFlash_u32(i_u32 +
                                               _new_app_begin_addr_vu32);

        sum_u32 += (read_word_u32 & 0xFF) + ((read_word_u32>>8) & 0xFF) +
                   ((read_word_u32>>16)&0xFF)+((read_word_u32>>24)&0xFF);
    }

    uint8_t checksum_u8 = sum_u32 % 256;
    if(checksum_u8 != _new_app_checksum_vu8){
        lockFlash_v();
        return S_FAILURE;
    }
    lockFlash_v();
    return S_END_OTA;
}
```

Code A.35: OTA new application checksum state.

The OTA end-state purpose is to signal the bootloader that a new application is ready to jump to and the application that a valid Over-The-Air update has been done. Therefore, it starts by erasing the bootloader hook variables and writing the new application address and magic key in the bootloader. Finally, it signals by an event flag the application and set the following state machine to wait for restart state.

```
static nextstate_t endOTAUpdateState(...){
    unlockFlash_v();

    eraseFlash_e(CONFIGS_PAGE_ADDR, 1);

    /* Set bootloader new application address */
    if(TO_UINT32(current_app_addr_cpv) == OTA_APP1_ADDR){
        /* Set APP2 address*/
        flashWord_e(BOOTLOADER_NEW_APP_ADDR, OTA_APP2_ADDR);
    }
    else{
        /* Set APP1 address*/
        flashWord_e(TO_UINT32(BOOTLOADER_NEW_APP_ADDR), OTA_APP1_ADDR);
    }

    /* Set the Magic Key */
    flashWord_e(BOOTLOADER_MAGIC_KEY_ADDR, OTA_MAGIC_KEY);

    lockFlash_v();

    /* Signal application to reset */
    tx_event_flags_set(&a_eventflags, a_ota_update_ef, TX_OR);

    return S_WAIT_FOR_RESTART;
}
```

Code A.36: OTA end state.

The wait for restart state, represented in Code A.37, as the end state signals the application that a valid update has been performed and returns the current state. This state maintains the integrity of the received new application to protect against new OTA updates before application restart.

```
static nextstate_t waitForRestartState(...){
    /* Signal application to reset */
    tx_event_flags_set(&a_eventflags, a_ota_update_ef, TX_OR);

    return S_WAIT_FOR_RESTART;
}
```

Code A.37: OTA wait for restart state.

A.1.8 Application

The application has the application manager thread. This thread, depicted in Code A.38, is responsible for the system startup and the send and application error handling loop.

```
void appManagerThreadEntry_v(unsigned long thread_input_ul){

    /* Waits until the sensors are initialized
       and then enable sensors sampling */
    ...

    /* Waits until the modem initializes and connects to the network */
    ...

    systemControlSetDatetimeUpdateFunction_v(communicationGetDateTime);

    /*Enables system control thread RTC update*/
    tx_event_flags_set(&sc_eventflags, sc_enable_rtc_update, TX_OR);

    /* Gets the board ID from modem sim card*/
    id_u64 = getModemID_u64();

    /* If the sensors are initialized, waits until the rtc is updated to
   set the first alarm */
    ...

    /* Initializes RSA and AES encryption */
    ...

    /* Starts the command manager thread */
    tx_thread_resume(&cmd_manager_thread);

    /* Activates send config flag */
    tx_event_flags_set(&a_eventflags, a_config_ef, TX_OR);

    while(1){
        /* Wait for application event flags */
        tx_event_flags_get(&a_eventflags, app_mng_efs_cu32, TX_OR_CLEAR,
                              &actual_flags_ul, TX_WAIT_FOREVER);

        /* Handle event flag */
        ...
    }
}
```

Code A.38: Application manager thread entry.

The system initialization starts by waiting for a signal that the sensors are initialized. It enables the sensors' sampling if there are sensors. Then, it waits for the modem initialization. When the modem is initialized, the system control date-time update function pointer is set to the `communicationGetDateTime` and enabled the system control refresh RTC event flag. After, this thread gets the board id, which is the modem ID. If there are sensors and the RTC is updated, it is set the first regular alarm. Moreover, it is initialized the RSA and AES encryption. The system is properly started, the commands manager thread is resumed, and the configuration event flag is set.

The main loop handles the configuration, emergency, regular, command response and error application event flags. Code A.39 represents the configuration event flag handling as exemplifying a transmission. It starts by getting the shared data binary semaphore. This shared data will be used to save the configuration's datagram objects. Then a new transmission and the configuration microservice are set up. The transmission is ready for appending its datagram objects. Therefore, it appends the objects and sends them. The transmission is ended and released the shared data binary semaphore. The other transmissions follow a similar structure.

```
    if(actual_flags_ul & a_config_ef){
        tx_bin_semaphore_get(&sharedata_bsem, TX_WAIT_FOREVER);

        communicationSetupNewTransmission_v();

        communicationSetupMicroservice_v(config_microservice);

        _appendConfigDatagramObjects_iv();

        communicationSendDatagram_e(2);

        communicationEndTransmission_v();

        tx_bin_semaphore_put(&sharedata_bsem);
    }
```

Code A.39: Configuration event flag handle.

The error handling is composed of the error signal and handle. The signal can be performed with the function represented in A.40. It receives the error, changes the error variable and sets the error event flag. There is no thread synchronisation since there is only one application error (the getting date time error). When the errors increase, a binary semaphore will be added.

```
void signalError_v(status_et error_e){
    unsigned long actual_flags_ul;

    _error_vu8 = error_e;

    tx_event_flags_set(&a_eventflags, a_error_ef, TX_OR);

    tx_event_flags_get(&a_eventflags, a_error_solved_ef,
                       TX_OR_CLEAR, &unused_var, TX_WAIT_FOREVER);

    return;
}
```

Code A.40: Error signal function.

The application error handling code is illustrated in Code A.41. If the error event flag is set, it switches to the respective error case. If the date-time error is signalled, it gets the modem binary semaphore and signals a communication date-time error. When the error is fixed, it is set the application error solved event flag.

```
    if(actual_flags_ul & a_error_ef){
        switch(_error_vu8){
            case getdatetime:
                tx_bin_semaphore_get(&modem_bsem, TX_WAIT_FOREVER);
                signalCommunicationError_v(datetime_error_ef);
                tx_bin_semaphore_put(&modem_bsem);
                break;
            default: break;
        }
        tx_event_flags_set(&a_eventflags, a_error_solved_ef, TX_OR);
    }
```

Code A.41: Application error handling.

The application manager thread agglomerates all the transmissions and error handling to save memory RAM. As a result, it is only required that the application manager stack size be large enough to the RSA encryption functions.

A.2 OTA Result Code Changes

```c
uint32_t end_i_u32;
uint32_t begin_i_u32 = 2;
char* token_pc = NULL; /* useful for parsing */

while(1){
  /* Retrieve a message from the queue.  */
  tx_queue_receive(&m_rx_queue, &end_i_u32, TX_WAIT_FOREVER);

  /* Process Received Message */
  if (circStrstr((const char*)m_rx_au8,"OK", begin_i_u32,
      M_RX_BUFFER_SIZE) != NULL){
    tx_event_flags_set(&m_eventflags, ok_ef, TX_OR);
  }
  else if(circStrstr((const char*)m_rx_au8,"QIRD", begin_i_u32,
          M_RX_BUFFER_SIZE) != NULL){
    /* gets the string until the token_pc ':' */
    token_pc = circStrtok((char*)m_rx_au8,":", begin_i_u32,
                           M_RX_BUFFER_SIZE);
    if(token_pc != NULL){  /* alwyas true if message received */
      token_pc = circStrtok(NULL, "\0", NULL, M_RX_BUFFER_SIZE);
      if(token_pc != NULL){ /* all ':' strings start with + */
        char *ptr_pc;

        uint16_t len_rcv_u16 = TO_UINT16(circStrtoul(CHAR_PTR(m_rx_au8),
                &ptr_pc, 10,
                begin_i_u32+(token_pc-CHAR_PTR(&m_rx_au8[begin_i_u32])),
                M_RX_BUFFER_SIZE));

        circStrtoul(CHAR_PTR(m_rx_au8), &ptr_pc, 10,
              begin_i_u32+(&ptr_pc[1]-CHAR_PTR(&m_rx_au8[begin_i_u32])),
              M_RX_BUFFER_SIZE);

        if(ptr_pc[0] == ',') {
          circMemcpy(nb_iot.cloud_dest_pau8 + nb_iot.data_rcv_u16,
            m_rx_au8, len_rcv_u16,
            begin_i_u32+(&ptr_pc[1]-CHAR_PTR(&m_rx_au8[begin_i_u32])),
            M_RX_BUFFER_SIZE);

          nb_iot.data_rcv_u16 += len_rcv_u16;

          /* Set event flag to alert URC received.  */
          tx_event_flags_set(&m_eventflags, rcv_data_ef, TX_OR);
        }
      }
    }
  }
  else if (circStrstr((const char*)m_rx_au8, ">", begin_i_u32,
           M_RX_BUFFER_SIZE) != NULL){
    tx_event_flags_set(&m_eventflags, data_mode_ef, TX_OR);
  }
  else if (circStrstr((const char*)m_rx_au8,"ERROR", begin_i_u32,
           M_RX_BUFFER_SIZE) != 0){
    /* gets the string until the token_pc ':' */
    token_pc = circStrtok((char*)m_rx_au8,":", begin_i_u32,
                          M_RX_BUFFER_SIZE);
    if(token_pc != NULL){  /* alwyas true if message received */
      /* checks if in fact there is a string terminated in ':' */
      token_pc = circStrtok(NULL, "\0", NULL, M_RX_BUFFER_SIZE);
      if(token_pc != NULL){ /* all ':' strings start with + */
        /* gets the informative message wether it was an error
           or a   simple response */
        circStrlcpy((char*)nb_iot.last_msg,(const char*)m_rx_au8,
                       sizeof(nb_iot.last_msg),
                       begin_i_u32+(token_pc-(char*)&m_rx_au8[begin_i_u32]),
   M_RX_BUFFER_SIZE);
      }
    }
    else{
      tx_event_flags_set(&m_eventflags, error_ef, TX_OR);
    }
  }
  else{
    /* gets the string until the token_pc ':' */
    token_pc = circStrtok((char*)m_rx_au8,":", begin_i_u32,
                            M_RX_BUFFER_SIZE);
    if(token_pc != NULL){  /* alwyas true if message received */
      /* checks if in fact there is a string terminated in ':' */
      token_pc = circStrtok(NULL, "\0", NULL, M_RX_BUFFER_SIZE);
      if(token_pc != NULL){ /* all ':' strings start with + */
        /* gets the informative message wether it was an error
           or a simple response */
        circStrlcpy((char*)nb_iot.last_msg,
          (const char*)m_rx_au8, sizeof(nb_iot.last_msg),
          begin_i_u32+(token_pc-(char*)&m_rx_au8[begin_i_u32]),
          M_RX_BUFFER_SIZE);

        if(circStrstr((const char*)m_rx_au8,"QIURC", begin_i_u32,
                           M_RX_BUFFER_SIZE) != NULL){
          memcpy(nb_iot.cloud_urc_au8, nb_iot.last_msg,
                 CLOUD_URC_BUFFER_SIZE);

          /* Set event flag to alert URC received.  */
          tx_event_flags_set(&m_eventflags, urc_ef, TX_OR);
        }
      #ifdef USE_TCP
        else if(circStrstr((const char*)m_rx_au8,"QISTATE", begin_i_u32,
                M_RX_BUFFER_SIZE) != NULL){
            /* Set event flag to alert URC received.  */
            tx_event_flags_set(&m_eventflags, socket_state_ef, TX_OR);
        }
      #endif
        else if(circStrstr((const char*)m_rx_au8,"QPING", begin_i_u32,
                M_RX_BUFFER_SIZE) != NULL){
          /* Set event flag to alert URC received.  */
          tx_event_flags_set(&m_eventflags, ping_ef, TX_OR);
        }
        else if(circStrstr((const char*)m_rx_au8,"QIOPEN", begin_i_u32,
                M_RX_BUFFER_SIZE) != NULL){
          /* Answer example: +QIOPEN: 0,0 */
          /* Set event flag to alert Open Socket received.  */
          tx_event_flags_set(&m_eventflags, socket_open_ef, TX_OR);
        }
        else if(circStrstr((const char*)m_rx_au8, "IP", begin_i_u32,
                M_RX_BUFFER_SIZE) != NULL){
          /* Set event flag to alert that is connected to the network.  */
          tx_event_flags_set(&m_eventflags, m_connected_ef, TX_OR);
        }
        else if (circStrstr((const char*)m_rx_au8, "CPIN", begin_i_u32,
                 M_RX_BUFFER_SIZE) != NULL){
          if(circStrstr((const char*)m_rx_au8, "NOT",
              begin_i_u32+(token_pc-(char*)&m_rx_au8[begin_i_u32]),
              M_RX_BUFFER_SIZE) == NULL){
              tx_event_flags_set(&m_eventflags, sim_rdy_ef, TX_OR);
          }
        }
      }
      else{
        /* there is no string terminated in ':' */
        circStrlcpy((char*)nb_iot.last_msg,(const char*)m_rx_au8,
                     sizeof(nb_iot.last_msg),
                    begin_i_u32, M_RX_BUFFER_SIZE);
          /* gets the informative message */
      }
    }
    if (circStrstr((const char*)m_rx_au8,"RDY", begin_i_u32,
        M_RX_BUFFER_SIZE) != 0){
      tx_event_flags_set(&m_eventflags, rdy_ef, TX_OR);
    }
  }
#ifdef M_RX_BUFFER_AND_MASK
    begin_i_u32 = (end_i_u32 + 1)  & (M_RX_BUFFER_SIZE-1);
#else
    begin_i_u32 = (end_i_u32 + 1)  % M_RX_BUFFER_SIZE;
#endif
}
```

Code A.42: Old modem parse UART messages thread code.

```
uint32_t end_i_u32;
uint32_t begin_i_u32 = 2;
char* token_pc = NULL; /* useful for parsing */
uint32_t total_u32 = 0;
uint32_t index_u32;

startRx_IT(MODEM);

while(1){
  /* Retrieve a message from the queue.  */
  tx_queue_receive(&m_rx_queue, &end_i_u32, TX_WAIT_FOREVER);

  index_u32 = begin_i_u32;
  total_u32 = 0;

  while(m_rx_au8[index_u32] != '\0' && m_rx_au8[index_u32] != ':') {
    if(m_rx_au8[index_u32] != '\n' && m_rx_au8[index_u32] != '\r') {
      /* Sum every byte in order to have a unique "key" so it can
          be parsed accordingly */
      total_u32 += m_rx_au8[index_u32];
    }
    index_u32 += 1;
    index_u32 &= (M_RX_BUFFER_SIZE - 1);
  }

  /* Parse the command received */
  switch(total_u32) {
  case OK_CMD:
    if (circStrstr((const char*)m_rx_au8,"OK", begin_i_u32,
        M_RX_BUFFER_SIZE) != NULL){
        tx_event_flags_set(&m_eventflags, ok_ef, TX_OR);
    }
    break;
  case QIRD_CMD:
    if (circStrstr((const char*)m_rx_au8,"QIRD", begin_i_u32,
        M_RX_BUFFER_SIZE) != NULL){
      char *ptr_pc;
      token_pc = (char*)&m_rx_au8[index_u32 + 1];

      uint16_t len_rcv_u16 = TO_UINT16(
        circStrtoul(CHAR_PTR(m_rx_au8), &ptr_pc, 10,
          begin_i_u32+(token_pc-CHAR_PTR(&m_rx_au8[begin_i_u32])),
          M_RX_BUFFER_SIZE));

      circStrtoul(CHAR_PTR(m_rx_au8), &ptr_pc, 10,
        begin_i_u32+(&ptr_pc[1]-CHAR_PTR(&m_rx_au8[begin_i_u32])),
        M_RX_BUFFER_SIZE);

      if(ptr_pc[0] == ',') {
        circMemcpy(nb_iot.cloud_dest_pau8 + nb_iot.data_rcv_u16,
          m_rx_au8, len_rcv_u16,
          begin_i_u32+(&ptr_pc[1]-CHAR_PTR(&m_rx_au8[begin_i_u32])),
          M_RX_BUFFER_SIZE);

        nb_iot.data_rcv_u16 += len_rcv_u16;

        /* Set event flag to alert URC received.  */
        tx_event_flags_set(&m_eventflags, rcv_data_ef, TX_OR);
      }
    }
    break;

  case DATA_MODE_CMD:
    if (circStrstr((const char*)m_rx_au8,">", begin_i_u32,
        M_RX_BUFFER_SIZE) != NULL){
      tx_event_flags_set(&m_eventflags, data_mode_ef, TX_OR);
    }
    break;

  case ERROR_CMD:
case CME_ERROR_CMD:
  if (circStrstr((const char*)m_rx_au8,"ERROR", begin_i_u32,
      M_RX_BUFFER_SIZE) != NULL){
    if(m_rx_au8[index_u32] == ':') {
      token_pc = (char*)&m_rx_au8[index_u32 + 1];

      /* Gets the informative message wether it was an error or
         a simple response */
      circStrlcpy((char*)nb_iot.last_msg,(const char*)m_rx_au8,
                  sizeof(nb_iot.last_msg), begin_i_u32+(token_pc-
                  (char*)&m_rx_au8[begin_i_u32]),
                  M_RX_BUFFER_SIZE);
    }
    tx_event_flags_set(&m_eventflags, error_ef, TX_OR);
  }
  break;

  case QENG_CMD:
    if (circStrstr((const char*)m_rx_au8,"QENG", begin_i_u32,
        M_RX_BUFFER_SIZE) != NULL){
      token_pc = (char*)&m_rx_au8[index_u32 + 1];
      token_pc+1 == (char*) &m_rx_au8[M_RX_BUFFER_SIZE] ?
                        token_pc = (char*)m_rx_au8 : token_pc++;

      if (token_pc[0] != '1'){
        circStrlcpy((char*)nb_iot.last_msg,(const char*)m_rx_au8,
          sizeof(nb_iot.last_msg),
          begin_i_u32+(token_pc-(char*)&m_rx_au8[begin_i_u32]),
          M_RX_BUFFER_SIZE);
      }
    }
    break;

  case QPING_CMD:
    if (circStrstr((const char*)m_rx_au8,"QPING", begin_i_u32,
        M_RX_BUFFER_SIZE) != NULL){
      token_pc = (char*)&m_rx_au8[index_u32 + 1];

      /* Gets the informative message wether it was an error or
         a simple response */
      circStrlcpy((char*)nb_iot.last_msg,(const char*)m_rx_au8,
          sizeof(nb_iot.last_msg),
          begin_i_u32+(token_pc-(char*)&m_rx_au8[begin_i_u32]),
          M_RX_BUFFER_SIZE);

      /* Set event flag to alert URC received.  */
      tx_event_flags_set(&m_eventflags, ping_ef, TX_OR);
    }
    break;

  case QIURC_CMD:
    if (circStrstr((const char*)m_rx_au8,"QIURC", begin_i_u32,
        M_RX_BUFFER_SIZE) != NULL){
      token_pc = (char*)&m_rx_au8[index_u32 + 1];

      /* Gets the informative message wether it was an error or
         a simple response */
      circStrlcpy((char*)nb_iot.last_msg,(const char*)m_rx_au8,
                   sizeof(nb_iot.last_msg),
                  begin_i_u32+(token_pc-(char*)&m_rx_au8[begin_i_u32]),
                  M_RX_BUFFER_SIZE);

      memcpy(nb_iot.cloud_urc_au8, nb_iot.last_msg, CLOUD_URC_BUFFER_SIZE);

      /* Set event flag to alert URC received.  */
      tx_event_flags_set(&m_eventflags, urc_ef, TX_OR);
    }
    break;

#ifdef USE_TCP
  case QISTATE_CMD:
    if (circStrstr((const char*)m_rx_au8,"QISTATE", begin_i_u32,
        M_RX_BUFFER_SIZE) != NULL){
      token_pc = (char*)&m_rx_au8[index_u32 + 1];

      /* Gets the informative message wether it was an error or
         a simple response */
      circStrlcpy((char*)nb_iot.last_msg,(const char*)m_rx_au8,
                sizeof(nb_iot.last_msg),
                begin_i_u32+(token_pc-(char*)&m_rx_au8[begin_i_u32]),
                M_RX_BUFFER_SIZE);

      /* Set event flag to alert URC received.  */
      tx_event_flags_set(&m_eventflags, socket_state_ef, TX_OR);
    }
    break;
#endif

  case QIOPEN_CMD:
    if (circStrstr((const char*)m_rx_au8,"QIOPEN", begin_i_u32,
        M_RX_BUFFER_SIZE) != NULL){
      token_pc = (char*)&m_rx_au8[index_u32 + 1];

      /* Gets the informative message wether it was an error or a simple
   response */
      circStrlcpy((char*)nb_iot.last_msg,(const char*)m_rx_au8, sizeof(
   nb_iot.last_msg),
                  begin_i_u32+(token_pc-(char*)&m_rx_au8[begin_i_u32]),
    M_RX_BUFFER_SIZE);

      /* Answer example: +QIOPEN: 0,0 */
      /* Set event flag to alert Open Socket received.  */
      tx_event_flags_set(&m_eventflags, socket_open_ef, TX_OR);
    }
    break;

  case IP_CMD:
    if (circStrstr((const char*)m_rx_au8,"IP", begin_i_u32,
                    M_RX_BUFFER_SIZE) != NULL){
      token_pc = (char*)&m_rx_au8[index_u32 + 1];

      /* Gets the informative message wether it was an error
         or a simple response */
      circStrlcpy((char*)nb_iot.last_msg,(const char*)m_rx_au8,
                  sizeof(nb_iot.last_msg),
                  begin_i_u32+(token_pc-(char*)&m_rx_au8[begin_i_u32]),
                  M_RX_BUFFER_SIZE);

      /* Set event flag to alert that is connected to the network.  */
      tx_event_flags_set(&m_eventflags, m_connected_ef, TX_OR);
    }
    break;

  case CPIN_CMD:
    if (circStrstr((const char*)m_rx_au8,"CPIN", begin_i_u32,
        M_RX_BUFFER_SIZE) != NULL){
      token_pc = (char*)&m_rx_au8[index_u32 + 1];

      if(circStrstr((const char*)m_rx_au8, "NOT", begin_i_u32+
        (token_pc-(char*)&m_rx_au8[begin_i_u32]), M_RX_BUFFER_SIZE)==NULL){
          tx_event_flags_set(&m_eventflags, sim_rdy_ef, TX_OR);
      }
    }
      break;

  case RDY_CMD:
    if (circStrstr((const char*)m_rx_au8,"RDY", begin_i_u32,
        M_RX_BUFFER_SIZE) != NULL){
      token_pc = (char*)&m_rx_au8[index_u32 + 1];

      /* Gets the informative message wether it was an error
         or a simple response */
      circStrlcpy((char*)nb_iot.last_msg,(const char*)m_rx_au8,
                  sizeof(nb_iot.last_msg),
                 begin_i_u32+(token_pc-(char*)&m_rx_au8[begin_i_u32]),
                 M_RX_BUFFER_SIZE);

      tx_event_flags_set(&m_eventflags, rdy_ef, TX_OR);
    }
      break;

  case SEND_OK_CMD:
    if (circStrstr((const char*)m_rx_au8,"SEND OK", begin_i_u32,
        M_RX_BUFFER_SIZE) != NULL){
      tx_event_flags_set(&m_eventflags, send_ok_ef, TX_OR);
    }
    break;

  case SEND_FAIL_CMD:
    if (circStrstr((const char*)m_rx_au8,"SEND FAIL", begin_i_u32,
        M_RX_BUFFER_SIZE) != NULL){
      tx_event_flags_set(&m_eventflags, error_ef, TX_OR);
    }
```

```
      break;
    case CLOSE_OK_CMD:
      if (circStrstr((const char*)m_rx_au8,"CLOSE OK", begin_i_u32,
          M_RX_BUFFER_SIZE) != NULL){
        tx_event_flags_set(&m_eventflags, close_ok_ef, TX_OR);
      }
      break;

    default:
      if(m_rx_au8[index_u32] == ':') {
        token_pc = (char*)&m_rx_au8[index_u32 + 1];

        /* Gets the informative message wether it was an error
           or a simple response */
        circStrlcpy((char*)nb_iot.last_msg,(const char*)m_rx_au8,
                    sizeof(nb_iot.last_msg),
                    begin_i_u32+(token_pc-(char*)&m_rx_au8[begin_i_u32]),
                    M_RX_BUFFER_SIZE);
      }
      else {
        /* Gets the informative message */
        circStrlcpy((char*)nb_iot.last_msg,(const char*)m_rx_au8,
                      sizeof(nb_iot.last_msg),
                      begin_i_u32, M_RX_BUFFER_SIZE);
      }
      break;
    }

#ifdef M_RX_BUFFER_AND_MASK
  begin_i_u32 = (end_i_u32 + 1)  & (M_RX_BUFFER_SIZE-1);
#else
  begin_i_u32 = (end_i_u32 + 1)  % M_RX_BUFFER_SIZE;
#endif
}
```

Code A.43: New modem parse UART messages thread code.

A.3 Code

A.3.1 Cryptography

```
#include "cryptography.h"

#ifdef CRYPTO_EN
/* ThreadX Variables Declaration. */
static TX_SEMAPHORE crypto_bsem;

#ifdef USE_ST_AES_ACC_HW
__align(8) static const uint8_t _key[16] = {...};
__align(8) static const uint8_t _iv[16] = {...};
#endif

typedef struct aes_data{
#ifdef USE_ATE
  uint8_t key[16];
  uint8_t iv[16];
#endif
#ifdef USE_ST_AES_ACC_HW
  const uint8_t *key_pu8c;
  const uint8_t *iv_pu8c;
  AccHw_AESECBctx_stt ctx;
#endif
}aes_t;

#if !defined(USE_STATIC_CRYPTO)
typedef struct rsa_data{
#if defined(USE_ST_RSA_SW)
#ifdef RSA_ENCRYPTION
  RSApubKey_stt PubKey_st;
#endif
#ifdef RSA_DECRYPTION
  RSAprivKey_stt PrivKey_st;
  uint8_t PrivateExponent[RSA_SIZE/8];
#endif
#endif
#ifdef RSA_ENCRYPTION
#if defined(USE_ST_RSA_ACC_HW)
  AccHw_RSApubKey_stt PubKey_st;
#endif
  uint8_t Modulus[RSA_SIZE/8];
#endif
}rsa_t;


#if defined(USE_ST_RSA_SW)
uint8_t entropy_data[32] = { ... };
#endif

#if defined(USE_ST_RSA_SW) || defined(USE_ST_RSA_ACC_HW)
const uint8_t PublicExponent[] =
  {
  0x01, 0x00, 0x01
  };
static rsa_t rsa;
#endif

/* buffer required for internal allocation of memory */
#if defined(USE_ST_RSA_SW) || defined(USE_ST_RSA_ACC_HW)
uint8_t preallocated_buffer[4096];
#endif

#endif

aes_t aes = {0};

```

```
#ifdef USE_ST_AES_ACC_HW
static inline int32_t aesSTEnc(uint8_t * plaintext_pu8, uint32_t
    size_plaintext_u32, uint8_t * ciphertext_pu8, int32_t* output_msg_size);
static inline int32_t aesSTDec(uint8_t * ciphertext_pu8, uint32_t
    size_cyphertext_u32, uint8_t * plaintext_pu8,int32_t* output_msg_size);
#endif


#if !defined(USE_STATIC_CRYPTO)
/**
  * @brief  Initialises RSA Encryption with PKCS#1v1.5
  * @retval error status: SUCCESS in case of success and
  * FAILURE in case of failure
*/
status_et RSA_Init(void){

  /* Enable CRC clock */
  __CRC_CLK_ENABLE();

#ifdef RSA_ENCRYPTION
#if defined(USE_ST_RSA_SW) || defined(USE_ST_RSA_ACC_HW)
  /* Preparing for Encryption */
  rsa.PubKey_st.mExponentSize = sizeof(PublicExponent);
  rsa.PubKey_st.mModulusSize = sizeof(rsa.Modulus);
  rsa.PubKey_st.pmExponent = (uint8_t *) PublicExponent;
  rsa.PubKey_st.pmModulus = (uint8_t *)rsa.Modulus;
#endif
#endif

#ifdef RSA_DECRYPTION
#if defined(USE_ST_RSA_SW) || defined(USE_ST_RSA_ACC_HW)
  /* Preparing for Decryption */
  rsa.PrivKey_st.mExponentSize = sizeof(PrivateExponent);
  rsa.PrivKey_st.mModulusSize = sizeof(rsa.Modulus);
  rsa.PrivKey_st.pmExponent = (uint8_t *) PrivateExponent;
  rsa.PrivKey_st.pmModulus = (uint8_t *) rsa.Modulus;
#endif
#endif
  return success;
}

/**
  * @brief  Refresh the modulus with the modulus received in @ref rsa_t
    structure
  * @retval None
*/
void RSA_setModulus(const uint8_t *new_modulus){
  uint16_t i;

  for(i=0; i<RSA_SIZE/8; i++){
    rsa.Modulus[i] = new_modulus[i];
  }

  return;
}

#ifdef RSA_ENCRYPTION
/**
  * @brief  RSA Encryption with PKCS#1v1.5
  * @param  P_pPubKey The RSA public key structure, already initialized
  * @param  P_pInputMessage Input Message to be signed
  * @param  P_MessageSize Size of input message
  * @param  P_pOutput Pointer to output buffer
  * @retval error status: can be RSA_SUCCESS if success or one of
  * RSA_ERR_BAD_PARAMETER, RSA_ERR_MESSAGE_TOO_LONG, RSA_ERR_BAD_OPERATION
*/
status_et RSA_Encrypt(const uint8_t *P_pInputMessage,
          int32_t P_InputSize,
          uint8_t *P_pOutput)
{
  int32_t status = RNG_SUCCESS ;
#ifdef USE_ST_RSA_SW
  RNGstate_stt RNGstate;
  RNGinitInput_stt RNGinit_st;
  RNGinit_st.pmEntropyData = entropy_data;
  RNGinit_st.mEntropyDataSize = sizeof(entropy_data);
  RNGinit_st.mPersDataSize = 0;
  RNGinit_st.mNonceSize = 0;

  status = RNGinit(&RNGinit_st, &RNGstate);
  if (status == RNG_SUCCESS){
    RSAinOut_stt inOut_st;
    membuf_stt mb;

    mb.mSize = sizeof(preallocated_buffer);
    mb.mUsed = 0;
    mb.pmBuf = preallocated_buffer;

    /* Fill the RSAinOut_stt */
    inOut_st.pmInput = P_pInputMessage;
    inOut_st.mInputSize = P_InputSize;
    inOut_st.pmOutput = P_pOutput;

    /* Encrypt the message, this function will write sizeof(modulus) data
   */
    status = RSA_PKCS1v15_Encrypt(&rsa.PubKey_st, &inOut_st, &RNGstate, &mb
```

```c
    );
    }
#endif
#ifdef USE_ST_RSA_ACC_HW

  AccHw_RSAinOut_stt inOut_st;
  membuf_stt mb;

  mb.mSize = sizeof(preallocated_buffer);
  mb.mUsed = 0;
  mb.pmBuf = preallocated_buffer;

  /* Fill the RSAinOut_stt */
  inOut_st.pmInput = P_pInputMessage;
  inOut_st.mInputSize = P_InputSize;
  inOut_st.pmOutput = P_pOutput;

  /* Encrypt the message, this function will write sizeof(modulus) data */
  status = AccHw_RSA_PKCS1v15_Encrypt(&rsa.PubKey_st, &inOut_st,  &mb);
#endif
  if(status == RNG_SUCCESS){
    return success;
  }

  return failure;
}
#endif

#ifdef RSA_DECRYPTION
/**
  * @brief  RSA Decryption with PKCS#1v1.5
  * @param  P_pPrivKey The RSA private key structure, already initialized
  * @param  P_pInputMessage Input Message to be signed
  * @param  P_MessageSize Size of input message
  * @param  P_pOutput Pointer to output buffer
  * @retval error status: can be RSA_SUCCESS if success or RSA_ERR_GENERIC
    in case of fail
*/
status_et RSA_Decrypt(const uint8_t * P_pInputMessage,
          uint8_t *P_pOutput,
          int32_t *P_OutputSize)
{
  int32_t status = RSA_SUCCESS ;
#ifdef USE_ST_RSA_SW
  RSAinOut_stt inOut_st;
  membuf_stt mb;

  mb.mSize = sizeof(preallocated_buffer);
  mb.mUsed = 0;
  mb.pmBuf = preallocated_buffer;

  /* Fill the RSAinOut_stt */
  inOut_st.pmInput = P_pInputMessage;
  inOut_st.mInputSize = rsa.PrivKey_st.mModulusSize;
  inOut_st.pmOutput = P_pOutput;

  /* Encrypt the message, this function will write sizeof(modulus) data */
  status = RSA_PKCS1v15_Decrypt(&rsa.PrivKey_st, &inOut_st, P_OutputSize, &
    mb);
#endif
#ifdef USE_ST_RSA_ACC_HW
  AccHw_RSAinOut_stt inOut_st;
  membuf_stt mb;

  mb.mSize = sizeof(preallocated_buffer);
  mb.mUsed = 0;
  mb.pmBuf = preallocated_buffer;

  /* Fill the RSAinOut_stt */
  inOut_st.pmInput = P_pInputMessage;
  inOut_st.mInputSize = rsa.PrivKey_st.mModulusSize;
  inOut_st.pmOutput = P_pOutput;

  /* Encrypt the message, this function will write sizeof(modulus) data */
  status = AccHw_RSA_PKCS1v15_Decrypt(&rsa.PrivKey_st, &inOut_st,
    P_OutputSize, &mb);
#endif
  if(status == RSA_SUCCESS){
    return success;
  }
  return failure;
}
#endif

#endif

/**
  * @brief  Initialises the EAS128 encrypton
  * @retval error status: success in case of success and
  * failure in case of failure
*/
status_et AES_Init(void){
  status_et status_e = failure;

  tx_bin_semaphore_get(&crypto_bsem, TX_WAIT_FOREVER); /* Locks the crypto
    binary mutex*/
```

```c
#ifdef USE_ATE
  uint8_t iv_aux[32];

  if(atecc608a_init_config() == success){
    atecc608a_random(iv_aux);
    memcpy(aes.iv, iv_aux, 16);

    status_e = (
    status_et)atecc608a_aes_key(iv_aux);
    memcpy(aes.key, iv_aux, 16);
    tx_bin_semaphore_put(&crypto_bsem); /* Releases the crypto binary semaphore */
  }
#endif
#ifdef USE_ST_AES_ACC_HW
  aes.iv_pu8c = _iv;
  aes.key_pu8c = _key;

  /* Set flag field to default value */
  aes.ctx.mFlags = AccHw_E_SK_DEFAULT;

  /* Set key size to 16 (corresponding to AES-128) */
  aes.ctx.mKeySize = 16;

  /* Set iv size field to IvLength*/
  aes.ctx.mIvSize = 16;
#endif
  tx_bin_semaphore_put(&crypto_bsem); /* Releases the crypto binary semaphore */
  return status_e;
}


/**
  * @brief Copies the AES key to the received pointer
  * @param kdf_gen_AES_key is where the key will be copied
  * @retval error status: success in case of success and
  * failure in case of failure
*/
status_et AES_getKey(uint8_t* dest_key, uint8_t offset, uint8_t size){
#ifdef USE_ATE
  memcpy(dest_key, aes.key+offset, 16-offset);
#endif
#ifdef USE_ST_AES_ACC_HW
  memcpy(dest_key, aes.key_pu8c+offset, 16-offset);
#endif
  return success;
}
```

```

/**
  * @brief Copies the IV array to the received pointer
  * @param dest_iv is where the IV will be copied
  * @retval error status: success in case of success and
  * failure in case of failure
*/
status_et AES_getIV(uint8_t* dest_iv, uint8_t offset, uint8_t size){

#ifdef USE_ATE
  memcpy(dest_iv, aes.iv+offset, size);
#endif
#ifdef USE_ST_AES_ACC_HW
  memcpy(dest_iv, aes.iv_pu8c+offset, 16-offset);
#endif
  return success;
}


/**
  * @brief Receives the plain text to be encrypted and it IV and encrypts
    it
  * @param kdf_gen_AES_key plain text to be encrypted
  * @param size_plaintext length of the plain text
  * @param ciphertext output plain text encrypted
  * @param iv_16bytes initialization vector used in AES encryption
  * @retval error status: success in case of success and
  * failure in case of failure
*/
status_et AES_Encrypt(uint8_t * plaintext_pu8, uint32_t size_plaintext_u32,
     uint8_t * ciphertext_pu8){
  status_et status_e = failure;
  tx_bin_semaphore_get(&crypto_bsem, TX_WAIT_FOREVER); /* Locks the crypto
    binary mutex*/
#ifdef USE_ATE
  status_e = (status_et)atecc608a_enc(plaintext_pu8, size_plaintext_u32,
    ciphertext_pu8, aes.iv);
#endif
#ifdef USE_ST_AES_ACC_HW
  int32_t output_msg_size_s16;
  if(aesSTEnc(plaintext_pu8, size_plaintext_u32, ciphertext_pu8, &
    output_msg_size_s16) != AES_SUCCESS){
     status_e = success;
  }
#endif
  tx_bin_semaphore_put(&crypto_bsem); /* Releases the crypto binary
    semaphore */
  return status_e;
```

```
}


/**
  * @brief Receives the cypher test to be decrypted and it IV and decrypts
    it
  * @param kdf_gen_AES_key plain text to be encrypted
  * @param size_plaintext length of the plain text
  * @param ciphertext output plain text encrypted
  * @param iv_16bytes initialization vector used in AES encryption
  * @retval error status: success in case of success and
  * failure in case of failure
*/
status_et AES_Decrypt(uint8_t * ciphertext_pu8, uint32_t
    size_cyphertext_u32,
                      uint8_t * plaintext_pu8){
  status_et status_e = failure;

  tx_bin_semaphore_get(&crypto_bsem, TX_WAIT_FOREVER);
#ifdef USE_ATE
  status_e = (status_et)atecc608a_dec(ciphertext_pu8, size_cyphertext_u32,
                                      plaintext_pu8, aes.iv);
#endif
#ifdef USE_ST_AES_ACC_HW
  int32_t output_msg_size_s16;
  if(aesSTDec(ciphertext_pu8, size_cyphertext_u32, plaintext_pu8,
              &output_msg_size_s16) == AES_SUCCESS){
    status_e = success;
  }
#endif
  tx_bin_semaphore_put(&crypto_bsem);
  return status_e;
}


void cryptographyTXAppDefine_v(void){
  /* Create the mutex to Modem mutex.  */
  tx_semaphore_create(&crypto_bsem, "Crypto BSEM", 1);

  return;
}

#ifdef USE_ST_AES_ACC_HW
static inline int32_t aesSTEnc(uint8_t * plaintext_pu8, uint32_t
    size_plaintext_u32,
                               uint8_t * ciphertext_pu8, int32_t*
    output_msg_size){
  int32_t out_lengtth_s32 = 0;
  uint32_t error_status = AES_SUCCESS;


  /* Initialize the operation, by passing the key.*/
  error_status = AccHw_AES_CFB_Encrypt_Init(&aes.ctx, aes.key_pu8c, aes.iv_pu8c);

  /* check for initialization errors */
  if (error_status == AES_SUCCESS){
    /* Encrypt Data */
    error_status = AccHw_AES_CFB_Encrypt_Append(&aes.ctx,
             plaintext_pu8,
             size_plaintext_u32,
             ciphertext_pu8,
             &out_lengtth_s32);

    if (error_status == AES_SUCCESS){
      /* Write the number of data written*/
      *output_msg_size = out_lengtth_s32;
      /* Do the Finalization */
      error_status = AccHw_AES_CFB_Encrypt_Finish(&aes.ctx, ciphertext_pu8 +
                                          *output_msg_size, &out_lengtth_s32);
      /* Add data written to the information to be returned */
      *output_msg_size += out_lengtth_s32;
    }
  }

  return error_status;
}

static inline int32_t aesSTDec(uint8_t * ciphertext_pu8, uint32_t size_cyphertext_u32,
                               uint8_t * plaintext_pu8, int32_t* output_msg_size){
  uint32_t error_status = AES_SUCCESS;
  int32_t outputLength;

  /* Initialize the operation, by passing the key. */
  error_status = AccHw_AES_CFB_Decrypt_Init(&aes.ctx, aes.key_pu8c, aes.iv_pu8c);

  /* check for initialization errors */
  if (error_status == AES_SUCCESS){
    /* Decrypt Data */
    error_status = AccHw_AES_CFB_Decrypt_Append(&aes.ctx,
             ciphertext_pu8,
```

```c
             size_cyphertext_u32,
             plaintext_pu8,
             &outputLength);
    if (error_status == AES_SUCCESS){
      /* Write the number of data written*/
      *output_msg_size = outputLength;
      /* Do the Finalization */
      error_status = AccHw_AES_CFB_Decrypt_Finish(&aes.ctx, plaintext_pu8 + *output_msg_size,
                                                   &outputLength);
      /* Add data written to the information to be returned */
      *output_msg_size += outputLength;
    }
  }
  return error_status;
}

#endif
#endif

#if defined(USE_MD5) && defined(USE_ST_MD5)
/**
  * @brief  MD5 HASH digest compute example.
  * @param  in_msg_pu8: pointer to input message to be hashed.
  * @param  in_msg_len_u32: input data message length in byte.
  * @param  digest_pu8: pointer to output parameter that will handle message digest
  * @param  digest_len_ps32: pointer to output digest length.
  * @retval success or failure according to @ref status_et
  */
status_et md5(uint8_t* in_msg_pu8, uint32_t in_msg_len_u32, uint8_t * digest_pu8)
{
  MD5ctx_stt P_pMD5ctx;
  uint32_t error_status = HASH_SUCCESS;
  int32_t  digest_len_s32;

  /* Set the size of the desired hash digest */
  P_pMD5ctx.mTagSize = CRL_MD5_SIZE;

  /* Set flag field to default value */
  P_pMD5ctx.mFlags = E_HASH_DEFAULT;

  error_status = MD5_Init(&P_pMD5ctx);

  /* check for initialization errors */
  if (error_status == HASH_SUCCESS)
  {
```

```
  /* Add data to be hashed */
  error_status = MD5_Append(&P_pMD5ctx,
                 in_msg_pu8,
                 in_msg_len_u32);

  if (error_status == HASH_SUCCESS)
  {
    /* retrieve */
    error_status = MD5_Finish(&P_pMD5ctx, digest_pu8, &digest_len_s32);
    return success;
  }
  }

  return failure;
}
#endif
```