Hochschule Luzern - Technik & Architektur
IET

Master of Science in Engineering
MASTER THESIS, DOCUMENTATION

PARALLEL MULTI-DEVICE FIRMWARE UPDATE OVER THE AIR WITH A LORAWAN NETWORK

Corsin Obrist
17. Juni 2022

Master-Thesis an der Hochschule Luzern - Technik & Architektur

Titel	**PARALLEL MULTI-DEVICE FIRMWARE UPDATE OVER THE AIR WITH A LORAWAN NETWORK**
Diplomandin/Diplomand	**Obrist, Corsin**
Master-Studiengang	**Master in Engineering**
Semester	**FS22**
Dozentin/Dozent	**Styger, Erich**
Expertin/Experte	**Vetterli, Christian**

Abstract Deutsch
Im Rahmen dieser Arbeit wurde eine Demonstrator-Infrastructur entwickelt, welches als Konzeptachweises den Prozess eines «Firmware updadates over the air» in einem LoRaWAN Netzwerk an mehrere «Nodes» gleichzeitig aufzeigen kann. Neben einer selbst entwickelten Hardware, die jene Nodes im Feld simuliert, und einem FirmwareOverTheAirUpdate-server (FUOTA), wurde ein Protokoll erarbeitet, das mit Hilfe von vordefinierten Spezifikationen der LoRa-Alliance ermöglicht, Firmware Patches parallel an eine Gruppe von Nodes zu senden. Ein speziell entwickelter Bootloader kann diesen Patch dann mit der aktuellen Firmware zu einer neuen Firmwareversion zusammenführen. Der schlussendlich entwickelte Demonstrator umfasst einen FUOTA-server, LoRaWAN Netzwerkserver, Gateway und eine LoRaWAN node. Der Demonstrator ist einer der ersten kompletten Beispielimplementierung des Konzepts «Firmware update over the air» in einem LoRaWAN Netzwerk und kann für zukünftige Entwicklungen und Recherchen als Anleitung dienen.

Abstract Englisch
In the scope of this thesis, a demonstrator infrastructure was developed as a proof-of-concept, which can demonstrate the process of a firmware update over the air in a LoRaWAN network to several nodes simultaneously. Besides a self-developed hardware, which simulates the nodes in the field and a FirmwareOverTheAirUpdate-server (FUOTA), a protocol was developed that allows to send firmware patches in parallel to a group of nodes using predefined specifications of the LoRa-Alliance. A specially developed bootloader can then merge these patches with the current firmware to create a new firmware version. The final version of the developed demonstrator includes a FUOTA server, LoRaWAN network server, gateway and a LoRaWAN node. The demonstrator is one of the first complete example implementations of the concept of firmware update over the air in a LoRaWAN network and can serve as a basis for future developments and research.

Ort, Datum Horw, 17.06.22

Contents

1. Introduction

This document contains the documentation of the project work done in the context of the thesis for the Master of Science in Engineering by Corsin Obrist. The client of the project is the company SensDRB and the Lucerne University of Applied Sciences and Arts, which already did different projects in cooperation in the area of LoRaWAN. In the latest of these projects, a LoRaWAN node, titled Tardigrade, was developed. Since IoT devices can be deployed individually and their functionalities are constantly improved, it is necessary to be able to update such devices in the field. Because such IoT devices are often installed in poorly accessible terrain, it is necessary to make the process of firmware updating customer-friendly and resource-optimized. One possible solution is updating these IoT devices over the air.
The goal of the project is to channel the knowledge gained in the previous work [52] [53] and to develop a demonstrator to enable a parallel multi-device firmware update over the air with the use of the ultra low bandwidth LoRaWAN network. Many factors come into play for the development of such a demonstration. First, it must be ensured that all necessary security aspects are fulfilled. On the other hand, the restrictions of the LoRaWAN technology have to be met. Furthermore, nodes have to be developed and must be configured in the field in such way that a new, secure firmware can be loaded and started. Since there are usually several nodes within in a radius of a few kilometers, all these nodes should be updated in parallel to gain efficiency.
This work focuses on four key points. One is the research of existing publication in the field of LoRaWAN and the firmware update over the air process. Another is to create a hardware shield for the LPC55S16-EVK to enable a LoRaWAN connectivity and additional functionality which then simulates the LoRaWAN nodes in the field. Further, a LoRaWAN FUOTA server that handles the "over the air" process and the whole communication has to be set up. Then, the node needs to be able to reprogram itself after a full update is received. After that, the whole process has to be tested and its results have to be presented.

2. Terms and Definitions

The following chapter provides an overview of the abbreviations used in the document and their full term.

ABP	Activation by Personalization
CLI	Command Line Interface
DSBL	Dual image Secondary Boot Loader
EVK	Evalution Kit
FW	Firmware
FUOTA	Firmware Update Over The Air
GNSS	Global Navigation Satellite System
HW	Hardware
IoT	Internet-of-Things
ISM	Industrial, Scientific and Medical Band
ISR	Interrupt Service Routine
LDPC	low-density parity-check
LNS	LoRaWAN Network Server
LPWAN	Low Power Wide Area Network
OS	Operating System
OTA	Over The Air
SBL	Secondary Boot Loader
SW	Software
TTN	The Things Network
TTS	The Things Network

NVM	non volatile memory
UUID	Universally Unique Identifier

3. Background

3.1. Context

The company SensDRB develops smart sensor systems which are used for monitoring the content of farming industries, among other things. The distributed sensors transmit measured values such as temperature, humidity, water or grain quality or they are used to monitor the vital parameters of productive livestock via LoRa to an available LoRaWAN network. However, these sensors are often installed in non-urban areas. In these non-urban areas there is often no coverage of terrestrial communication techniques like wireless LAN, GSM or even LoRaWAN.

In previous work, a sensor node was developed, which is able to communicate in terrestrial LoRaWAN infrastructures as well in non-terrestrial satellite systems. For the non-terrestrial communication, the sensor uses the infrastructure of Lacuna. In this setup the sensor can only communicate in an unidirectional way. This means the sensor is only able to send uplink messages from the sensor to the satellite.

3.2. Purpose of this Thesis

As IoT devices take on increasingly important roles in industry, the security of these devices must be ensured. Devices that deliver incorrect data due to software errors or devices that are deliberately manipulated, must be able to be updated through software patches. There is always the possibility to update such IoT devices physically. But regarding the fact, that usually such devices are installed in non-urban areas the approach to update these IoT devices over the air is crucial. This includes the possibility to update devices, which deliver their data unidirectional via a satellite network.
This thesis is intended to connect the various components of a LoRaWAN firmware update over the air to serve as a basic fully operating example for the general public and to perform first performance tests of the FUOTA technology in the LoRaWAN network.

3.3. Preliminary Work

As part of another project in prior to this thesis, the basic setup to run a firmware update process over the air in a LoRaWAN network was established. This included a literature search using the keywords LoRaWAN, FUOTA, OTA, LoRaWAN security, patch programming and bootloader. The components for such an update were detected and the corresponding software stacks and server implementations where installed and ported to fit the use-case. A bootloader was developed, which is able to boot an image from different flash regions in the LPC55S16 microcontroller. The basic process of sending software image fragments form server side per unicast to a LPC55S16 based node via a private LoRaWAN network was established.

3.4. Requirements

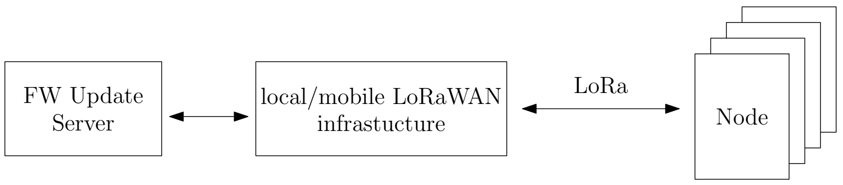


Figure 3.1.: Blockdiagram demostrator.

From the goal of this thesis, visualized in the block diagram 3.1, following vision is derived:

Building a demonstrator, that allows running parallel multi-devices firmware updates over the air using a local LoRaWAN network.

Based on this vision, following key requirements were extracted:

- Update server with a user interface, where the user can control the whole process
- Own hosted LoRaWAN network server
- Gateway hardware and gateway software driver
- LoRaWAN node based on the LPC55S16-EVK
- Integration of the LoRaWAN multicast and fragmented data block transport protocol
- Secure bootloader
- Patch algorithm to build a delta update file

3.5. Procedure

At the beginning of this project, an extensive literature research was conducted in various areas, namely firmware update over the air, LoRaWAN multicast, LoRaWAN fragmented data block transport and delta update. The results and functions of the preliminary work were analyzed in order to improve the implementation of the new firmware update process. Subsequently, the project was divided into individual areas, hence smaller subtasks. After the planning and research, the hardware based demonstrator node as a shield for the LPC55S16-EVK had to be developed. As a next step the full porting of the LoRaMac-node software stack had be done to give the node the possibility to communicate in the LoRaWAN network.
After that, the server side had to be implemented and set up, which includes the LoRaWAN network server with its own gateway and an FUOTA server. Next to it the patch file generation had to be taken into account to build a delta update and reduce the amount of data which has to be sent over the infrastructure. Then the multicast protocol and fragmented data block protocol had to be implemented on the server as on node side. As the final step, a bootloader had to be implemented on the node to merge the delta file back to a full image and to boot the new image on the node.

3.6. Approach

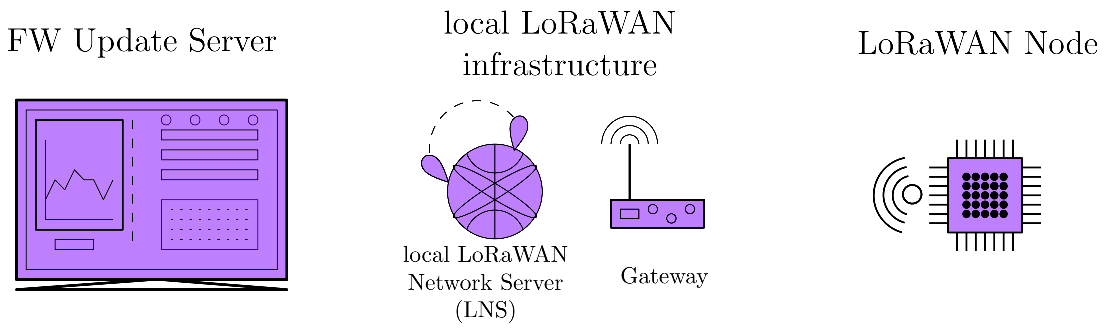

FW Update Server
local LoRaWAN infrastructure
LoRaWAN Node
local LoRaWAN Network Server (LNS)
Gateway

Figure 3.2.: Building Blocks of Approach.

Figure 3.2 shows the building blocks of the approach. A single block tackles one or multiple challenges, but all together build the complete firmware update over the air infrastructure which meets the requirements. While some details of a single block can be defined or managed separately, other parts are related and need to be designed togehter. The approach used in this project was to execute research for every building block separately and combine the output in the concept for the whole infrastructure needed. The *FW Update Server* can be split into user interface and process control algorithm. *Local LoRaWAN infrastructure* is split into LoRaWAN network server and gateway. The LoRaWAN network server includes a software stack which handles the LoRaWAN protocol and forwards the messages to application server and to the gateways. The gateway includes a LoRaWAN hardware concentrator unit and a software stack, which drives the concentrator and forwards the messages to the node or to the network server. The LoRaWAN node includes the hardware for a demonstrator node and its software drivers. This contains the LoRAMac-node software stack for the communication in the LoRaWAN net and its hardware like the LoRa transceiver and an antenna. It can be seen, that the identified building blocks are related to each other. Nevertheless, it was considered useful to treat individual blocks separately from each other in order to simplify the implementation process.

3.7. Challenges

The main challenge is to set up the infrastructure needed to perform a firmware update over the air. This includes the presence of a FUOTA server, which is able to set up a FUOTA process and guaranties a stable connection to the node. Furthermore, the node has to be able to receive such data-packages and store them locally to then assemble these packages to a FW image and load it via a bootloader. As a next step, this data should be encrypted and signed, so that authorized data can be loaded by the node.
To enable such a process the new LoRaMac-node softwate stack of Semtech [58] has to be ported to the LPC55S16 controller. This controller will be used to simulate a node in the field. Like the ported stack also a bootloader has to be implemented which allows to reboot the node automatically with the new firmware.

4. Scientific research

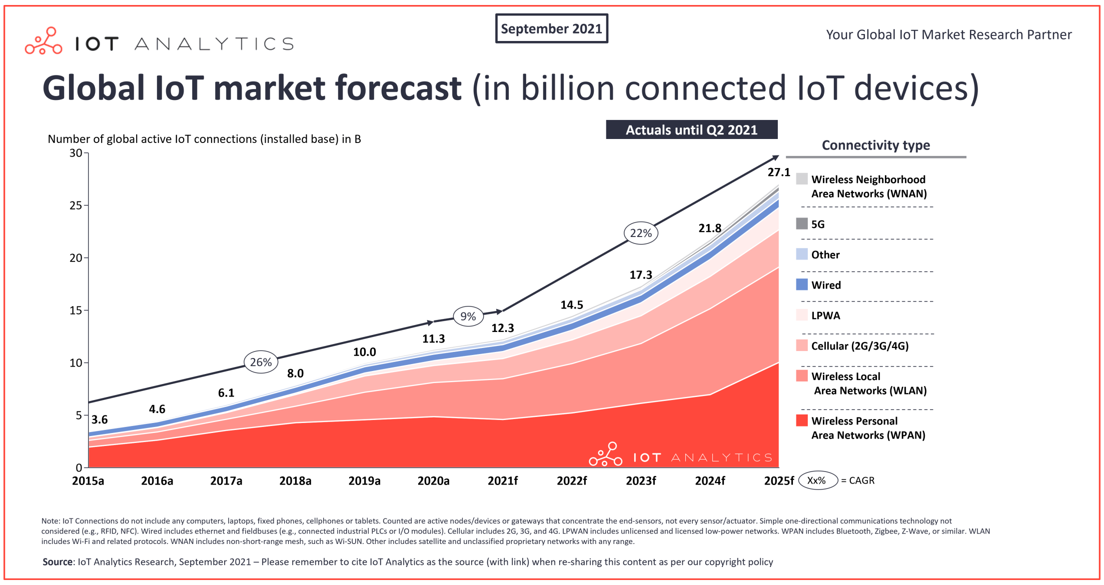
In this chapter, the basics regarding IoT, LoRa and LoRaWAN will be explained followed by the results of the literature search using the keywords *FOTA* and *DELTA* update.

4.1. Basics

This section will introduce the basic communication technologies used in this master thesis.

4.1.1. Internet of Things in general

The IoT is an ever-growing part of the information world. In the year 2021, there were already over 11 billion active devices, and this number is expected to grow to 27 billion by 2025 (see 4.1). Especially in the home automation sector, new use cases are constantly emerging for smart devices that network with each other, to perform more and more functions.



Note: IoT Connections do not include any computers, laptops, fixed phones, cellphones or tablets. Counted are active nodes/devices or gateways that concentrate the end-sensors, not every sensor/actuator. Simple one-directional communications technology not considered (e.g., RFID, NFC). Wired includes ethernet and fieldbuses (e.g., connected industrial PLCs or I/O modules). Cellular includes 2G, 3G, and 4G. LPWAN includes unlicensed and licensed low-power networks. WPAN includes Bluetooth, Zigbee, Z-Wave, or similar. WLAN includes Wi-Fi and related protocols. WNAN includes non-short-range mesh, such as Wi-SUN. Other includes satellite and unclassified proprietary networks with any range.

Source: IoT Analytics Research, September 2021 – Please remember to cite IoT Analytics as the source (with link) when re-sharing this content as per our copyright policy

Figure 4.1.: Number of IoT devices 2021 [32].

The most important capability of IoT devices is the low-power wireless communication. There are a wide variety of protocols for this, each with different areas of application. While W-LAN, Bluetooth, Zigbee or other radio standards are often used for short distances in the home, devices in other locations must rely on alternative technologies. LPWAN transmissions are one such group of technologies. These have the advantage of a much longer range and better energy efficiency than most other radio standards, but at the cost of bandwidth. Figure 4.2 shows LPWAN compared to other wireless communication protocols. LPWAN technologies are mostly used when end devices are installed in locations that are difficult to reach with other technologies or when energy efficiency is of high importance. In this work, LoRaWAN is used as LPWAN technology.

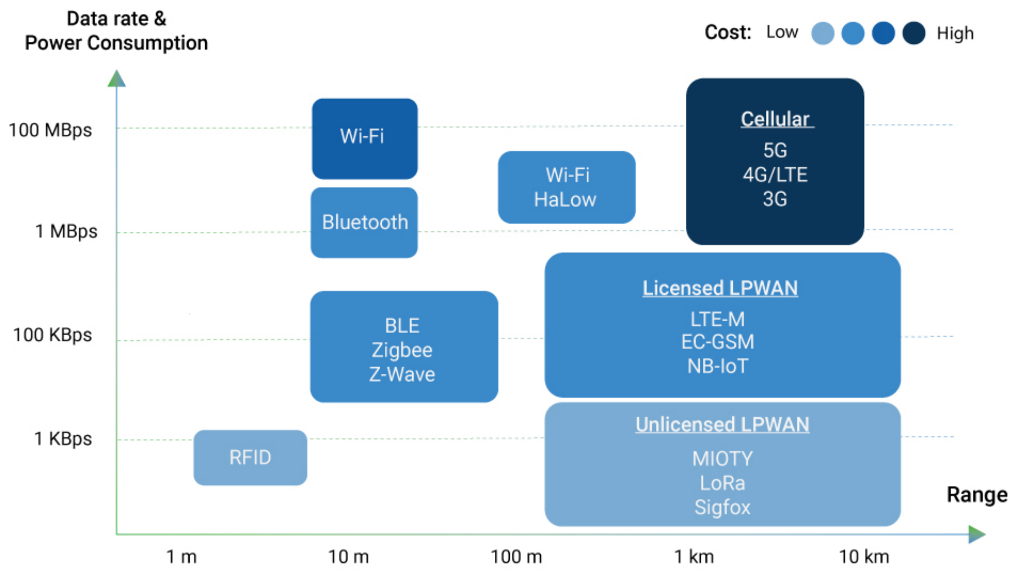


Figure 4.2.: Different IoT communication technologies [16].

However, range and power consumption are not the only important issues in IoT. One issue that is often neglected is the security of connected devices. Between 2017 and 2018 alone, the number of known malware for IoT devices nearly quadrupled [38]. But malware is not the only threat. Many IoT devices send data inadequately protected, enabling attacks on that data and by extension, IoT infrastructure. Espionage, manipulation of data and complete takeover of systems are exemplary attack scenarios.

Another relevant topic in IoT security are firmware updates; they enable manufacturers to introduce new functions to devices and in the event of security incidents, to fix them without the user having to take action. It is extremely important that the updates are carried out in a secure manner, so attackers are not able to insert counterfeit firmware into a device.
Combining firmware updates and end devices, that use an LPWAN protocol for data

transmission, creates a whole new challenge. While some techniques for IP-based technologies already exist, such as W-LAN, updates via LPWAN are still largely unexplored. The reason for not being able to use the classical protocols can be found in the limitations of LPWAN technologies. LoRaWAN, for example, has high limitations in terms of data rate and transmission time and does not have a standardized transport protocol, which could be used to compensate for losses during data transmission[15].

4.1.2. LoRaWAN

LoRaWAN is a LPWAN solution for IoT applications that allows small amounts of data to be transmitted wirelessly over long distances in an energy-efficient manner. It consists of LoRa radio, a protocol for physical data transmission, and LoRaWAN itself, a MAC protocol that builds on LoRa and provides a standardized method to transfer data over LoRa[7].

4.1.2.1. What is LoRa

LoRa is a frequency modulation method developed by Semtech, which allows wireless communication between two communication partners [39]. It is thus a physical protocol (OSI layer 1), which only performs the modulation of the physical data transmission. LoRa uses frequency modulated chirps to encode symbols. The chirp modulation used uses so-called "chirps" to transmit symbols. In this process, the frequency continuously changes over a defined period of time across the bandwidth. The transmitted symbols are defined by the beginning of the chirp. Figure 4.3 shows what such a message looks like.

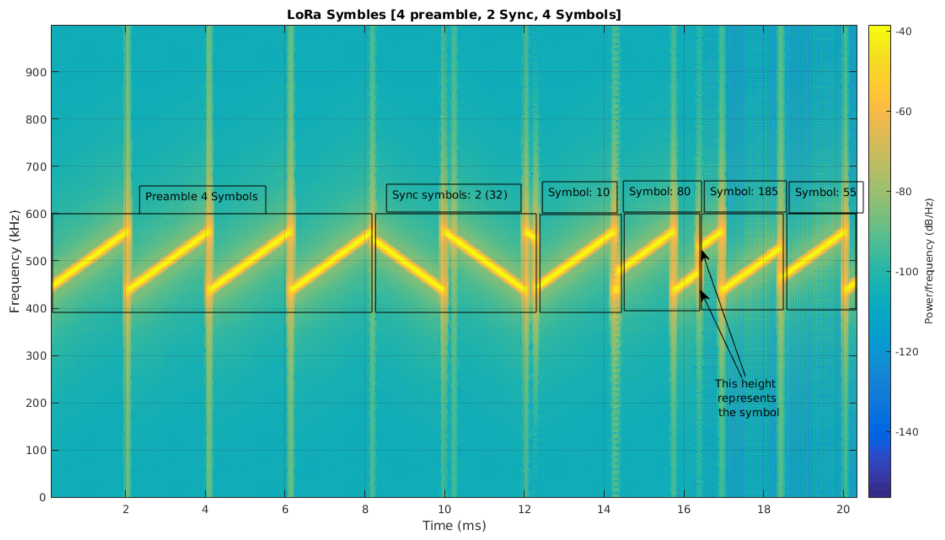


Figure 4.3.: LoRa frequenz hop symboles [28].

The primary advantages offered by this modulation, compared to FSK or PSK, are its long range and robustness against noise. Both are determined by the spreading factor used and the bandwidth [56]. The spreading factor determines how long a single chirp lasts, i.e. how wide it is "spread". A higher factor means wider symbols, which provides longer transmission ranges, but also leads to slower data transmission. In LoRa, spreading factors of 7 to 12 are defined, allowing transmission speeds from a maximum of 37.5 kbit/s to a minimum of 300 bit/s [31]. The bandwidth is fixed at 125 kHz, 250 kHz or 500 kHz and also affects the range and speed of the signal. The specific choice of these parameters is determined by LoRaWAN.
The frequencies that LoRa uses depend on the region. In Europe, it can transmit on 868 MHz or on 433 MHz. It is important to mention that these frequencies are license-free spectra, so no license fee has to be paid for their use. In compensation, there are time restrictions on transmission that all devices must adhere to. These restrictions are explained in chapter 4.1.2.7.

4.1.2.2. What is LoRaWAN

LoRaWAN is a MAC protocol (OSI layer 2) that builds on LoRa (but can also be used with FSK), and contains some elements of a network protocol (OSI layer 3) [39]. It defines a message format, as well as MAC commands to control the transmission. The parameters for the underlying LoRa transmission are defined by LoRaWAN.
The specification is divided into two parts. The first part is the specification itself, which defines the message format, the MAC commands, and the flow [7]. As an extension to this, there are the regional parameters, which define specific settings for LoRa, as well as some adaptations or additions to the LoRaWAN protocol, depending on the geographical region [5]. A LoRaWAN network consists of several groups of nodes and is organized in a star topology, as shown in Figure 4.4. In the center is the network server, which handles server-side management of the LoRaWAN network and provides an API for client LoRaWAN applications to manage, send and receive messages. This server communicates with multiple gateways over an TCPIP connection. The primary task of these gateways is to send the LoRaWAN packages received by the network server to the end devices via LoRa and vice versa. Accordingly, they serve as an interface for the change of the physical medium. The end devices communicate with one or more gateways to transmit their data.

In this process, the LoRaWAN protocol is only used between the gateway and the end devices. No standard is defined for the remaining paths and the format thus depends on the specific applications used. In this context, LoRaWAN performs a number of tasks, which are explained further below. These include the different communication classes that can be used to transmit data, the two ways to add devices to a LoRaWAN application, the encryption and integrity checking of transmitted data as well as the different MAC commands for controlling the connection [4].

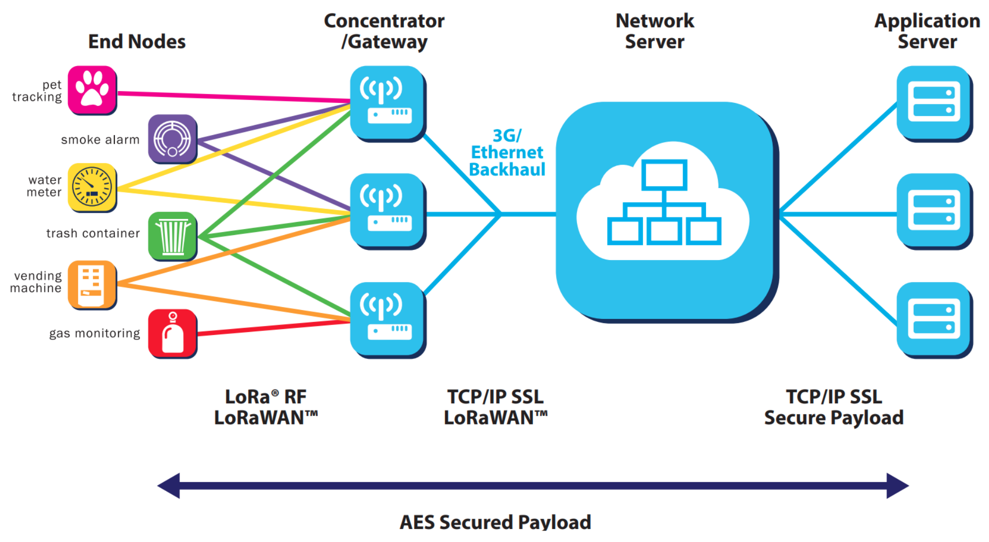


Figure 4.4.: LoRaWAN topology [6].

4.1.2.3. Communication modes in LoRaWAN

LoRaWAN supports three different modes for data transmission. Each of these modes have specific use cases, as well as advantages and disadvantages, which are listed below.

Class A
Class A enables bidirectional communication between the node and the gateway. The node is able to send uplink messages (node to gateway) at any time. After sending an uplink message, the node opens sequently two timeslot after each other during which it can receive a message from the gateway. These two timeslots are the only possibility for the server to send a message down to the node. In Figure 4.5, the timeslots in which the node is waiting for a downlink message from the gateway after an uplink message (red) are marked in green.

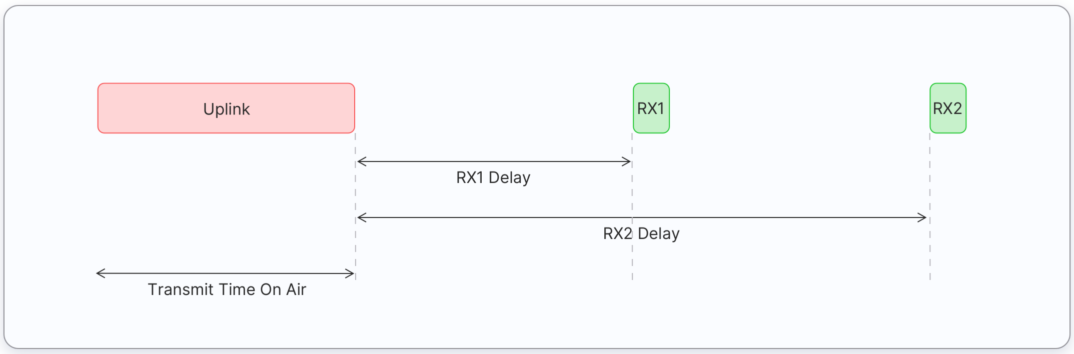


Figure 4.5.: LoRaWAN **Class A** communication timing [70].

Class A Nodes:

- are often battery operated
- have the lowest energy consumption
- have long interval times (sleep mode) between uplinks
- have high downlink latency

4.1.2.4. Class B

Class B adds scheduled receive times to the node. This means that the node can not only receive messages in the two fixed timeslots after an uplink, it also periodically opens additional timeslots for downlink messages. To use this technology, a time synchronization between node and server is necessary. Nodes operating in class B mode are also called beacons. In figure 4.6 the periodic timeslot (ping slot) is shown in orange. Further it can be seen that the class A function (green timeslots) remains.

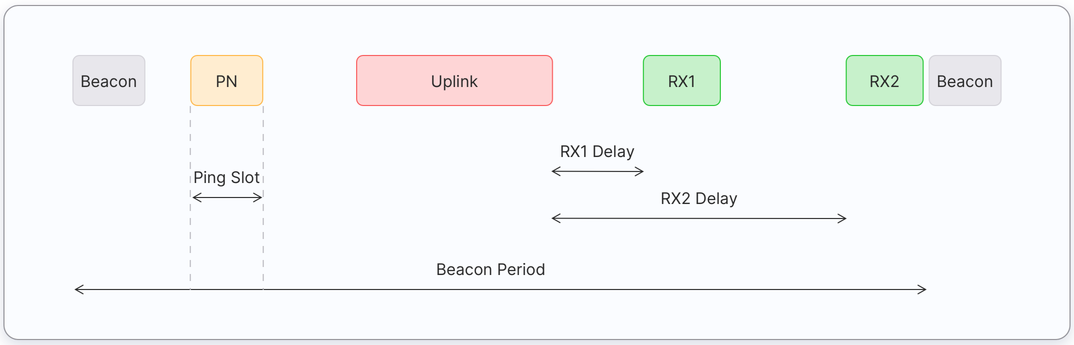


Figure 4.6.: LoRaWAN **Class B** communication timing [70].

Class B nodes have lower latency than class A nodes because they are reachable at periodically preconfigured times and do not need to send an uplink message to receive a downlink. The energy consumption is higher for class B nodes than for class A, since the node spends more time in active mode (orange, red, green boxes in Figure 4.6).

4.1.2.5. Class C

Class C nodes extend Class A by keeping receive windows open continuously. The advantage of Class C is that data can be received at any time. However, the price for this is a high energy consumption, since the terminal device must keep the LoRa transceiver active at all times. Multicast transmission can be take into account here. Class-C should only be used when large amounts of data have to be transmitted over a short period of time, or when time-critical transmissions take place. In this case, if nodes needs to spend a lot of time in class C mode, a permanent power supply need to be installed, since this mode consumes too much energy for a battery operation (figure 4.7).

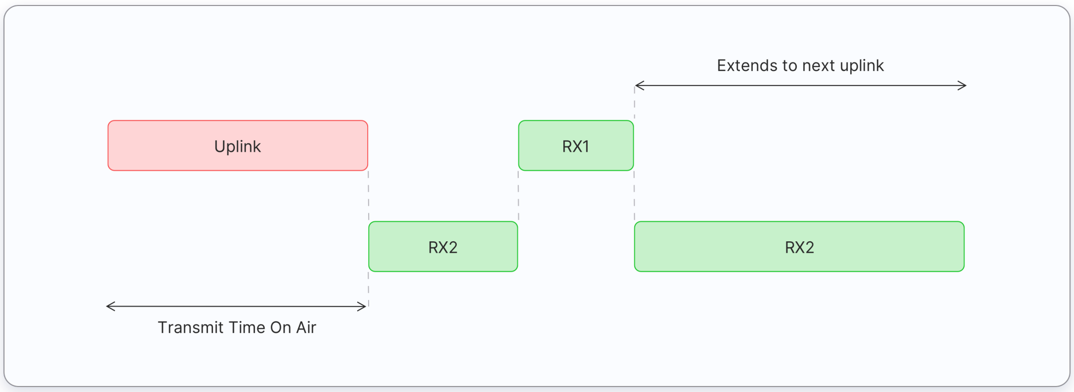


Figure 4.7.: LoRaWAN **Class C** communication timing [70].

4.1.2.6. Enctyption in LoRaWAN

The LoRaWAN specifications implements a series of keys which provides a secure communication in the LoRaWAN network. This means, that the communication between LoRaWAN network server and the node is encryptet plus there is a end-to-end encryption of the application itself to the node. All keys used in the communication protocol are 128 bits long and are based on the AES method [72].

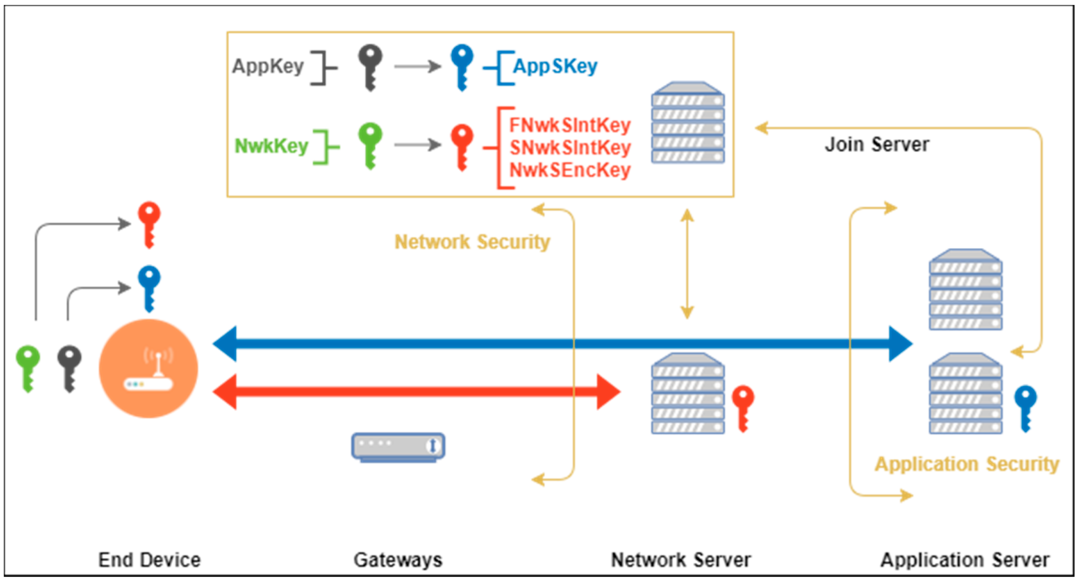


Figure 4.8.: Overview: LoRaWAN encryption [30].

- **NwkSKey: Network Session Key**
  The network session key is used for interaction between the node and the network server. This key is used to validate the integrity of each message through its *Message Integrity Code* (MIC) check. This MIC is similar to a checksum, except that it prevents intentional tampering of a message. In the backend of network server, this validation is also used to map a non-unique device address (DevAddr) to a unique DevEUI and AppEUI.

- **AppSKey: Application Session Key**
  The Application Session Key (AppSKey) is used to encrypt and decrypt the payload data. The user data is fully encrypted between the node and the application server. This means that no one other than the user is able to read the contents of messages that are sent or received.

These two session keys (NwkSKey and AppSKey) are unique per device and per session. If the node is activated dynamically (Over The Air Activation: OTAA), these keys are regenerated with each activation. If the node is statically activated (Activation by personalization: ABP), these keys remain the same.

- **AppKey: Application Key**
  The application key (AppKey) is known only to the node and the application. Dynamically activated devices (OTAA) use the application key to derive the two session keys during the activation process.

Figure 4.8 shows an overview of the security concept behind the LoRaWAN protocol. A detailed description of this security concept can be found in the project work VM1 by Mr. Bienz [17].

4.1.2.7. Restrictiond of LoRaWAN

In principle, the country-specific guidelines specified by the LoRa Alliance [1] apply to the use of the LoRaWAN protocol.
The main regulation is the duty cycle. This specifies how often (in terms of time) a node is allowed to send and receive data in the ISM band over the course of a day (table 4.1). Different data rates can be set via the parameterization of the network. Basically, the parameterization must be adapted to the respective situation and to the particular use case that the network/gateway/node must cover. With a small „airtime„ time setting, a higher data rate can be achieved, but the data can only be sent and received over smaller distances. Table 4.2 gives the setting for the ISM band 863-870, which is reserved for LPWAN applications in Europe, to achieve the highest data rate in the LoRaWAN network.

[1]The LoRa Alliance® is the fastest growing technology alliance. A non-profit association that has become one of the largest alliances in the technology sector, committed to enabling large scale deployment of Low Power Wide Area Networks (LPWAN) IoT through the development and promotion of the LoRaWAN® open standard. [41]

continents / countries	frequency band	bandwidth	duty cycle
Europa	EU683-870	125KHz, 250kHz	1%
	EU433	500kHz	
Amerika	US902-928	125kHz, 500kHz	no limits
China	CN470-512	125kHz, 250kHz	1%
Asia	AS923	125kHz	1%
Korea	KR920	125kHz, 250kHz	1%
India	IN865	125kHz, 250kHz	no limits

Table 4.1.: LoRaWAN dutycycle restrictions [71].

Network	max Payload	airtime	1% max duty cycle
SRF7/BW250	222Bytes	184.4ms	195msg/hour 43.29kByte/hour 1204Byts/s
SRF7/BW125	222Bytes	368.9ms	97msg/hour 21.53kByte/hour 602Byts/s

Table 4.2.: LoRaWAN max bandwidth.

On the infrastructure side, provider-dependent rules apply. For example, various providers have a fair access policy with which they want to prevent overloading of the infrastructure. The fair access policy specifies the transmission and reception rate at which the user of the infrastructure may use the channel. In other words, how many times a day the customer may send downlink and uplink data. Table 4.3 lists the fair access policy for two providers.

provider	uplink	downlink
TTN	30s	10 msg/day
Swisscom	144 msg/day	14 msg/day

Table 4.3.: Fair Access Policy.

4.2. Over the air firmware update

The scientific research was split into three categories *FUOTA in LoRaWAN*, *Binary patch files* and *Bootloader*. The authors in publication [10] presented a specified overview of the different components which are needed for the process of a firmware update over the air for IoT devices.
Because communication channels used in IoT devices are often limited in the bandwidth and the fact, that the IoT devices are often battery driven, the concept of frequently sending a complete updated firmware image to these nodes does not fit. This limitation triggered the development of the first incremental programming schemes. These schemes avoid sending the whole firmware image every time a new update has been released and just transmit commands to the nodes, on how to reconstruct the new firmware locally, utilizing parts of the currently run firmware, each node already has stored in its flash memory.
In order to update the modern IoT networks incrementally, a firmware update server should first create the new firmware image and then the resulting delta script, computing the common segments between the new and the previous firmware versions. Afterwards, the delta script is disseminated within the network, utilizing a multi-hop protocol, in order to reach all the nodes. Further, each node should interpret the received script, execute the commands found inside, and reconstruct the new firmware locally. Once this last step has been completed, the node can be updated replacing the firmware it currently runs, with the one it just reconstructed (loading phase). This process is visualized in figure 4.9.

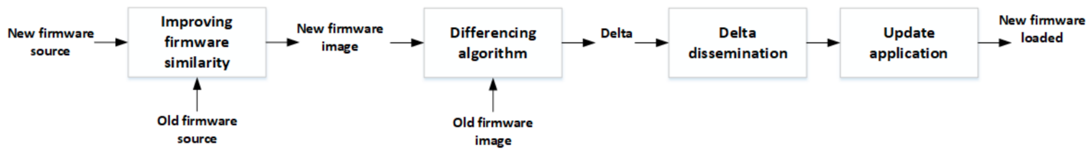


Figure 4.9.: Firmware update process essential stages [36].

The authors divided the firmware update over the air process in four essential operations.

- **Firmware similarity improvement:**
  Comparing of the new firmware source code with the old one, in order to mitigate function or variable shifts and increase the similarity between the built firmware versions [10].

- **Differencing algorithm application:**
  Production of a delta or patch script by using a differencing algorithm for comparing the old firmware image with the new one. The delta script encodes a set of instructions that, once applied on the old firmware, enable the reconstruction of the new one. In general, a delta script should be of minimal size (smaller than the original firmware image), since the goal is to reduce the data transmitted to the node [10].

- **Delta script dissemination:**
  Responsible for orchestrating the efficient and reliable firmware (delta script) distribution to the IoT nodes by applying a suitable dissemination protocol that focuses on transmitted data minimization [10].
- **Update application:**
  Refers to the OTAP stage that takes place on the node and includes reconstructing, verifying, installing and executing the new firmware [10].

The authors of the work [10] presented some techniques to improve the firmware similarity. Following these techniques will be sum up.

Slop-Regions:
A slop region is defined as the free area of memory immediately after a function's code in flash memory where a function can grow or shrink without moving other functions. When a function grows, part of its slop region is used without moving other functions. When a function shrinks, its slop region grows to occupy the removed portion of the function, so other functions that follow are not displaced. In addition to program code being mapped to flash memory, slop regions can also be used between the .data and .bss sections in RAM to avoid global variable offsets. In order to implement this feature, the linker must be modified, with the risk of degrading the performance of the generated code. A disadvantage of using slop regions is that excessive fragmentation of memory may occur, as some regions may contain code while others remain unused. Finally, special care must be taken when a function outgrows its slop region and may need to be relocated to a completely different memory region.
Position-Independent Code (PIC):
Position-Independent Code is an option that can be set during code compilation, where the code is compiled to execute normally, regardless of the absolute memory address in which it is stored. All references and destination addresses refer to the memory address of the calling instruction. Thus, when displacements occur, the „relative„ destination addresses are not affected. However, due to embedded device hardware limitations, these relative (instruction) jumps can only be executed within certain offsets.
Indirection-Tables:
When linking the firmware image, an indirection table is created that is stored in a fixed location in flash memory. In these tables, there are entries for each function called, along with the memory address where it is stored. All function calls are replaced by appropriate jumps to the corresponding entries in the table. The advantage of this technique is that when a function is relocated, only its entry in the indirection table is affected (updating the memory address), while the calling instructions are not. Since the function calls are executed indirectly through the table, the runtime latency of the function call increases. Further, the size of the table is proportional to the number of functions called. This means that the table also takes up a large area in flash memory for complex programs with many function calls.
Interrupt-Service-Routines-Pinning:
Changes in the firmware code can shift the interrupt service routines, which affects the

memory addresses contained in the interrupt vector table. One way around this is to map the interrupt service routines to fixed memory locations in the program flash.
Address pinning of global variables:
This technique was proposed in Hermes [54] to ensure that the global variables in each firmware version are stored in a specific order and thus at the same address. Both the defined and undefined global variables are recognized and stored in two different structures, so that if the update does not define additional global variables, it is ensured that the memory addresses of the current variables are not affected. Further, a slop area is inserted between the .data and .bss sections to avoid address shifts of the undefined variables when the .data section is reduced or increased.
In-Place-Patching:
With the strategy of Over-The-Air-Programming (OTAP), which is based on in-place code updating and uses code patches, system reboots can be avoided and the available memory can be used more efficient. The key idea in OTAP is to only transfer the parts of the firmware that have actually been changed (patches). The update module, which must already be included in the base firmware, then copies the patches directly into the flash memory. A risk with this technique is that by updating the firmware image in real time, the state of the firmware stored in memory can become inconsistent if the patches are transferred incorrectly. A countermeasure is to pause all instructions that contain references to functions that are being updated until the update process is complete. Since a patch can not only add to an existing firmware but also change it, meaning that changes can be made to the already existing functions, these code changes must be atomic[2]. The principle of OTAP is the same as that of delta update, except that the flash process is executed directly without a boot sequence. To keep the patches as small as possible, the same techniques described here must be used.
Dynamic linking of modified firmware patches:
With this OTAP scheme, only the modified sections need to be transferred, since the dynamic linker relinks the image when it is received on the node and reloads it, replacing the previous image in the program flash. A limitation of this technique, however, is that it requires an OS or sophisticated linker that can properly resolve the addresses. Such a linker or OS is memory intensive and not usable on many IoT devices.
Replaceable Components:
The concept of replaceable components is based on the Elon reprogramming scheme [22] and has been implemented and tested for the TinyOS operating system. However, part of the basic principle can be applied to other platforms and baremetal programs. It is based on the fact that specific sections are allocated to the program code in Flash. When the first (base) version of the firmware is created, these sections are defined and not changed throughout the life of the node. For example, the flash can be divided into the sections Initialization, LoRa, Peripherals and Update. If a change is now made to the LoRaWAN stack, in $worst - case$ only the entire LoRa section needs to be updated.

[2]In computer science, a process (which can consist of any number of individual pieces) can be called atomic if it is ensured that it cannot be influenced by other processes that may be running simultaneously. [13]

4.2.1. FUOTA in LoRaWAN

There are not many publications regarding firmware updates over the air in the field of LoRaWAN. The authors from the publication „How to make Firmware Updates over LoRaWAN Possible„ [2] investigated the new multicast, fragmentation and time synchronization specifications from the LoRa-Alliance regarding the integration of a firmware update process in the LoRaWAN network. A basic FUOTA architecture figure 4.10 was presented. The interfaces with solid lines are described in the LoRa Alliance specifications, otherwise, they are out of the LoRa specifications scope and have to be implemented by the user. The authors developed a simulation tool, which gives some

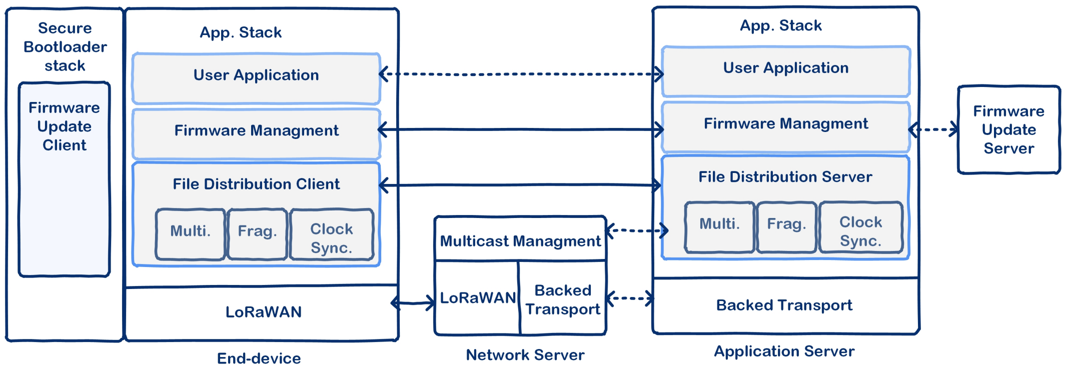


Figure 4.10.: FUOTA architecture blockdiagram [2].

insights about the scalability and the performance of the protocols specified by the LoRa Alliance. In the following section these protocols well be further explained.

4.2.1.1. Multicast

The objective of a multicast design is to let a group of class A devices receive the same downlink transmission at the same time. This requires that the group of devices are at the same time in a mode where they always listen to messages coming from the network and share the same security keys to be able to decrypt the same downlink messages. For this, the multicast specification defines a command to set up a receive-window of class C into a group of class A devices. Additionally, the specification defines commands to instruct the group of devices to switch to class C and switch back to class A at the end of the receive-window. All commands of this specification are sent to each device individually using unicast messages on port 200.

4.2.1.2. Clock synchronization

LoRaWAN devices usually do not have access to accurate clocks via GPS or other techniques. Consequently, due to the clock drifts, their time keeping is not reliable enough to perform McClassCSessionReq commands. Therefore, the clock synchronization specification defines a way for the devices to correct their clock skews. The basic idea is that the network has access to an accurate GPS clock that can be used to correct the devices' clocks. All commands of this specification are sent as application messages on port 202. The Command AppTimeReq is sent by a device to ask for a clock correction. The command includes the device time, which indicates the current device clock. The time is again expressed as the time in seconds since start of the GPS epoch [3] modulo 2^{32}. Next, the device gets AppTimeAns back, including the time correction in seconds. The expected accuracy of this approach is around one second, which is enough to run the multicast commands efficiently.

4.2.1.3. Fragmentation

A firmware image is usually quite big and cannot fit into one downlink packet but needs several packets. LoRaWAN links are lossy and thus packet losses are inevitable. Consequently, there is no an efficient way to know which packets were lost at which devices during a multicast session. Therefore, a mechanism to handle big data blocks and to recover packet losses in a scalable manner is required. For this, the fragmentation specification supports all necessary commands to transport a large data block to one device or to a group of devices. All commands of this specification are sent as application messages on port number 201. The fragmentation algorithm proposes adding a simple forward error correction code to the original firmware image before sending it. This allows devices to autonomously recover a certain ratio (based on the code used) of the lost transmissions without requesting re-transmission of lost fragments. This is done by first, chunking the original firmware image to fragments equal in size and then adding redundancy fragments, which are XORed to some of the original fragments. Devices can use redundant fragments to reconstruct their missing fragments. In this case, 5% redundancy added to the original firmware image allows devices to loose roughly 5% of the incoming transmissions and still be able to reconstruct the original firmware.
The algorithm is based on the LDPC-codes, but differs by using fragments. It is further assumed that fragments are only either transmitted correctly or lost completely in the case of an error-ridden data transmission. Bit errors within a fragment can therefore not be corrected by this algorithm. From the specification [9], the following section explains the algorithm with a small example.

[3]Sunday 6th of January 1980 at 00:00:00

To encode a message, it is first divided into m fragments of length l_f. Each fragment represents a data bit, so the complete fragmented message is the data word $F = (f1; f2; :::; fm)$, consisting of all the data fragments. All operations that are normally performed on data bits are now applied to the fragments. When calculating parity fragments, this means that the complete fragments are XOR calculated in each case. This results in a new fragment of length l_f. The parity matrix C still remains a binary matrix and has the dimension $m * m$. It is randomly generated, but does not have to fulfill all properties of regular LDPC codes. Only the 2nd and 3rd properties must be met. From this a parity check matrix $P = (I|C^T)^T$ of the dimension $m * n$ is generated, which is needed for the coding as well as the decoding.

Encoding: To encode a message of length l_d with this algorithm, the fragment length l_f and the code rate R must first be defined. This calculates m and n to $m = [l_d/l_f]$ and $n = [m/R]$. With this basis, the following steps can be performed to encode the data. The data fragments created by splitting the message are converted into code fragments using these steps. The code fragments are then transmitted to the receiver.

1. Create the parity check matrix P. Here the number of 1's in C are determined by the code rate and is calculated as $m * R$.

2. Code each code fragment c_i according to the following rule.

```
c_i = 0
for k=0 to m
    if P[i][k]
        c_i = c_i ^ f_k
```

3. From the code fragments generated, the complete codeword is thus obtained $C = (c1; c2; :::; cn)$. Here, for the first m code fragments: $c_i = f_i$.

4. The individual code fragments can now be sent one after the other. It should be noted that often only a part of the correction fragments is required by the node.

Decoding: To be able to decode the received code fragments, the receiver must also know l_f , R and P. Furthermore, a decoder matrix D must be created. This matrix has the dimension of $m*m$ and is initially a zero matrix. To recover the data fragments and thus the original message, the following steps must be performed:

1. Create the parity check matrix P and the decoder matrix D.

2. Perform the following calculation for each code fragment c_i. This is calculated with the already existing fragments, in order to remove all redundant information, which the receiver already knows. If the fragment still contains useful information, it is stored.

```
for k=0 to m
    if P[i][k] and D[k] != 0
        c_i = c_i ^ f_k
        P[i] = P[i] ^ D[k]
```

```
if P[i] != 0
    j = first_non_null_index(P[i])
        D[j] = P[i]
        f_j = c_i
```

3. As soon as D is a triangular matrix with a diagonal consisting only of ones, the receiving can be aborted. Depending on the error rate of the channel, less than n fragments are needed. If this condition is not met after the nth fragment is received, more fragments have been lost than the algorithm can correct, and the process must be terminated with an error. To be able to reconstruct the uncoded fragments the receiver must receive at least m ***linearly independent*** coded fragments.

4. In the last step, the obtained fragments are recomputed to create from D a unit matrix. This is achieved as follows.

```
for k=m-1 to 0
    for l=k+1 to m
        if D[k][l]
            f_k = f_k ^ f_l
            D[k][l] = 0
```

5. After this calculation, $D = I$ and all fragments have been recovered. The received data word is thus $F = (f1; f2; :::; fm)$.

Example: The algorithm is to be explained by means of a simple example. For this purpose, the parameters are set as followed.

- $d = "CorsinObristMT2022" = 436f7273696e204f6272697374204d5432303232$
- $l_d = 20$
- $l_f = 4$
- $R = 0.5$
- $m = 5$
- $n = 10$

In the next step, the parity check matrix P must be created. For this purpose, the rows $[m + 1, n]$ are filled randomly, with two ones per row. For this example, the matrix was generated from equation 4.1.

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix} \tag{4.1}$$

This matrix can now be used to calculate the code fragments. Equation 4.2 shows the results of the calculations after step 2 of the coding.

$$\begin{aligned} c_0 &= f_0 = 436f7273 \\ c_1 &= f_1 = 696e204f \\ c_2 &= f_2 = 62726973 \\ c_3 &= f_3 = 74204d54 \\ c_4 &= f_4 = 32303232 \\ c_5 &= f_0 \oplus f_2 = 211d1b00 \\ c_6 &= f_1 \oplus f_3 = 1d4e6d1b \\ c_7 &= f_3 \oplus f_4 = 46107f66 \\ c_8 &= f_0 \oplus f_1 = 2a01523c \\ c_9 &= f_2 \oplus f_4 = 50425b41 \end{aligned} \tag{4.2}$$

The code fragments created in this way are transmitted to the receiver in the next step. In this example, a packet loss is simulated in that the fragments $c1$ and $c3$ do not arrive at the receiver. The calculation after step 2 of the decoding is trivial for $i < m$, since no calculations need to be performed after the first loop. The reason for this is that for $i < m$ P has a one only at the position P_{ii}, i.e. exactly the position of the received fragment, which means that $D_i = 0$. After the reception of c_0, c_2 and c_4 the status of the receiver corresponds to equation 4.4.

$$D = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.3}$$

$$\begin{aligned} f'_0 &= c_0 = 436f7273 \\ f'_0 &= c_0 = 00000000 \\ f'_2 &= c_3 = 62726973 \\ f'_0 &= c_0 = 00000000 \\ f'_4 &= c_4 = 32303232 \end{aligned} \tag{4.4}$$

Next, c_5 is received. $\overrightarrow{P_5}$ has a one at positions 0 and 2. When processed with step 2, $\overrightarrow{P_5}$ is updated according to equation 4.5, resulting in a zero vector.Thus, the received code fragment contains no new information and is discarded.

$$\overrightarrow{P'_5} = \overrightarrow{P_5} \oplus \overrightarrow{P_0} \oplus \overrightarrow{P_2}$$

$$\overrightarrow{P'_5} = \begin{pmatrix}1\\0\\1\\0\\0\end{pmatrix} \oplus \begin{pmatrix}1\\0\\0\\0\\0\end{pmatrix} \oplus \begin{pmatrix}0\\0\\1\\0\\0\end{pmatrix} = \overrightarrow{0} \tag{4.5}$$

The code fragment c_6 is processed next. Here, the same calculations are performed. Unlike c_5, however, at this point a $\overrightarrow{P_6}$ is produced that is not zero, since both $\overrightarrow{D_1}$ and $\overrightarrow{D_3}$ are zero vectors. $\overrightarrow{P_6}$ is thus used unchanged as a new $\overrightarrow{D_1}$, while c_6 is stored as f'_1.

$$\overrightarrow{P_6} =< 0,1,0,1,0 >$$

$$D = \begin{pmatrix}1&0&0&0&0\\0&1&0&1&0\\0&0&1&0&0\\0&0&0&0&0\\0&0&0&0&1\end{pmatrix}$$

$$\begin{aligned} f'_0 &= c_0 = 436f7273 \\ f'_1 &= c_6 = 1d4e6d1b \\ f'_2 &= c_2 = 62726973 \\ f'_3 &=00000000 \\ f'_4 &= c_4 = 32303232 \end{aligned} \tag{4.6}$$

The next code fragment is c_7. This also contains useful information, since $P_{73} = 1$. Using step 3, $\overrightarrow{P_7}$ is xored with $\overrightarrow{D_4}$, which removes the one at this point. Thus, the status of the receiver after this operation is given by equation 4.7.

$$\overrightarrow{P_7} = < 0, 0, 0, 1, 1 >$$

$$D = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{aligned} f'_0 &= c_0 = 436f7273 \\ f'_1 &= c_6 = 1d4e6d1b \\ f'_2 &= c_2 = 62726973 \\ f'_3 &= c_7 \oplus c_4 = 74204d54 \\ f'_4 &= c_4 = 32303232 \end{aligned} \tag{4.7}$$

At this point, the transmission of further fragments can be terminated, since the prerequisite from step 3, the triangular matrix, has been created. With step 4, the rows of the decoder matrix D are now offset against each other until a unit matrix is created. For this example, this is only true for $k = 1$ and $l = 3$, resulting in $f'_1 = f'_1 \oplus f_3$ being calculated. The result is the decoded result shown in equation 4.8.

$$D = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{aligned} f'_0 &= c_0 = 436f7273 = f_0 \\ f'_1 &= c_6 \oplus c_7 \oplus c_4 = 696e204f = f_1 \\ f'_2 &= c_2 = 62726973 = f_2 \\ f'_3 &= c_7 \oplus c_4 = 74204d54 = f_3 \\ f'_4 &= c_4 = 32303232 = f_4 \end{aligned} \tag{4.8}$$

Comparing the fragments calculated in this way with those that were originally encoded, we find that the message could be successfully recovered: $F = f'_0|f'_1|f'_2|f'_3|f'_4 = 436f7273696e204f6272697374204d5432303232 = "CorsinObristMT2022"$.

4.2.1.4. Summary FUOTA LoRaWAN

The theory and specifications for a firmware update over the air in a LoRaWAN network are defined. Simulations [2] have shown that the protocol defined by the LoRa Alliance work but detailed field tests have not been published yet. The protocol workflow shown in figure 4.11 will be taken as the base for the implementation of this thesis.

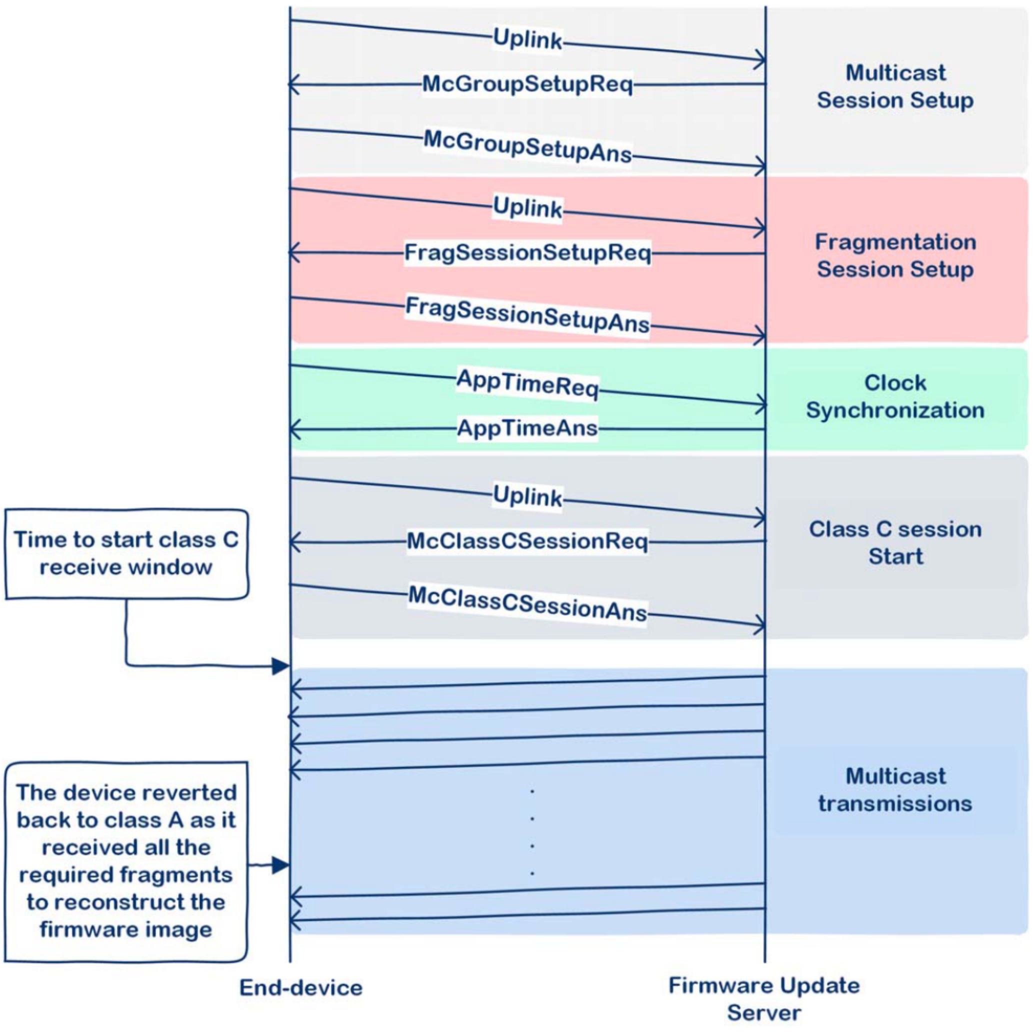


Figure 4.11.: FUOTA protocol flow-diagram [2].

4.2.2. Binary patch files

The authors in paper [10] are listing and comparing differencing algorithms. Following section presents the key-findig of their work.
Differencing algorithms can be of two types, either block-level or byte-level, depending on the granularity level they are able to detect matching segments. The block-level algorithms split the firmware images into fixed-size blocks, aiming to detect non-common segments between the two images, hence, their accuracy is highly affected by the block size. On the other hand, the byte-level algorithms are able to find non-common segments between two firmware versions using blocks of variable lengths and can utilise more fine-grained approaches in order to achieve better accuracy, for example dynamic programming. Regarding algorithms' performance, the block-level ones can detect alimited number of non-common segments, since they are not able detect those with size smaller than the size of a block. However, these algorithms typically have smaller time and memory footprint. Byte-level algorithms, on the other hand, can detect more non-common segments but typically require more time to complete. The following list and table 4.4 summerizes the different algorithms presented in the work and their performance.

- **FBC:**
  Fixed block comparison splits the two images into blocks and then compares each corresponding block. For each matching block pair, a COPY instruction is inserted into the produced delta script, while the non-matching ones are transmitted along with the delta script. In order to encode the latter blocks, an ADD instruction needs to be inserted in the delta script. The main benefit of this technique is the low time and space overhead, as well as the ease of implementation.

- **Rsync:**
  Is initially developed for binary files exchange over low-bandwidth channels.This is a block-level differencing algorithm that splits the firmware images into fixed-size blocks, and then uses a sliding window with a size equal to the block size, to scan the two firmware images for detecting matching segments. Like typical sliding window protocols, when a match is found, the window moves forward one block, otherwise it moves one byte, signing this block as unmatched. All unmatched blocks are accumulated for transmission either when a next block is matched, or the current window reaches the end of the new image.

- **RMTD:**
  Reprogramming with minimal transferred data is a byte-level algorithm that aims to find the optimal combination of common sequences between two images, in order to minimize the number of transmitted bytes. RMTD uses a 2D matrix to record the pairs of the common bytes found for the two firmware images, with the comparisons performed in both forward and backward order to achieve higher accuracy. The result of this operation consists of two lists that contain the matching segments of the two images, as well as the matching segments between the partially reconstructed new image and the rest of the new image, respectively. It must be

noted that the algorithm's complexity depends on the size of the images, which makes it unsuitable for increasingly complex programs. Experiment have shown, that RMTD crashes when the code size becomes too large (~42Kb), due to a lack of memory.

- **Hirschberg's trick:**
  Is a method for computing the longest common sequences between two strings, while saving space, utilizing a dynamic programming approach. Hirschberg also presented a modified version of this algorithm, which follows a divide and conquer approach and is able to compute the LCS of two strings in $O(min(m, n))$.

- **R3diff:**
  Is a byte-level comparison algorithm that complies with the overall design of the R3 OTAP scheme. Initially, the algorithm computes the hash values for every three continuous bytes of the current image.

- **DASA:**
  An efficient differencing algorithm based on suffix array is a differencing algorithm that focuses on minimizing the space and time complexity for computing the optimal delta script. In order to accomplish this, it utilizes an efficient data structure, called *suffix array*.

- **DG:**
  Is a differencing algorithm that is developed special for nodes that lack external memory. The algorithm places the two images side-by-side and executes an XOR operation between the corresponding bytes, aiming to reveal the sequences of the non-matching bytes. In a other work a comparison of R3diff and DG was conducted, using various image sizes and code shifts. The authors inferred that DG outputs significantly smaller delta scripts than D3diff for small-sized images but this does not stand true, as more data and code is shifted. Moreover, the authors found that the number of ADD instructions in the delta script gets smaller, as code shifts increase. The authors conclude that DG is not able to provide optimization for a high number of small changes. Instead, it generates a number of ADD instructions that encode regions with a few bytes for each non-matching segment. When the code shifts increase, these non-matching segments expand together and are merged under one common ADD instruction. This results to a larger delta script with fewer ADD instructions.

Algorithm	Type	Time complexity	Space complexity
FBC	block-level	O(n)	O(n)
Rsync	block-level	O(n^2)	O(n)
RMTD	byte-level	O(n^3)	O(n^2)
Hirschberg's trick	byte-level	O(n^2)	O(n)
R3diff	byte-level	O(n^3)	O(n)
DASA	byte-level	O($nlogn$)	O(n)
DG	byte-level	O(n^2)	O(n)

Table 4.4.: Summary of differencing algorithms commonly used for OTAP schemes
n: the combined length of the two firmware images in bytes [10].

JojoDiff
In other work [34] the authors used the JojoDiff algorithm [1] which tries to find a minimal set of differences between two files using a heuristic algorithm with constant space and linear time complexity. This means that accuracy is traded over speed. JDIFF will therefore, in general, not always find the smallest set of differences, but will try to be fast and will use a fixed amount of memory. This work provides a javascript interface to build the patch file and a small C library to merge the patch file to the actual image. In previous work this algorithm was already used to build patch files.

The authors in work [11] were presenting the BSDIFF and XDELTA algorithm. Further they presented performance analysis for all algorithm which will be shown at the end of this section.

BSDIFF
BSDiff is a differencing algorithm that focuses on executable files using suffix sorting for the efficient computation of delta scripts. Initially, the two files are scanned both forwards and backwards, so that segments of the new file that exactly match with others of the current file are detected. Next, BSDiff computes approximate matches, expanding the detected matching segments in both directions, so that every suffix/prefix of the extension matches at least half of its bytes. These matches correspond to slightly modified segments of the new firmware that have small differences between the two versions. The algorithm relies on common change patterns observed in executable code, so that it produces smaller delta scripts for executable files, compared to other similar tools, in $O((n+m)logn)$ time complexity, where m and n are the sizes of the two file versions. On the other hand, BSDiff is memory-intensive, requiring $max(17*n, 9n*m)+O(1)$ bytes of memory for the patch computation.

XDELTA
XDELTA is a linear time and space differencing algorithm that operates at block-level and produces delta scripts in VCDIFF format. The algorithm appends the two file versions constructing a new file, which is compressed using LZ77 or a similar compression algorithm, producing a compression only for the part that consists of the new version. As the two file versions share many common segments, the new version will be efficiently

compressed. In this sense, data compression can be considered as a special case of differencing, where no current file is provided. XDELTA produces small delta scripts by optimizing the generated instruction set and merging small instructions into one.
In table 4.6 to 4.8 the performance of the algorithm is presented. The used application files are explained in table 4.5.

Version number	Description	Codesize (bytes)
V1	Base version	41608
V2	Two initialized variables are added, and their values are printed using an additional printf call	41628
V3	A new function is implemented and called	41628
V4	An additional instruction is added in the implementation of the new function	41628
V5	No modifications done (same as V4)	41628
V6	An additional printf call is added	41740
V7	Three new functions are implemented and called	41740

Table 4.5.: The different firmware versions used to feed the differencing algorithms [11].

Firmware update	RMTD	DG	Dfinder (i-p)	Dfinder (o-o-p)	Rdiff	R3	BSDiff	Xdelta3	JojoDiff
V1, V2	3771	5910	8286	2392	41639	5465	1057	3568	3939
V2, V3	13	11	31	31	2065	16	160	63	15
V3, V4	5	2	26	26	9	5	140	53	–
V4, V5	5	2	26	26	9	5	140	53	–
V5, V6	3871	6026	8362	2479	41751	5569	1175	3719	4043
V6, V7	46	63	71	57	2065	54	209	92	60

Table 4.6.: The delta script size (in bytes) produced by the various differencing algorithms when optimised code is used [11].

Firmware update	RMTD	Dfinder(i-p)	Dfinder(o-o-p)	Rdiff	R3	BSDiff	Xdelta3	JojoDiff
V1, V2	151.480	0.072	0.033	0.020	0.483	0.023	0.049	1.807
V2, V3	154.524	0.039	0.039	0.015	7.345	0.0162	0.050	1.167
V3, V4	153.790	0.052	0.037	0.014	8.800	0.015	0.011	–
V4, V5	161.040	0.042	0.039	0.016	10.412	0.018	0.008	–
V5, V6	133.800	0.045	0.022	0.019	0.292	0.023	0.029	1.532
V6, V7	104.939	0.021	0.023	0.011	4.358	0.011	0.033	0.779

Table 4.7.: Mean execution time (in seconds) of the various differencing algorithms for the optimised code case [11].

Firmware update	RMTD	Dfinder(i-p)	Dfinder(o-o-p)	Rdiff	R3	BSDiff	Xdelta3	JojoDiff
V1, V2	1695291	4071	4069	122	3834	7595	100845	7595
V2, V3	1695317	4065	4069	38	3829	7595	100845	7595
V3, V4	1695317	4065	4069	32	3829	7595	66925	–
V4, V5	1695317	4065	4069	32	3829	7595	66925	–
V5, V6	1704450	4079	4078	122	3843	7595	100845	7595
V6, V7	1704436	4076	4080	38	3840	7595	100845	7595

Table 4.8.: Peak memory utilisation (in Kbytes) during differencing algorithms' execution for the optimised code case [11].

4.2.3. Bootloader

A Bootloader is the first piece of firmware that gets executed once the microcontroller is turned-on/reset. The primary objective of the Bootloader is to initialize the system and provide control to the application. The other objective of the Bootloader is to support the data loading feature. The bootloader typically contains minimal operations such as kernel release steps and cryptographic operations to read and verify the integrity of the firmware stored and/or received via an upgrade process [3][35].

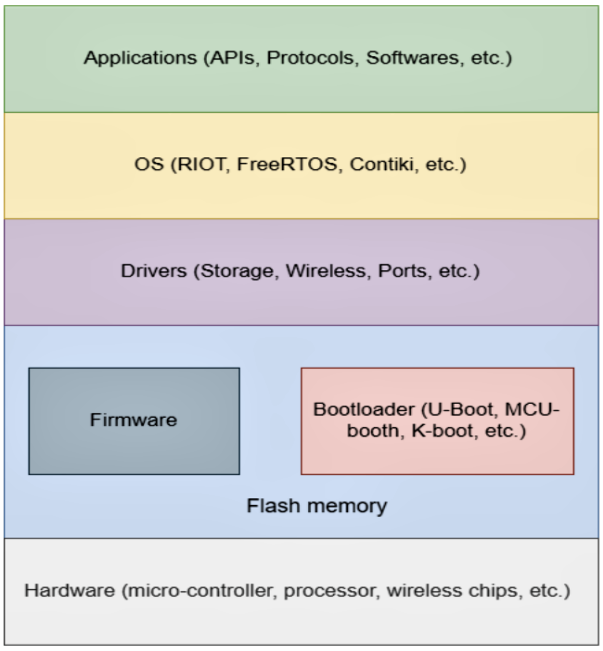

Applications (APIs, Protocols, Softwares, etc.)
OS (RIOT, FreeRTOS, Contiki, etc.)
Drivers (Storage, Wireless, Ports, etc.)
Firmware
Bootloader (U-Boot, MCU-booth, K-boot, etc.)
Flash memory
Hardware (micro-controller, processor, wireless chips, etc.)

Figure 4.12.: Typical IoT device stack [35].

The authors [35] developed a proof of concept FreeRTOS bootloader for IoT devices in context of OTA firmware update. These IoT devices will receive their update over a wireless channel and the bootloader will then implement checksum operations, verify the integrity and authenticity if possible, install it and then reboot.

The authors present two levels where the OTA update module can be implemented.

- At the bootloader level:
  The OTA and firmware verification process are executed at the bootloader stage and is independent of the application. This choice allows a bootloader/ application separation but increases the footprint of the first one (bootloader size).

- At the application level:
  The implementation of the ota process at this level allows to decrease the bootloader footprint. The OTA process is dependent on the application, but independent of the bootloader.

In the presented work the authors gave special interest to the OTA update at the application level, to reduce the highest possible additional footprint impact on the constrained device. In their own research the authors scanned a variety of papers regarding bootloader for IoT devices. The most interesting finding from this work was a bootloader for a board using a Cortex-M0+ with 256kB Flash and 32kB of RAM. The solution there uses different partitions, or areas, in the flash memory to run the firmware update. The firmware is loaded into an update area which is used to run integrity and version checks before copying the image into a live partition. Moreover, the last installed version is copied to a backup area, and in case of failure, a rollback to this version is done. Further the authors referred to a document by ATMEL [12] which discusses several design considerations in developing this kind of software. The most important ones are:

- The bootloader sequence diagram for firmware upgrade, which includes firmware integrity verification and code encryption

- The memory partitioning (i.e., single or dual banked memory), which makes it possible to have at least one working version of the firmware in the device at anytime, to avoid firmware corruption in case of issues such as power or connection loss.

- Safety solutions to prevent safety related errors(i.e., transmission error, transmission failure, information loss).

- Security solution, by enforcing the privacy, integrity and authenticity features, via hash functions, digital signatures, message authentication codes (MACs) and encryption in order to prevent attacks(i.e., unauthorized device, third party firmware, firmware alteration, reverse-engineering).

Another work [37] was focusing on two different possible memory allocations regarding a OTA update on an IoT node.

- Single-bank – firmware is rewritten right in its destination location, making it unavailable during an update

- Dual-bank – firmware is reconstructed in another part of the memory (second bank) and later copied into its destination location. If the procedure fails mid-update, old firmware is still available.

The paper provides experiments show that single-bank updates can significantly decrease memory requirements for an update. The main advantage of single update approach is less memory required. The main disadvantage is the possibility to cause non-recoverable firmware corruption in case of a mid-update error. The implementation of single-bank updates also requires more logic. The-main advantage of dual-bank updates is that a working copy of firmware is always present on a device. It is also simpler to implement. The main disadvantage is that more memory is required. The developers must consider their options and choose which approach to use. The ideal scenario is to implement both options and make the solution configurable.

For this thesis, with the knowledge gained in previous work [53] a solution compared to a single-bank firmware update bootloader with the use of an external flash will be implemented. The use of a FreeRTOS with a file system running will also simplify the implementation and firmware architecture but will increase the memory footprint of the bootloader. But having an external flash will give us the opportunity to plan more internal flash for the bootloader.

5. Hardware Design

As explained in chapter 3.6, the three main blocks *FW Update Server, LoRaWAN Network Server & Gateway and the LoRaWAN node* will build the infrastructure for the project. All of these blocks include software stacks as well as hardware. The following chapter will show which hardware setup was developed and installed.

5.1. LoRaWAN node

For the LoRaWAN node, which should simulate a sensor node in the field, the LPC55S16-EVK (figure 5.1) is used. The evaluation kit (EVK) is a development board, which already has a variety of peripheries on it. This helps during prototyping and to test and further implement different functionalities of the LPC55S16 microcontroller. Table 5.1 lists the basic features of the EVK.

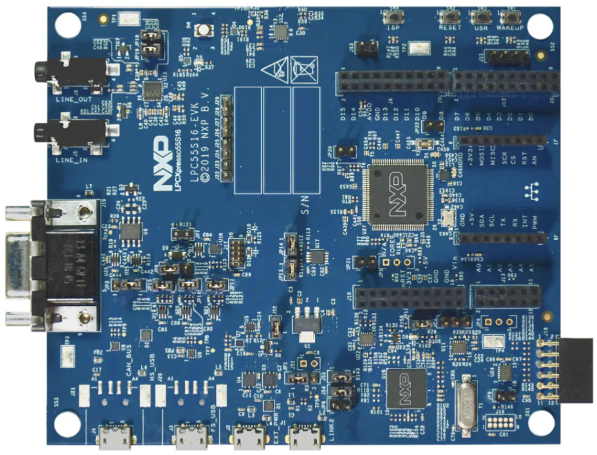


Figure 5.1.: LPC55S16-EVK board by NXP [50].

The EVK by itself has no on-board hardware functionality for LPWAN communication. This means there is no LPWAN modulation chip and antenna connector already placed. To enable LoRa functionality to this EVK board, the LPCXpresso expansion could be used. Due to the fact, that the LPCXpresso expansion is Arduino UNO pin header compatible, there are suitable LoRa shields available. One of these shields is the SX126xMB2xAS (figure 5.2) shield from SEMTECH. In previous projects this shield

was already with EVK board to simulate a basic node.
For this project, the idea was to create a demonstrator, which can receive firmware update over the air in a LoRaWAN network. To give the EVK board some demonstrator characteristic and to be able to handle large amount of firmware data, an own shield was developed.

Processors	LPC55S16 Arm® Cortex®-M33 PSA Level 2 certified microcontroller running at up to 150 MHz 256 KB flash and 96 KB SRAM on-chip HLQFP100 package
Debug Capabilities	LPC-Link2 debugs high-speed USB probe with VCOM port I2C and SPI USB bridging to the LPC device via LPC-Link2 probe SWO trace support (MCUXpresso IDE) Debug connector to allow debug of target microcontroller using an external probe
Expansion Connectors	MikroElektronika Click expansion option LPCXpresso expansion connectors compatible with Arduino UNO PMod compatible expansion / host connector
User Interface	Reset, ISP, wake and user buttons for easy testing of software functionality Tri-color LED
Connectivity	Full-speed USB device / host port High-speed USB device / host port UART header for external serial to USB cable CAN Transceiver

Table 5.1.: LPC55S16-EVK features[].

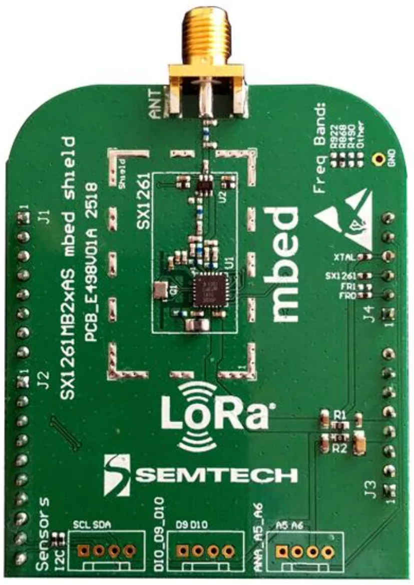


Figure 5.2.: SX126xMB2xAS by SEMTECH [48].

5.1.1. Demonstrator shield

In following section the development process of the demonstrator shield is explained.

5.1.1.1. Demonstrator requirements

In the following section, the requirements for the demonstrator are listed. The requirements were defined on the knowledge gained in previous work and through the scientific research in chapter 4.

- **External memory**
  The internal memory of the LPC55s16 microcontroller is limited to 256kB flash. The last 10kB of this flash are reserved for internal function. This means there are 246kB of internal flash to use for applications. This internal flash has to host a bootloader application and a LoRa application which integrate the LoRaWAN stack and has to provide functionality of firmware updates over the air. For the basic bootloader tested in previous work, which flashed the received application to the running section (concept of dual-bank flash memory), a memory footprint of 20kB had to be reserved. For the LoRaMac-node software stack, Semtech recommends at least 128kB of flash [61]. A fully size-optimized basic LoRaWAN example with the LoRaMac-node stack uses already more than 64kB of flash. Not only the LoRa functionality has to be provided, the demonstrator also needs to have sensor/actor integrations and will use a RTOS for a powerful software architecture. The demonstrator always need a running backup image ready for safety reasons.

The update image has to be cached. This update image could in the worst case have a memory footprint of a complete application. That means the flash has to provide at least the recommended 128kB flash on top of the bootloader, running image and backup image. Figure 5.3 summs this memory footprints calculation up. It can be seen that, even if the cached update image is directly in the running application via the bootloader, the total memory footprint would exceed the 246kB of internal flash. Regarding this memory footprint calculation it is recommended,

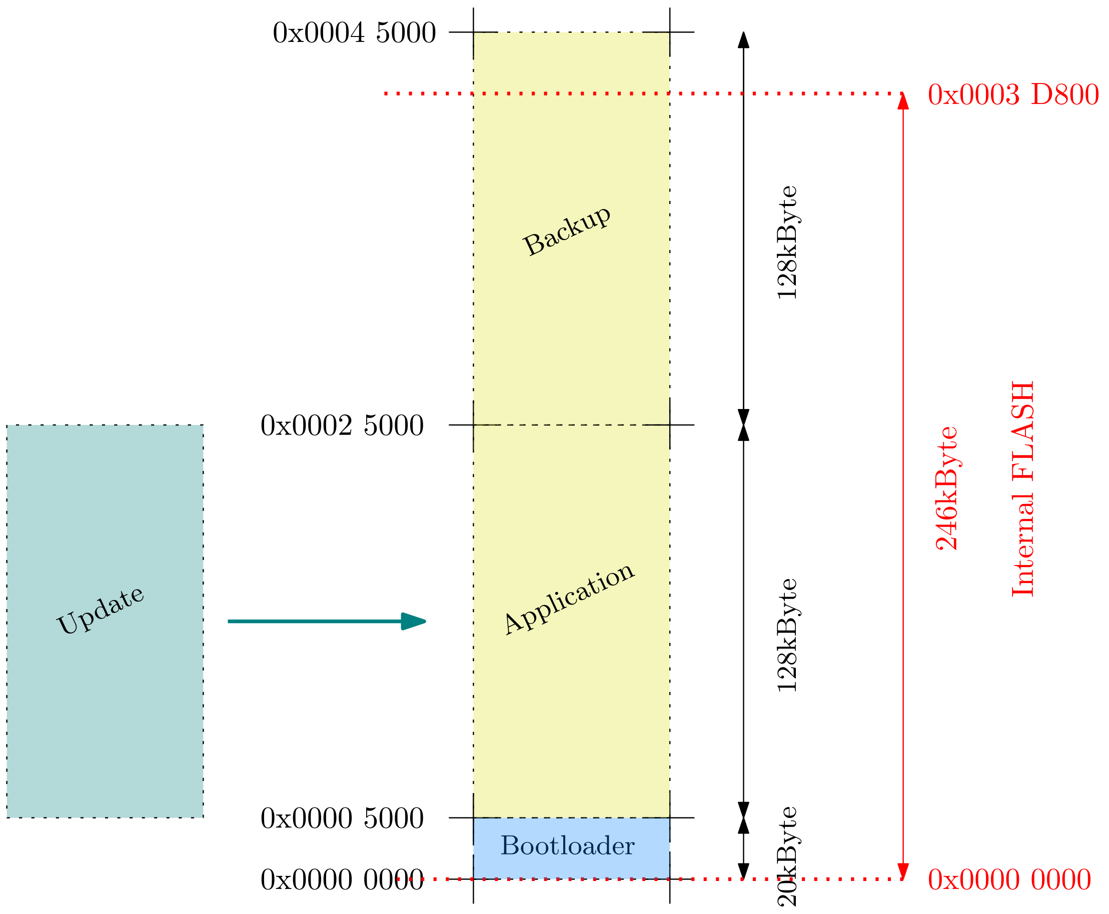


Figure 5.3.: Internal flash problem.

to use an external memory. An external memory would allow the bootloader to integrate a simpler image handling and the running application does not have to be designed as low memory usage as possible and could therefore provide more functionalities and options. For the external memory the following criteria apply:

- Memory $>$ 256kB
- Divisible in sections
- SPI interface
- Driver in C available

With an external memory the memory footprint could be organized as shown in figure 5.4.

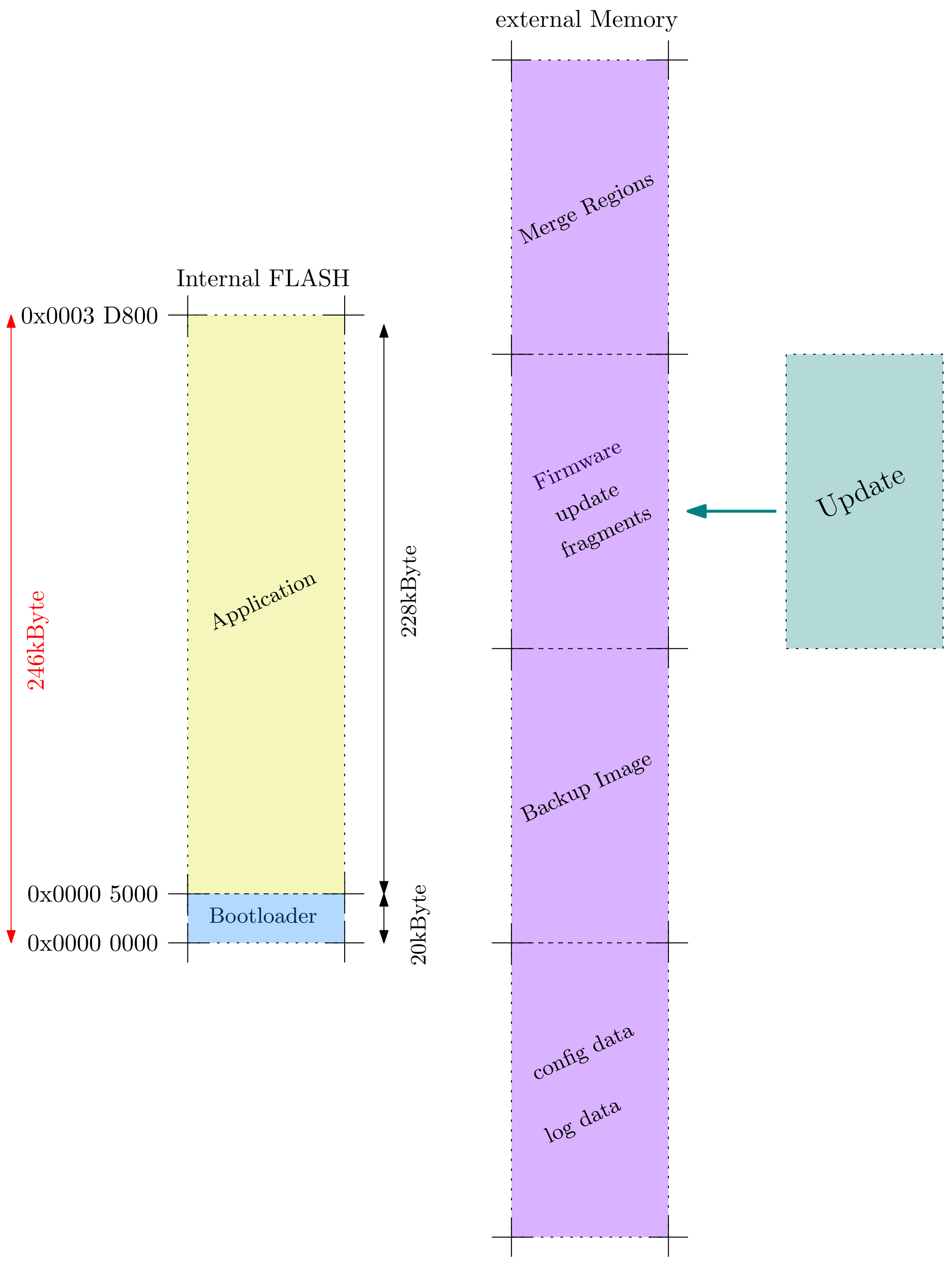


Figure 5.4.: Possible memory setup

- **External RTC**
  As described in chapter 4.2.1, the firmware update over the air protocol needs a time synchronization between the server and the node. To have an actual and accurate time on the node, a RTC chip is needed. This chip needs to be powered

by an own battery. This guaranties that, even if the node loses power, the time would not be lost.

- **Sensors/Actors**
  For demonstration purposes, the node should have sensors and actors on it. For sensors basic temperature and humidity data should be provided. As actors, a display should show the firmware update process. Further the node should have buttons so the user can interact with the demonstrator node.

- **LoRa**
  To provide LoRa functionality to the Node a LoRa chip has to be integrated. This LoRa chip has to be supported from the LoRaMac-node sotware stack from Semtech. Following chips are supported:

  – SX1261

  – SX1262

  – SX1272

  – SX1276

- **SE**
  SE stands for SecureElement. These elements provide a secure key storage for IoT devices. A SE chip shoul be *optionally* taken into account.

Figure 5.5 shows an overview of the different hardware components the demonstrator node should provide.

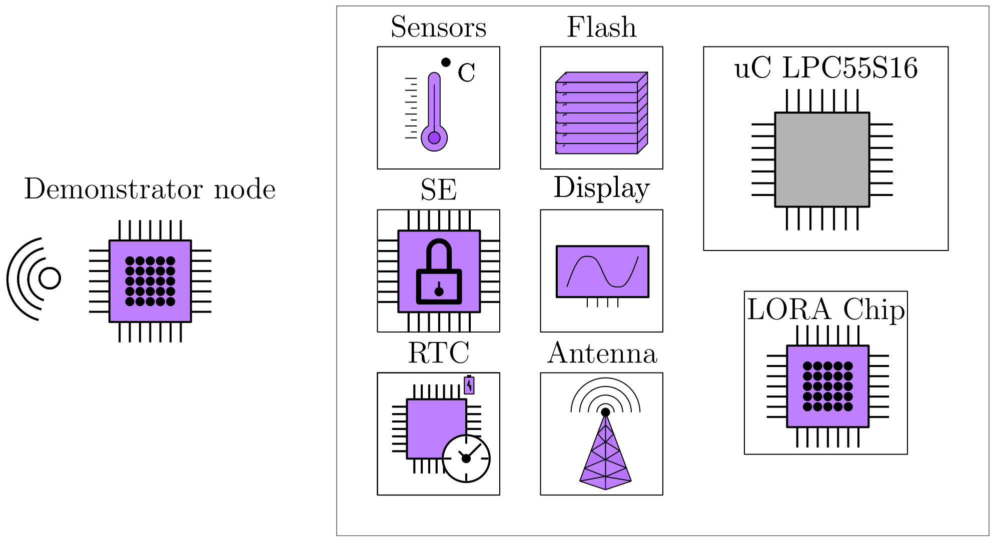


Figure 5.5.: Demonstrator node hardware blocks.

5.1.1.2. Demonstrator components selection

External memory:
At the university, different projects with external memory chips took place. The most common used external on-board memory chips for logging and data cache purposes, is the SPI NOR-Flash chip from Winbond. Winbond has a variety of SPI chips with different memory capacities. The W25 NOR-Flash array is organized into programmable pages of 256-bytes. These pages can be erased in groups of 16 (4kB sector erase), groups of 128 (32kB block erase), groups of 256 (64kB block erase) or the entire chip (chip erase). The small 4kB erase sectors allow greater flexibility in applications that require data and parameter storage, which fits the use-case of the FUOTA application. Further, this NOR-flash guaranties at least 100'000 Program-Erase cycles and a data retention of more than 20years. With a max of 1uA power consumption in low-power mode it fits also requirements for low-power battery driven IoT nodes.
For the demonstrator node a W25Q128 chip will be used. This will provide 16MByte of memory. The reason to choose the Q128 variant is, that the university and common distributors have this chip in stock. Table 5.2 sums up the specification of the NOR-Flash.

interface	133MHz SPI
max transfer rate	66MB/S continuous data transfer rate
write/erase cycles	100'000
data retention	>20years
capacity	16MByte
page size	256byte
sector sizes	4, 16, 32 and 64kByte
low-power current	1uA

Table 5.2.: W25Q128 features [74].

External RTC:
Like the external memory, the university often works with RTC chips for time management on devices. Regarding the chip market crisis RTC chips are rare components and there are long waiting times for the distributors. The university itself has the RTC chip DS3232 in stock. The DS3232 is a low-cost, extremely accurate, I2C real-time clock. The RTC maintains seconds, minute, hour, day, date, month, and year information. There are already generic implemented I2C libraries in C which have to be ported to the LPC55S16. Table 5.3 lists the specification of the DS3232 RTC.

timekeeping Accuracy	±5ppm (±0.432 Second/Day)
interface	400kHz I2C
features	Battery Backup
	Two Time-of-Day Alarms
	1Hz and 32.768kHz Output

Table 5.3.: DS3232 features [47].

Sensors/Actors:
For the sensors the SH31 temperature and humidity sensor from Sensirion will be used. This sensor is commonly used in the industry and often integrated in IoT devices. The sensor provides following features5.4.

interface	I2C
average supply current	1.7 uA
relative humidity accuracy	2%RH
relative humidity range	0 - 100%RH
response time	8s
relative humidity accuracy	0.3 °C
relative humidity range	-40 - 125 °C
response time	2s

Table 5.4.: SHT31 features [63].

For the actors and interaction with the user a display, LEDs and some buttons have to be integrated. For the display, a small size I2C display is used. As like for the other components, the SSD1306 OLED display is often used for prototyping purposes and the university has it in stock. Figure 5.6 shows the SSD1306 OLED display. The 128*64 pixel display provides a I2C interface and each pixel can be manually triggered. As further interaction options, some basic LEDs and PUSH-buttons will be placed on the demonstrator node.

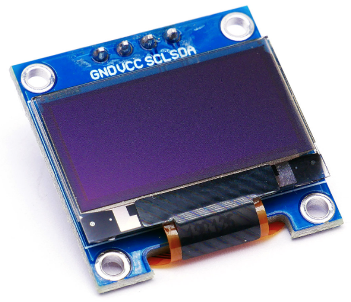


Figure 5.6.: SSD1306 128x64 OLED display [49].

LoRa:
For the LoRa functionality a LoRa module was needed. This means, it is not planned to build the LoRa radio electrical circuit for the common LoRa IC (SX126x, SX1272), but to use a LoRa module which has the electrical circuit already on the module.
A market check showed, that there are plenty of LoRa module available but most of them already have a microcontroller integrated. These microcontrollers have to be used to build the LoRa application or they build an interface, such that you can communicate with AT commands with the LoRa module. For the demonstrator node, which will be a shield on top of the LPC55S16 EVK, these mentioned modules are not suitable.
One module which fits these requirements and has a small form factor is the RFM96 module from HOPERF with a SX1276 Semtech LoRa IC on it. Following figure 5.7 shows the module.

Figure 5.7.: RFM96 LoRa module [31].

5.1.1.3. Demonstrator node hardware development

External memory:
The schematic for the external memory can be seen in figure 5.9. The main part is the IC *U3* which is the W25WQ memory component. During research about the flash component W25Q, a second interesting flash component was mentioned. The W77Q NOR-Flash is a pin-compatible to the W25Q, which means it has the same footprint and pin mapping. The W77Q is a secure flash chip. It provides as add on to the same basic functionality of the W25Q, a secure mechanism, which allows to generate a secure channel from the microcontroller to the flash or even from the remote user (server) to the flash (figure 5.8). To have the possibility to use either one of these two flash chips, a levelshifter and 1.8V power supply for the W77QW had to be developed. The reason for this is that the W25Q chip with logical levels from 0-1.8V is not in stock, and the W77Q is designed for these logical levels. The IC5, ADP150 chip, provides a stable 1.8V power supply with low noise and a maximal current output of 150mA. The IC4, NTB0104, is a specially designed level shifter for the SPI communication bus. With the jumper JP6 to JP9 it can be chosen, if either the SPI communication is directly mapped to the flash chip (W25Q variant) or if it will be level shifted from 3.3V to 1.8V (W77Q variant). With jumper JP5, the 1.8 power supply can be enabled or disabled. This was developed, if the power of 1.8V is not needed, it can be disabled, so there are no additional leak currents from the IC5 and IC4.

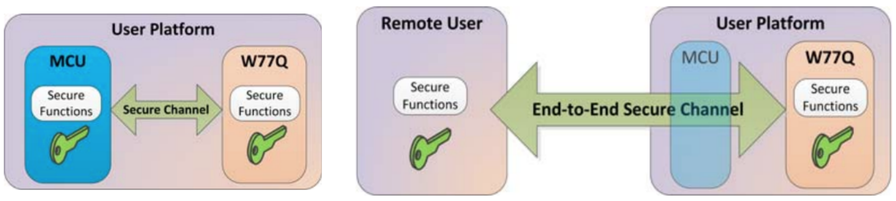


Figure 5.8.: W77Q use case [75].

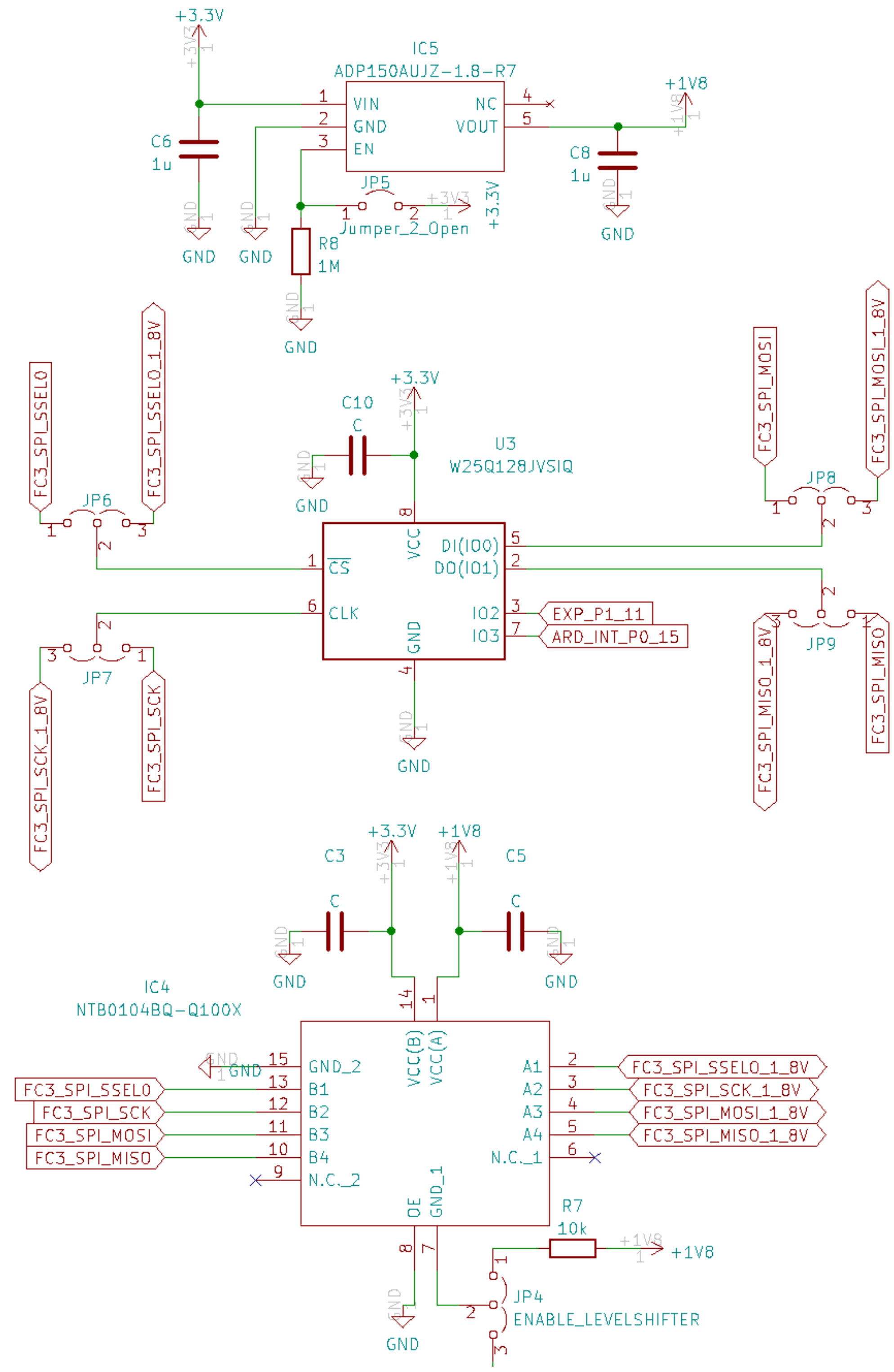


Figure 5.9.: Schematic of W25Q flash.

External RTC:
For the external RTC the electrical schematic is shown in figure 5.10. The RTC connection provides a basic use. The 32kHz output is not used. To the VBat pin a 3V chip battery will be connected. This Vbat pin will also be mapped to an analog input of the microcontroller. This allows to check the capacity of the RTC battery.

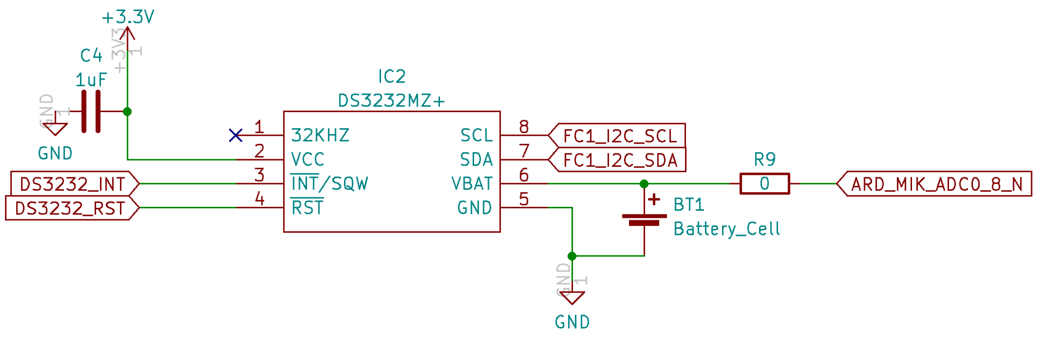


Figure 5.10.: Schematic of RTC DS3232.

Sensors/Actors:
The SHT31 (figure 5.11) is connected to provide the basic functionality. The ADDR pin is connected to ground, which sets the I2C address of the SHT31 to *0x44*.
The SSD1306 OLED display will be connected to the same I2C bus. The display will be connected over basic 4pin header connectors. For the buttons BT1 and BT2 (figure 5.12) additional LEDs were placed, which give a light feedback to the user if the button is pushed. LED D1 shows if the shield is well powered and the LED D5 can be software triggered.

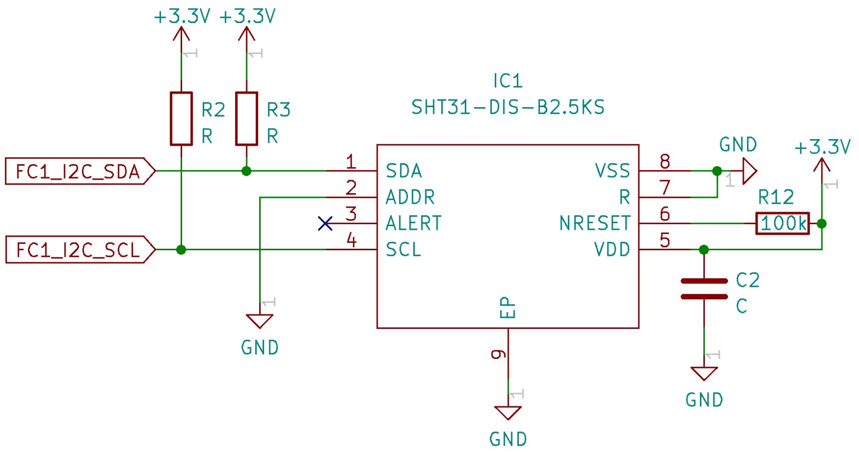


Figure 5.11.: Schematic of SHT31 sensor.

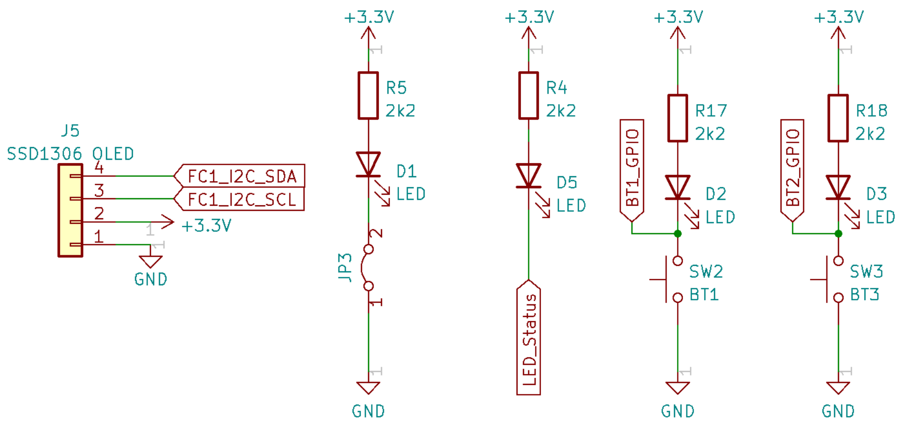


Figure 5.12.: Schematic buttons, LEDs and Diplay connector.

LoRa:
The schematic of the RFM96 LoRa module can be found in figure 5.13. All digital output pins (DIO0-DIO5) are mapped to GPIOs of the microcontroller. An additional button SW1 allows the user to reset the module manually. The jumper JP2 is placed to enable/disable the module and to measure the power usage of the module.

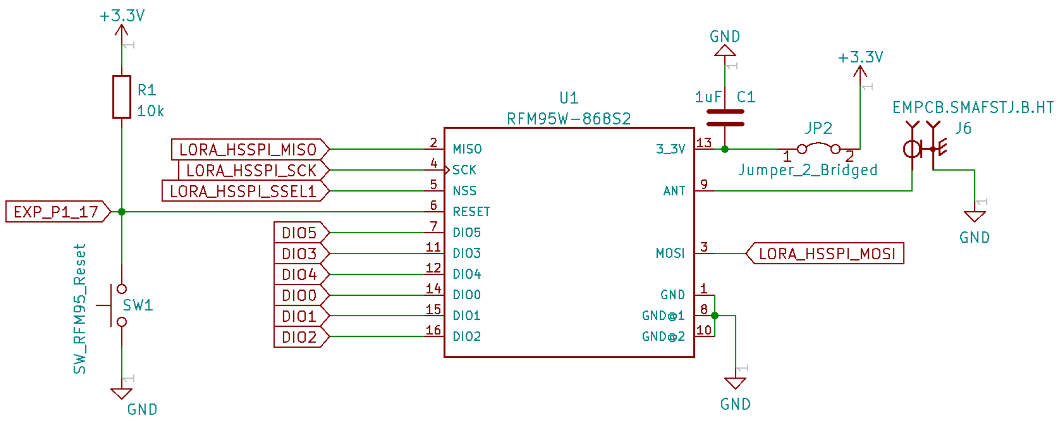


Figure 5.13.: Schematic of RFM96 LoRa module.

5.1.1.4. Layout

Figure 5.14 shows the rendered PCB.

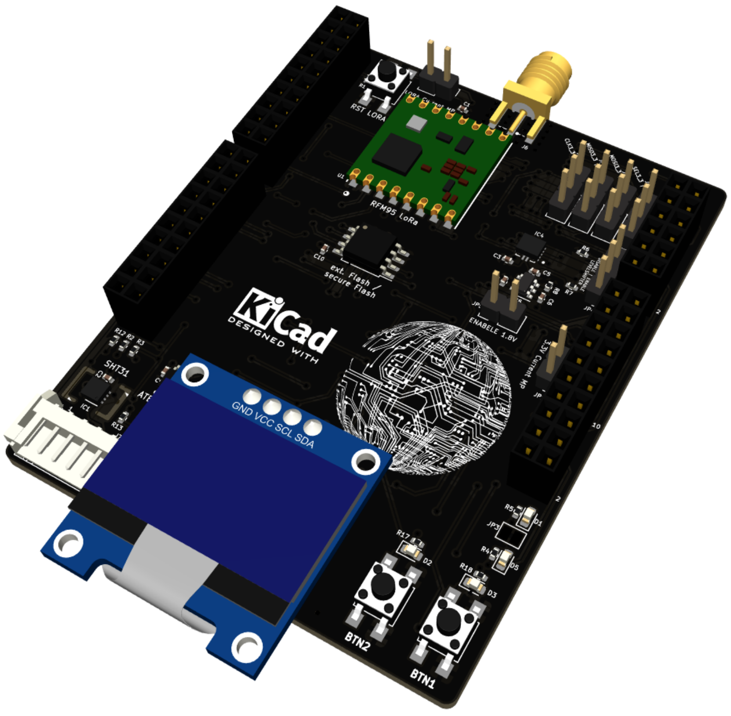


Figure 5.14.: Demonstrator shield rendered

In figure 5.15 the PCB is graphically divided in the main functionalities. Table 5.5 lists what functionalities these parts represent. For the PCB stack a two layer variant is chosen, where the top chopper layer is connected to ground and the bottom chopper layer is connected to 3.3V. The components are placed in such a way, that the LoRa module is not in the region where the user interacts with the shield and next to the boarder to guarantee a minimal trace to the antenna connector.

1 (orange)	This part represents the flash function, which maps to the schematic from figure 5.9. On the right side and on top are all jumpers and in the center is the flash IC.
2 (red)	Red represents the RFM96 LoRa functionality (schematic figure 5.13). The RFM96 modem us green colored and on top the SMA antenna connector is placed.
3 (green)	Green with the number 3, the buttons, LEDs and diplay connector are marked.
4 (blue)	The blue box shows the SHT31 sensor placement. The SHT31 sensor needs cutouts around the PCB. This is needed so there is enough ambient air around the IC and that the copper surfaces of the PCB do not affect the temperature measurement.
5 (purple)	On the backside in blue, the RTC circuit with the battery connector is placed.

Table 5.5.: PCB parts

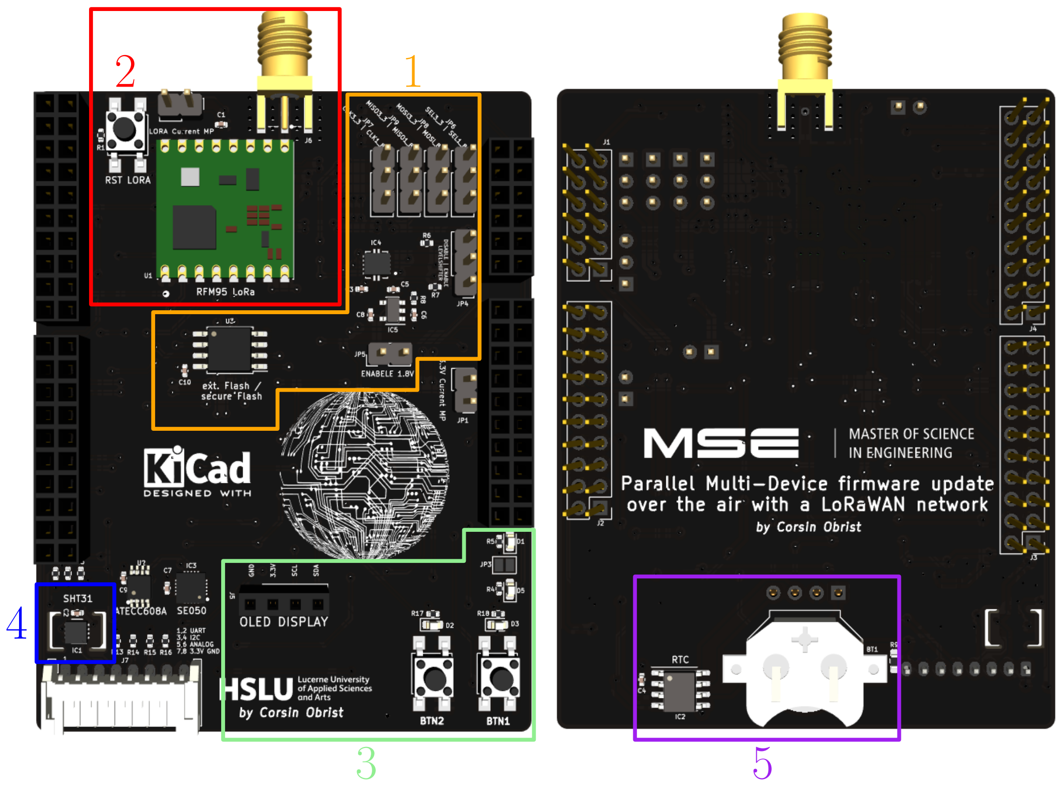

2
1
4
3
5
RST LORA
LORA Current MP
RFM95 LoRa
ext. Flash / secure Flash
ENABELE 1.8V
KiCad DESIGNED WITH
SHT31
ATECC608A
SE050
OLED DISPLAY
HSLU Lucerne University of Applied Sciences and Arts
by Corsin Obrist
BTN2
BTN1
MSE
MASTER OF SCIENCE IN ENGINEERING
Parallel Multi-Device firmware update over the air with a LoRaWAN network
by Corsin Obrist
RTC

Figure 5.15.: Demonstrator shield divided in main parts.

5.2. LNS & Gateway hardware

In previous work[53], a gateway provided by Prof. Erich Styger was used. The gateway a is modular hardware stack containing an RAK831 WisLink LPWAN HAT together with a GPS HAT. Both modules can be placed on top of a Raspberry Pi, which runs the gateway driver. Figure 5.16 shows the two hardware modules next to each other. Figure 5.17 shows the complete hardware stack including a Raspberry Pi 3b+. To have a Raspberry Pi included in this gateway hardware stack provides the flexibility to run other services next to the gateway driver on the same hardware. The experience gained in previous work with this hardware stack lead to the decision to use it for this master thesis project.

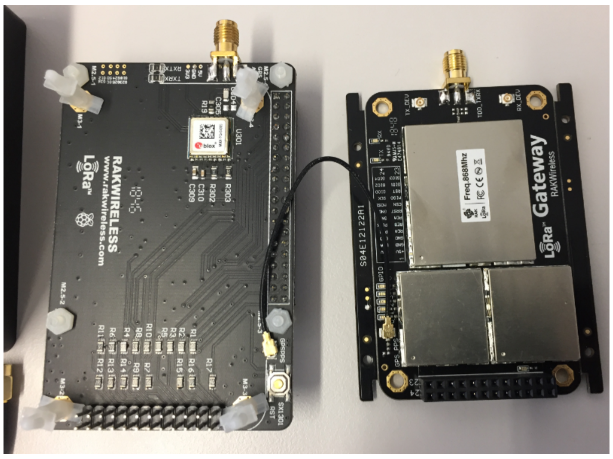


Figure 5.16.: RAK831 with GPS module [68].

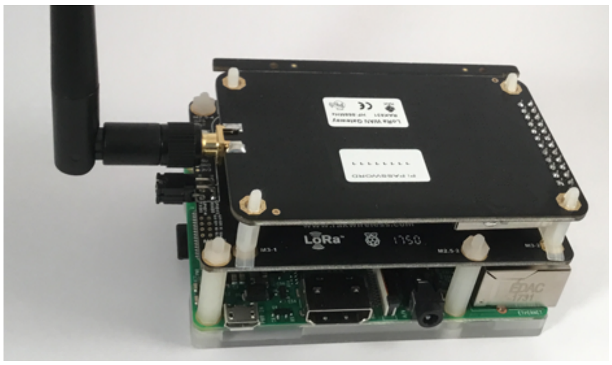


Figure 5.17.: RAK831 Stack by TTN [69].

There are a few open-source and non-open-source LoRaWAN network servers from different providers (table 5.6). These providers combine the application and network server in one software package. The ChirpStack and TTS network server fulfill all the features necessary for this project. In previous work both server were used with basic functionality. The hardware requirements of these servers for a private network are minimal. It is recommended to use a Unix based OS which is able to run Docker container on it. For

the Hardware a Lenovo Thinkpad notebook with an Ubuntu 20.04LTS OS installed will be used. First a solution with The Things Stack was set up, but this solution caused problems regarding TLS certificates between the gateway and the LNS as well as the CLI host interface and the LNS.
These problems lead to switch to the ChirpStack implementation. In chapter **??** this setup is explained.

Provider	OpenSource	Private-Network possibility	Community	FUOTA ready	Cost free
Loriot		x		-	
Acklio		x		x	
ChirpStack	x	x	x	x	x
The Things Stack	x	x	x	x	x

Table 5.6.: LNS provider.

5.3. FUOTA server

The FUOTA server will be a software service, which can be run on the same hardware as the LNS server. The hardware should be able to run Python an JavaScript scripts. For the FUOTA server the same host as for the LNS will be used.

6. Software Design

The following chapter explains the different software and firmware components used and how they are implemented on both the server side as well as the demonstrator node. First, the process flow of the firmware update over the air on a high level of abstraction will be presented, then a deeper insight on components level will be made.

6.1. FUOTA server

In figure 6.1 the block graphic gives an overview how the FUOTA process flow is organized.

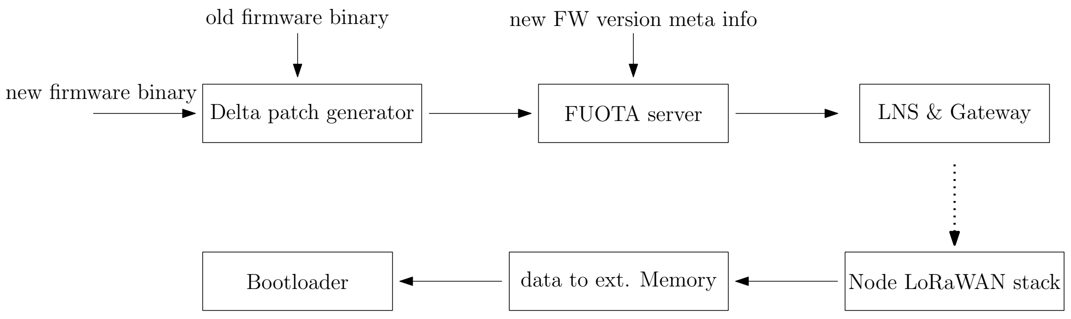


Figure 6.1.: FUOTA process overview.

- **Delta patch generator:**
  First the new firmware has to be build with the same compiler optimization level as the old firmware version. The *Delta patch generator* will then build the delta from the new version with the old version and generate a patch binary as a output.

- **FUOTA server:**
  The FUOTA server has the task, to generate meta file information for the update process and then start the process with setting up the time synchronization, multicast and fragmented data block transfer protocol. After the setup the FUOTA server will send the patch file divided in fragments via the LNS&Gateway to the demonstrator node.

- **LNS & Gateway:**
  The LNS (LoRaWAN network server) is responsible to handle the LoRaWAN protocol and it's MAC commands. Together with the gateway it builds the minimum

setup to establish a LoRa communication between the LoRaWAN network and the nodes in the field. The gateway acts as a package broker. This means it converts the messages received by the LNS to a LoRa modulated signal and vice versa.

- **Node LoRaWAN stack:**
  The LoRaWAN stack on the node is responsible to control the LoRa transceiver. This includes the communication with the transceiver via SPI and the interpreting of the LoRaWAN communication messages. It also handles the clock synchronization, multicast and fragmented data block transport protocol.

- **Data to external Memory:**
  Here the node will save the received meta info and fragmented data into the external memory.

- **Bootloader:**
  Depending on the metadata stored in the external memory, the bootloader is responsible to merge a new received patch file with the actual image (stored in the external memory) and boot the new generated firmware image. Further the bootloader should handle failed boot process and should be able to boot a backup image or the last run application if something went wrong with the merge process.

6.1.1. Delta patch generator

For the patch file, a JavaScript library [34] will be used. There, the JojoDiff [1] differencing algorithm was ported to an executable script to create a patch file from two binary images. This library was used, because there is a memory efficient C library available, which allows the node to rebuild a firmware binary from a patch file created with the JojoDiff algorithm. As an example two different firmware versions were build on the base of a *hello_wolrd* example project from the MCUXpresso IDE (table 6.1). Table 6.2 shows the size of the patch file generated from these two firmware versions. It shows that the BSDiff algorithm could build a smaller patch file, but because the BSDiff algorithem has a bigger memory footprint, it is not possible to use it on the LPC55S16 demostrator node.

Version number	Description	Codesize (bytes)
V1	Base version, prints uart input as a echo to the uart output (basic hello_world example)	9180
V2	adds an GPIO driver to toggle a LED every time, an input on the uart interface arrives	9320

Table 6.1.: Different firmware versions for testing purposes.

Compared versions	Description	Codesize (bytes)
V1 -> V2	JojoDiff algorithm	1501
V1 -> V2	BSDiff algorithm	657

Table 6.2.: Example JojoDiff patch file.

6.1.2. FUOTA server process overview

The FUOTA server uses a MQTT interface as a communication protocol between the LNS and the FUOTA server. As a base the FUOTA server uses the implementation by Ahmed Elsalahy [23].
The FUOTA server was adapted to the developed FUOTA protocol which is shown in figure 6.5 and explained in following section. The protocol is split into seven parts (figure 6.2).

1	FUOTA metadata information
2	Clock synchronization
3	Multicast session setup
4	Fragmentatioin session setup
5	Multicast class C request
6	Firmware multicast fragments
7	FW failed/complete message

Figure 6.2.: FUOTA protocol main parts.

In code listing 6.1 and 6.2, the JavaScript implementation for the server MQTT configuration and device configuration are shown. At code listing 6.2, the user has to write all the device EUIs, which represent the devices that should receive a firmware update, in the constant array *devices*.

```
//———————— Start of config area ——————————//
// MQTT TheThingsStack config, see https://www.thethingsindustries.com/docs/integrations/mqtt/
// MQTT Chirpstaclconfig, see https://www.chirpstack.io/application-server/integrations/mqtt/
//###### MQTT connection config ######//
var HOST = '192.168.1.112';
var APP_ID = "";//'fuota-app';
var API_KEY = "";// API key from the Application server
var PORT = 1883 ; // 1883 or 8883 for TLS
var options = {
    host: HOST,
    username: APP_ID,
    password: API_KEY,
    port: PORT,
    rejectUnauthorized: false,
    protocol: 'mqtt' //mqtts for TLS
}
```

Listing 6.1: FUOTA server configuration.

```
// Device IDs and EUIs config
var GATEWAY_ID= 'mt-gateway';

//###### Multicast group details config ######//
var MULTICAST_APP_ID= 'fuota-app';
var MULTICAST_DEV_ID= 'multicast-dev';

//###### All devices which receive FUOTA ######//
// device EUIs config with no spaces, Example: `FA23A01E61AE4F65`
const devices = [
    '1111111111111111'
];
//————————— End of config area ————————————//
```

Listing 6.2: FUOTA EUI and IDs definition.

In the code listing 6.2, only one device is added *EUI* (0x11 0x11 0x11 0x11 0x11 0x11 0x11 0x11), which should receive a firmware update.

6.1.3. FUOTA metadata information

As a first step, the FUOTA server generates metadata, which are necessary for the node for a successful firmware update over the air. Following table 6.3 lists the metadata that have to be calculated by the server.

data description	size (bytes)	example data
size of the patch file	4	1501
new firmware version	4	0x01020304
# of patch files	4	1
size of a fragments	4	50
# of fragments per patch (this includes redundancy fragments)	4	48
HASH of patch file	32	0x7ca12506e88cf8b814e20848b229460f 91fc0370c44a7c4fee786960ce30c36d
HASH of new firmware	32	0x3b2730fa78b1bf326d33c0739908354c 9f03ff724627c1c448dfe9f91d4f8f29

Table 6.3.: Metadata explanation.

In code listing 6.4, the calculation of this metadata in the server are shown. To calculate this data, the user have to pass the file path of the fragmentation file, patch file and new firmware file at the start of the FUOTA service. Additionally, the user has to pass a string with the new firmware version. Code listing 6.3 shows how these arguments are passed while executing the FUOTA server.

```
// node.js instance, server script, fragmented data file, patch binaray,
    new firmware binary, firmware version
node server.js fragmented_patch.txt patch.bin newFirmware.bin 01020304
```

Listing 6.3: Server execution arguments

```
//###### Script arguments ######//
const client = mqtt.connect(options);
const PACKET_FILE = process.argv[2];        //path to fragment file "
    fragmets.txt"
const PATCH_BIN_FILE = process.argv[3];     //path to patch file "patch.bin
    "
const NEW_IMG_BIN_FILE = process.argv[4];   //path to new firmware file "
    new_FW.bin"
const FW_VERSION = process.argv[5];         //new firmware version number
    "01020304" (0x01, 0x02, 0x03, 0x04) --> Version 1.2.3.4


//###### Metadate information creation ######//
// reading binary files
const fileBufferPatchBIN = fs.readFileSync(PATCH_BIN_FILE);
const fileBufferNewBIN = fs.readFileSync(NEW_IMG_BIN_FILE);
// patch hash calculation
// patch size calculation
const patch_SHA = crypto.createHash('sha256');
const patch_SHA_HEX = patch_SHA.update(fileBufferPatchBIN).digest('hex');
const patch_State = fs.statSync(PATCH_BIN_FILE);
const patch_Size = patch_State.size;
// new firmware hash calculation
const newIMG_SHA = crypto.createHash('sha256');
const newIMG_SHA_HEX = newIMG_SHA.update(fileBufferNewBIN).digest('hex');
// #fragments and fragments size calculation
let packets = parsePackets();
const frag_CNT = packets.length-1;
const frag_Size = packets[1].length-3;
//###### Metadate converting to Buffer for correct use ######//
var patch_SHA_buffer = patch_SHA_HEX.match(/[0-9a-z]{2}/gi);  // splits the
     string into segments of two including a remainder => {1,2}
var patch_SHA_buffer_res = patch_SHA_buffer.map(t => parseInt(t, 16));

var newBin_SHA_buffer = newIMG_SHA_HEX.match(/[0-9a-z]{2}/gi);  // splits
    the string into segments of two including a remainder => {1,2}
var newBin_SHA_buffer_res = newBin_SHA_buffer.map(t => parseInt(t, 16));

var FWversion_SHA_buffer = FW_VERSION.match(/[0-9a-z]{2}/gi);  // splits
    the string into segments of two including a remainder => {1,2}
var FWversion_SHA_buffer_res = FWversion_SHA_buffer.map(t => parseInt(t,
    16));

var patchSize_buf = getBufferFromInt(patch_Size,3);
patchSize_buf = Buffer.from(patchSize_buf);
```

Listing 6.4: FUOTA metadata calculation

This metadata will be sent as two unicast messages to each node, over the LoRaWAN Port 144 and 145. These ports (ports 1-199 are free to use) were randomly chosen and have no further importance. The two messages sent are show in figure 6.3 and will be sent before the general FUOTA protocol will be set up.

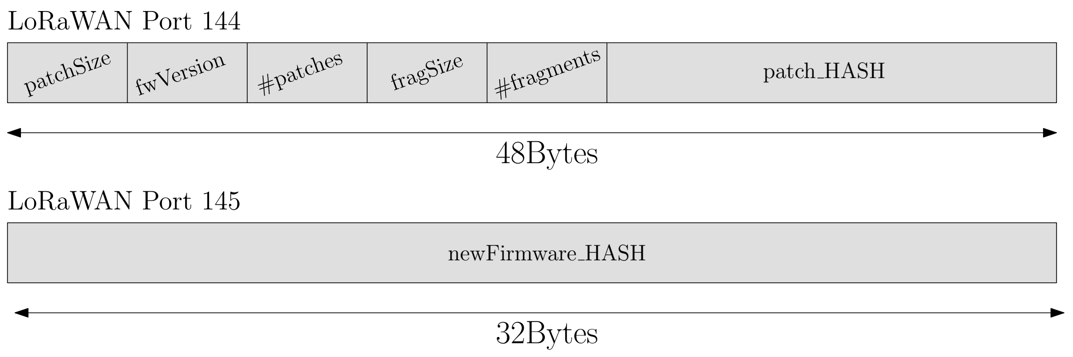


Figure 6.3.: Metadata unicast messages.

6.1.4. Clock synchronization

As soon as the metadata information is received by the node, the node is requesting for a time synchronization from the server. This request messages include the actual device time. The *DeviceTime* is the current end-device clock and is expressed as the time in seconds since 00:00:00, Sunday 6th of January 1980 (start of the GPS epoch) modulo 2^{32} [42]. The FUOTA server then has to calculate the time difference between the node and itself. This can be seen in code listing 6.5.

```
    ...
    ...
    ...
    let deviceTime = body[1] + (body[2] << 8) + (body[3] << 16) + (body[4]
        << 24);
    let serverTime = gpsTime.toGPSMS(Date.now()) / 1000 | 0;

    let adjust = serverTime - deviceTime | 0;
    let resp = [ 1, adjust & 0xff, (adjust >> 8) & 0xff, (adjust >> 16) & 0
        xff, (adjust >> 24) & 0xff, 0b0000 /* tokenAns */ ];
    let responseMessage = {
        "downlinks": [{
            "priority": "NORMAL",
            "f_port": 202,
            "frm_payload": Buffer.from(resp).toString('base64')
        }]
    };
```

Listing 6.5: Clock sync server answer

6.1.5. Multicast session setup

For the multicast session setup, the server has to send a message with the data listed in table 6.4.

	McGroupIDHeader	McAddr	McKey_encrypted	minMcFCount	maxMcFCount
size (bytes)	1	4	16	4	4

Table 6.4.: McGroupSetupReq [8].

- **McGroupIDHeader**
  This value gives an ID for the multicast group the device joins.
- **McAddr**
  The multicast address is the address of the virtual device in the LNS. Over this multicast device all the multicast messages will be sent. From this device address and the McKey_encrypted, the relevant session keys will be derived.
- **McKey_encrypted**
  The McKey_encrypted is the encrypted multicast group key from which McAppSKey and McNetSKey will be derived. With the McKey_encrypted and the McKEKey the McKey will be derived. This McKey will then be used with the McAddr to derive the mentioned McAppSKey and McNetSKey.
  The McKEKey is a lifetime end-device specific key used to encrypt a multicast key transported over the air (Key Encryption Key). For the LoRaMac-node software stack version 1.1, which runs on the demonstrator node, this key will be generated as followed:

  $$McRootKey = aes128\_encrypt(AppKey, 0x20|pad_{16})$$
  $$McKEKey = aes128\_encrypt(McRootKey, 0x00|pad_{16})$$

  The mentioned AppKey is the device unique key, that the node needs to communicate in the LoRaWAN network in a normal unicast way. The key management for the multicast sessions are complicated. With the help of Miguel Luis, a SEMTECH application engineer, who developed a python script [46] to generate the different multicast key, the correct McKey_encrypted key can be derived for all the nodes in the field. The script can be found in the attachment A.2. An example output of the script can be seen in code listening 6.6.

```
LoRaWAN 1.1.0
AppKey          : 000102030405060708090A0B0C0D0E0F
McAddr          : 0x01FFFFFF
McRootKey       : 430BFF9B049F19279455BD564133C73B **
McKeKey         : 0FC43A2A45FDB753DD065270B50AB9F2 **
McKey           : 0102030405060708090A0B0C0D0E0F10
McKeyEncrypted  : 67608274FDD6C3937DA6C58030273C60 **
McAppSKey       : C3F6C39B6B6496C29629F7E7E9B0CD29
McNwkSKey       : BB75C362588F5D65FCC61C080B76DBA3
```

Listing 6.6: Example multicast keys.

Further, figure 6.4 gives a schematic view of the key derivation process.

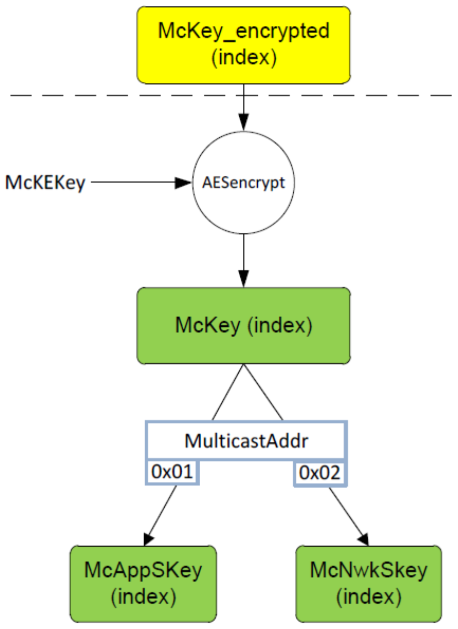


Figure 6.4.: Multicast key derivation process [8].

- **minMcFCount**
  The minMcFCount field is the next frame counter value of the multicast downlink to be sent by the server for this group. This information is required in case an end-device is added to a group that already exists. The end-device MUST reject any downlink multicast frame using this group multicast address if the frame counter is < minMcFCount.

- **maxMcFCount**
  Specifies the lifetime of this multicast group expressed as a maximum number of frames. The end-device will only accept a multicast downlink frame if the 32bits frame counter value $minMcFCount \leqslant McFCount < maxMcFCount$.

  In code listing 6.7 an example of the downlink message from the server to the node for the multicast session request is shown. The McKey_encrypted is hardcoded in this variant. This is only possible if all nodes have the same API_Key, which is not secure, but appropriate for testing the system it is simpler.

```
console.log('sendMcGroupSetup\r\n\r\n');
// mcgroupsetup
let mcGroupSetup = {
    "downlinks": [{
        "priority": "NORMAL",
        "f_port": 200,
        "frm_payload": Buffer.from([ 0x02, 0x00,
            0xFF, 0xFF, 0xFF, 0x01, // McAddr
            0x58, 0x2D, 0x3B, 0x83, 0xEA, 0xD5, 0x18, 0x70, 0x72, 0x19
                , 0x83, 0x2B, 0x39, 0x09, 0x3D, 0xE9, //
                McKey_encrypted
            0x0, 0x0, 0x0, 0x0, // minFCnt
            0xff, 0xff, 0x0, 0x0 // maxFCnt
        ]).toString('base64')
    }]
};
```

Listing 6.7: Example multicast session request message.

6.1.6. Fragmentation session setup

For the fragmentation session setup, the server has to send a FragSessionSetupReq message with the data listed in table 6.5.

	FragSession	NbFrag	FragSize	Control	Padding	Descriptor
size (bytes)	1	2	1	1	1	4

Table 6.5.: FragSessionSetupReq [9].

- **FragSession**
  The FragSession byte sets which fragmentation session will be used (there are 4 simultaneously possible fragmentation sessions). It also decides from which multicast session the set fragmentation session is able to receive fragments.

- **NbFrag**
  NbFrag specifies the total number of fragments of the patch file or generally of the data block to be transported during the multicast fragmentation session.

- **FragSize**
  FragSize defines the size in byte of each fragment.

- **Control**
  With the control byte the used fragmentation algorithm can be chosen and a device specific delay time can be set, helps to avoid that all devices answering at the same time to the server.

- **Padding**
  The binary data block size may not be a multiple of the FragSize. Therefore, some padding bytes must be added to fill the last fragment. This field encodes the number of padding byte used.

- **Descriptor**
  The descriptor field is a freely allocated 4 bytes field describing the file that is going to be transported through the fragmentation session. This can be defined by the user.

```
    console.log('sendFragSessionSetup\r\n\r\n');
    let msg = {
        "downlinks": [{
        "priority": "NORMAL",
        "f_port": 201,
        "frm_payload": Buffer.from(parsePackets()[0]).toString('base64')
    }]
    };
```

Listing 6.8: Fragmentation session setup downlink message.

In code listing 6.8 an example of the downlink message from the server to the node for the fragmentation session setup is shown. What is special, is, that this message is directly parsed from the fragmented file (fragmented_patch.txt) passed during the start of the server (see code listing 6.3). In this file the first raw has all the information needed for the explained setup ready and correct aligned. For an example check in code listing 6.9 the first row. The first number 02 indicates the FragSessionSetupReq and then the next numbers would map to:

$$\begin{aligned} FragSession &= 0x00 \\ NbFrag &= 0x0015 \\ FragSize &= 0x30 \\ Control &= 0x00 \\ Padding &= 0x0d \\ Descriptor &= 0x00000000 \end{aligned}$$

```
02 00 15 00 30 00 0d 00 00 00 00

08 01 00 6a b8 00 4d 25 18 2f 58 65 89 6b af 34 e0 07 ab 6a 8e 24 33 cc f7
    50 b1 df e4 16 0a 2b e0 19 6b f4 88 b0 50 af d5 05 bf fc bf 48 c9 1a 72
    e8 4b

08 02 00 49 21 05 ad 36 44 7d 0e 92 25 96 b2 1a 6a b1 6a 0d 89 a9 83 f1 96
    e6 23 24 ce 65 2a ed 59 bf b9 e1 9f bf 59 45 e4 6d f6 a9 9f 1d f2 73 93
    68 3c

08 03 00 e6 92 01 9d 53 86 e2 5b e8 9f 04 ed b0 fe 2c 23 ee b2 11 c1 c0 eb
    b5 20 05 07 07 3f db 89 8d d7 20 47 95 f7 8f 05 cc 3b ce 63 dc bf 8e 23
    0f 1f

08 04 00 0b 1b 3a b6 8f 03 12 f7 91 34 40 be 32 94 f7 d8 3d 7e 9e d3 28 05
    12 7c 9b 91 51 fc 94 61 8c be ab 44 11 5e a1 36 c0 11 42 cd 31 a3 86 ec
    43 11

08 05 00 bb c5 e7 ee 04 3c 9d f2 f2 8e e9 a4 63 7c 9a 8c ec ea 65 f6 2c 20
    22 81 05 f2 d3 04 2d 55 21 ce 58 f0 94 e0 d7 e7 75 8b cc 49 7d 99 04 97
    6f ce

08 06 00 50 e4 93 2f c9 9c 0e 6b ad ea 00 f9 60 f8 9a d3 93 a0 49 96 bf db
    c1 b2 dc 52 ab 73 61 69 66 51 9f 6e f8 ad 33 6d 58 cb e0 e6 a9 45 40 da
    87 cd
...
...
...
```

Listing 6.9: Example of a fragmented data block file.

6.1.7. Multicast class C request

For the multicast class c session start request, the server has to send a McClassCSessionReq message with the data listed in table 6.6.

	McGroupIDHeader	Session Time	SessionTimeOut	DLFrequ	DR
size (bytes)	1	4	1	3	1

Table 6.6.: McClassCSessionReq [8].

- **McGroupIDHeader**
  McGroupID is the identifier of the multicast group being used. It was defined during the multicast session setup (table 6.4).
- **Session Time**
  This defines the time of the start of the Class C window, and is expressed as the time in seconds since 00:00:00, Sunday 6th of January 1980 (start of the GPS epoch) modulo 2^{32}. Note, that this is the same format as the time field in the beacon frame.
- **SessionTimeOut**
  This encodes the maximum length in seconds of the multicast session. After this timeout the node will switch back to class A.
- **DLFrequ**
  Encodes the frequency used for the multicast. This field is a 24 bits unsigned integer. The actual channel frequency in Hz is $100xDlFrequ$. This allows setting the frequency of a channel anywhere between 100 MHz to 1.67 GHz in 100 Hz steps.
- **DR**
  DR is the index of the data rate used for the multicast. As a reference which DR exists check the presented values in the appendix A.1.

In code listing 6.10, an example of the downlink message from the server to the node for the multicast class C session request is shown.

```
console.log('SENDING sendMcClassCSessionReq!\r\n');
let msg = {
    "downlinks": [{
    "priority": "NORMAL",
    "f_port": 200,
    "frm_payload": Buffer.from([
        0x4,
        0x0, // mcgroupidheader
        startTime & 0xff, (startTime >> 8) & 0xff, (startTime >> 16) & 0xff
            , (startTime >> 24) & 0xff,
        0xFF, // session timeout
        0x9d, 0xba, 0x84, // 8'698'525Hz
        0x03 // Datarate 3
    ]).toString('base64')
}]
};
```

Listing 6.10: Example multicast class C session request message.

6.1.8. Firmware multicast fragments

In this section of the FUOTA protocol, the node has already switched to class C and is now expecting the fragmented data blocks. The FUOTA server sends these packages after each other to the node. The code listing 6.11 shows the function in the FUOTA server, which takes on after another raw from the fragments file (see code listing 6.9) and sends it as a multicast message.

```

for (let p of packets) {
    // first row is header, don't use that one
    if (counter === 0) {
        counter++;
        continue;
    }

    let msg = {
        "downlinks": [{
        "priority": "NORMAL",
        "f_port": 201,
        "frm_payload": Buffer.from(p).toString('base64'),
        "class_b_c": {
            "gateways": [
                {
                    "gateway_ids": {
                        "gateway_id": GATEWAY_ID
                    }
                }
            ]
        }
    }]
    };

    client.publish(`v3/${mcDetails.application_id}/devices/${mcDetails.
        device_id}/down/push`, Buffer.from(JSON.stringify(msg), 'utf8'));
    await sleep(2100);
}
```

Listing 6.11: Example loop for sending the fragmented data blocks.

6.1.9. FW failed/complete message

At the end of the protocol, the FUOTA server expects a message on port 146. This message tells the server if the received patch on the node has the same $HASH$ value as the server calculated. By this, the server knows if the fragmented data block transfer protocol has worked correctly and stitched the fragments back to the correct patch binary. After that, the node will further process it. The tasks on FUOTA server side are done.

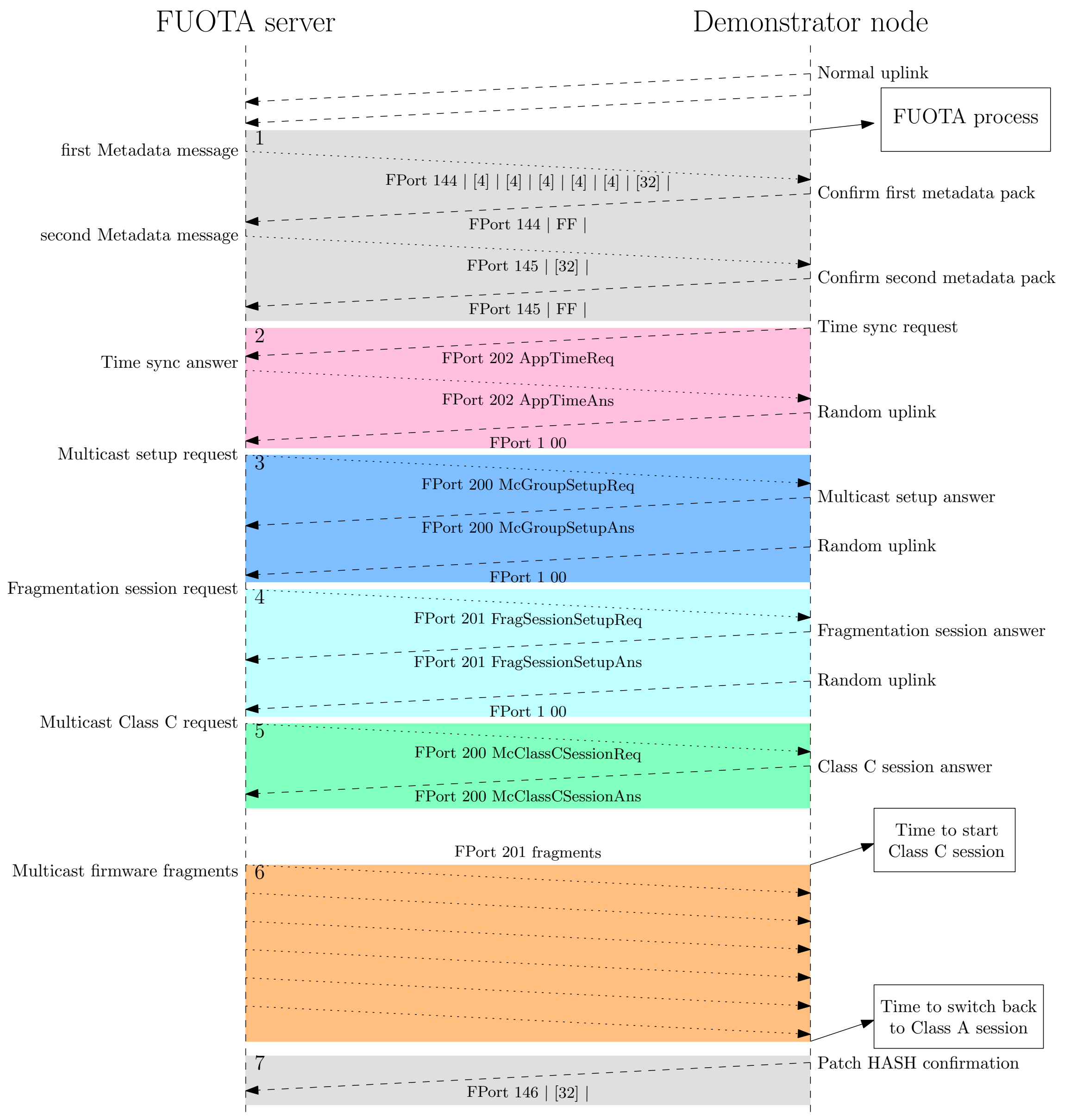

FUOTA server
Demonstrator node
Normal uplink
FUOTA process
1
first Metadata message
FPort 144 | [4] | [4] | [4] | [4] | [4] | [32] |
Confirm first metadata pack
FPort 144 | FF |
second Metadata message
FPort 145 | [32] |
Confirm second metadata pack
FPort 145 | FF |
2
Time sync request
Time sync answer
FPort 202 AppTimeReq
FPort 202 AppTimeAns
Random uplink
FPort 1 00
3
Multicast setup request
FPort 200 McGroupSetupReq
Multicast setup answer
FPort 200 McGroupSetupAns
Random uplink
FPort 1 00
4
Fragmentation session request
FPort 201 FragSessionSetupReq
Fragmentation session answer
FPort 201 FragSessionSetupAns
Random uplink
FPort 1 00
5
Multicast Class C request
FPort 200 McClassCSessionReq
Class C session answer
FPort 200 McClassCSessionAns
Time to start Class C session
FPort 201 fragments
6
Multicast firmware fragments
Time to switch back to Class A session
7
Patch HASH confirmation
FPort 146 | [32] |

Figure 6.5.: FUOTA detailed protocol.

6.2. LNS & Gateway

In the following section the used gateway and LNS environment in this work are explained.

6.2.1. Gateway

As mentioned in chapter 5.2, a hardware stack with a Raspberry Pi was selected for the gateway. For the gateway stack in previous work, the LoRaBasicStation [62] service was installed. This service is an implementation of a LoRa packet forwarder, which forwarder is a program running on the host of a LoRa-based gateway (with or without GPS). It forwards LoRa packets received by the concentrator to a LoRaWAN Network Server (LNS) through a secured IP link [53].
For the LNS in previous work the Things Stack [73] and the Balena environment [14] were used. The experience showed that there are some problems regarding certificates on the different services (gateway driver, LoRaWAN network server and application server). The basic functionalities were working but the advanced function regarding multicast caused problems.
One way to simplify the setup, was to install a UDP package broker on the gateway. This was done using the chirpstack-gateway-OS-base package provided by the Chirpstack environment. The chirpstack-gateway-OS-base supports the used gateway hardware stack (figure 5.17, chapter 5.2) with the Raspberry Pi and the RAK831 LoRa concentrator module. This service provides the ChirpStack concentrator and ChirpStack Gateway Bridge pre-installed including a CLI utility for gateway configuration. The gateway was then able to connect to after the Things Stack LNS [73] or the Chirpstack LNS [19].

6.2.2. LNS

For the LoRaWan network server, the Chirpstack implementation will be used. The reason for using the Chirpstack is, that it is providing more flexibility and the example FUOTA application [59] from Semtech was tested using the Chirpstack server. For the installation the published documentation [19] will give a guide.
In code listing 6.12, the initialization of the database instances for the network server and application server is made. The name and password have then to be placed in the configuration file (.toml file) of the network server (code listing 6.13) and application server (code listing 6.14). In the network server configuration the used frequency plan has to be defined. In the application server the web-frontend address and port as well as login credentials have to be set.
With the following commands the networks server and application server will be started and can be accessed via the cli interface or the browser over the address http://localhost:8080.

$$sudo\ systemctl\ start\ chirpstack - network - server$$
$$sudo\ systemctl\ start\ chirpstack - application - server$$

```
-- set up the users and the passwords
-- (note that it is important to use single quotes and a semicolon at the
   end!)
create role chirpstack_as with login password 'dbpassword';
create role chirpstack_ns with login password 'dbpassword';

-- create the database for the servers
create database chirpstack_as with owner chirpstack_as;
create database chirpstack_ns with owner chirpstack_ns;

-- change to the ChirpStack Application Server database
\c chirpstack_as

-- enable the pq_trgm and hstore extensions
-- (this is needed to facilitate the search feature)
create extension pg_trgm;
-- (this is needed to store additional k/v meta-data)
create extension hstore;

-- exit psql
\q
```

Listing 6.12: Chirpstack postgres database init.

```
[general]
log_level=4

[postgresql]
dsn="postgres://chirpstack_ns:dbpassword@localhost/chirpstack_ns?sslmode=
   disable"

[network_server]
net_id="000000"

[network_server.band]
name="EU_863_870"
...
...
```

Listing 6.13: Chirpstack network server configeration.

```
[general]
log_level=4

[postgresql]
dsn="postgres://chirpstack_as:dbpassword@localhost/chirpstack_as?sslmode=
   disable"

[application_server.external_api]
jwt_secret="verysecret"
...
...
```

Listing 6.14: Chirpstack application server configeration.

6.3. Demonstrator node firmware

In this section, the firmware implementation of the bootloader and the LoRaWAN FUOTA application will be presented. First the general firmware architecture will be shown and further the different components are considered in detail.

6.3.1. Firmware architecture

In figure 6.6, the firmware stack for the demonstrator node (chapter 5.1) is shown. This stack shows on an abstract level which different modules are used for the demonstrator node to fulfill its task of a firmware update over the air. The different modules are explained in following list.

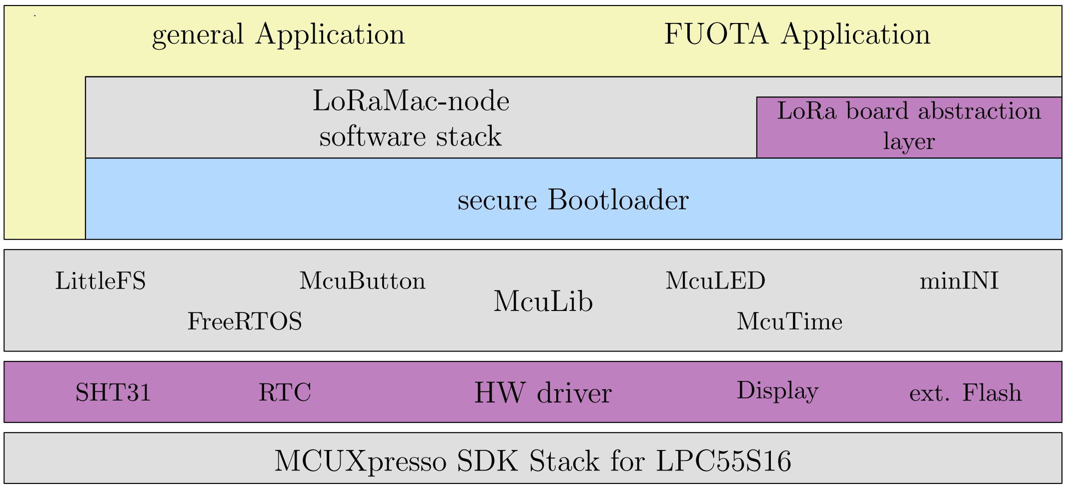


Figure 6.6.: Firmware stack.

- **MCUXpresso SDK**
  The MCUXpresso SDK is a comprehensive software enablement package designed to simplify and accelerate application development with Arm® Cortex®-M-based devices from NXP. The MCUXpresso SDK includes a production-grade software with integrated RTOS (optional), integrated enabling software technologies (stacks and middleware), reference software, and more[51]. The stack is shown in figure 6.7

- **HW driver**
  The hardware drivers are implementing the functionality of the hardware components used on the demonstrator node. This includes the I2C driven SHT31 temperature and humidity sensor, RTC and OLED display aswell as the SPI driven W25Q NOR flash chip.

- **McuLib**
  The McuLib is a scalable C/C++ library from the McuOnEclipse [64] project by

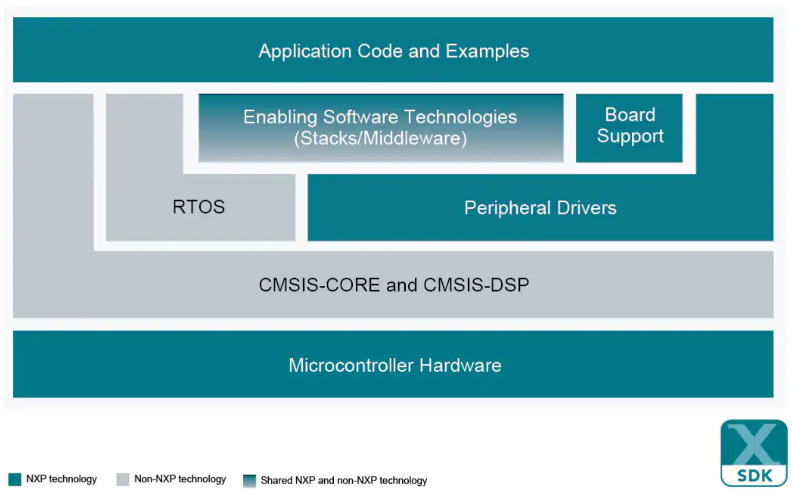


Figure 6.7.: MCUXpresso SDK stack [51].

Erich Styger. This library adds additional functionality to the hardware driver. For the demonstrator node the integations of the LitteFS, FreeRTOS, minIni, SSD1306, McuTime, McuSHT31, McuLEDs and McuButtons were used [65].

- **LittleFs**
  The LittleFs is a little fail-safe filesystem designed for microcontrollers. The LittleFs is designed to handle random power failures. All file operations have strong copy-on-write guarantees and if power is lost the filesystem will fall back to the last known good state (**Power-loss resilience**). Further the LittleFs is designed with flash in mind, and provides wear leveling over dynamic blocks. Additionally, LittleFs can detect corrupted blocks and work around them (**Dynamic wear leveling**). It is also designed to work with a small amount of memory. RAM usage is strictly bounded, which means RAM consumption does not change as the file system grows. The file system contains no unbounded recursion and dynamic memory is limited to configurable buffers that can be provided statically (**Bounded RAM/ROM**) [40].

- **FreeRTOS**
  The FreeRTOS is a real-time operating system (RTOS) for microcontrollers and small microprocessors. It is scalable in size, with usable program memory footprint as small as 9KB [27]. The McuLib addes to the common FreeRTOS kernel configuration possibilities and an easy integration to the other components. In this thesis the FreeRTOS will be used to simplify the functionality with the LittleFs and memory management.

- **minIni**
  The minIni is s file parser for initialization data in flash memory. The data will be saved as a key-value pair in different sections. This function enables to store metadata from the firmware update in initialization files which than

  the bootloader will be able to read in an optimized way.

  - **SSD1306**
    The SSD1306 library will give the functionality to draw or write with different fonts to the OLED display.

  - **McuTime**
    The McuTime gives the functionality to read from different RTC times and convert it to human-readable time formats. Further it can be configured that the McuLib will calibrate the internal RTC from an external RTC in a continuous time period. This is often used because the external RTC is more accurate but the reading over the I2C takes more time and can be blocking. Calibrating the internal RTC from time to time gives the system a faster and accurate time management.

  - **McuSHT31**
    McuSHT31 give a generic interface to request the status and the temperature as well the humidity from the sensor.

  - **McuLEDs**
    The McuLED driver builds a generic interface to switch on/off different configured LEDs.

  - **McuButtons**
    The McuButton library give the user the possibility to add different buttons to the application. Further it is possible to add a debouncing function to the button GPIO, which will add a software debouncing to the logic level switch during a push/pull of the buttons.

  - **Shell**
    A useful functionality, that the McuLib provides, is the Shell. The Shell is a command line interface which can uses the UART or RTT interfaces. This allows the user to interact with the node via a terminal. Every here mentioned library has a Shell interface. This helps to debug the node in the developing phase. Fore example it is possible to write and read files on the external flash via the Shell interface through an external terminal.

- **secure Bootloader**
  The secure Bootloader section has the task after every call through a restart, reset or by an application, to check if there is a new patch file ready in the memory. Further it has to check the integrity of the patch file, like the HASH and firmware version it contains. From this information the bootloader will run a merge of the patch and actual firmware (both stored in the external flash) and then boot the new generated image with loading it to the internal flash. In chapter 6.3.3 the functionality will be explained in detail.

- **LoRaMac-node**
  The LoRaMac-node software stack is the official LoRaWAN protocol stack im-

plementation by Semtech [58]. In this software stack, many microcontroller are covered except the LPC55S16. A former master thesis [18] has then integrated the stack for the LPC55S16 microcontroller. With this project as a base and the porting guideline of Semtech [60], Erich Styger ported the stack for the use on the LPC55S16-EVK. All steps are well explained on the McuOnEclipse blog [67].
The basic LoRaMac-node stack from Samtech has the following folder structure:

```
src/
 ├──── apps
 ├──── boards
 ├──── mac
 ├──── peripherals
 ├──── radio
 └──── system
```

- **Apps:**
  In the app folder, the user specific application source files will be placed. There are already some basic examples in this folder, which helps the developer to start.

- **Boards:**
  The boards folder contains board specific implementations of hardware platform drivers

- **Mac:**
  The mac folder contains the source files, which provide the functionality of the official LoRaWAN protocol.

- **Peripherals:**
  The peripherals folder contain the key handling and the drivers for the secure elements.

- **Radio:**
  The radio folder inculdes the driver for the common LoRa radio chips.

- **System:**
  The system folder contains the generic abstraction layer for the different hardware platforms.

- **Application**
  The application includes the basic application functionality the demonstrator node should provide. The basic functionality should be periodically sending the temperature and humidity via the LoRaWAN network to the server and simultaneous showing the sensor date on the display on the node. Next to the basic application, the FUOTA application will be implemented as well. The detailed explanation of the FUOTA application is presented in chapter 6.3.2.

These modules will build the functionality shown in the top-level firmware architecture figure 6.8.

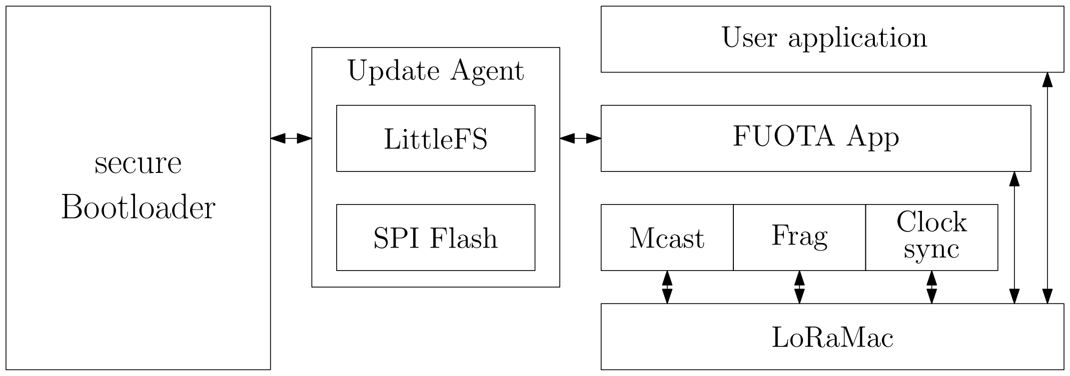


Figure 6.8.: Firmware top-level design.

6.3.2. FUOTA application

The FUOTA application will have to fulfill the following main task.

- **Metadata handling:**
  The application has to handle the receiving metadata presented in chapter 6.1.3 figure 6.3.

- **Time synchronization:**
  For a successful multicast session the node time has to be synchronized with the server. This behavior is presented in section 6.1.8.

- **Multicast session initialization:**
  The multicast session request by the server 6.1.5 has to be handled.

- **Fragmented data block transfer initialization:**
  Then the fragmented data request from the server 6.1.6 has to be checked and answered.

- **Class C switch:**
  After the class C switch timer, defined by the multicast class C session request, is triggered, the node has to switch to the class C.

- **Fragment handling:**
  The sent multicast fragments have to be cached and then checked for completeness. After received all fragments, they have to be stored in a patch file on the external memory. A HASH integrity value will then be sent to the server as a confirmation message, that the session in finised.

- **Bootloader call:**
  After a patch was successfully received, the bootloader have to be called.

This process is shown in the FUOTA application flow diagram in figure 6.9. For a better overview, the flow diagram is divided into two parts. The second part (marked with grey colored background and dotted boundaries) is described in figure 6.10.

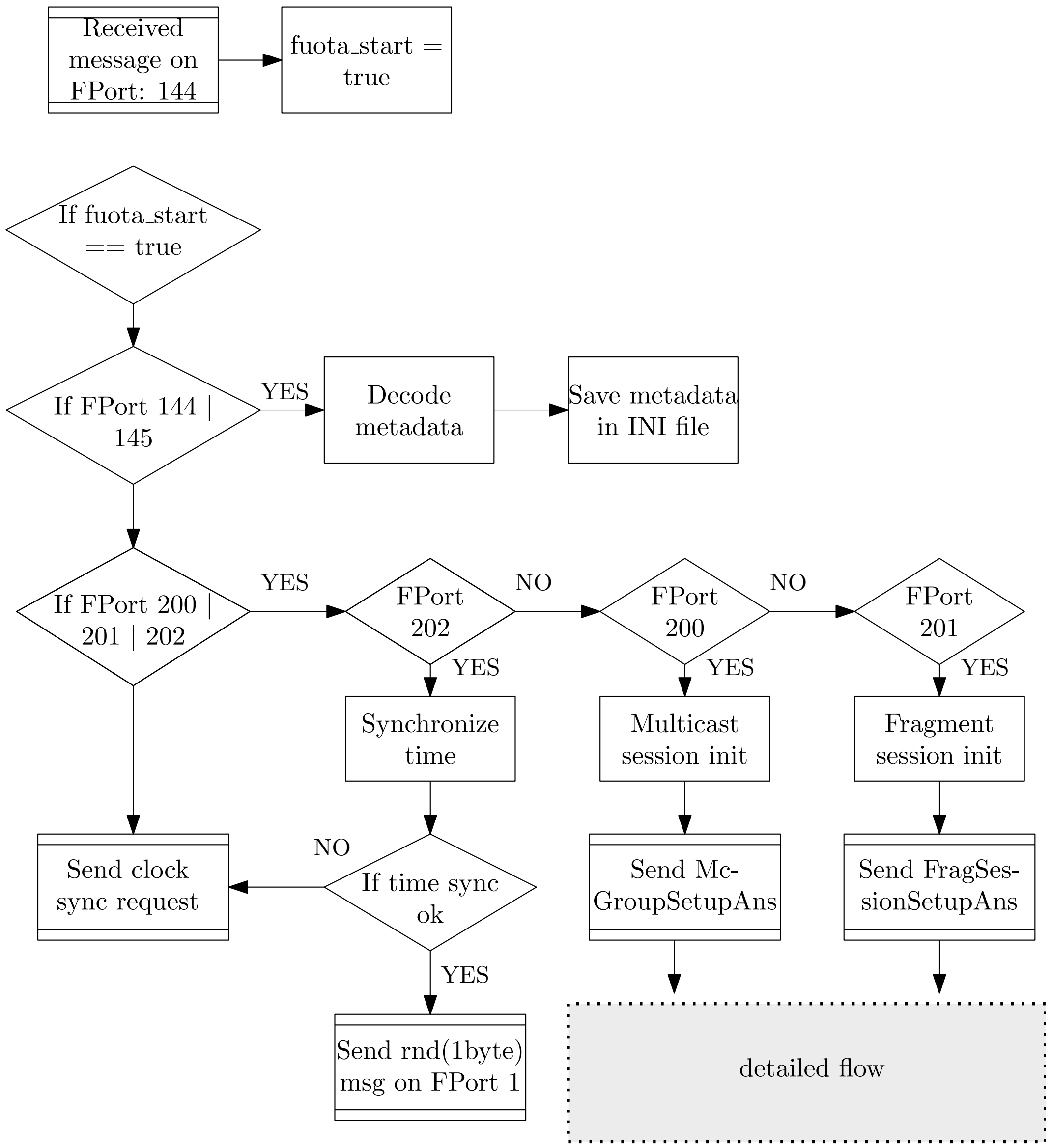


Figure 6.9.: FUOTA application flow diagram.

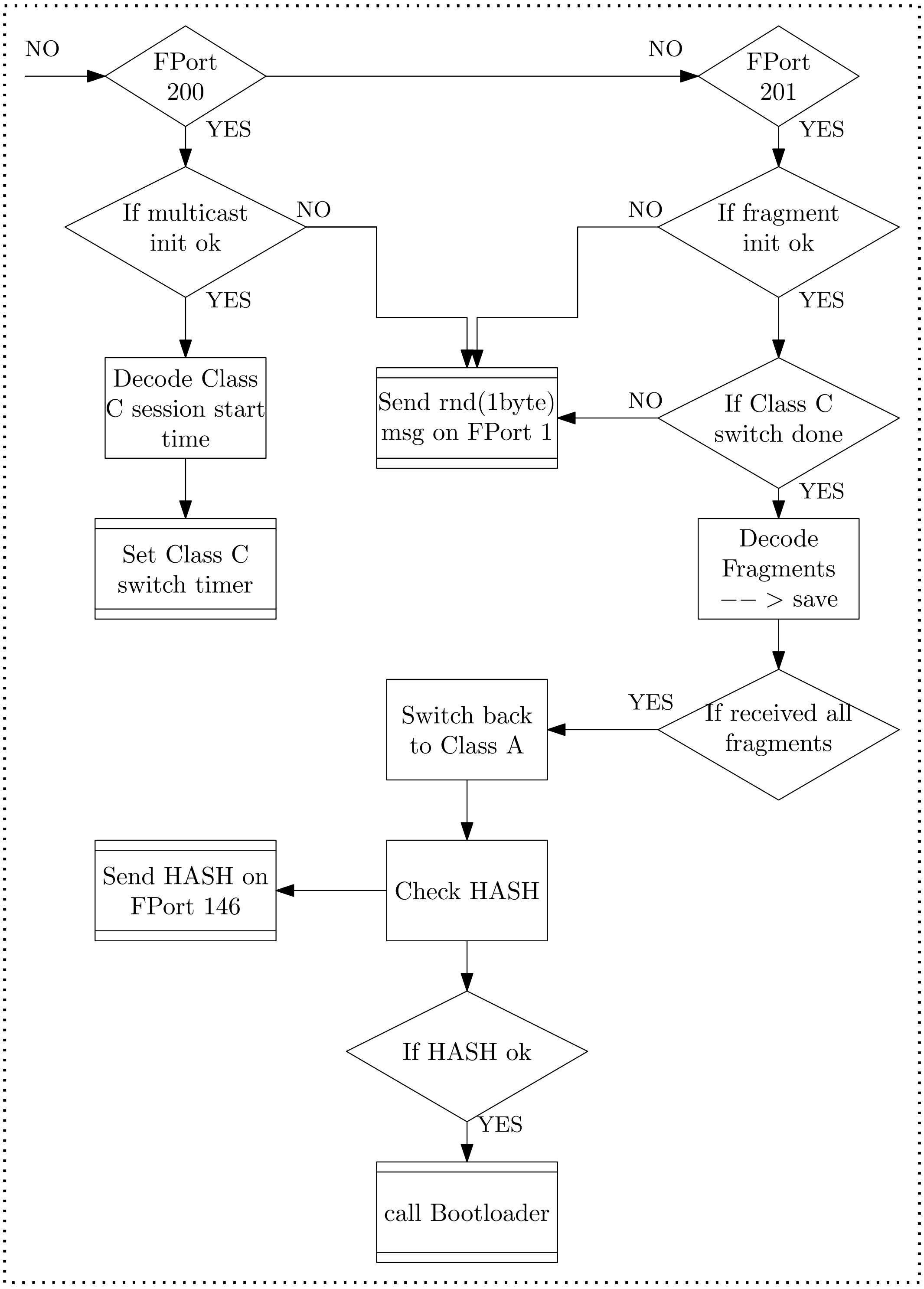


Figure 6.10.: FUOTA application detailed flow diagram.

6.3.2.1. Metadata handling

Figure 6.11 shows the detailed flow of the metadata handling when it is received on the node. To have a better metadata handling a struct (code listing 6.15) was defined storing all the data. The function *FUOTA_FW_Meta_Data()* in code listing 6.16 then parses the received payload and stores the data in the presented struct. The metadata will then be displayed and stored in the external flash in the file called *FWconf.ini*. Code listing 6.17 shows how this *FWconf.ini* is configured after receiving the metadata.

```c
typedef struct
{
    uint32_t patchSize_INI;
    uint8_t fwVersion_INI[4];
    uint32_t patchesCNT_INI;
    uint32_t fragSize_INI;
    uint32_t fragCNT_INI;
    uint8_t patchSHA_INI[32];
    uint8_t newImgSHA_INI[32];
    uint32_t fwCNT_INI;
}fw_conf_INI_t;
```

Listing 6.15: Metadata struct.

```c
uint8_t FUOTA_FW_Meta_Data(LmHandlerAppData_t* appData)
{

    if(appData->BufferSize==43)
    {
        fwConf_data.patchSize_INI = (appData->Buffer[2]<<16) + (appData->
            Buffer[1]<<8) + appData->Buffer[0];
        fwConf_data.fwVersion_INI[0] = appData->Buffer[3];
        fwConf_data.fwVersion_INI[1] = appData->Buffer[4];
        fwConf_data.fwVersion_INI[2] = appData->Buffer[5];
        fwConf_data.fwVersion_INI[3] = appData->Buffer[6];
        fwConf_data.patchesCNT_INI = appData->Buffer[7];
        fwConf_data.fragSize_INI = appData->Buffer[8];
        fwConf_data.fragCNT_INI = (appData->Buffer[10]<<8) + appData->
            Buffer[9];
        for(int i = 0; i<32; i++){
            fwConf_data.patchSHA_INI[i] = appData->Buffer[i+11];
        }

        //TODO: Display FW Data
        update_and_diplayFW_info(&fwConf_data, false, false);

        return ERR_OK;
    }
    else{
            return ERR_FAILED;
    }
}
```

Listing 6.16: Metadata parsed and stored.

```
#[1] marks the section 1
 [1]
patchSHA=0ee2f03bb307031c9c764301184164ae75187f846aaab489062d30da5b9da7f7

 patchSize=0

fwCNT=10

fragCNT=10

patchCNT=10

 fwVersion=01010101

newImgSHA=420c96f0094042b04e9c45fff15df56993c446ed9cce7f53ed7c006c8ac92d0d

 fragSize=0
```
Listing 6.17: Example config of FWconf.ini file.

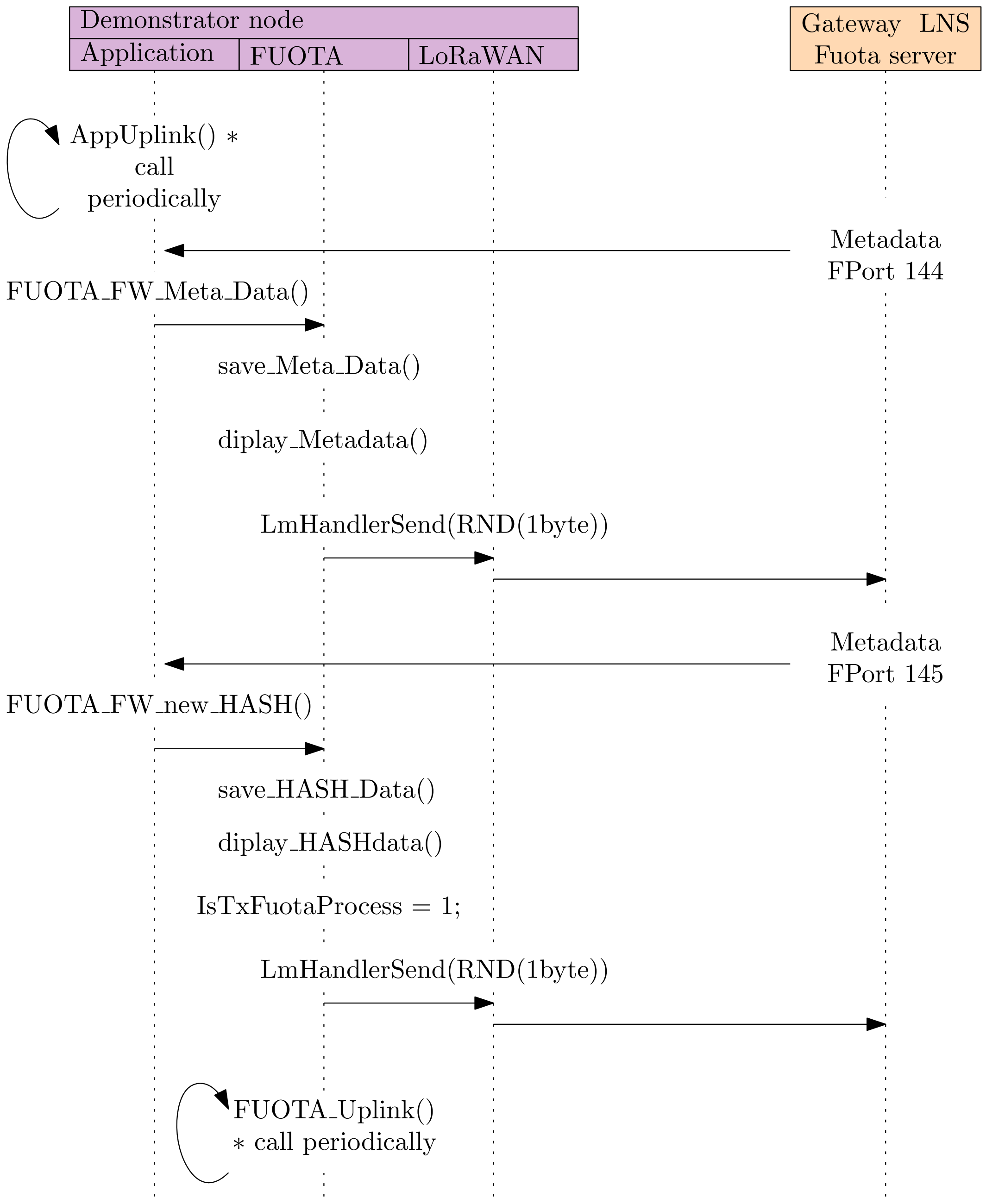


Figure 6.11.: Flow metadata handling.

6.3.2.2. Time synchronization

After the metadata is received and stored, the node switches to the FUOTA application. This means it will run the *Prepare_FuotaTxFrame* function periodically. First, it will send the ClockSyncRequest messages to the server to synchronize with the server. This flow is shown in figure 6.12. The following actions for the complete setup for the FUOTA session will be handled in the file shown in figure 6.13. These files belong to the official LoRaMac-node stack form Semtech.

```c
static void Prepare_FuotaTxFrame(void) {
    LmHandlerErrorStatus_t status = LORAMAC_HANDLER_ERROR;
    if( LmHandlerIsBusy( ) == true ) {return;}
    if( IsMcSessionStarted == false )
    {
        if( IsFileTransferDone == false )
        {
            if( IsClockSynched == false )
            {
                status = LmhpClockSyncAppTimeReq( );
            }
            else
            {
                AppDataBuffer[0] = randr( 0, 255 );
                // Send random packet
                LmHandlerAppData_t appData =
                {
                        .Buffer = AppDataBuffer,
                        .BufferSize = 1,
                        .Port = 1,
                };
                status = LmHandlerSend( &appData,
                    LORAMAC_HANDLER_UNCONFIRMED_MSG);
            }
        }
        else{
            AppDataBuffer = calc_patchHASH();
            // Send HASH
            LmHandlerAppData_t appData =
            {
                    .Buffer = AppDataBuffer,
                    .BufferSize = 32,
                    .Port = 146,
            };
            status = LmHandlerSend( &appData,
                LORAMAC_HANDLER_UNCONFIRMED_MSG);
        }
        if( status == LORAMAC_HANDLER_SUCCESS )
        {}
    }
}
```

Listing 6.18: Periodic FUOTA function.

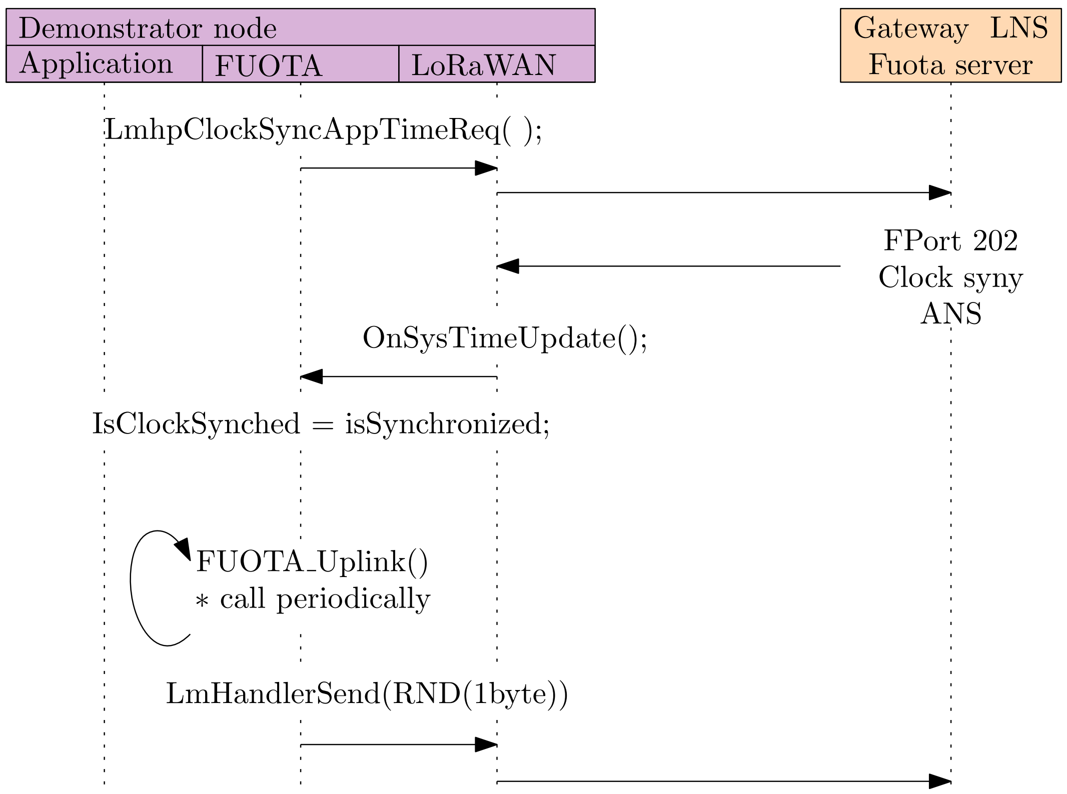


Figure 6.12.: Flow clock synchronization.

```
LoRaMac/
└──────── common/
          └──────── LmHandler/
                    └──────── packages/
                              ├──────── FragDecoder.c|.h
                              ├──────── LmhpClockSync.c|.h
                              ├──────── LmhpCompilance.c|.h
                              ├──────── LmhpFragmentaion.c|.h
                              └──────── LmphRemoteMcasSetup.c|.h
```

Figure 6.13.: FUOTA session important files.

6.3.2.3. Multicast session initialization

In figure 6.14 the flow of the MulticastSessionSetup is shown. In code listing 6.19, the setup of the confirmation payload is shown. If the multicast channel couldn't be initialized, the answer will send error $0x00$ with the group ID back to the server.

```
uint8_t idError = 0x01; // One bit value
if( LoRaMacMcChannelSetup( &channel ) == LORAMAC_STATUS_OK )
{
    idError = 0x00;
}
LmhpRemoteMcastSetupState.DataBuffer[dataBufferIndex++] =
    REMOTE_MCAST_SETUP_MC_GROUP_SETUP_ANS;
LmhpRemoteMcastSetupState.DataBuffer[dataBufferIndex++] = ( idError << 2 )
    | McSessionData[id].McGroupData.IdHeader.Fields.McGroupId;
```

Listing 6.19: Multicast session setup answer.

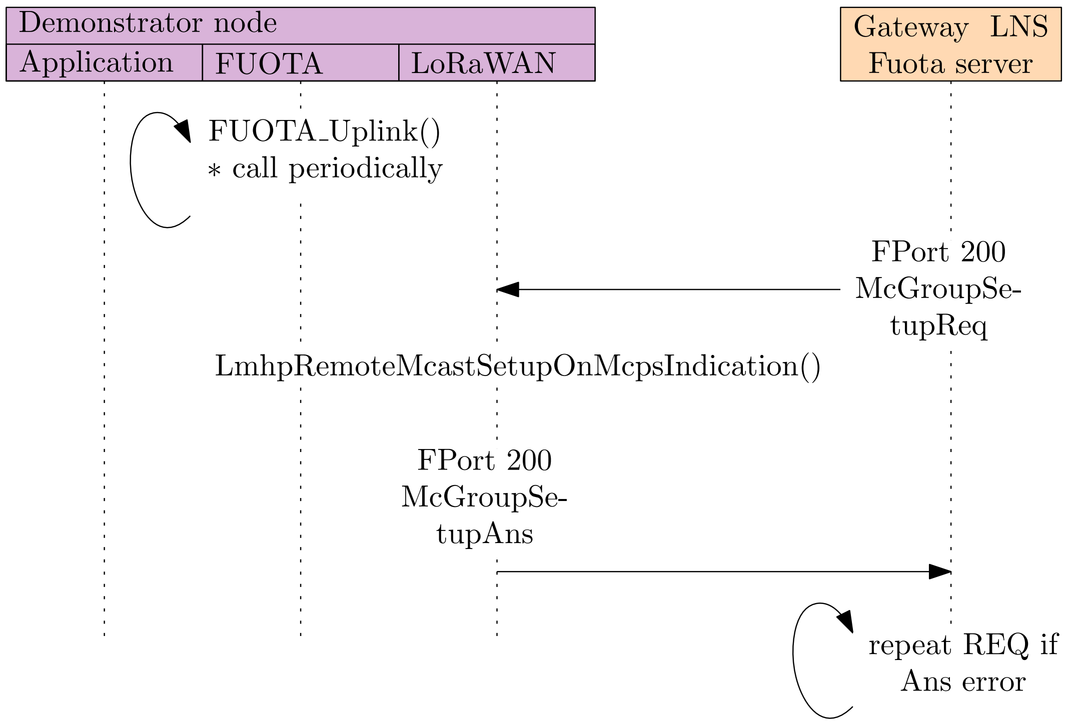


Figure 6.14.: Flow Multicast session setup request

6.3.2.4. Fragmented data block transfer initialization

In figure 6.15, the flow of the FragmentationSessionSetup is shown. In code listing 6.20 the setup answer payload is shown. The *status* variable will contain the error code if the initialization fails.

```

if( fragSessionData.FragGroupData.Control.Fields.FragAlgo > 0 )
{
    status |= 0x01; // Encoding unsupported
}

if( ( fragSessionData.FragGroupData.FragNb > FRAG_MAX_NB ) ||
    ( fragSessionData.FragGroupData.FragSize > FRAG_MAX_SIZE ) ||
    ( ( fragSessionData.FragGroupData.FragNb * fragSessionData.
        FragGroupData.FragSize ) > FragDecoderGetMaxFileSize( ) ) )
{
    status |= 0x02; // Not enough Memory
}

status |= ( fragSessionData.FragGroupData.FragSession.Fields.FragIndex << 6
    ) & 0xC0;
if( fragSessionData.FragGroupData.FragSession.Fields.FragIndex >=
   FRAGMENTATION_MAX_SESSIONS )
{
    status |= 0x04; // FragSession index not supported
}

// Descriptor is not really defined in the specification
// Not clear how to handle this.
// Currently the descriptor is always correct
if( fragSessionData.FragGroupData.Descriptor != 0x01020304 )
{
    //status |= 0x08; // Wrong Descriptor
}

if( ( status & 0x0F ) == 0 )
{
    // The FragSessionSetup is accepted
    fragSessionData.FragGroupData.IsActive = true;
    fragSessionData.FragDecoderPorcessStatus = FRAG_SESSION_ONGOING;
    FragSessionData[fragSessionData.FragGroupData.FragSession.Fields.
        FragIndex] = fragSessionData;

    FragDecoderInit( fragSessionData.FragGroupData.FragNb,
                     fragSessionData.FragGroupData.FragSize,
                     &LmhpFragmentationParams->DecoderCallbacks );

}
LmhpFragmentationState.DataBuffer[dataBufferIndex++] =
   FRAGMENTATION_FRAG_SESSION_SETUP_ANS;
LmhpFragmentationState.DataBuffer[dataBufferIndex++] = status;
isAnswerDelayed = false;
```

Listing 6.20: Fragmentation session setup answer.

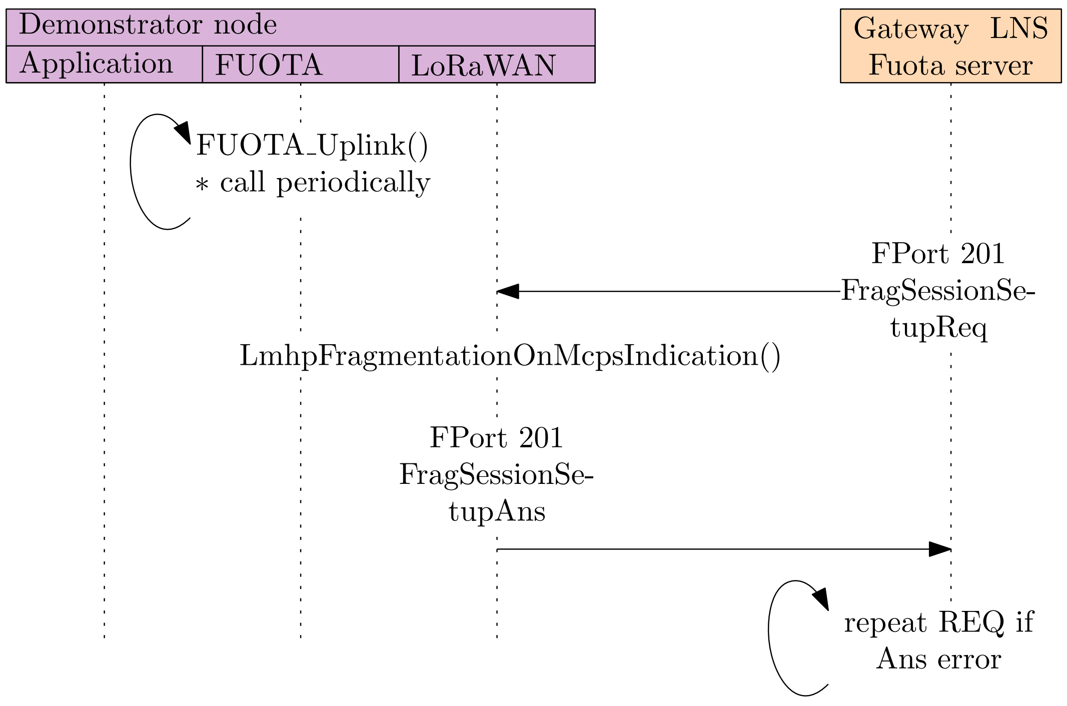


Figure 6.15.: Flow Fragmented session setup request.

6.3.2.5. Class C switch

In figure 6.16 the flow of the multicastClassC request is shown. In code listing 6.21 the setup initialization of the timer depending on the receive time for starting the session is shown. If after this *SessionStartTimer* triggers, the LoRaMacHandler will request with following call: *LmHandlerRequestClass(CLASS_C);* an Class C switch. If there was an error with the time management the node will send an error message with $0x10$ back to the server on FPort 201.

```
SysTime_t curTime = { .Seconds = 0, .SubSeconds = 0 };
curTime = SysTimeGet( );

int32_t timeToSessionStart = McSessionData[id].SessionTime - (curTime.
    Seconds);
if( timeToSessionStart > 0 )
{
    // Start session start timer
    TimerSetValue( &SessionStartTimer, timeToSessionStart * 1000 );
    TimerStart( &SessionStartTimer );

    DBG( "Time2SessionStart: %ld ms\n", timeToSessionStart * 1000 );

    LmhpRemoteMcastSetupState.DataBuffer[dataBufferIndex++] = status;
    LmhpRemoteMcastSetupState.DataBuffer[dataBufferIndex++] = (
        timeToSessionStart >> 0  ) & 0xFF;
    LmhpRemoteMcastSetupState.DataBuffer[dataBufferIndex++] = (
        timeToSessionStart >> 8  ) & 0xFF;
    LmhpRemoteMcastSetupState.DataBuffer[dataBufferIndex++] = (
        timeToSessionStart >> 16 ) & 0xFF;
    break;
}
else
{
    // Session start time before current device time
    status |= 0x10;
}
```

Listing 6.21: Class C switch timer init.

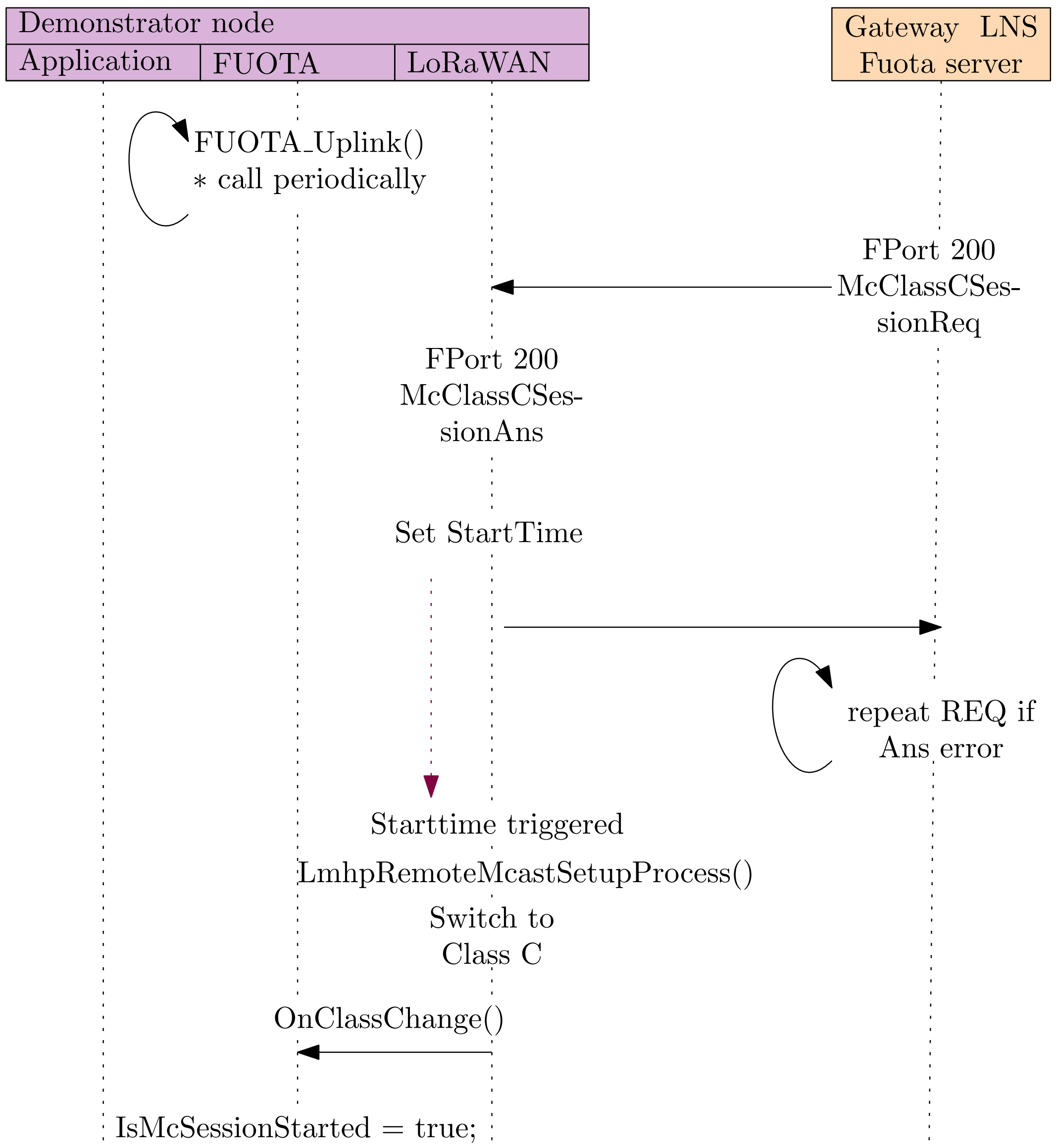


Figure 6.16.: Flow multicast class C switch.

6.3.2.6. Fragment handling

After a class C switch, the node will receive the fragments as a multicast message. In code listing 6.22, the process for receiving fragments is shown. The function call *FragDecoderProcess* will decode the fragments as it is described in chapter 4.2.1.3. This function will also use the callback function *FragDecoderWrite* and *FragDecoderRead* in the FUOTA application and the functions *OnProgress* and *OnDone* will print feedback to the console. All these callback functions are shown in code listing 6.23.

```c

uint8_t fragIndex = 0;
uint16_t fragCounter = 0;
fragCounter = ( mcpsIndication->Buffer[cmdIndex++] << 0 ) & 0x00FF;
fragCounter |= ( mcpsIndication->Buffer[cmdIndex++] << 8 ) & 0xFF00;
fragIndex = ( fragCounter >> 14 ) & 0x03;
fragCounter &= 0x3FFF;

if( FragSessionData[fragIndex].FragDecoderPorcessStatus ==
    FRAG_SESSION_ONGOING )
{
    FragSessionData[fragIndex].FragDecoderPorcessStatus =
        FragDecoderProcess( fragCounter ,
     &mcpsIndication->Buffer[cmdIndex] );
    FragSessionData[fragIndex].FragDecoderStatus = FragDecoderGetStatus( );
    if( LmhpFragmentationParams->OnProgress != NULL )
    {
        LmhpFragmentationParams->OnProgress(
            FragSessionData[fragIndex].FragDecoderStatus.FragNbRx,
            FragSessionData[fragIndex].FragGroupData.FragNb,
            FragSessionData[fragIndex].FragGroupData.FragSize,
            FragSessionData[fragIndex].FragDecoderStatus.FragNbLost );
    }
}
else
{
    if( FragSessionData[fragIndex].FragDecoderPorcessStatus >= 0 )
    {
        // Fragmentation successfully done
        FragSessionData[fragIndex].FragDecoderPorcessStatus =
            FRAG_SESSION_NOT_STARTED;
        if( LmhpFragmentationParams->OnDone != NULL )
        {
            LmhpFragmentationParams->OnDone(
                FragSessionData[fragIndex].FragDecoderPorcessStatus,
                ( FragSessionData[fragIndex].FragGroupData.FragNb *
                FragSessionData[fragIndex].FragGroupData.FragSize ) -
                FragSessionData[fragIndex].FragGroupData.Padding );
        }
    }
}
cmdIndex += FragSessionData[fragIndex].FragGroupData.FragSize;
break;
}
```

Listing 6.22: Fragment handler.

```c

static int8_t FragDecoderWrite( uint32_t addr, uint8_t *data, uint32_t size )
{
    if( size >= UNFRAGMENTED_DATA_SIZE )
    {return -1; // Fail}
    for(uint32_t i = 0; i < size; i++ )
    {UnfragmentedData[addr + i] = data[i];}
    return 0; // Success
}

static int8_t FragDecoderRead( uint32_t addr, uint8_t *data, uint32_t size )
{
    if( size >= UNFRAGMENTED_DATA_SIZE )
    {return -1; // Fail}
    for(uint32_t i = 0; i < size; i++ )
    {data[i] = UnfragmentedData[addr + i];}
    return 0; // Success
}

static void OnFragProgress( uint16_t fragCounter, uint16_t fragNb, uint8_t fragSize, uint16_t fragNbLost )
{
    printf( "\n###### =========== FRAG_DECODER ============ ######\n" );
    printf( "######               PROGRESS                ######\n");
    printf( "###### ==================================== ######\n");
    printf( "RECEIVED    : %5d / %5d Fragments\n", fragCounter, fragNb );
    printf( "%5d / %5d Bytes\n", fragCounter * fragSize, fragNb * fragSize );
    printf( "LOST        :       %7d Fragments\n\n", fragNbLost );
}


static void OnFragDone( int32_t status, uint32_t size )
{
    FileRxCrc = Crc32( UnfragmentedData, size );
    IsFileTransferDone = true;
    printf( "\n###### =========== FRAG_DECODER ============ ######\n" );
    printf( "######               FINISHED                ######\n");
    printf( "###### ==================================== ######\n");
    printf( "STATUS      : %ld\n", status );
    printf( "CRC         : %08lX\n\n", FileRxCrc );
}
```

Listing 6.23: Fragmentation callback functions.

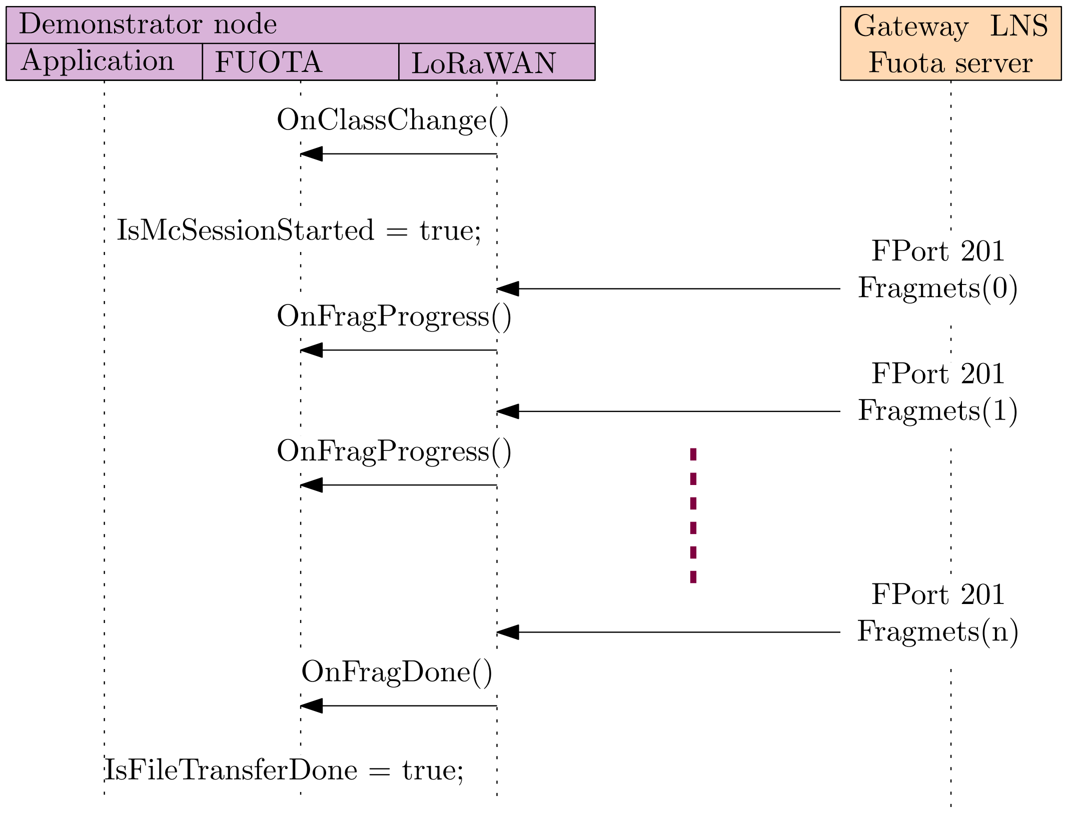


Figure 6.17.: Flow fragmented data transport.

6.3.2.7. Bootloader call

At the end of the FUOTA process, the application will switch back to class A and calculate the HASH value of the decoded fragmented data in the array *UnfragmentedData*[ ]. Then the HASH value will be sent on the FPort 146 back to the server. If the calculate HASH is equal to the stored *patchSHA_INI* (code listing 6.15) in the *FW_config.ini* file, the file will be stored as *patch.bin* to the external flash and the bootloader (chapter 6.3.3) will be called.

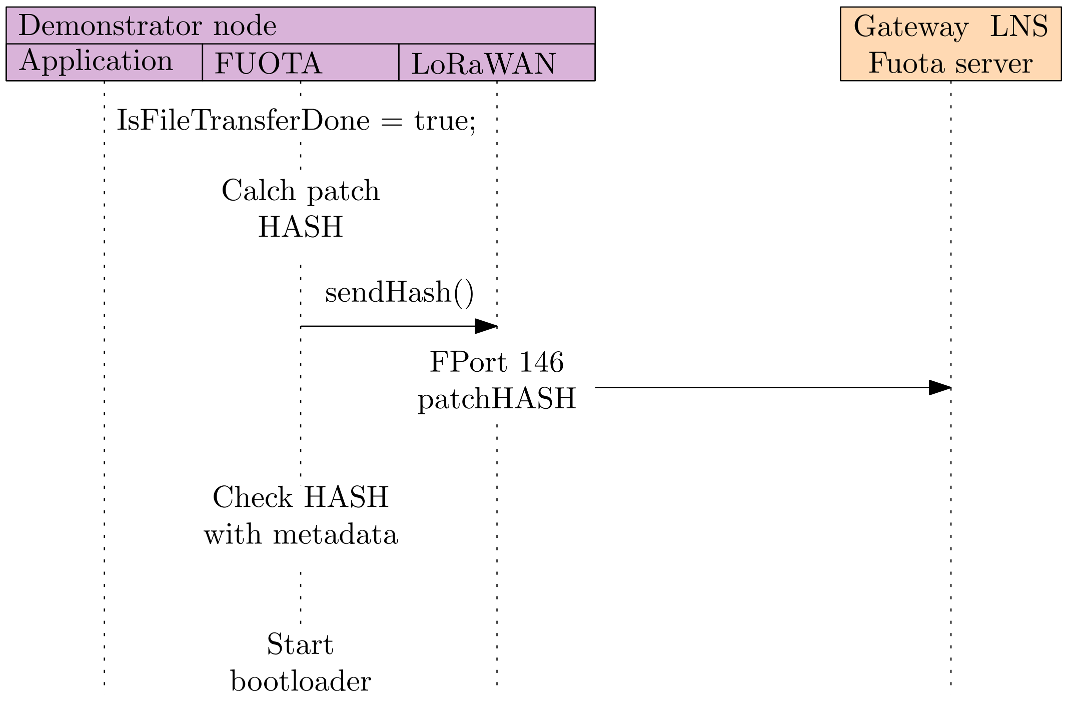


Figure 6.18.: Flow finishing FUOTA session.

6.3.3. Bootloader

In figure 6.19 the flow graph of the bootloader process is shown. The bootloader is designed to use the LittelFS as a file system to access the external W25Q flash, as already described in 6.3.1. For the bootloader itself, a new configuration file, named *Boot_conf.ini* is generated. Next to it, this data from the *Boot_conf.ini* file (example configuration shown in listing 6.25) will be stored in a struct shown in code listing 6.24.

- **bootCNT_INI:**
  This variable indicated how often the bootloader has been trying to boot. It will be set to 0 of the boot of a image was sucessful. If anything with the boot process went wrong, and the bootloader is called more than three times in a row, it will boot the backup image.

- **actFW_INI**[4]**:**
  This variable represents the actual firmware version that is running.

- **newFW_INI**[4]**:**
  This variable will be taken from the FW_conf.ini file. Here, if a new firmware is ready to boot, its firmware version will be stored.

- **patchState_INI:**
  In the patchState_INI information about the patch could be stored. This variable is however not used at the moment.

- **boot_Img_INI:**
  The boot_Img_INI is a flag that indicates if the bootloader is called on purpose.

```c
typedef struct
{
    uint32_t bootCNT_INI;
    uint8_t actFW_INI[4];
    uint8_t newFW_INI[4];
    uint8_t patchState_INI;
    bool boot_Img_INI;
}boot_conf_INI_t;
```

Listing 6.24: Bootdata struct.

```ini
#[1] marks the section 1
[1]
actFW=01020304

newFW=01020304

bootIMG=0

bootCNT=0

patchState=1
```

Listing 6.25: Example config of Boot_conf.ini file.

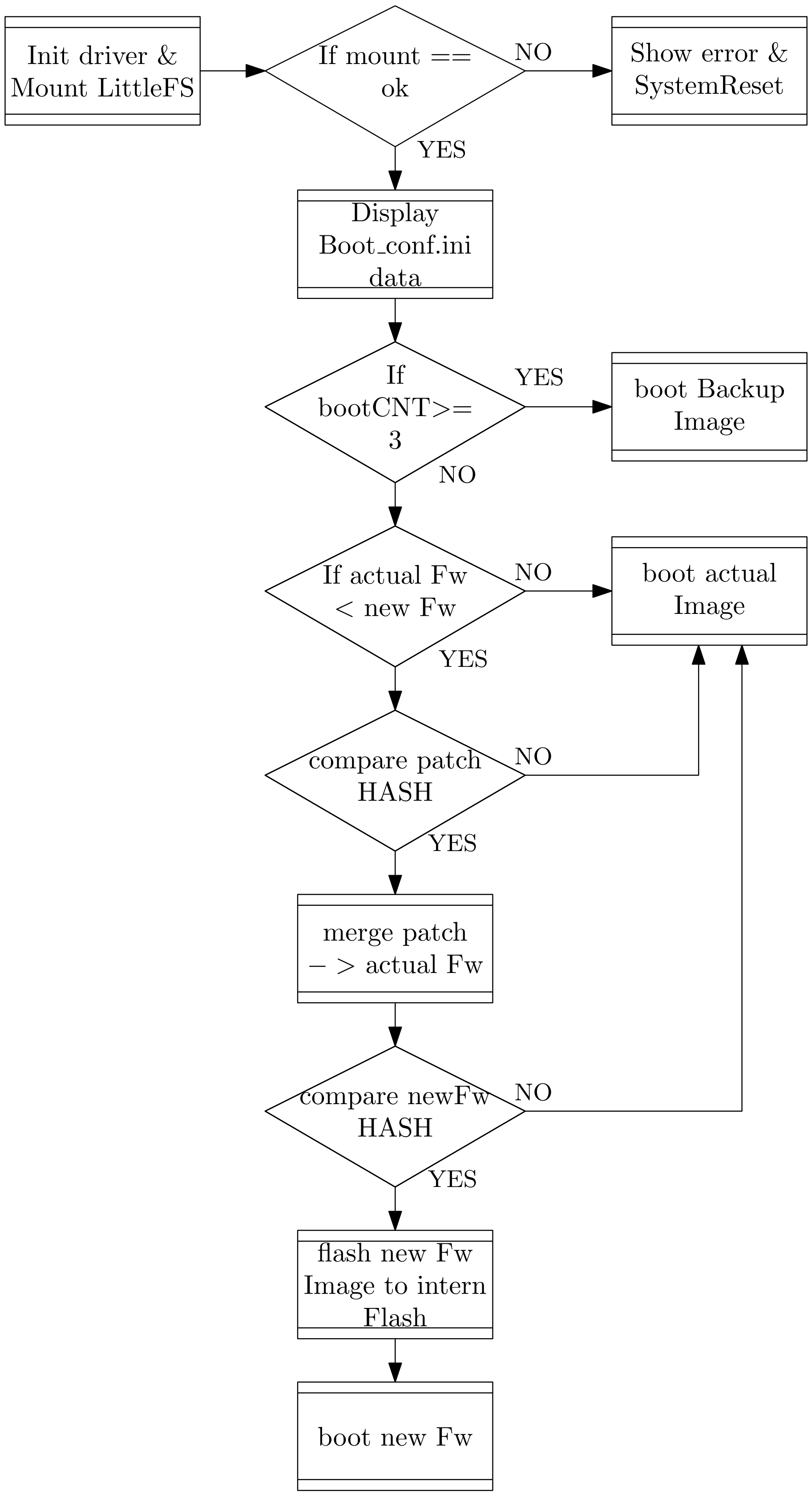


Figure 6.19.: Bootloader top level flow diagram.

The most important task the bootloader has to fulfill, is, that there is always a LoRa application running. To guarantee this, an external file named *Backup.bin* will be placed in the external flash. This backup image has to be able to connect to a LoRaWAN network and receive a new image. This means, the backup image will run the basic tasks explained in chapter 6.3.2. Next to this, the bootloader has to guarantee the integrity of the new image which has to be flashed. This is done with checking the firmware version and mainly with calculating and comparing the HASH of the patch file as well as the HASH of the newly merged and built image with the metadata stored in the *FW_conf.ini* file.

If the integrity is guaranteed, the bootloader should build a new image from the actual running firmware and the patch file received from the FUOTA process. Both binary files are stored in the external flash named *act_FW.bin* and *patch.bin*. To merge this file to a new binary file, named *new_FW.bin*, the *janpatch* [33] library will be used with the LittleFs as its filesystem. In code listing 6.27 the function *patchNewFwFile(unsigned char* sha)* integrads the janpatch library in the bootloader application. First the file system has to open all three files. Then the library needs information about the buffer sizes it can use and what the callback function (*bd_fread, bd_fwrite, bd_fseek, bd_ftell, progress*) are. The library will use these callback functions as posixs file system function calls. The code listing 6.26 shows the wrapper function, that the library can be used with the LittleFs function calls. With these informations initialized and the needed files created, the patch function *janpatch()* is called. After a successful patch the file system will close all files and the HASH of the *newFW.bin* will be calculated.

```c
int bd_fseek(lfs_file_t *file, long int pos, int origin) {
    return (int)lfs_file_seek(lfs, file, (lfs_soff_t)pos, origin);;
}

long int bd_ftell(lfs_file_t *file) {
    return (long int)lfs_file_tell(lfs, file);
}

size_t bd_fread(void *buffer, size_t elements, size_t size, lfs_file_t *
    file) {
    return (size_t)lfs_file_read(lfs, file, buffer, (lfs_size_t)size);
}

size_t bd_fwrite(const void *buffer, size_t elements, size_t size,
    lfs_file_t *file) {
    return lfs_file_write(lfs, file, buffer, (lfs_size_t)size);
}
```

Listing 6.26: Mapping posix calls to LittlFS.

```
uint8_t patchNewFwFile(unsigned char* sha){
int res;
int result;
lfs = McuLFS_GetFileSystem();
JANPATCH_STREAM actFW;
JANPATCH_STREAM patchFile;
JANPATCH_STREAM newFW;
res = lfs_remove(lfs, "newFW.bin");
result = lfs_file_open(lfs, &actFW, "act_FW.bin", LFS_O_RDWR | LFS_O_CREAT|
     LFS_O_APPEND);
if (result < 0) {return ERR_FAILED;}

result = lfs_file_open(lfs, &patchFile, "patch.bin", LFS_O_RDWR |
    LFS_O_CREAT| LFS_O_APPEND);
if (result < 0) {return ERR_FAILED;}

result = lfs_file_seek(lfs, &actFW, 0, LFS_SEEK_END);
if (result < 0) {
    (void)lfs_file_close(lfs, &actFW);
    return ERR_FAILED;}

result = lfs_file_seek(lfs, &patchFile, 0, LFS_SEEK_END);
if (result < 0) {
    (void)lfs_file_close(lfs, &patchFile);
    return ERR_FAILED;}

result = lfs_file_open(lfs, &newFW, "newFW.bin",LFS_O_RDWR | LFS_O_CREAT|
    LFS_O_APPEND);
if (result < 0) {return ERR_FAILED;}

result = lfs_file_seek(lfs, &patchFile, 0, LFS_SEEK_END);
if (result < 0) {
    (void)lfs_file_close(lfs, &patchFile);
    return ERR_FAILED;}

janpatch_ctx ctx = {
    { (unsigned char*)malloc(256), 256 }, // source buffer
    { (unsigned char*)malloc(256), 256 }, // diff buffer
    { (unsigned char*)malloc(256), 256 }, // target buffer
    &bd_fread, &bd_fwrite, &bd_fseek, &bd_ftell, &progress
};

int jpr = janpatch(ctx, &actFW, &patchFile, &newFW);
if (jpr != 0) {return ERR_FAILED;}

lfs_file_close(lfs, &actFW);
lfs_file_close(lfs, &patchFile);
lfs_file_close(lfs, &newFW);

if(W25_read_File_for_SHA("newFW.bin", 512, sha)!=ERR_OK)
{return ERR_FAILED;}

return ERR_OK;
}
```

Listing 6.27: janpatch library integration.

After a successful passed integrity check, the bootloader has to flash the *newFW.bin* to the LoRaWAN application section in the internal flash. This is done with the function call *W25_read_and_flash* shown in code listing 6.28. The *newFW.bin* will be flashed in 512byte blocks to the internal flash. Important to mention is that before a page in the internal flash can be programmed, the address space has to be erased.

```c
uint8_t W25_read_and_flash(const char* filename, bool readFromBeginning,
    size_t nofBytes){
    uint8_t result;
    static int32_t filePos;
    size_t fileSize;
    uint8_t buf[512];
    if( nofBytes > 512) {nofBytes = 512;}

    lfs = McuLFS_GetFileSystem();
    lfs_file_t file;
    lfs_file_open(lfs, &file, filename , LFS_O_RDWR | LFS_O_CREAT|
        LFS_O_APPEND);

    if(readFromBeginning) {
        lfs_file_rewind(lfs, &file);
        filePos = 0;
    } else {lfs_file_seek(lfs, &file, filePos,LFS_SEEK_SET);}

    fileSize = lfs_file_size(lfs, &file);
    filePos = lfs_file_tell(lfs, &file);
    fileSize = fileSize - filePos;
    while(fileSize >0){
        if(fileSize > nofBytes)  {
            if (lfs_file_read(lfs, &file, buf, nofBytes) < 0) {return
                ERR_FAILED;}
        } else {
            if (lfs_file_read(lfs, &file, buf, fileSize) < 0) {return
                ERR_FAILED;}
            if(!(memory_erase((STARTADDRESS_IMAGE+filePos), 512))){
                memory_write((STARTADDRESS_IMAGE+filePos), buf, 512);
            }else{return ERR_FAILED;}

            result = lfs_file_close(lfs, &file);
            if (result < 0) {return ERR_FAILED;}
            return ERR_OK; //EOF
        }
        if(!(memory_erase((STARTADDRESS_IMAGE+filePos), 512))){
            memory_write((STARTADDRESS_IMAGE+filePos), buf, 512);
        }
        else{return ERR_FAILED;}

        filePos = filePos + nofBytes;
        bzero(buf, nofBytes);
        filePos = lfs_file_tell(lfs, &file);
        fileSize = fileSize - nofBytes;
    }
}
```

Listing 6.28: Flashing files to internal flash.

If the internal flashing is done, the bootloader is ready to load the new application. This is done with the function call *load_new_Application()* shown in 6.29. As argument (*addr*) the start address has to be passed. This is the address the LoRa application is flashed to. In the next chapter 6.3.4 an overview about the whole system and the demonstrator node memory map is presented.

```
void load_new_Application(uint32_t addr)
{
    #if McuLib_CONFIG_SDK_USE_FREERTOS
    portDISABLE_ALL_INTERRUPTS(); /* disable all interrupts, they get
        enabled in vStartScheduler() */
    vPortStopTickTimer(); /* tick timer shall not run until the RTOS
        scheduler is started */
    vTaskSuspendAll();
    #endif

    uint32_t *vectorTable = (uint32_t*)addr;
    uint32_t sp = vectorTable[0];
    uint32_t pc = vectorTable[1];

    typedef void(*app_entry_t)(void);
    uint32_t ss_stackPointer = 0;
    uint32_t ss_applicationEntry = 0;
    app_entry_t ss_application = 0;

    ss_stackPointer = sp;
    ss_applicationEntry = pc;
    ss_application = (app_entry_t)ss_applicationEntry;

    // Change MSP and PSP
    __set_MSP(ss_stackPointer);
    __set_PSP(ss_stackPointer);
    SCB->VTOR = addr;

    // Jump to application
    ss_application();
}
```

Listing 6.29: Bootloader boot application function.

6.3.4. System overview

The presented implementation (chapter 6) of the server software and microcontroller firmware builds the full stack of the FUOTA infrastructure used in this thesis. Figure 6.20 gives a concluding overview of the system design developed in this work. On the node, a dual application architecture with the bootloader and the LoRaWAN application was built. For the FUOTA image handling, an external flash is included in the architecture. On this flash, six files named backup.bin, newFW.bin, actFW.bin, patch.bin, Boot_conf.ini and FW_conf.ini will allow a secure bootloader process. The memory map of the demonstrator node is presented in figure 6.21.

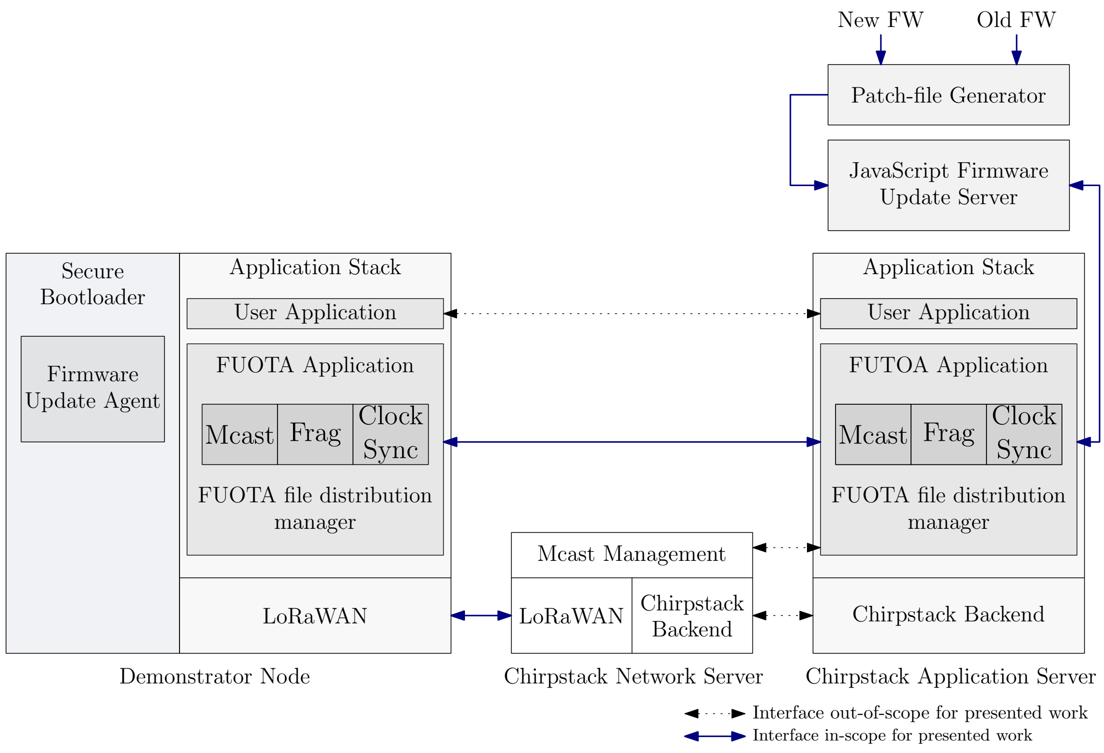


Figure 6.20.: System design of presented work.

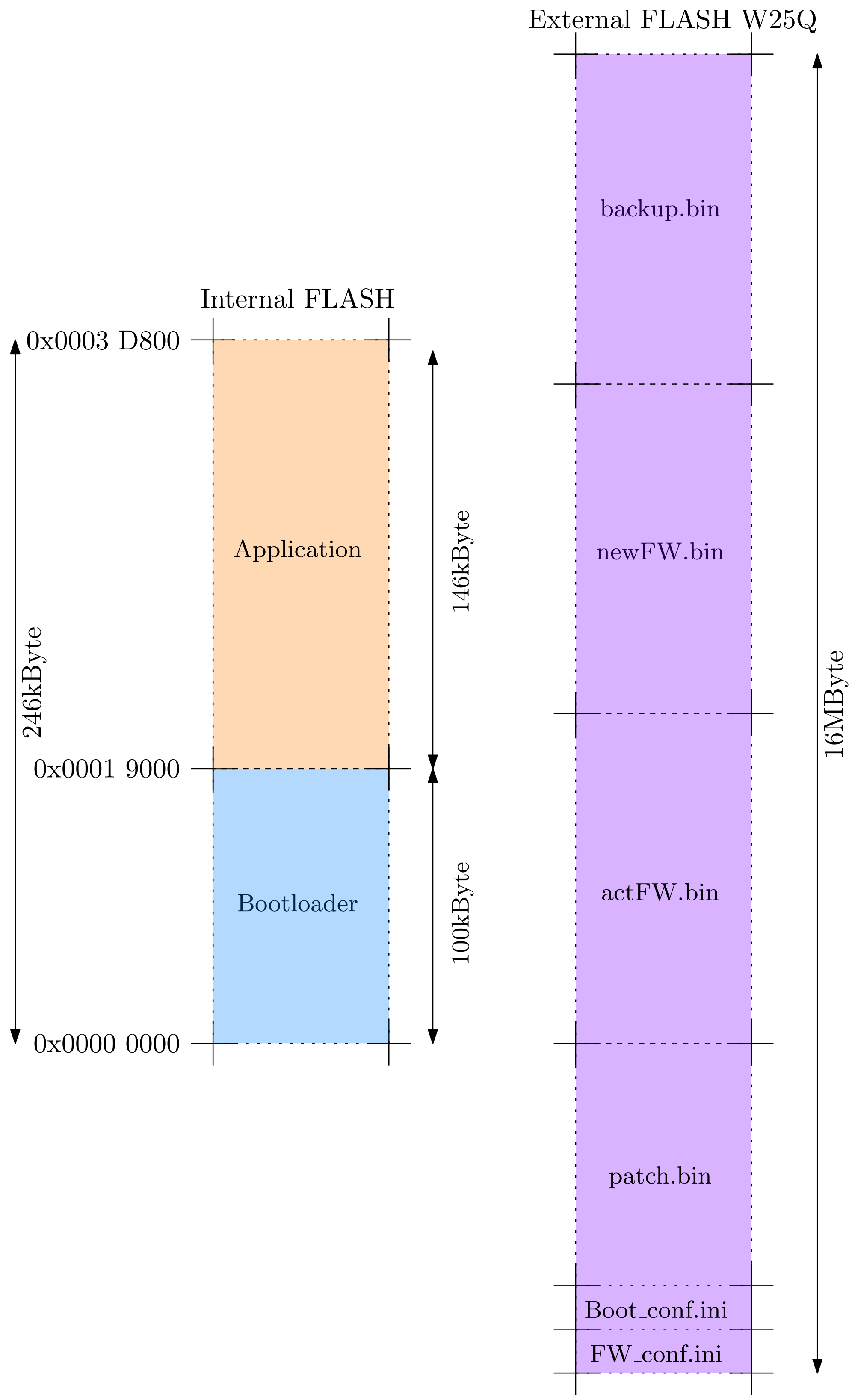


Figure 6.21.: Actual memory map of the demonstrator node.

7. System tests

In this chapter, the developed infrastructure that was presented in the previous chapters will be tested. For the tests, the system will be divides into units which first will be tested separately to then be combined for system-wide integration tests.

7.1. Bootloader

For the bootloader, first the external memory will be tested and then the merge process of the patch file to an actual image on the basis of a *hello_world* example application. The new created firmware image has to be flashed to the application sector in the internal flash and needs to be booted.

7.1.1. External flash

To test the external flash, a benchmark for writing, reading and copying files on the flash is run. To interact with the flash from extern via the microcontroller a basic shell application [66] on the demonstrator node is running, which is able to communicate via the debug probe and a SEGGER RTT interface with the host machine running a serial terminal (figure 7.1). With the command shown in code listing 7.1 the benchmark can be run. In figure 7.2 the results of the benchmark are plotted. At the time of testing the SPI clock frequency was set to 12MHz. The W25Q flash chip can be run up to 104MHz. This means that, if faster data rates are required, the SPI clock has to be attached to a higher clock. The LPC55S16 has the possibility to work up to 150MHz.

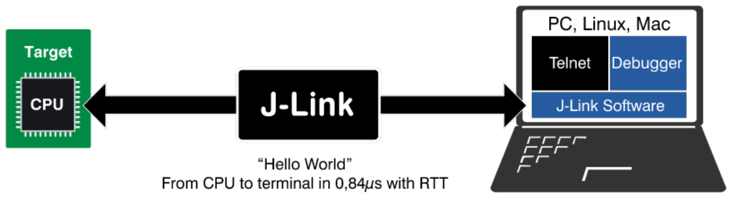


Figure 7.1.: Segger RTT communication [57].

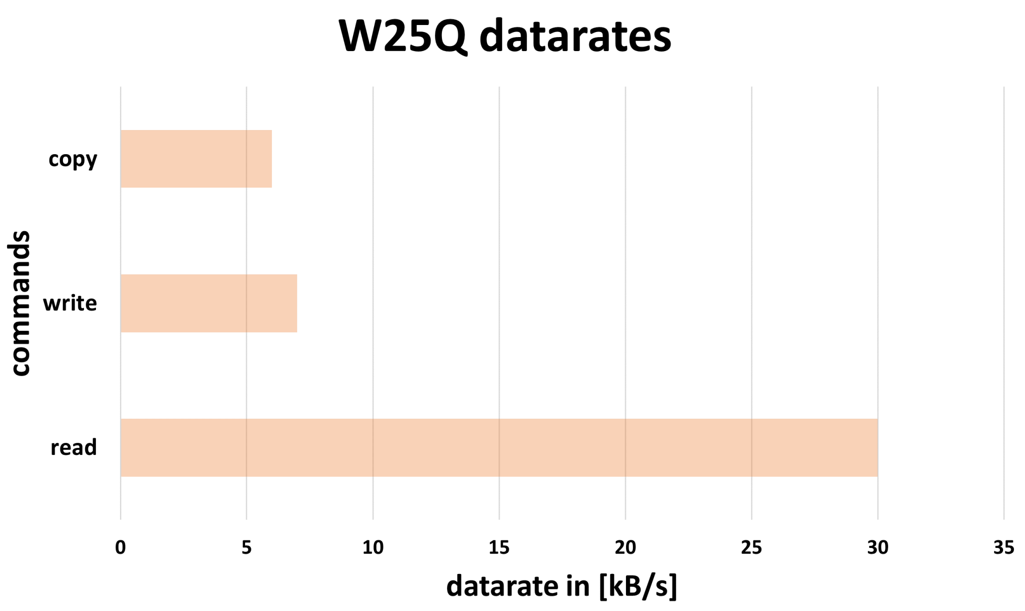


Figure 7.2.: W25Q benchmark results.

```
>McuLittleFS format
    Formatting ... done.

>McuLittleFS mount
    Mounting ... done.

>McuLittleFS status
    McuLittleFS  : McuLittleFS status
    version     : 0x00020005
    mounted     : yes
    space       : 67108864 bytes
    read_size   : 256
    prog_size   : 256
    block_size  : 4096
    block_count: 16384
    lookahead   : 256

>McuLittleFS benchmark
    Benchmark: write/copy/read a 100kB file:
    Delete existing benchmark files...
    ERROR: Failed removing file.
    ERROR: Failed removing file.
    Create benchmark file...
    13000 ms for writing (7 kB/s)
    Read 100kB benchmark file...
    5000 ms for reading (20 kB/s)
    Copy 100kB file...
    15000 ms for copy (6 kB/s)
    done!
```

Listing 7.1: Shell commands W25Q benchmark.

7.1.2. Merging patch & actual image

To test the algorithm [33] for merging the patch file to an existing firmware image, without having a LoRaWAN communication channel, the data has to be transmitted from a host machine over a UART interface (figure 7.3). To send the files from the host directly into the flash, a Python script (code listing 7.2) is sending the file in 32byte packages to the shell interface on the demonstrator node. With this setup, a file named *actFW.bin*, which is a basic hello_wolrd example (code listing 7.3) is stored to the external flash on the demonstrator node. On the host, a patch file with the JDiff library [34] was generated. The patch file is the calculated difference between the actFW.bin (code listing 7.3, 9180 bytes) and the newFW.bin (code listing 7.3 with the uncommented lines included, this means the firmware flashes a LED every time a character is arrived at the UART interface) with the size of 9320 bytes. The generated patch.bin file has the size of 1501 bytes.
With the patch.bin and actFW.bin stored in the external flash, the bootloader is ready to merge the patch.bin and the actFW.bin to the newFW.bin and stores the newFW.bin in the external Flash. The newFW.bin file was then print via the SHELL to the RTT terminal in HEX format. This output was then compared with the newFW.bin file on the host machine, with the result of zero differences. This means the merge process on the demonstrator node for the *hello_world* firmware example worked.

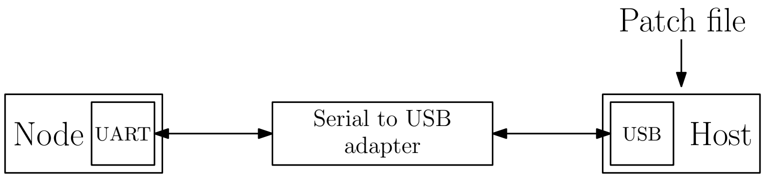


Figure 7.3.: Serial to USB interface.

```python
parser = argparse.ArgumentParser()
parser.add_argument("FILE", help="the name of the file that you wish to
    dump", type=str)
parser.add_argument("FILE_NAME", help="name which the file will be saved on
     thedevice", type=str)
parser.add_argument("-b", "--binary", help="display bytes in binary format
    instead of hexadecimal", action="store_true")
args = parser.parse_args()
myRawData = []
serialPort = serial.Serial(port = "COM31", baudrate=115200, bytesize=8,
    timeout=2, stopbits=serial.STOPBITS_ONE)

mcLib_bincat_string = "McuLittleFS bincat " + args.FILE_NAME + " "

waitTime = 1.4
n = 0
y = 0
blocksize = 32
binary16_string_2 = ""
msg_16byte = ""

file = open(args.FILE, "rb")
binary_block = file.read(blocksize)

while binary_block:
    n += 1
    y += 1
    str = ""
    hex_str = ""
    for ch in binary_block:

        tmp = hex(ch)[2:].zfill(2) + " "
        str += f'{ch}' + " "
        hex_str += tmp
        binary16_string = str

    print(hex_str)
    print("\r\n")
    msg = mcLib_bincat_string + binary16_string + "\r\n"
    time.sleep(waitTime/2)
    serialPort.write(msg.encode('utf-8'))
    serialPort.flush()
    binary_block = file.read(blocksize)
    time.sleep(waitTime/2)

print("\n\nFINISED\n\n\n\n")
serialPort.close()
```

Listing 7.2: Serial file transfer script.

```c
int main(void)
{
    char ch;
    /* Init output LED GPIO. */
    //GPIO_PortInit(GPIO, BOARD_LED_PORT);

    /* Init board hardware. */
    POWER_SetBodVbatLevel(kPOWER_BodVbatLevel1650mv,
        kPOWER_BodHystLevel50mv, false);
    CLOCK_AttachClk(BOARD_DEBUG_UART_CLK_ATTACH);
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole();

    PRINTF("hello world.\r\n");
    //PRINTF("This is the new Version.\r\n");
    //PRINTF("Blinks LED after UART input.\r\n");
    while (1)
    {
        ch = GETCHAR();
        PUTCHAR(ch);
        //GPIO_PortToggle(GPIO, BOARD_LED_PORT, 1u << BOARD_LED_PIN);
    }
}
```

Listing 7.3: Basic hello_world firmware example.

7.2. LoRaWAN

To test the LoRaWAN communication, a basic periodic uplink application will be installed on the node. On server side the node has to be added to the application server with the same keys, as on the node in the se-identity.h file. On power up, the demonstrator node application will send a join request to the server and if it receives a join accept message, the application will send actual temperature and humidity values, each as a 4byte value periodically up to the server.
In code listing 7.4 the log of the demonstrator node is shown. In figure 7.7 the log on the application server is presented. For this test the demonstrator node and the LoRaWAN gateway were placed 2m apart from each other with no obstacle in between. Further analysis of the provided meta information about the received LoRa messages by the server has shown, that the signal strength, represented in the RSSI value, is very weak. The Received Signal Strength Indication (RSSI) is the received signal power in milliwatts and is measured in dBm.
To have a better understanding of the problem, a LPC55S16-EVK will be used with a SX1261MB2BAS shield (5.2) as a second node and the demonstrator node will be compared to this shield, which is using a SX1261 LoRa chip from Semtech. The test setup is shown in figure 7.4. Figure 7.5 shows the signal strength of the two nodes for the exact same setup. It can be seen that the SX1261MB2BAS shield has a significantly better signal strength. Further in 7.6 the SNR (signal to noise ration) for the same measurement is shown. From this it can be concluded, that the RFM96 module with the SX1276 LoRa chip assembled, has a problem with its power setting. The reason for this could be a wrong firmware configuration or a poor hardware layout and antenna setting. Due to time constraints, and the fact that the demonstrator node works for the basic setup, the problem with the signal strength must be further investigated after this work is completed.

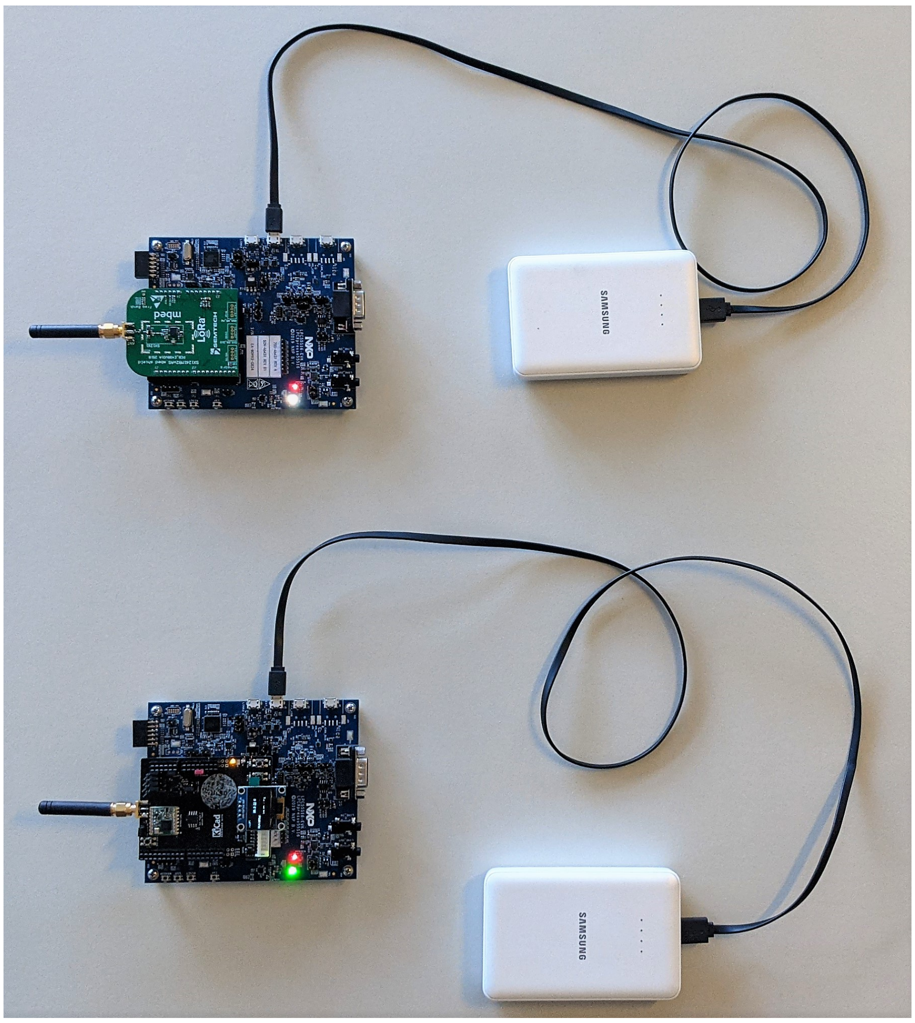


Figure 7.4.: RSSI test setup.

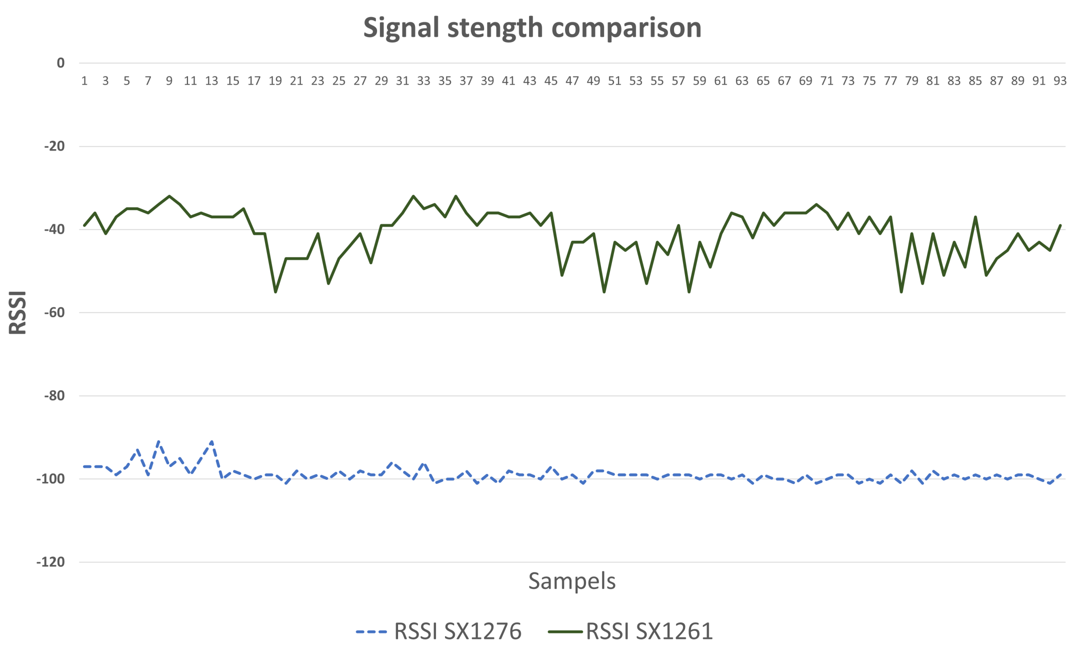


Figure 7.5.: RSSI comparison.

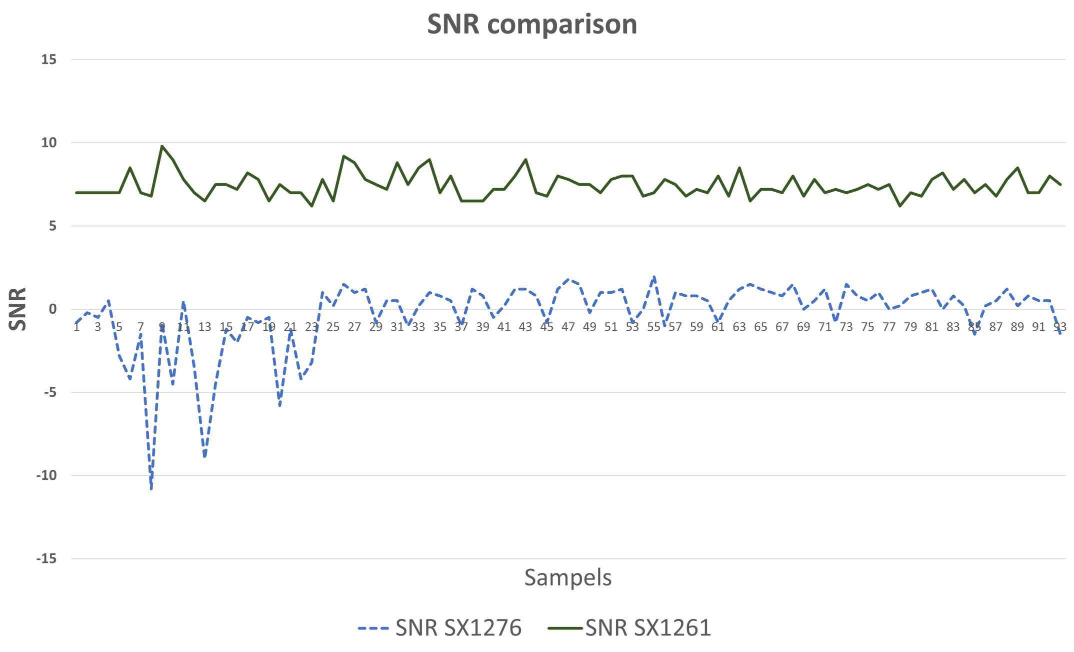
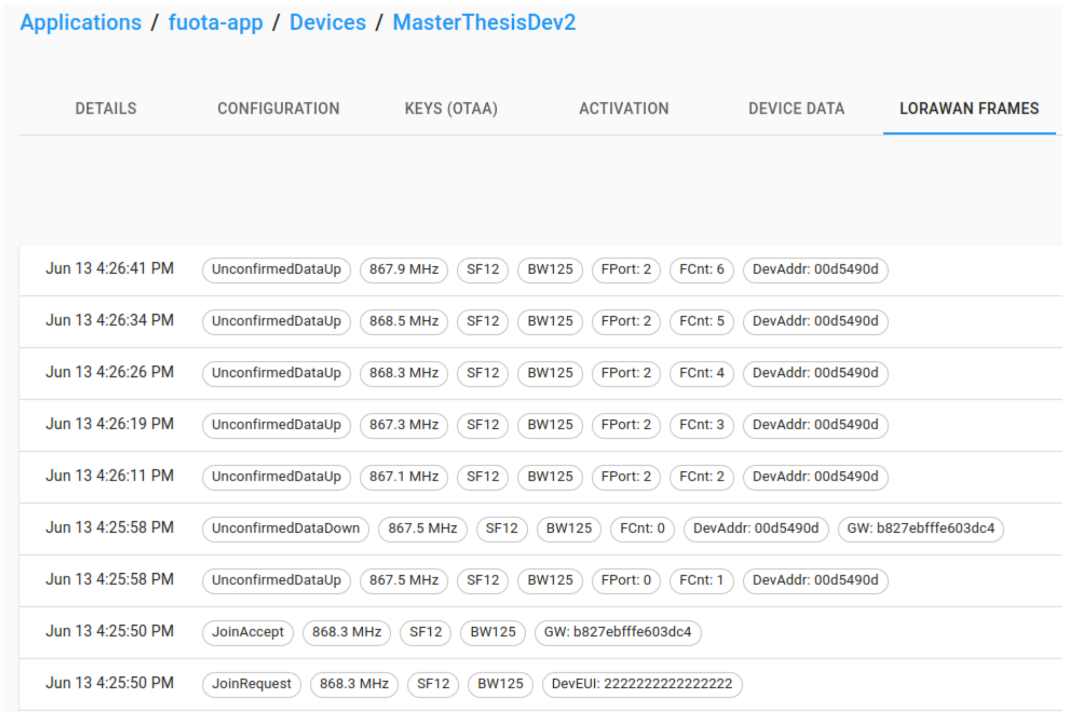


Figure 7.6.: SNR comparison.



```
Jun 13 4:26:41 PM   UnconfirmedDataUp    867.9 MHz  SF12  BW125  FPort: 2  FCnt: 6  DevAddr: 00d5490d
Jun 13 4:26:34 PM   UnconfirmedDataUp    868.5 MHz  SF12  BW125  FPort: 2  FCnt: 5  DevAddr: 00d5490d
Jun 13 4:26:26 PM   UnconfirmedDataUp    868.3 MHz  SF12  BW125  FPort: 2  FCnt: 4  DevAddr: 00d5490d
Jun 13 4:26:19 PM   UnconfirmedDataUp    867.3 MHz  SF12  BW125  FPort: 2  FCnt: 3  DevAddr: 00d5490d
Jun 13 4:26:11 PM   UnconfirmedDataUp    867.1 MHz  SF12  BW125  FPort: 2  FCnt: 2  DevAddr: 00d5490d
Jun 13 4:25:58 PM   UnconfirmedDataDown  867.5 MHz  SF12  BW125  FCnt: 0  DevAddr: 00d5490d  GW: b827ebfffe603dc4
Jun 13 4:25:58 PM   UnconfirmedDataUp    867.5 MHz  SF12  BW125  FPort: 0  FCnt: 1  DevAddr: 00d5490d
Jun 13 4:25:50 PM   JoinAccept  868.3 MHz  SF12  BW125  GW: b827ebfffe603dc4
Jun 13 4:25:50 PM   JoinRequest  868.3 MHz  SF12  BW125  DevEUI: 2222222222222222
```

Figure 7.7.: Basic Chirpstack LoRaWAN message log.

```
###### ======================================== ######

Application name    : periodic-uplink-lpp
Application version: 1.2.0
GitHub base version: 5.0.0

###### ======================================== ######

######   Board UUID: {0x16,0xCD,0xBA,0xFB,0xEC,0x16,0xDF,0x5B,0xA4,0xA2,0
    xFA,0xF1,0xD5,0xB4,0x59,0xEF}   ######

13.06.2022 16:24:47,00 INFO  LoRaWAN.c:937: Mounting litteFS volume.
Mounting ... done.
13.06.2022 16:24:47,60 INFO  application.c:68: App Task started.
13.06.2022 16:24:49,10 INFO  LoRaWAN.c:1050: start joining ...
13.06.2022 16:25:00,90 INFO  LmHandlerMsgDisplay.c:328: CLASS: A, TX PORT 0

###### =========== MCPS-Indication =========== ######
STATUS        : OK

###### =====  DOWNLINK FRAME          0    ===== ######
RX WINDOW     : 1
RX PORT       : 0

DATA RATE     : DR_0
RX RSSI       : -41
RX SNR        : 8

13.06.2022 16:25:03,20 INFO  LoRaWAN.c:1061: ... connected
13.06.2022 16:25:03,20 INFO  LoRaWAN.c:1085: request to tx data
13.06.2022 16:25:14,20 INFO  LmHandlerMsgDisplay.c:328: CLASS: A, TX PORT 2
00 00 0B 97 00 00 10 17
13.06.2022 16:25:22,70 INFO  LmHandlerMsgDisplay.c:328: CLASS: A, TX PORT 2
00 00 0B 9A 00 00 10 0E
13.06.2022 16:25:29,80 INFO  LmHandlerMsgDisplay.c:328: CLASS: A, TX PORT 2
00 00 0B 9A 00 00 10 0E
13.06.2022 16:25:36,90 INFO  LmHandlerMsgDisplay.c:328: CLASS: A, TX PORT 2
00 00 0B 9E 00 00 10 05
13.06.2022 16:25:44,00 INFO  LmHandlerMsgDisplay.c:328: CLASS: A, TX PORT 2
00 00 0B 9D 00 00 0F FA
13.06.2022 16:25:51,10 INFO  LmHandlerMsgDisplay.c:328: CLASS: A, TX PORT 2
00 00 0B 9F 00 00 0F FF
```

Listing 7.4: Demonstrator node log information for basic LoRaWAN application.

7.3. FUOTA

For the FUOTA application on both, the server side and the demonstrator node side following tests were run.

7.3.1. FUOTA server and demonstrator tests

To test the FUOTA server and the demonstrator node application a test setup as described in the following section was developed. The presented logs are shortened, because to show them full they would take to many pages into account. The original logs and test data can be found in the appendix.
To test the FUOTA session setup, the following files where generated:

- V1.bin
  The firmware binary V1.bin is the demonstrator node firmware running at address 0x19000 on the node and which is sending periodically just the temperature value to the LoRaWAN network.

- V2.bin
  The firmware binary V2.bin is the demonstrator node firmware, which is sending periodically both the temperature and humidity value to the LoRaWAN network. This binary is built but not flashed to the demonstrator node. The only sourcecode change is shown in code listing 7.5.

```
    readSensorSHT31(&tmp, &hum);
    tmpInt = (uint32_t)(tmp*100);
    /*
    *#################################################
    *This is the source code change from version 1  to verison 2
    *#################################################
    */
    humInt = (uint32_t)(hum*100);
    //humInt = (uint32_t)(0);

    uint8_t appdata[8];
    appdata[3] = tmpInt & 0x000000FF;
    appdata[2] = ( tmpInt >> 8 ) & 0x000000FF;
    appdata[1] = ( tmpInt >> 16 ) & 0x000000FF;
    appdata[0] = ( tmpInt >> 24 ) & 0x000000FF;
    appdata[7] = humInt & 0x000000FF;
    appdata[6] = ( humInt >> 8 ) & 0x000000FF;
    appdata[5] = ( humInt >> 16 ) & 0x000000FF;
    appdata[4] = ( humInt >> 24 ) & 0x000000FF;
```

Listing 7.5: Sourcecode change from V1 to version V2.

- lora_diff.bin
  This is the patch binary built with the jDiff library [34]. Following command generates the patch:

  ```
  >jdiff.exe V1.bin V2.bin lora_diff.bin
  ```

  The patch file has the size of 14'477 Bytes.

- lora_patch_frag_218_20.txt
  With the following command the patch file is divided in fragments of size 218bytes and twenty redundancy fragments were added.

  ```
  >lorawan-fota-signing-tool create-frag-packets -i lora_diff.
      bin --output-format plain --frag-size 218 --redundancy-
      packets 20 -o lora_patch_frag_218_20.txt
  ```

With this setup the FUOTA session setup is tested. On the node the bootloader is already running as the log 7.6 shows. To start the FUOTA application the button 2 on the node has to be pressed to tell the bootloader to load the actual stored application in the flash.

```
13:22:33: Bootloader started

Mounting litteFS volume.

###### ========== Boot Config Data =========== ######
Boot counter: 0
Booting new Image?: 0
Patch state: 0
actual FW Version:  1  0  0  0
new FW Version:  1  0  0  0
13:22:34:


Ready for Button interaction.
```

Listing 7.6: FUOTA session setup demonstrator LOG 1.

By pushing the BTN2 on the demonstrator node the actual firmware is booted. There the node connects to the LoRaWAN network and is sending the temperature value periodically (log output 7.7).

```

13:22:45: User Up pressed.
------> ------> ------>
No Boot flag
act IMAGE boots
------> ------> ------>
13:22:49:
###### ======================================== ######
Application name    : periodic-uplink-lpp
Application version: 1.2.0
GitHub base version: 5.0.0
###### ======================================== ######
Mounting litteFS volume.LORA APP Started.
13:22:52: start joining ...
13:23:00:

JOINED (OTAA), DevAddr:  1B3B20F, DATA RATE: DR_0
13:23:04:
###### =========== MCPS-Indication =========== ######
STATUS       : OK
###### =====   DOWNLINK FRAME          0   ===== ######
RX WINDOW    : 1
RX PORT      : 0

13:23:06: ... connectedrequest to tx data
13:23:13:
###### ============ SENSOR DATA ============= ######
###### ======================================== ######
Temperature        : 28 \xB0
Humidity           : 42  RH

13:23:17:
MCPS-Confirm: OK
UPLINK FRAME:           2
 CLASS: A, TX PORT 2
 Payload: 0  0  B 33  0  0  0  0

13:23:22:
###### ============ SENSOR DATA ============= ######
###### ======================================== ######
Temperature        : 28 \xB0
Humidity           : 42  RH

MCPS-Confirm: OK
UPLINK FRAME:           3
CLASS: A, TX PORT 2
Payload: 0  0  B 33  0  0  0  0
```

Listing 7.7: FUOTA session setup demonstrator node LOG 2.

While the basic FUOTA application is sending senosor data, the FUOTA server is ready to start setting up the FUOTA session. By the following command, the FUOTA server starts and calculates the meta data for the FUOTA session (shown in log outout 7.8).

```
    node prepareFuota.js lora_patch_frag_218_20.txt Images/lora_diff.bin
        Images/V2.bin 01020305
```

```
    NR Fragments 87 Type number

    Patch binary Size 14477 Type number

    Patch fragment size 218 Type number

    Patch HASH 8
        c88814fc5ec6bde8f96c930c47b705dc6847b52946a6b4d33a358bee7fcd6ed
        Type string

    New binary HASH 2
        e0ae35326bd71ebf70c7c08750e6fff843e17fa01383864f39e369b576efcd1
        Type string


    MQTT client connected!
    Subscribed to all application events
```

Listing 7.8: FUOTA session setup server LOG 1.

After the metadata are calculated the server first runs the clock synchronization on the demonstrator node and the sets up the multicast and fragmented data session as presented in the communication flow diagram in figure 6.5. The shorted session set up log of the node is shown in listing 7.10 and the one from the server in 7.9.

```
Port144 msg is being sent:
 publishing as 2 3333333333333333 following msg { confirmed: false,
  fPort: 144,
  data:
   'jTgAAQIDBQHaVwCMiIFPxexr3o+WyTDEe3BdxoR7UpRqa00zo1i+5/zW7Q==' }
------------------------>>>>>>>>>>>>>>>>>>>>>>>>>
EUI 3333333333333333 fPort 144 payload Buffer <Buffer ff>
<<<<<<<<<<<<<<<<<<<<<<<<<------------------------

New Bin HASH 3333333333333333 2
    e0ae35326bd71ebf70c7c08750e6fff843e17fa01383864f39e369b576efcd1 seconds
publishing as 2 3333333333333333 following msg { confirmed: false,
  fPort: 145,
  data: 'LgrjUya9cev3DHwIdQ5v/4Q+F/oBODhk8542m1du/NE=' }

------------------------>>>>>>>>>>>>>>>>>>>>>>>>>
EUI 3333333333333333 fPort 145 payload Buffer <Buffer ff>
<<<<<<<<<<<<<<<<<<<<<<<<<------------------------
All device ready for setup...

------------------------>>>>>>>>>>>>>>>>>>>>>>>>>
EUI 3333333333333333 fPort 202 payload Buffer <Buffer 01 06 84 d4 4f 00>
<<<<<<<<<<<<<<<<<<<<<<<<<------------------------
```

```
###################################################################
deviceTime 1339327494 serverTime 1339327476
deviceTime 10:24:54 serverTime 10:24:36
Adjust time in Seconds -18
###################################################################
Clock sync for device 3333333333333333 -18 seconds
All devices have had their clocks synced, setting up mc group...

publishing as 2 3333333333333333 following msg { confirmed: false, fPort:
    202, data: 'Ae7///8A' }
sendMcGroupSetup

publishing as 2 3333333333333333 following msg { confirmed: false,
  fPort: 200,
  data: 'AgD///8BWC07g+rVGHByGYMrOQk96QAAAAD//wAA' }

------------------->>>>>>>>>>>>>>>>>>>>>>>>>
EUI 3333333333333333 fPort 200 payload Buffer <Buffer 02 00>
<<<<<<<<<<<<<<<<<<<<<<-----------------------

All devices have received multicast group, setting up fragsession...
sendFragSessionSetup
publishing as 2 3333333333333333 following msg { confirmed: false, fPort:
    201, data: 'AgBDANoAgQAAAAA=' }

All devices have received frag session, sending mc start msg...
sendMcClassCSessionReq
publishing as 2 3333333333333333 following msg { confirmed: false, fPort:
    200, data: 'BABIhNRP/526hAU=' }

------------------->>>>>>>>>>>>>>>>>>>>>>>>>
EUI 3333333333333333 fPort 200 payload Buffer <Buffer 04 00 35 00 00>
<<<<<<<<<<<<<<<<<<<<<<-----------------------
#####################################################################
3333333333333333 time to start 53 startTime is 1339327560 currtime is
    1339327514
3333333333333333 time to start 00:00:53 startTime is 10:26:00 currtime is
    10:25:14
startSendingClassCPackets
All device ready? { '3333333333333333':
   { clockSynced: true,
     fragSessionAns: true,
     mcSetupAns: true,
     mcStartAns: true,
     applicationID: '2',
     msgWaiting: null,
     ready: true } }
```

Listing 7.9: FUOTA session setup server LOG 2.

```

    ###### =====  DOWNLINK FRAME          0    ===== ######
    RX WINDOW     : 1
    RX PORT       : 144
    RX DATA       :
    8D 38  0  1  2  3  5  1 DA 57  0 8C 88 81 4F C5
    EC 6B DE 8F 96 C9 30 C4 7B 70 5D C6 84 7B 52 94
    6A 6B 4D 33 A3 58 BE E7 FC D6 ED
    13:24:06:
    ###### =========== Firmware Metadata ============ ######
    Patch Size: 14477
    Fragment Size: 218
    FW Version:  1  2  3  5
    patch Hash: 8C 88 81 4F C5 EC 6B DE 8F 96 C9 30 C4 7B 70 5D
    C6 84 7B 52 94 6A 6B 4D 33 A3 58 BE E7 FC D6 ED

    13:24:11:
    MCPS-Confirm: OK
    UPLINK FRAME:          9
     CLASS: A, TX PORT 144
     Payload: FF
    ###### =====  DOWNLINK FRAME          1    ===== ######
    RX WINDOW     : 1
    RX PORT       : 145
    RX DATA       :
    2E  A E3 53 26 BD 71 EB F7  C 7C  8 75  E 6F FF
    84 3E 17 FA  1 38 38 64 F3 9E 36 9B 57 6E FC D1
    DATA RATE     : DR_0
    RX RSSI       : 4294967276
    RX SNR        : 7
    ###### =========== Firmware Metadata ============ ######
    new firmware version Hash: 2E  A E3 53 26 BD 71 EB F7  C 7C  8 75  E 6F
         FF
    84 3E 17 FA  1 38 38 64 F3 9E 36 9B 57 6E FC D1

    13:24:14:
    MCPS-Confirm: OK
    UPLINK FRAME:         10
     CLASS: A, TX PORT 145
     Payload: FF

    MCPS-Confirm: OK
    UPLINK FRAME:         11
     CLASS: A, TX PORT 202
     Payload: 1  6 84 D4 4F  0
    ###### =====  DOWNLINK FRAME          2    ===== ######
    RX WINDOW     : 1
    RX PORT       : 202
    RX DATA       : 1 EE FF FF FF  0

    ----------------><----------------
    TIME SYNC OK!!!!!!!
    ----------------><----------------


     ###### =====  DOWNLINK FRAME          3    ===== ######
    RX WINDOW     : 1
```

```
RX PORT       :  200
RX DATA       :
  2   0 FF FF FF   1 58 2D 3B 83 EA D5 18 70 72 19
 83 2B 39   9 3D E9   0   0   0   0 FF FF   0   0

ID            :  0
McAddr        :   1FFFFFF
McKey         :  58-2D-3B-83-EA-D5-18-70-72-19-83-2B-39- 9-3D-E9
McFCountMin   :  0
McFCountMax   :  65535
SessionTime   :  0
SessionTimeT  :  0
Rx Freq       :  0
Rx DR         :  DR_0
13:24:30:

MCPS-Confirm: OK
UPLINK FRAME:          13
 CLASS: A, TX PORT 200
 Payload: 2   0
###### =====  DOWNLINK FRAME          4   ===== ######
RX WINDOW     :  1
RX PORT       :  201
RX DATA       :  2   0 43   0 DA   0 81   0   0   0   0

13:24:43:
MCPS-Confirm: OK
UPLINK FRAME:          15
 CLASS: A, TX PORT 201
 Payload: 2   0
###### =====  DOWNLINK FRAME          5   ===== ######
RX WINDOW     :  1
RX PORT       :  200
RX DATA       :  4   0 48 84 D4 4F FF 9D BA 84   5

Time2SessionStart: 53000 ms
ID            :  0
McAddr        :   1FFFFFF
McKey         :  58-2D-3B-83-EA-D5-18-70-72-19-83-2B-39- 9-3D-E9
McFCountMin   :  0
McFCountMax   :  65535
SessionTime   :  1655292360
SessionTimeT  :  15
Rx Freq       :  869852500
Rx DR         :  DR_5
13:24:58:

DATA RATE: DR_0, TX POWER: 0
U/L FREQ: 867900000
13:25:47:
###### ===== Switch to Class C done.  ===== ######
----------------->8-----------------
OnClassChange
----------------->8-----------------

13:27:58:
MCPS-Confirm: OK
```

```
UPLINK FRAME:          24
 CLASS: C, TX PORT 0
 Payload:
###### ========== MCPS-Indication ========== ######
STATUS        : OK
```

Listing 7.10: FUOTA session setup demonstrator node LOG 1.

If the setup worked as planed on both sides, the fragments should have been sent over the virtual multicast device in the Chirpstack application as a multicast downlink to the demonstrator node. The problem was that these messages where never sent. A deeper analysis has shown, that the fragments where sent from the JavaScript FUOTA server via MQTT to the Chirpstack application server, but the application server rejected these messages. Different information sources [25][24] have shown, that multicast downlink can not be sent via the MQTT protocol. To be able to send multicast downlinks, the Chirpstack Python SDK using gRPC [1] and the Chirpstack API have to be used. This had the consequence, that for the multicast downlink messages, another Python FUOTA server had to be implemented, which can send the fragments to the demonstrator node. The change in the system design is shown in figure 7.8. The main part of the Python FUTOA server is shown in code listing 7.11.

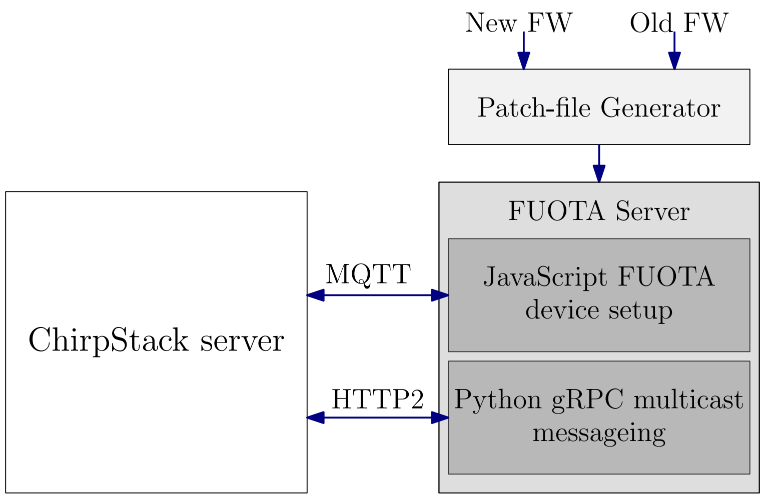


Figure 7.8.: Two-stage FUOTA server.

This second stage Pyhton server has to be called with following command:

```
python3 test-python.py ../test-fuota-server/test-fuota-server/
    lora_patch_frag_218_20.txt
```

At the moment of testing, this command has to be started manually after the devices have switched to class C. For future work, this hast to be automated and the two server have to be combined.

[1] "gRPC is a modern open source high performance Remote Procedure Call (RPC) framework that can run in any environment."[29]

```python
# Configuration.
# This must point to the API interface.
server = "localhost:8080"
# The API token (retrieved using the web-interface).
api_token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
              eyJhcGlfa2V5X2lkIjoiZTUyZDE3MzUtZTZiO
              C0OMmRiLWE1ODgtNTIxOWVjMmU1NTcyIiwiYX
              VkIjoiYXMiLCJpc3MiOiJhcyIsIm5iZiI6MTY
              1NDExNTM0Miwic3ViIjoiYXBpX2tleSJ9.OMt
              nV2yJECFRd9QqrD0a4Fm5xKElFJVUhPhbLozyEN4"

if __name__ == "__main__":
    # Connect without using TLS.
    channel = grpc.insecure_channel(server)

    # Device-queue API client.
    client = api.DeviceQueueServiceStub(channel)
    multicastclient = api.MulticastGroupServiceStub(channel)

    # Define the API key meta-data.
    auth_token = [("authorization", "Bearer %s" % api_token)]

    # Construct request.
counter = 15
for line in Lines[1:]:
    counter = counter + 1
    byte_strings = line.split()
    byte_arrays = [int(byte_string, 16) for byte_string in byte_strings
        ]
    reqMulti = api.EnqueueMulticastQueueItemRequest()
    reqMulti.multicast_queue_item.multicast_group_id = "
        ec3c22c436fb4652beb288e294adf880"
    reqMulti.multicast_queue_item.data = bytes(byte_arrays)
    reqMulti.multicast_queue_item.f_cnt = counter
    reqMulti.multicast_queue_item.f_port = 201
    respMuti = multicastclient.Enqueue(reqMulti, metadata=auth_token)
    print(respMuti)
    print([f"{hex(b)}" for b in byte_arrays], end="")
    print("")
    time.sleep(2.5)
```

Listing 7.11: Python FUOTA server.

7.4. Multicast-Fragments handling

After the new server was implemented, the test was run again. The same test setup as in the previous tests where used. Additionally, an OttiARC power-analyser [55] is plugged to the RFM96 power supply, to measure the used power of the module during the FUOTA session. Further, the logs presented, will not include the session setup which is already presented in previous test. In the log 7.12 it can be seen, that the fragments were received by the node. At the end 2 fragments were lost and the node needed in total 69 fragments to rebuild the patch file. This is in total a size of 15042bytes sent as multicast downlink messages. At the end the demonstrator node switches back to the class A and confirms the correct received patch by sending back the calculated HASH of the patch file, which is identical with the one calculated by the FUOTA server.
Figure 7.9 shows the power consumption of the RFM96 node during the FUOTA session setup and the received multicast fragments. The RFM96 module needs approximately 40mW for receiving and transmitting the data. However it must be kept in mind, that this value is not the radio power, but the power the whole module needs. From the power log the different parts in the FUOTA session can be extracted.

1. Part one is fits to the joining process.
2. Part two is the periodical uplink with the sensor data.
3. Part three is the FUOTA sessions set up.
4. In part four the node switched to class C and waits for the multicast fragments. This took more time than expected since, the logfiles had to be saved on the host machine and the Python server was then manually started.
5. In part five the multicast fragments were received.
6. At the end in part six, the HSAH of the patch file was transmitted to the server to confirm a successful transmission.

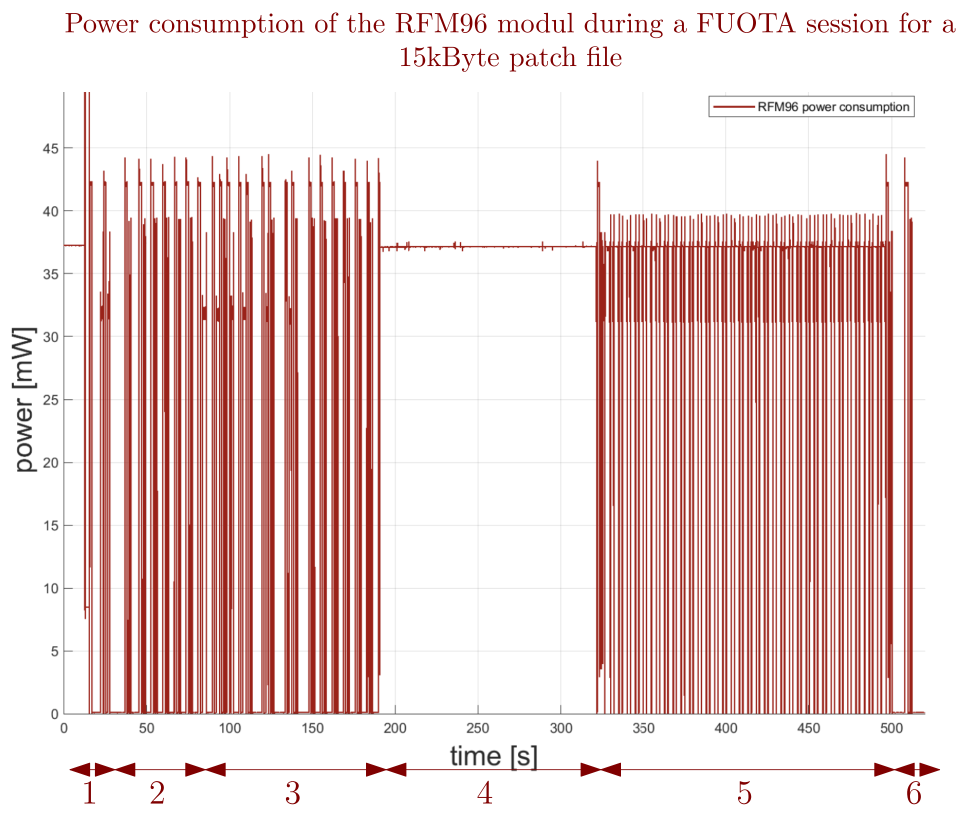


Figure 7.9.: Power consumption of the RFM96 module during the FUOTA session.

```

    ------------------------><------------------------
    OnClassChange
    ------------------------><------------------------


      ###### =====  DOWNLINK FRAME     2530   ===== ######
    RX WINDOW      : C Multicast
    RX PORT        : 201
    RX DATA        :
     8  1  0 A7 A3  B A7 A6 1F A7 A3 2E A7 A4  7 A7
    A6 27  E  3 A7 A3 38 A7 A6 15 ED  2  0 1D ED  2
     0 29 D7  1 A7 A3 A8 A7 A5 75 EE  2  0 7D EE  2
     0 A7 A6  4 71 A7 A3  9 A7 A6  4 71 A7 A3  9 A7
    A6  4 71 A7 A3 F7 A7 A6 1A A7 A3  A A7 A6 B7 A7
    A3  E A7 A6 AF A7 A3 16 A7 A6 56 A7 A3  8 A7 A6
    CB A7 A3  8 A7 A6 F0 A7 A3  6 A7 A6  A FF 14 F0
    A2 A7 A3  8 A7 A6 41 A7 A3  6 A7 A6 B7 A7 A3  8
    A7 A6 DC F9 14 F0 92 A7 A3 FD  2 ED A7 A6 1D A7
    A3 90 A7 A6 62 A7 A3 14 A7 A6 2D A7 A3  6 A7 A6
    7B A7 A3  4 A7 A6 74 A7 A3  A A7 A6 20 A7 A3 22
    A7 A6 88 A7 A3 44 A7 A6 CD A7 A3 12 A7 A6 E3 A7
    A3  6 A7 A6 2F A7 A3  4 A7 A6 18 A7 A3  A A7 A6
    D4 A7 A3  C A7 A6 85 A7 A3 FC 4A A7 A6

    DATA RATE      : DR_5
    RX RSSI        : 4294967276
```

```
RX SNR        : 7
###### ============ FRAG_DECODER ============ ######
######              PROGRESS                ######
###### ==================================== ######
RECEIVED      :      1 /     67 Fragments
                   218 / 14606 Bytes
LOST          :                0 Fragments

13:28:07:
###### =====  DOWNLINK FRAME      2533   ===== ######
RX WINDOW     : C Multicast
RX PORT       : 201
RX DATA       :
8  4  0 A7 A6 43 A7 A3 D6 A7 A6 59 A7 A3  C A7
A6 24 A7 A3 60 A7 A6 59 A7 A3  8 A7 A6 E0 A7 A3
E A7 A6 D8 A7 A3 28 A7 A6 33 A7 A3 50 A7 A6  A
A7 A3 1A A7 A6 FC A7 A3  C A7 A6 74 A7 A3  6 A7
A6 E7 41  3  0 6A 40  3  0 72 40  3  0 82 40  3
0 88 40  3  0 8E 40  3  0 99 A7 A3 26 A7 A6 D1
A7 A3 34 A7 A6 B6 A7 A3 1A A7 A6 A8 A7 A3 1A A7
A6 9A A7 A3 52 A7 A6 70 A7 A3  8 A7 A6 99 A7 A3
6 A7 A6 E7 41  3  0 A1 40  3  0 72 40  3  0 AB
40  3  0 B5 40  3  0 BE 40  3  0 C7 40  3  0 CF
40  3  0 D8 40  3  0 E2 A7 A3  E A7 A6 51 A7 A3
8 A7 A6 F8 A7 A3  8 A7 A6 D3 A7 A3 1A A7 A6 39
A7 A3 12 A7 A6 BB A7 A3 1A A7 A6 21 A7 A3 14 A7
A6 A2 A7 A3 18 A7 A6  9 A7 A3  8 A7 A6

DATA RATE     : DR_5
RX RSSI       : 4294967277
RX SNR        : 8
###### ============ FRAG_DECODER ============ ######
######              PROGRESS                ######
###### ==================================== ######
RECEIVED      :      4 /     67 Fragments
                   872 / 14606 Bytes
LOST          :                2 Fragments

...
...
...

###### ============ FRAG_DECODER ============ ######
######              PROGRESS                ######
###### ==================================== ######
RECEIVED      :     69 /     67 Fragments
                 15042 / 14606 Bytes
LOST          :                2 Fragments

13:30:53:
###### =====  DOWNLINK FRAME      2599   ===== ######
RX WINDOW     : C Multicast
RX PORT       : 201
RX DATA       :
8 46  0 4A EC A9 99 7F 2B 2C 41 42  C 56 6B 2A
D0 DA 45 E4 8D 35 AD 6D A2 2F B8 E3 60 41 C9 1E
36 5B  8 97 94 36 87 49 6A 2D 64 F2 75 D6 5D 51
```

```
B9 B6 62 AC D2 9C  7 8D 2B 95 A6 17 A1 7A F1 B1
D3 39 30 69 DE BF 88  F A7 A7 10 86 1F 52 BF 2B
E8 43 CA B6 A4 31 73  8 5B 2B D3 EE 2C 85 5B 23
1C EA 5E 1D B3 DB 2C C9 39 A8 ED 84 28 19 8B 38
DD AF  7  6 DA ED 49 92 30 10 9C 7D 4B  A 59  1
C3 EA 62 DF 99 B4 54 96 63 C9 E5 B5 E9 24 54 2B
58 61 85 7F 49 2E 55 47 12 8B D7 18 ED 90 2D 71
82 E8 D9 9C 9F D0 7A 45 88 B4 2C 66 19 24 8E 2F
58 EB DC FE  9 82 47 4D FF 38 53 F4 76 66 8C  E
52 BA 47 DA 76 EC 78 72 ED B1 E1 D1 CE A5 BD 12
20 AC 83 CD 21 80 18 C2 3A 52 ED 3A 27

DATA RATE    : DR_5
RX RSSI      : 4294967277
RX SNR       : 7
###### =========== FRAG_DECODER ============ ######
######              FINISHED                 ######
###### ===================================== ######
STATUS       : 4294967294
CRC          : 49999466

 Size        : 14477
###### ===== Switch to Class A done.  ===== ######
----------------><----------------
 OnClassChange
----------------><----------------

 13:31:09:
MCPS-Confirm: OK
UPLINK FRAME:        27
CLASS: A, TX PORT 146
 Payload:     8C 88 81 4F C5 EC 6B DE 8F 96 C9 30 C4 7B 70 5D
              C6 84 7B 52 94 6A 6B 4D 33 A3 58 BE E7 FC D6 ED
```

Listing 7.12: Fragmented multicast downlinks LOG.

7.5. Merge and boot process

For the merge and boot, the same setup is used as in previous test (section 7.3). The part of the FUOTA session setup and transmission of the fragments over multicast will not be presented again. The log 7.13 starts after all fragments were received and the node has confirmed a successful transaction with the HASH of the patch file. The merge process took approximately 100s. Interesting is that after 89% of the data are patched the process jumped immediately to 100%. The flash process to the internal flash took about 7s for the newFw.bin file which has a size of 120kByte. After the firmware file is flashed, the bootloader tried to boot the new image but failed. No application was running after the boot sequence. An analysis of the generated new firmware binary after the patch process has shown, that from address 0x88C5 the file does not match with the original firmware version V2. As the merge and boot process worked in a simpler setting with a different file transfer (section 7.1.2) and the HASH of the patch file was identical with the one on the server, the problem has to be related to the size of the patch or file alignment, for example that big endian and little endian are incorrectly swapped. Due to time constrains this analysis of the error, while merging the new file, has to be delayed to the project end or to subsequent work.

```
 13:31:09:
MCPS-Confirm: OK
UPLINK FRAME:        27
 CLASS: A, TX PORT 146
 Payload:
8C 88 81 4F C5 EC 6B DE 8F 96 C9 30 C4 7B 70 5D
C6 84 7B 52 94 6A 6B 4D 33 A3 58 BE E7 FC D6 ED
DATA RATE: DR_0, TX POWER: 0
U/L FREQ: 868300000
 13:31:34: Bootloader started

Mounting litteFS volume.
###### =========== Boot Config Data ============ ######
Boot counter: 0
Booting new Image?: 1
Patch state: 0
actual FW Version:  0  0  0  0
new FW Version:  1  2  3  5

Ready for Button interaction.
13:34:08: User Up pressed.
------> ------> ------>
patch process starts
------> ------> ------>
13:34:10:
########## ========== PATCH PROCESS STATE ========== ##########
Patchprocess:  0
########## ========== PATCH PROCESS STATE ========== ##########
Patchprocess:  1
...
...
...
########## ========== PATCH PROCESS STATE ========== ##########
Patchprocess:  89
```

```
 13:35:46:
##########  ==========  PATCH PROCESS STATE  ==========  ##########
 Patchprocess:  100
 13:35:47:
##########  ==========  FLASHING DATA TO MEMORY  ==========  ##########
 Flashed Block to address 0x   19000
 File size remaning 0x   1E1AC
##########  ==========  FLASHING DATA TO MEMORY  ==========  ##########
 Flashed Block to address 0x   19200
 File size remaning 0x   1DFAC
 ...
 ...
 ...
##########  ==========  FLASHING DATA TO MEMORY  ==========  ##########
 Flashed Block to address 0x   37000
 File size remaning 0x     1AC

##########  ==========  FLASHING DATA TO MEMORY  ==========  ##########
 Flashed Block to address 0x   37200
 File size remaning 0xFFFFFFAC
--------->  --------->  --------->
BOOTING NEW IMAGE
--------->  --------->  --------->
```

Listing 7.13: Merge and boot log.

7.6. Complete system test

For the complete system test, the same test setup is used as in previous test (section 7.3), but additionally one more demonstrator node was added to the network. To demonstrate the process, the same patch will be sent several times. For each run the results and logs will be saved. In table 7.1 the results of the test are presented. Row "TOT. FRAG" shows the total number of fragments received by the demonstrator node. In the row "LOST" the number of fragments lost during the multicast session are listed. The "PATCH HASH" and "NEWFW HASH" indicates, if the calculated HASH of the received patch file and the one form the new merged firmware file are correct.
From test three on, the "DEV2" was better aligned with the gateway to improve the connection. It can be seen, that the patch file was in total correctly received 10 from 12 times. Regarding the described problem with the weak signal strength (figure 7.5), this result proofs that the multicast protocol in general works as mentioned in the theory. The problem with "DEV2" and the wrong HASH of the patched firmware was, that the node did not store the patch file properly in the external flash, it was stored empty. Due to this the merging process didn't even started. Later the error was detected by as a wrong key - value pair in the FW_conf.ini file caused, that the patch file was not flashed to the external W25Q SPI flash.
The calculation of the new merged firmware HASH and printing it out to the console was first done in this test. Seeing, that the HASH is correct, the bootloader should be able to read it from the external flash, write it to the correct location in the internal flash and boot the new version.
As already mentioned, a further analysis of the presented founding in these tests have to be shifted to consequent work.

DEV1				DEV2			
TOT. FRAG	LOST	PATCH HASH	NEWFW HASH	TOT. FRAG	LOST	PATCH HASH	NEWFW HASH
69	0	✓	✓	77	5	✓	✗
71	2	✓	✓	71	2	✓	✗
69	0	✓	✓	69	24	✗	✗
69	0	✓	✓	69	0	✓	✗
69	0	✓	✓	69	0	✓	✗
69	0	✓	✓	69	0	✗	✗

Table 7.1.: Results for multiple devices test.

8. Results and Outlook

This thesis has been separated into two main parts: hardware and software design. These two parts contained the sub categories: FUOTA server, LNS&Gateway and demonstrator node. Together they built the infrastructure for a proof of concept in firmware updates over the air in a LoRaWAN network. In the following section, the characteristics of the demonstrator are listed, the results of this research work are recapitulated, and a potential continuation is discussed.

8.1. Review of the demonstrator node

Figure 8.1 shows the full hardware stack of the demonstrator node.

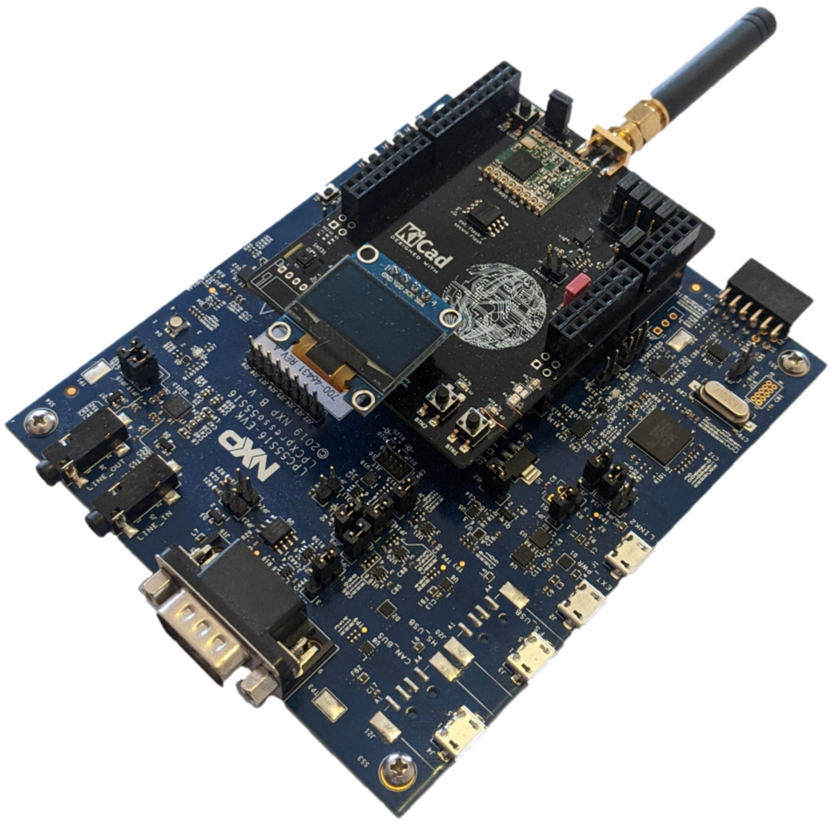


Figure 8.1.: Demonstrator node.

8.1.1. Provided hardware functionality

In the following list, the functionality of the presented demonstrator node is discussed.

- RFM96 module:
  The RFM96 LoRa module is connected to the SPI bus on FLEXCOM8 of the LPC55S16 and all provided GPIOs of the module are connected to the microcontroller. With the implementation of the sx1276_RFM96.c driver (figure 8.2), the demonstrator node is able to communicate in a LoRaWAN network.

```
LoRa/
└── src/
    └── boards/
        └── LPC55s16-EVK/
            ├── sx1276_RFM96.c/
            ├── .../
            └── .../
```

Figure 8.2.: RFM96 driver.

- W25Q NOR flash:
  The external 16MByte NOR flash is connected to the SPI bus on FLEXCOM3. Together with the LittleFS library, the demonstrator node provides a secure files system to store firmware image files in it.

- DS3232 RTC:
  The I2C RTC chip has its own battery for an accurate time management on the node. Together with the McuLib the node is able to synchronize the internal RTC.

- SHT31 sensor:
  For the sensors on the demonstrator node, the full functionality of the SHT31 over the I2C interface is provided.

- SSD1306 display:
  For the visualization of the process states, the I2C SSD1306 OLED display in combination with the McuLib gives the print messages on the display. This is used to show sensor values, FUOTA metadata and process states.

- Buttons:
  The two push buttons on the demonstrator node provide the possibility to start or stop processes on the board.

8.1.2. Implemented firmware stack

The firmware on the demonstrator node was divided into two applications, the bootloader and the FUOTA application. The bootloader is called at every startup and checks the metadata stored in the config files on the external flash. Based on the metadata it decides if a new firmware is ready to merge and boot or if the actual or back image has to be loaded. For the merge process, the jDiff library [34] is ported to the LittleFS file system and has the possibility to create a new image based on the patch file and the current image.
The FUOTA application is the main running application on the demonstrator node. With the integration of the LoRaMac-node [58] software stack it provides an interface to communicate with the LoRaWAN network. In regular mode, the demonstrator node is logging the sensor data of the SHT31 chip and periodically sends its data to the Chirpstack application server. With the use of the time sync, multicast and fragmentation protocol, the demonstrator node covers the possibility to receive big data blocks divided into fragments. These blocks could then be stored as a patch binary file in the external flash and with a software reset the bootloader application can be called. The following figure 8.3 shows the memory footprint of the two applications in the internal flash and the RAM usage calculated at compiler time. The applications are compiled for size optimization. The bootloader uses 90% of the provided flash and 66%of the RAM, while the FUOTA app is using 82% of the flash and 92%of the RAM. Regarding the memory usage, in chapter 8.3 some recommendation are presented.

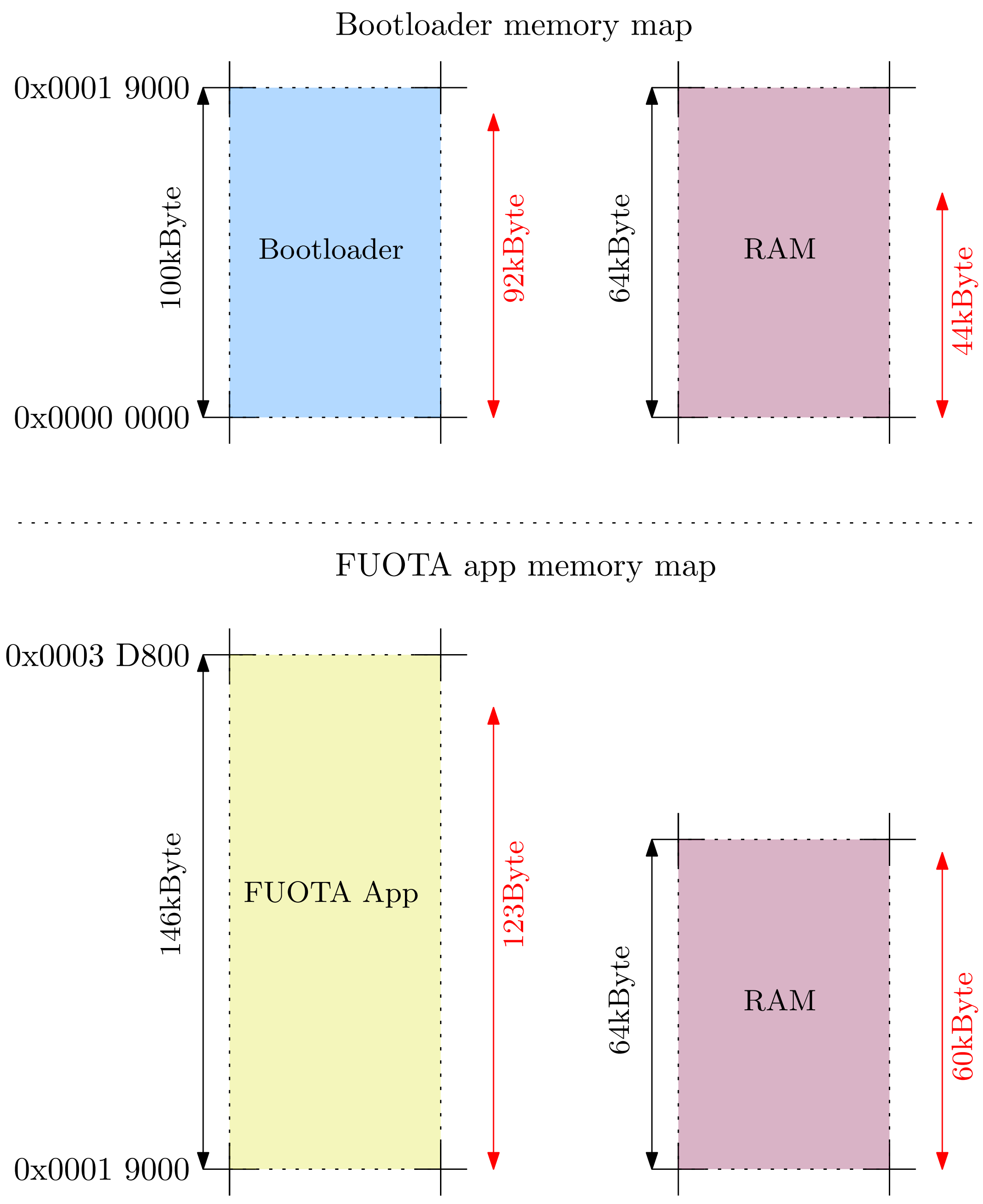


Figure 8.3.: Memory footprint of the two application.

8.2. FUOTA server

The FUOTA server is a JavaScript server which is communicating with the Chirpstack application server using the MQTT protocol. This server is able to parallel setup multiple predefined devices with the clock synchronization, multicast session and fragmentation session. The solution for the detected problem, to send multicast messages during the testing (chapter 7.3) was, to implement a second FUOTA server using a Python script and the Chirpsatck API. This Python script is using gRPC to call Chirpstack network server function directly form the FUOTA server. This solution leads to a two-stage FUOTA server. In the first stage the JavaScript server is setting up all devices for the FUOTA session and in stage-two, the Python gRPC server is sending the fragmented data as a multicast message to all devices. The following figure 8.4 shows the part of the new system design caused by the two-stage FUOTA server. With this new second stage server, the fragmented data block can be sent over the Chirpstack infrastructure to the demonstrator nodes.

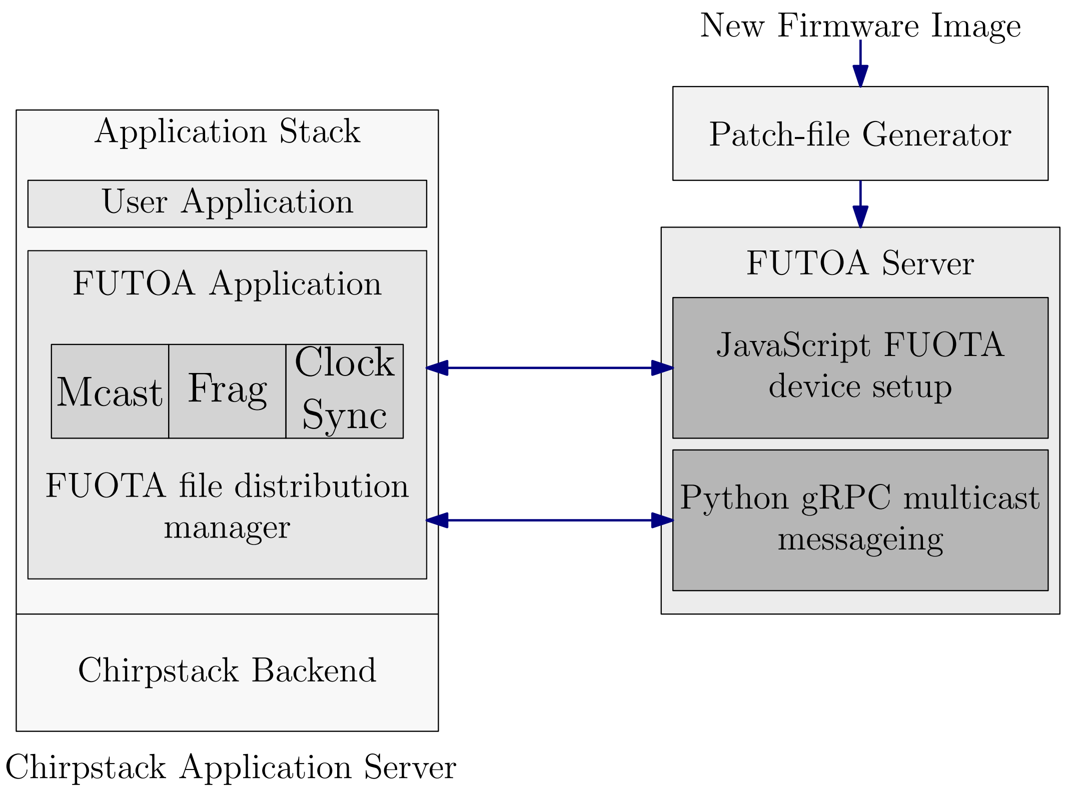


Figure 8.4.: Adjusted system design of the FUOTA server.

8.2.1. Verification

In hindsight, the objectives for this thesis were set high. The main goal was to build a demonstrator that runs parallel multi-device firmware updates over the air using a local LoRaWAN network. This demonstrator includes the complete infrastructure that is necessary for a successful procedure. The demonstrator shows, how a firmware image can be sent to multiple nodes in the filed using the multicast protocol in a LoRaWAN network.
An abstracted system design was introduced, from which the main task for the thesis was derived. A key task was developing a demonstrator node based on the LPC55S16 EVK. This board needed additional components for the LoRaWAN communication and the node should further provide additional flash memory and some demonstrator properties, such as sensors and actors. Already early in the project, a hardware shield was developed, which fits on the EVK extension header and provides the required functionalities. With the driver implementation for the SX1276 LoRa chip, the node was able to communicate in the LoRaWAN network. During the integration of the required functionalities, identified bugs of the hardware were documented continuously. Thus, a second as well as improved version was developed later on.
With the working demonstrator node, the implementation of the FUOTA application and bootloader were launched. Integrating the McuLib with the FreeRTOS and the LittleFS into two applications provided a flexible firmware development. For the bootloader test, a Python script enables the possibility to send a patch file over the serial interface to the demonstrator node. A jDiff patch library provides the functionality to merge a patch file to build a new firmware. These functionalities were tested with basic applications.
Furthermore, the FUOTA application was implemented and the functions of reading sensor values and displaying different messages on the OLED display were tested. The basic LoRaWAN application was adapted to run in a FreeRTOS task, in order to provide a flexible architecture of the firmware. Additionally, the key task of the clock synchronization, multicast session setup and fragmentation session functionalities were implemented. In parallel, the FUOTA server was implemented to send protocol messages for the FUOTA session using a MQTT interface to the Chirpstack application server. A second stage FUOTA server using a Python API with a gRPC integration had to be incorporated to be able to send fragmented multicast messages. The complete FUOTA session process has been tested, and the infrastructure is ready to send firmware images from the FUOTA server to the demonstrator node.
Running two different applications on the same microcontroller, caused some problems in debugging. Further, not having the McuShell functionality ready, due to a limited flash memory, presented an additional problem for debugging the complete process of the firmware update over the air.
In the end, the process has been tested on two demonstrator nodes running in parallel. The FUOTA sessions setup and multicast transmission of the fragments worked well. Some errors occurred while booting and merging the received patch file to a new image, writing it from the external flash to the internal and running it, which should be further analyzed in the future.
To briefly summarize the outcomes: The presented infrastructure is ready to build and

send a firmware patch file over the LoRaWAN network using the multicast protocol to multiple devices in the field in parallel. The following merge and boot process work in basic examples but caused problems in a complete run through.

8.2.2. Validation

To validate the functionality of the demonstrator, various parts were tested in a decapsulated state. During the implementation of the drivers for the demonstrator node, multiple aspects were tested using the debugging tools of the IDE. Single function calls or application segments were stepped through in order to check the behavior of the application and hardware. Together with a logic analyzer, the communication interfaces and protocols of I2C and SPI bus were reviewed. Once the board drivers were working as expected, the demonstrator node was then tested using a LoRaWAN sample application for a periodic uplink. The sample application sent random content to the backend of the Chirpstack server, running on the host machine.
In parallel, the secure bootloader was developed on the basis of the findings from the research as well as previous work [53]. The use of the McuLib components Shell, FreeRTOS and FatFS, simplified the debugging process with the external W25Q SPI flash. With the implementation of a Python script, that sends binary files in fixed block sizes over a serial interface directly to the external flash, the merge process [33] was tested using a simple hello_world firmware example. The bootloader was able to merge the patch file with the actual version of the hello_world application binary stored in the external flash to a new firmware binary. As a next step, the bootloader flashed the new built binary to the inter application flash region. Finally, the bootloader boots into this application region and the new hello_world application was ready to boot and run correctly with printing new data to the serial interface.
To debug and test the FUOTA application on the demonstrator node, the FUOTA server first needed to send correct setup messages. This was debugged with the help of the Chirpstack application server web interface. From there, it was able to check if the messages were sent to the LoRaWAN network and printed to the SEGGER RTT terminal. From there, the message was analyzed for its propriety. After the messages were correctly aligned, a code step through of the clock synchronization, multicast and fragmentation session setup were done with the debugger. It was observed that the time synchronization was made correctly, but the implemented software RTC could not save the values properly on the internal flash. Further, it detected that the class C switch was called but because the LoRaMAC handler was busy, the switch was not triggered. After a code change, the class C switch was forced to perform all requests.
During the tests (chapter 7), it was seen, that the fragmented multicast messages could not be sent from the JavaScript MQTT server since the Chirpstack API did not implement the multicast downlink for the MQTT interface. To send multicast downlinks, the Chirpstack API, with an integrated gPRC stack, has to be used. This had lead to an implementation of a second stage FUOTA server using Python with the installed Chirpstack API.
In the end, a complete system test has shown that the fragments can be sent as multicast messages to two different demonstrator nodes in parallel. These demonstrator

nodes were able to stick the fragments back to a patch binary file. This process could have been validated by calculating the HASH of the patch binary file and compare it with the HASH calculated by the FUOTA server. The bootloader call after the FUOTA session was seen in the serial output log with a specific message. The merge algorithm [33] then generated the new firmware version. With printing the calculated HASH from the new firmware file to the serial port, a comparison to the calculated HASH on the host machine was made manually. If these were identical, a pressed button triggered the flashing of the new version to the internal flash region. The validation of this aspect was not fully completed, as errors have occurred that could not be corrected before the submission deadline of the thesis.

8.3. Recommendations for improvement and future work

In the following chapter, the detected optimizations of the presented work for further use or follow-up projects are evaluated. The chapter is divided into two sections: Demonstrator node and FUOTA server.

8.3.1. Demonstrator node

8.3.1.1. Hardware

- **RFM96 module**
  The tests in chapter 7.2 have shown that there is a major problem regarding the signal strength of the RFM96 module. The source of the problem was not detected yet, but having an already integrated SX1261MB2BAS shield in the firmware stack, leads to the suggestion to use a SX1261 LoRa chip and develop the RF circuit directly on the demonstrator node. The work [18] could serve as a guideline for the hardware implementation.
  Nevertheless, a clean error analysis should first be made. A brief search has shown that the problem could have arisen due to a wrong configuration of the SX1276 LoRa chip [26].

- **Pinout changes**
  The button SW3 (BTN1) and the LED D5 (figure 8.5) were mapped to the GPIOs P0_12 and P0_11 of the LPC55S16. The problem is that these GPIOs are used for the debug interface. Therefore, the SW3 and LED D4 need to be remapped as well as the GPIO P1_9, which is the user button on the EVK mapped to the RFM96 DIO5. As a consequence, the USR button on the EVK can not be used anymore. It recommends mapping the EVK buttons on the same pins as the demonstrator shield buttons. Then either the EVK buttons or demonstrator node buttons can be used.

Another small thing to change is the position of the OLED display. It should be aligned to the border of the shield or placed in the middle of the PCB, so that the buttons are more accessible.

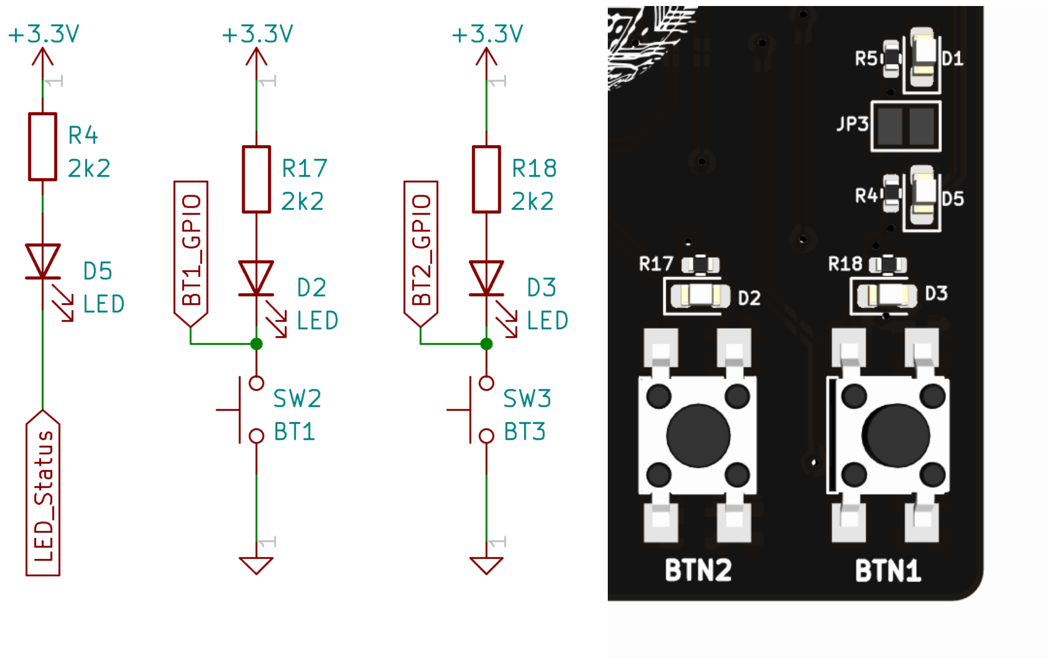


Figure 8.5.: Wrong pin mapping.

8.3.1.2. Firmware

- **ROM libraries**
  The actual firmware stack uses two applications, the bootloader and the FUOTA application. Both applications are using identical firmware components. Figure 8.6 shows the used components from each application and marks the overlapping ones. All these marked components are identical on boot application and therefore exists twice in the internal flash, which is not ideal regarding a small memory footprint. As an improvement these components should be built as a ROM library. This means that they are decoupled from the application and stored as a shared library in the internal flash. By this, both applications can access the library functions and are not duplicated in the flash. Figure 8.7 shows a possible memory footprint after using the approach of ROM libraries. This would drastically reduce the memory footprint of both application and therefore also reduce the patch firmware update size, which is desirable. The here explained approach is not tested, and a short research has not many solutions or example publications. Although the problem is known [20], a further deep research has to be performed. The other findings in the short research were, that often a dynamic linker will be used with a library compiled as position independent code. This approach has been presented in previous work [52] and should be further analyzed in a possible follow-up project.

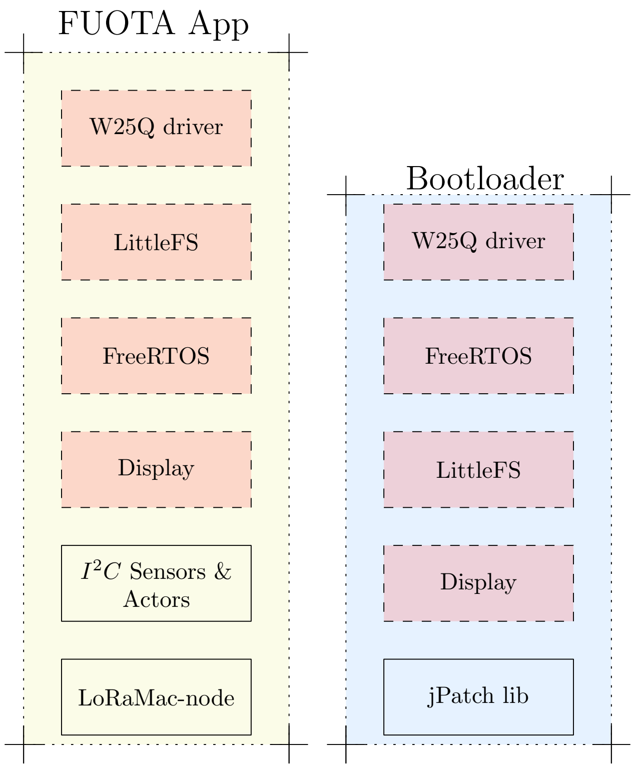


Figure 8.6.: Same firmware components used in both applications.

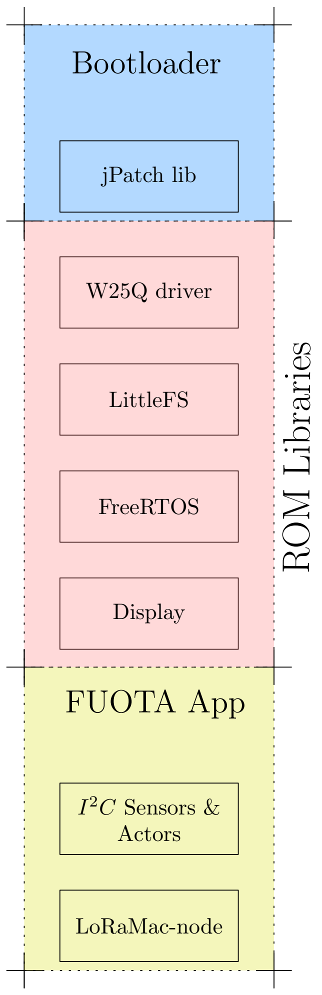


Figure 8.7.: ROM library approach.

- **Improving firmware similarity**
  During the testing phase of this thesis it was noticed that even small changes in the source code result in a large delta patch files. For example in the FUOTA application, a minor change in the LoRaWAN uplink payload was made (shown in code listing 8.1). Instead of sending the humidity as integer value, zero is sent. Generating the delta from the firmware version were the correct humidity data is sent to the one a zero is sent, results in a delta.bin file of the size of 14kByte. The binaries are built with the compiler option for size optimized code. For a better understanding of the big delta, the SRECORD file of both version were compared using the compare tool from notepad. It can be seen that there are multiple changes from the version 1 to version 2. Repeating this steps with the binaries built with the compiler option set for not optimizing at all, the delta binary did not result in a smaller size.
  The problem of these big delta binaries, even of just small changes are made in the source code, were addressed in previous work [52]. There are several techniques to gain firmware similarity. Some of these were explained in the research chapter 4.2. For the project in this thesis, these techniques were not considered. However, for further work and an optimized process, these techniques have to be taken into account.

```
        readSensorSHT31(&tmp, &hum);
        tmpInt = (uint32_t)(tmp*100);

        /*
        *#####################################################
        *This is the only source code change from version 1  to
            verison 2
        *#####################################################
        */
        //humInt = (uint32_t)(hum*100);
        humInt = (uint32_t)(0);

        uint8_t appdata[8];
        appdata[3] = tmpInt & 0x000000FF;
        appdata[2] = ( tmpInt >> 8  ) & 0x000000FF;
        appdata[1] = ( tmpInt >> 16  ) & 0x000000FF;
        appdata[0] = ( tmpInt >> 24  ) & 0x000000FF;
        appdata[7] = humInt & 0x000000FF;
        appdata[6] = ( humInt >> 8  ) & 0x000000FF;
        appdata[5] = ( humInt >> 16  ) & 0x000000FF;
        appdata[4] = ( humInt >> 24  ) & 0x000000FF;
```

Listing 8.1: Small sourcecode change for delta example.

- **LoRaMAC-node using the McuTimeDate**
  At the moment, the LoRaMAC-node is using an own software RTC implementation. Regarding the timing problems occurred during the testing phase, this implementation should be adapted. The LoRaMac-node stack should use the McuLib McuTimeDate functionalities. This would guarantee a correct and accurate time mechanism.

- **LoRaMac-node**
  The LoRaMac-node needd non-volatile memory to store keys, IDs and session counter in it. Having an external memory on the demonstrator node leads to the concept of storing this data on the external memory. To achieve this the board abstraction drivers of the LoRaMac-node stack have to be adapted. This data should be stored encrypted, to keep security risks minimal.

8.3.2. FUOTA server

The testing phase has shown that there are some optimization recommendations on the FUOTA server for a better process flow.

- **McKe_encrypted**
  In chapter 6.1.5 the use of the McKe_encrypted was presented. The calculation of this key for a defined application key and multicast group ID, the Python script in attachment A.2 generates the needed McKe_encrypted. The dynamic calculation of the McKe_encrypted for every device, that should be updated, is not yet implemented. At the moment, the demonstrator nodes uses the same application key and for this it only needs one McKe_encrypted. This solution was chosen to simplify the process for a more dynamic use and not weakening security when all devices have the same application key. This approach is yet be implemented.

- **Python API and gRPC**
  As described in section 8.2 the FUOTA server had to be divided into two different servers, due to the problem with sending multicast downlink messages. The solution was to implement the multicast messaging with the Chirpstack Python API using gRPC. The whole setup of the nodes for the clock synchronization, multicast and fragmentation session made with the JavaScript server should be ported to the Python API. This API provides all the Chirpstack API functionalities and having the whole FUOTA server in one service increases the robustness and dynamic of the whole process.

9. Conclusion

The goals as well as the expected results of this thesis were ambitious. The focus of this project was the implementation of a complete demonstrator infrastructure to handle a FUOTA process for multiple devices in parallel. The demonstrator is supposed to include the FUOTA server, LNS&gateway and the nodes in the field.
The chosen approach allowed an extensive research for the specified building blocks. Solutions that are targeting problems of the individual blocks were later combined in a concept for the demonstrator infrastructure. The implementation of the concept was then separated into three parts, that allowed partial success within the project and a reasonably independent development. The designed PCB for the demonstrator node successfully met the technical requirements needed to establish a communication in the LoRaWAN network and to store data on an external flash chip.
The developed demonstrator infrastructure offers the functionality to set up a group of demonstrator nodes into a multicast session. The FUOTA server can then send a generated patch binary divided into fragments over the multicast channel parallel to the demonstrator nodes. The node itself is able to merge the binary patch file to the actual image, and provide a new firmware image. The implemented bootloader will then flash the image into the internal flash of the microcontroller. After a successful merge and flash process, the bootloader will boot the new firmware.
The development of the demonstrator, including the server and nodes, builds its own environment. Together with the content of this documentation, the full work flow of a parallel multi device firmware update over the air can be taken as a guideline for a concrete industrial solution. Even if not all test criteria are fulfilled and there are a lot of potential optimizations, most of the objectives of the thesis have been achieved.

9.1. Industrial trend

As described in section 4.1, the growth in installed IoT devices will increase rapidly over the next 10 years. A major part of these devices communicate via the ultra low power LoRaWAN protocol. The manufacturers of the LoRaWAN infrastructure and the end nodes have recognized the importance of a firmware update over the air and are in the process of providing solutions to perform such updates. In May 2022 the LoRa Alliance® published new versions of the multicast, timesync and fragmentation specifications [44][43][43]. Further they developed a new protocol which should simplify the management of the firmware versions on the nodes [45]. These new publications show, that the need for a framework for firmware update over the air is urgent. For a stable integration of the FUOTA process in industrial solutions and to be able to record

real performance data, demonstrators like the one developed in this thesis are vital.
It can be assumed that in the next 5 years the firmware-update-over-the-air process for LoRaWAN nodes will be integrated as standard in industrial applications.

9.2. Lessons learned

During this thesis, I was able to further develop my skills in the area of embedded hardware and software solutions. This includes identifying hardware components to reach the set requirements, designing of own PCBs, implementing hardware drivers, porting libraries, low level debugging and working with embedded operating systems. Rather challenging was the server implementation which included Python and JavaScript scripting, understanding of network architectures, handling certificates, setting up software stacks, working with docker containers and using different network interfaces.
Personally, I believe that the area of full-stack development is an interesting subject, where you have the possibility to create a complete solution with frontend, server backend and even developing your own hardware. The dilemma that occurs while writing a master thesis that requires the usage of these features is that there are many possible sources for problems. Since the duration of the thesis is limited to one semester, the time slot for a full-stack implementation as an individual is very restricted. This limited period of time already conflicted the development in my last work and had the consequence that in the end, when the system was ready for extensive testing, the time ran out. This is also reflected in the quality of the presented tests and the way of working itself.
The most important lessons I have learned are that I am still in need of on an intense coaching support in the areas of time management and work planning, and that I still need to improve a lot in these fields to gain a better work balance.
Finally, it can be said that the topic has challenged me enormously. However, I would say in a positive way, since it gives me the aspiration to further develop it in order to optimize the system and to gain an even better knowledge of the individual topics.

9.3. An Epilogue to the Master of Science in Engineering

After three years, I am now completing my Master of Science in Engineering. At the end of my Bachelor of Science in electrical Engineering, I was sure to never start a master study. Luckily, I was fortunate to start working at the university of applied science, where they convinced me to pursue a master's degree. During this master study I have gained further knowledge in embedded hardware and software and started the interest in embedded linux and backend integrations. But not only in the technical aspect, also on a personal level I have gathered a lot of new experiences that will be important for the future.
I would like to thank my advisor, Erich Styger, for the support I received during this time. Without this good collaboration I would not have as this many positive experiences.

List of Figures

List of Tables

List of Snippets

Bibliography

[1] SourceForge, Jun 2021. [Online; accessed 10. Jun. 2021].

[2] Khaled Abdelfadeel, Tom Farrell, David McDonald, and Dirk Pesch. How to make firmware updates over lorawan possible. In 2020 IEEE 21st International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM), pages 16–25, 2020.

[3] accord global. Embedded bootloader. `https://accord-global.com/embedded_bootloader.html`. (Accessed on 06/05/2022).

[4] LoRa Alliance. Lorawan® protocol with lora radio - lora alliance®. `https://lora-alliance.org/resource_hub/protocol-choice-determines-success-lorawan-drives-business-value/`. (Accessed on 05/06/2022).

[5] LoRa Alliance. Rp2-1.0.2 lorawan® regional parameters - lora alliance®. `https://lora-alliance.org/resource_hub/rp2-102-lorawan-regional-parameters/`. (Accessed on 05/06/2022).

[6] LoRa Alliance. what-is-lorawan.pdf. `https://lora-alliance.org/wp-content/uploads/2020/11/what-is-lorawan.pdf`. (Accessed on 05/06/2022).

[7] LoRa Alliance. What is lorawan® specification - lora alliance®. `https://lora-alliance.org/about-lorawan/`. (Accessed on 05/06/2022).

[8] LoRa Alliance. Lorawan remote multicast setupspecificationv1.0.0, 2018. [Online; accessed 22. May 2021].

[9] LoRa Alliance. LoRaWAN® Fragmented Data Block Transport Specification v1.0.0 - LoRa Alliance®, Nov 2020. [Online; accessed 22. May 2021].

[10] Konstantinos Arakadakis, Pavlos Charalampidis, Antonis Makrogiannakis, and Alexandros Fragkiadakis. Firmware over-the-air programming techniques for iot networks - a survey. ACM Comput. Surv., 54(9), oct 2021.

[11] Konstantinos Arakadakis, Nikolaos Karamolegkos, and Alexandros Fragkiadakis. Dfinder an efficient differencing algorithm for incremental programming of constrained iot devices. Internet of Things, 17:100482, 2022.

[12] ATMEL. Atmel at02333: Safe and secure bootloader implemen-

tation for sam3/4. http://ww1.microchip.com/downloads/en/AppNotes/Safe-and-Secure-Bootloader-Implementation-for-SAM3-4_Application-Note.pdf. (Accessed on 06/05/2022).

[13] Autoren der Wikimedia-Projekte. atomar – Wikipedia, Jun 2005. [Online; accessed 16. May 2021].

[14] Balena.io. balena - what is balena and what do we do? https://www.balena.io/what-is-balena. (Accessed on 06/17/2022).

[15] Felix Barz. Sichere lpwan-Übertragungen und iot-updates am beispiel eines smarten briefkastens. Masterthesis, 1:74, 01 2019.

[16] Albert Behr. Best uses of wireless iot communication technology | industry today. https://industrytoday.com/best-uses-of-wireless-iot-communication-technology/. (Accessed on 05/06/2022).

[17] Diego Bienz. ,lora smartsilo satellite gateway. VM1, 1:74, 01 2020.

[18] Diego Bienz. tinylacuna: A low-power universal lorawan module. MASTER THESIS, 1:108, 01 2021.

[19] Chirpstack. Quickstart debian or ubuntu - chirpstack open-source lorawan<sup>®</sup> network server. https://www.chirpstack.io/project/guides/debian-ubuntu/. (Accessed on 06/07/2022).

[20] Arm Community. Create a shared library for multiple applications for arm cortex-m4 - keil forum - support forums - arm community. https://community.arm.com/support-forums/f/keil-forum/47440/create-a-shared-library-for-multiple-applications-for-arm-cortex-m4. (Accessed on 06/14/2022).

[21] DBRGN. Lorawan data rates - blog.dbrgn.ch. https://blog.dbrgn.ch/2017/6/23/lorawan-data-rates/. (Accessed on 06/07/2022).

[22] Wei Dong, Yunhao Liu, Xiaofan Wu, Lin Gu, and C. L. Philip Chen. Elon: Enabling efficient and long-term reprogramming for wireless sensor networks. In ACM Transactions on Embedded Computing Systems, pages 49–60, 2010.

[23] Ahmed Elsalahy. elsalahy/test-fuota-server: An experimental java script test server used for prototyping firmware updates over the air (fuota) on the things stack cloud. https://github.com/elsalahy/test-fuota-server. (Accessed on 06/06/2022).

[24] Chirpstack forum. How can i use mosquitto pub a frame to a multicast group - chirpstack application server - chirpstack community forum. https://forum.chirpstack.io/t/how-can-i-use-mosquitto-pub-a-frame-to-a-multicast-group/3931. (Ac-

cessed on 06/15/2022).

[25] Chirpstack forum. Send multicast message by using mqtt - chirpstack application server - chirpstack community forum. https://forum.chirpstack.io/t/send-multicast-message-by-using-mqtt/11764/5. (Accessed on 06/15/2022).

[26] TTN Forum. Poor rssi in sx1276 - end devices (nodes) - the things network. https://www.thethingsnetwork.org/forum/t/poor-rssi-in-sx1276/12478/10. (Accessed on 06/14/2022).

[27] FreeRTOS. Freertos - market leading rtos (real time operating system) for embedded systems with internet of things extensions. https://www.freertos.org/index.html. (Accessed on 06/08/2022).

[28] Sakshama Ghoslya. All about lora and lorawan: Lora decoding. https://www.sghoslya.com/p/lora_9.html. (Accessed on 05/06/2022).

[29] gRPC. grpc. https://grpc.io/. (Accessed on 06/15/2022).

[30] Jialuo Han and Jidong Wang. An enhanced key management scheme for lorawan. Cryptography, 2:34, 11 2018.

[31] HOPERF. Rfm96 433/470mhz rf transceiver module, long range wireless transceiver module _lora long range transceiver module | hoperf. https://www.hoperf.com/modules/lora/RFM96.html. (Accessed on 06/02/2022).

[32] iot analytics. State of iot 2021: Number of connected iot devices growing 9% to 12.3 b. https://iot-analytics.com/number-connected-iot-devices/. (Accessed on 05/06/2022).

[33] janjongboom. janpatch, Jun 2021. [Online; accessed 10. Jun. 2021].

[34] janjongboom. jdiff-js, Jun 2021. [Online; accessed 10. Jun. 2021].

[35] S. Jaouhari and E. Bouvet. Toward a generic and secure bootloader for iot device firmware ota update. In 2022 International Conference on Information Networking (ICOIN), pages 90–95, 2022.

[36] Ondrej Kachman. Effective multiplatform firmware update process for embedded low-power devices. 2018.

[37] Ondrej Kachman and Marcel Balaz. Efficient patch module for single-bank or dual-bank firmware updates for embedded devices. In 2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), pages 1–6, 2020.

[38] kaspersky. New iot-malware grew three-fold in h1 2018 | kaspersky. https://www.kaspersky.com/about/press-releases/2018_new-iot-malware-grew-three-fold-in-h1-2018. (Accessed on 05/06/2022).

[39] LinksLabs. Lora: A breakdown of what it is & how it works | link labs. https://www.link-labs.com/blog/what-is-lora. (Accessed on 05/06/2022).

[40] LittleFS-projects. littlefs-project/littlefs: A little fail-safe filesystem designed for microcontrollers. https://github.com/littlefs-project/littlefs. (Accessed on 06/08/2022).

[41] LoRa Alliance. Homepage - LoRa Alliance®, Apr 2021. [Online; accessed 30. Apr. 2021].

[42] LoraAlliance. Lorawan application layer clock synchronization. https://lora-alliance.org/wp-content/uploads/2020/11/application_layer_clock_synchronization_v1.0.0.pdf. (Accessed on 06/06/2022).

[43] LoraAlliance. Ts004-2.0.0 fragmented data block transport. https://resources.lora-alliance.org/document/ts004-2-0-0-fragmented-data-block-transport. (Accessed on 06/29/2022).

[44] LoraAlliance. Ts005-2.0.0 remote multicast setup. https://resources.lora-alliance.org/document/ts005-2-0-0-remote-multicast-setup. (Accessed on 06/29/2022).

[45] LoraAlliance. Ts006-1.0.0 firmware management protocol. https://resources.lora-alliance.org/document/ts006-1-0-0-firmware-management-protocol. (Accessed on 06/29/2022).

[46] Miguel Luis. Fuota multicast mic_fail · discussion #1317 · lora-net/loramac-node. https://github.com/Lora-net/LoRaMac-node/discussions/1317#discussioncomment-2766326. (Accessed on 06/06/2022).

[47] maxim integrated. Ds3232 ds. https://datasheets.maximintegrated.com/en/ds/DS3232.pdf. (Accessed on 06/29/2022).

[48] MBED. Sx126xmb2xas | mbed. https://os.mbed.com/components/SX126xMB2xAS/. (Accessed on 05/24/2022).

[49] NETTIGO. original.jpg (1920×1080). https://nettigo.eu/system/images/3580/original.JPG?1578141142. (Accessed on 06/02/2022).

[50] NXP. Lpcxpresso55s16 development board | nxp semiconductors. https://www.nxp.com/design/development-boards/lpcxpresso-boards/lpcxpresso55s16-development-board:LPC55S16-EVK. (Accessed on 05/24/2022).

[51] NXP. Mcuxpresso sdk | software development for kinetis, lpc, and i.mx mcus | nxp semiconductors. https://www.nxp.com/design/software/development-software/mcuxpresso-software-and-tools-/mcuxpresso-software-development-kit-sdk:MCUXpresso-SDK. (Accessed

on 06/08/2022).

[52] Corsin Obrist. Tardigrade mobile update. VM1, 1:93, 07 2021.

[53] Corsin Obrist. Secure firmware update. VM2, 1:97, 01 2022.

[54] R. K. Panta and S. Bagchi. Hermes: Fast and energy efficient incremental code updates for wireless sensor networks. In IEEE INFOCOM 2009, pages 639–647, 2009.

[55] QOITECH. Products - otii by qoitech. `https://www.qoitech.com/products/`. (Accessed on 06/15/2022).

[56] Brecht Reynders and Sofie Pollin. Chirp spread spectrum as a modulation technique for long range communication. In 2016 Symposium on Communications and Vehicular Technologies (SCVT), pages 1–5, 2016.

[57] SEGGER. J-link rtt – real time transfer. `https://www.segger.com/products/debug-probes/j-link/technology/about-real-time-transfer/`. (Accessed on 06/13/2022).

[58] Semtech. Lora-net/loramac-node: Reference implementation and documentation of a lora network node. `https://github.com/Lora-net/LoRaMac-node`. (Accessed on 01/13/2022).

[59] Semtech. Loramac-node/src/apps/loramac/fuota-test-01 at master · lora-net/loramac-node. `https://github.com/Lora-net/LoRaMac-node/tree/master/src/apps/LoRaMac/fuota-test-01`. (Accessed on 06/07/2022).

[60] Semtech. Loramac: Porting guide. `https://stackforce.github.io/LoRaMac-doc/LoRaMac-doc-v4.5.1/_p_o_r_t_i_n_g__g_u_i_d_e.html`. (Accessed on 01/14/2022).

[61] SEMTECH. Mcu memory management | developer portal. `https://lora-developers.semtech.com/documentation/tech-papers-and-guides/mcu-memory-management`. (Accessed on 06/02/2022).

[62] Semtech. Quick start — basic station 2.0.5 june 2020 documentation. `https://doc.sm.tc/station/quick.html`, 1 2020. (Accessed on 01/04/2022).

[63] Sensirion. Sensirion_humidity_sensors_sht3x_datasheet_digital.pdf. `https://sensirion.com/media/documents/213E6A3B/61641DC3/Sensirion_Humidity_Sensors_SHT3x_Datasheet_digital.pdf`. (Accessed on 06/29/2022).

[64] Erich Sryger. Mcu on eclipse | everything on eclipse, microcontrollers and software. `https://mcuoneclipse.com/`. (Accessed on 06/08/2022).

[65] Erich Styger. Erichstyger/mcuoneclipselibrary: Library of mcuoneclipse components. `https://github.com/ErichStyger/McuOnEclipseLibrary`. (Accessed on

06/08/2022).

[66] Erich Styger. Fatfs, minini, shell and freertos for the nxp k22fn512 | mcu on eclipse. `https://mcuoneclipse.com/2020/05/20/fatfs-minini-shell-and-freertos-for-the-nxp-k22fn512/`. (Accessed on 06/13/2022).

[67] Erich Styger. Lorawan with nxp lpc55s16 and arm cortex-m33 | mcu on eclipse. `https://mcuoneclipse.com/2021/12/30/lorawan-with-nxp-lpc55s16-and-arm-cortex-m33/`. (Accessed on 01/14/2022).

[68] Erich Styger. Contributing an iot lorawan raspberry pi rak831 gateway to the things network | mcu on eclipse. `https://mcuoneclipse.com/2019/02/25/contributing-an-iot-lorawan-raspberry-pi-rak831/`, 2 2019. (Accessed on 01/03/2022).

[69] TTN. Rak831 | the things network. `https://www.thethingsnetwork.org/docs/gateways/rak831/`, 5 2018. (Accessed on 01/03/2022).

[70] TTN. Device Classes, May 2021. [Online; accessed 22. May 2021].

[71] TTN. Regional Parameters, Apr 2021. [Online; accessed 1. May 2021].

[72] TTN. Security, May 2021. [Online; accessed 24. May 2021].

[73] TTS. Getting started | the things stack for lorawan. `https://www.thethingsindustries.com/docs/getting-started/`, 11 2021. (Accessed on 01/04/2022).

[74] winbond. Serial nor flash - code storage flash memory - winbond. `https://www.winbond.com/hq/product/code-storage-flash-memory/serial-nor-flash/?__locale=en`. (Accessed on 06/29/2022).

[75] winbond. W77q secure flash memory - trustme - winbond. `https://www.winbond.com/hq/product/trustme/w77q-trustme-secure-flash-memory/?__locale=en`. (Accessed on 06/29/2022).

A. Appendix

The appendix starts after this page. It contains the following documents in exact order:

- Problem definition
- Time scheduled
- Schematic of demonstrator node
- LoRaWAN datarates table
- Python script for multicast key derivation

Further, there is an electronic appendix that additionally contains the following files:

- Complete Documentation as PDF
- Test folder with binary files and log files
- Mid-Presentation as PDF
- End-Presentation as PDF
- Source-Code FUOTA app as ZIP
- Source-Code Bootloader as ZIP
- KiCAD Project demonstrator node as ZIP

MASTER OF SCIENCE
IN ENGINEERING

Lucerne University of
Applied Sciences and Arts
HOCHSCHULE
LUZERN
Technik & Architektur

MSE – Master Thesis

Horw, 16. Juni 2022

Aufgabenstellung für:

	Corsin Obrist____________________	(Masterstudent/-in)
	Information and Communication Technologies	(Fachgebiet/Profil)
von	Prof. Erich Styger________________	(Advisor/Advisorin)
	Dr. Christian Vetterli_____________	(Experte/Expertin)
	_______________________________	(E-Mail-Adresse Experte/Expertin)

1. **Arbeitstitel**

Parallel Multi-Device firmware update over the air with a LoRaWAN network

2. **Fremdmittelfinanziertes Forschungs-/Entwicklungsprojekt**

SmartIA

3. **Industrie-/Wirtschaftspartner**

Name des Industrie-/Wirtschaftspartners: SensDRB GmbH

Anschrift des Industrie-/Wirtschaftspartners: Technopark Luzern, Platz 4, 6039 Root D4

Kontaktperson beim Industrie-/Wirtschaftspartner (inkl. E-Mail-Adresse): Christian Di Battista, christian.dibattista@awaptec.ch

4. **Fachliche Schwerpunkte**

Mikrocontroller Firmware und Software

LoRa, LoRaWAN

Drahtlose Datenkommunikation

5. **Inhalt**

Die Firma SensDRB entwickelt und betreibt Geräte im Bereich IoT, welche die Umgebung, Infrastruktur oder Lager von Lebensmitteln oder Getreide überwachen. Da diese Geräte oft in Gebieten ohne normale Internet oder Mobilfunk eingesetzt werden, kommunizieren diese mit der LoRa Modulation im LoRaWAN Netzwerk, welches auch über Satelliten möglich ist. LoRaWAN *erlaubt nur kleine Datenmengen mit niedriger Bandbreite, und trotzdem müssen die Geräte im Feld mit neuer Firmware programmiert werden können, z.B. für Patches, Security Updates oder neue Funktionen. In früheren Arbeiten wurde untersucht, wie man trotz der Einschränkungen ein Update durchführen könnte. Das Ziel ist es, die Möglichkeit eines FUOTA (Firmware Update Over The Air) über das LoRaWAN zu realisieren. Zur Unterstützung kann ein mobiler Update Server eingesetzt werden. Da die Geräte im Feld oft über kleine Energiespeicher verfügen, soll auf einen möglichst kleinen Energieverbrauch geachtet werden.*

***Recherche**: Zuerst soll eine Recherche zum aktuellen Stand der Technik gemacht werden, inbesondere in den Bereichen Multicast, Fragmented Data Block Protocol und Patch Update für Microkontroller.*

***Konzepte**: Ein Multicast Ansatz soll gleichzeitig mehrere Geräte mit Informationen versorgen. Hierzu bedarf es einer robusten Fehlerbehandlung (Übertragungsfehler, verlorene Pakete). Die Speicheraufteilung und Verwendung des Speichers auf dem Zielgerät inkl. möglichem externen Speicher soll evaluiert und auf die Realisierbarkeit überprüft werden. Die Übertragung und Speicherung der Daten auf dem Gerät soll gemäss dem Stand der Technik erfolgen (SE, Hash, CRC, Verschlüsselung). Es sollte berücksichtigt werden, dass es im Feld Geräte mit verschiedenen Konfigurationen oder Firmware Versionen vorhanden sein können.*

***Realisierung**: Um den Ansatz zu validieren, soll dies mit drei oder mehr Knoten überprüft werden. Wegen der limitierten Verfügbarkeit von Halbleitern kann auf EVK Boards (z.B. LPC55S16-EVK) oder Module (z.B. SemTech SX126x Shield) zurückgegriffen werden, aber nach Bedarf auch neu entwickelt werden. Der FUOTA Server soll skalierbar sein und auch viele Geräte unterstützen können. Neben der eigentlichen Funktion sollen Messungen zur Update-Performance und Energieverbrauch gesammelt und interpretiert werden.*

6. Fachliteratur/Web-Links/Hilfsmittel

tinyLacuna, A Low-Power Universal LoRaWAN Module (MT Diego Bienz, Jan 2021)

Tardigrade Mobile Update (Corsin Obrist, Juni 2021)

Secure Firmware Update (Corsin Obrist, Jan 2022)

https:/ thethingsnetwork.org

LPC55S16-EVK mit SX126x Transceiver

7. Durchführung der Arbeit

Termine

Start der Arbeit:	Mo, 21.02.2022 (Semesterbeginn FS2022)
Abgabe der Aufgabenstellung:	bis spätestens Fr, 25.02.2022, 17.00 Uhr
Master Thesis Kolloquium:	Anfang Mai
Zwischenbesprechung:	während des Semesters
Abgabe Prüfungsexemplar:	bis spätestens Fr, 17.06.2022 um 17.00 Uhr als Upload auf ILIAS und direkt an Advisor/-in und Experte/-in
Verteidigung:	bis spätestens Mi, 29.06.2022
Meldung Grade:	bis spätestens Do, 07.07.2022 um 12.00 Uhr
Abgabe def. Master Thesis:	bis spätestens Fr, 15.07.2022 um 17.00 Uhr als Upload auf ILIAS und direkt an Advisor/-in und Experte/-in

Diplomposterausstellung: Sa, 08.07.2022
→ Weitere Termine gemäss Ablauf Master Thesis

8. Form der Abgabe der Master Thesis

Das Prüfungsexemplar der Master Thesis wird ausschliesslich online eingereicht (für Advisor/Advisorin und Experte/Expertin). Nach der Verteidigung wird die finale Version der Master Thesis im pdf_Format auf ILIAS hochgeladen und via Sekretariat BA&MA an die Bibliothek zur Archivierung gegeben. Die Angabe zur Sicherheitsstufe ist zwingend erforderlich: Öffentlich mit oder ohne Internet/intern/vertraulich, damit diese korrekt durch die Bibliothek archiviert werden kann.

9. Titelblatt

Für die Arbeit muss zwingend das von der Bibliothek vorgegebene Titelblatt verwendet werden. Dieses ist auf MyCampus abrufbar. Es besteht die Möglichkeit neben diesem noch ein eigenes Titelblatt einzufügen.

10. Selbstständigkeits- und Redlichkeitserklärung

Die Selbstständigkeits- und Redlichkeitserklärung muss zwingend zusammen mit der Master Thesis abgegeben werden. Dieses Dokument darf jedoch nicht in die Master Thesis eingebunden werden, sondern muss als separates Dokument mitabgeben werden. Es ist die Vorlage der Bibliothek zu verwenden. Diese ist auf MyCampus abrufbar.

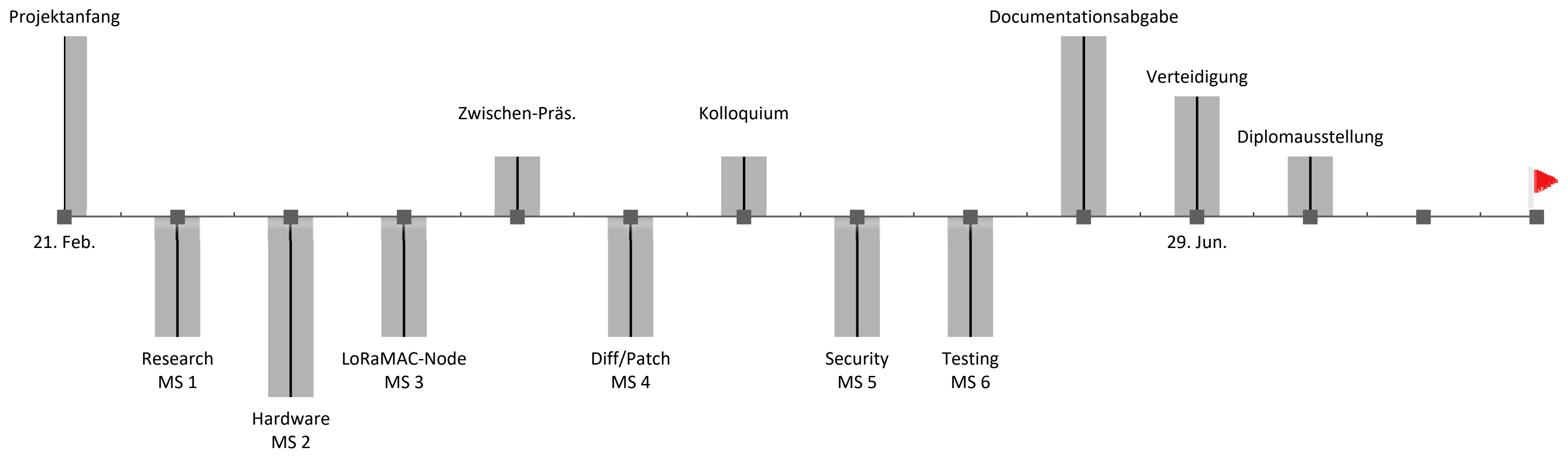
11. Zusätzliche Bemerkungen

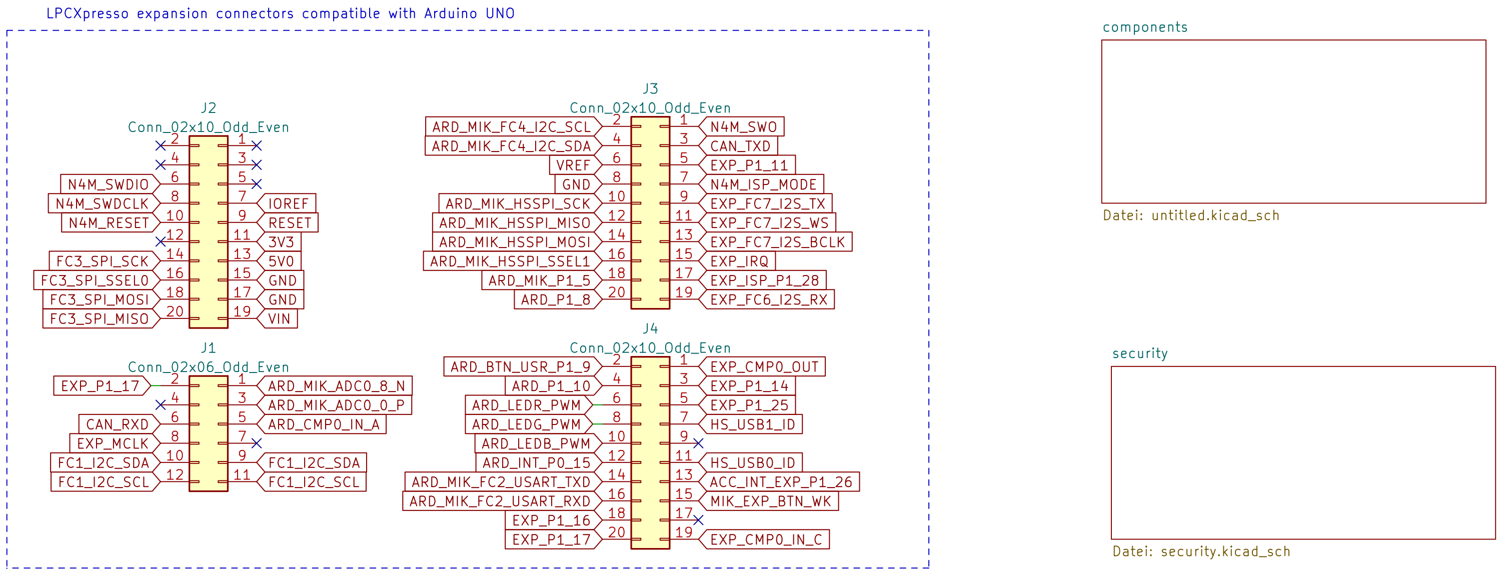
Betreffend Geheimhaltung und Rechte am Geistigen Eigentum ist die Vereinbarung zwischen dem Student bzw. der Studentin, der HSLU und dem Industrie-/Wirtschaftspartner massgeblich. Eine Vorlage hierfür ist auf MyCampus abrufbar.

Ort, Datum

Advisor/Advisorin	Experte/Expertin	Student/Studentin
______________________	______________________	______________________

Die Aufgabenstellung der Master Thesis muss mit allen Unterschriften bis **spätestens Ende der 1. Woche des Kontaktstudiums dem Sekretariat BA&MA via mse@hslu.ch** abgeben werden. Änderungen können danach jeweils per E-Mail übermittelt werden.


Projektanfang
Documentationsabgabe
Verteidigung
Zwischen-Präs.
Kolloquium
Diplomausstellung
21. Feb.
29. Jun.
Research
MS 1
Hardware
MS 2
LoRaMAC-Node
MS 3
Diff/Patch
MS 4
Security
MS 5
Testing
MS 6



HSLU T&A

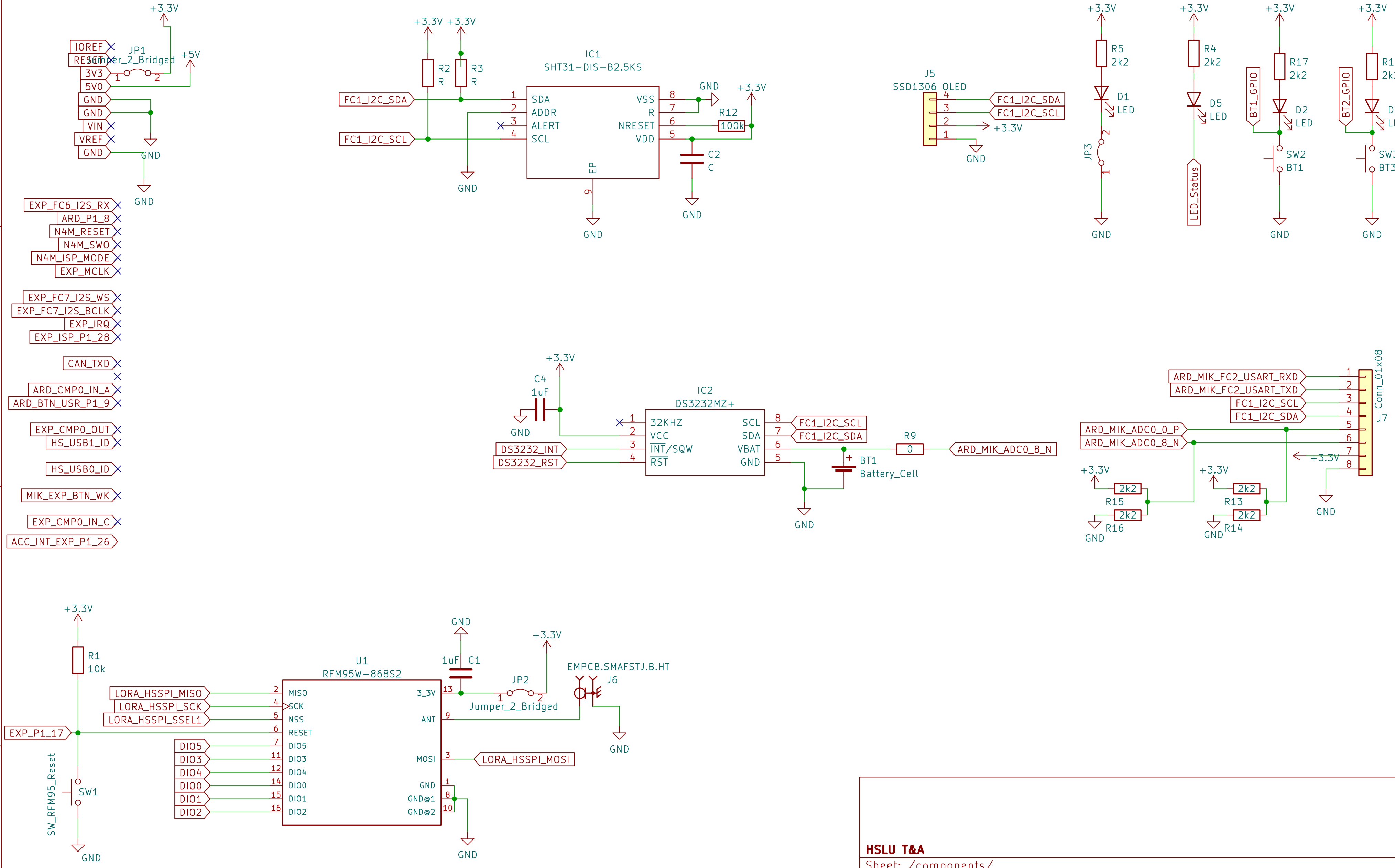
Sheet: /

File: peripherieHat.kicad_sch

Title: MT Corsin Obrist EVK-LPC55S16 LoRaWAN Shield

Size: A4 | Date: 2022-06-16 | **Rev:**

KiCad E.D.A. kicad (6.0.2) | Id: 1/3

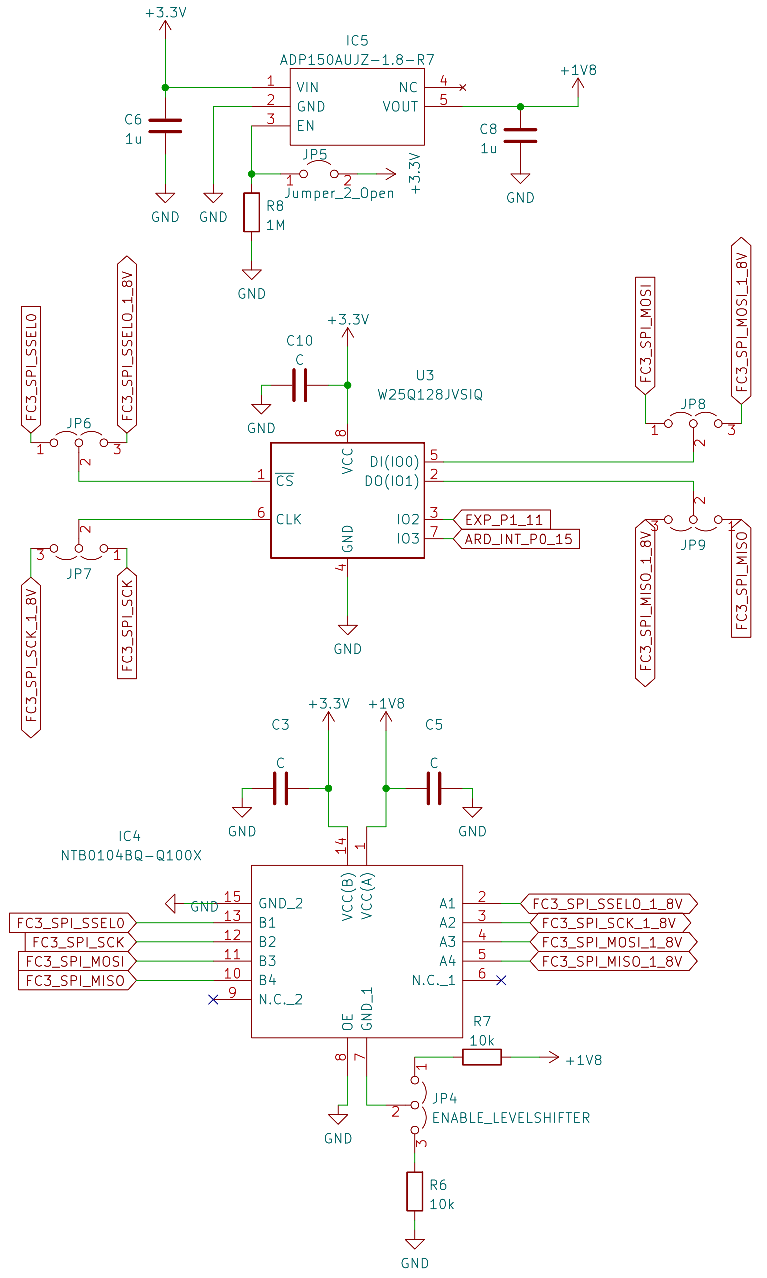


HSLU T&A

Sheet: /components/
File: untitled.kicad_sch

Title: MT Corsin Obrist EVK-LPC55S16 LoRaWAN Shield

Size: A4	Date: 2022-06-16	Rev:
KiCad E.D.A. kicad (6.0.2)		Id: 2/3



HSLU T&A

Sheet: /security/
File: security.kicad_sch

Title: MT Corsin Obrist EVK-LPC55S16 LoRaWAN Shield

Size: A4 Date: 2022-06-16 **Rev:**
KiCad E.D.A. kicad (6.0.2) Id: 3/3

A.1. LoRaWAN

DataRate	Modulation	SF	BW	bit/s
0	LoRa	12	125	250
1	LoRa	11	125	440
2	LoRa	10	125	980
3	LoRa	9	125	1760
4	LoRa	8	125	3125
5	LoRa	7	125	5470

Table A.1.: Datarates LoRaWAN [21].

A.2. Python script for multicast key derivation

```python
from Crypto.Hash import CMAC
from Crypto.Cipher import AES
import semantic_version


def decrypt(buffer: bytearray, root_key: bytes) -> bytearray:
    cipher = AES.new(root_key, AES.MODE_ECB)
    return cipher.decrypt(buffer)


def lorawan_derive_key(buffer: bytearray, root_key: bytes) -> bytearray:

    cipher = AES.new(root_key, AES.MODE_ECB)
    return cipher.encrypt(buffer)


def lorawan_derive_mc_root_key(lorawan_version: semantic_version.Version,
   root_key: str) -> str:

    key_in = bytes.fromhex(root_key)

    comp_base = bytearray(16)

    if lorawan_version.minor == 1:
        comp_base[0] = 0x20

    key_out = lorawan_derive_key(comp_base, key_in)

    return "".join("{:02X}".format(x) for x in key_out)


def lorawan_derive_mc_ke_key(root_key: str) -> str:

    key_in = bytes.fromhex(root_key)

    comp_base = bytearray(16)

    key_out = lorawan_derive_key(comp_base, key_in)

    return "".join("{:02X}".format(x) for x in key_out)


def lorawan_derive_mc_session_key_pair(mc_addr: int, root_key: str):
    key_in = bytes.fromhex(root_key)

    comp_base_app_s = bytearray(16)
    comp_base_nwk_s = bytearray(16)

    comp_base_app_s[0] = 0x01
    comp_base_app_s[1] = mc_addr & 0xFF
    comp_base_app_s[2] = (mc_addr >> 8) & 0xFF
    comp_base_app_s[3] = (mc_addr >> 16) & 0xFF
    comp_base_app_s[4] = (mc_addr >> 24) & 0xFF

    comp_base_nwk_s[0] = 0x02
    comp_base_nwk_s[1] = mc_addr & 0xFF
    comp_base_nwk_s[2] = (mc_addr >> 8) & 0xFF
    comp_base_nwk_s[3] = (mc_addr >> 16) & 0xFF
    comp_base_nwk_s[4] = (mc_addr >> 24) & 0xFF

    app_s_key = lorawan_derive_key(comp_base_app_s, key_in)
    nwk_s_key = lorawan_derive_key(comp_base_nwk_s, key_in)

    return "".join("{:02X}".format(x) for x in app_s_key), "".join("{:02X}"
        .format(x) for x in nwk_s_key)


if __name__ == "__main__":
    lorawan_version = semantic_version.Version("1.0.4")

    # Inputs
    app_key: str = "000102030405060708090A0B0C0D0E0F"
    mc_key = "0102030405060708090A0B0C0D0E0F10"
    mc_addr = 0x1FFFFFF

    # Outputs
    mc_root_key = lorawan_derive_mc_root_key(lorawan_version, app_key)
    mc_ke_key = lorawan_derive_mc_ke_key(mc_root_key)
    mc_key_encrypted = "".join("{:02X}".format(x) for x in decrypt(bytes.
        fromhex(mc_key), bytes.fromhex(mc_ke_key)))
    mc_app_s_key, mc_nwk_s_key = lorawan_derive_mc_session_key_pair(mc_addr
        , mc_key)

    print("LoRaWAN " + str(lorawan_version))

    print("AppKey          : " + app_key)
    print(f"McAddr          : 0x{mc_addr:08X}")
    print("McRootKey       : " + mc_root_key)
    print("McKeKey         : " + mc_ke_key)
    print("McKey           : " + mc_key)
    print("McKeyEncrypted  : " + mc_key_encrypted)
    print("McAppSKey       : " + mc_app_s_key)
    print("McNwkSKey       : " + mc_nwk_s_key)

    print()
    lorawan_version = semantic_version.Version("1.1.0")

    # Inputs
    app_key: str = "000102030405060708090A0B0C0D0E0F"
    mc_key = "0102030405060708090A0B0C0D0E0F10"
    mc_addr = 0x1FFFFFF

    # Outputs
    mc_root_key = lorawan_derive_mc_root_key(lorawan_version, app_key)
    mc_ke_key = lorawan_derive_mc_ke_key(mc_root_key)
    mc_key_encrypted = "".join("{:02X}".format(x) for x in decrypt(bytes.
        fromhex(mc_key), bytes.fromhex(mc_ke_key)))

    mc_app_s_key, mc_nwk_s_key = lorawan_derive_mc_session_key_pair(mc_addr
        , mc_key)

    print("LoRaWAN " + str(lorawan_version))

    print("AppKey          : " + app_key)
    print(f"McAddr          : 0x{mc_addr:08X}")
    print("McRootKey       : " + mc_root_key)
    print("McKeKey         : " + mc_ke_key)
    print("McKey           : " + mc_key)
    print("McKeyEncrypted  : " + mc_key_encrypted)
    print("McAppSKey       : " + mc_app_s_key)
    print("McNwkSKey       : " + mc_nwk_s_key)
```

Listing A.1: Multicast key generator script [46]