Department of Industrial Engineering

Mechatronics Engineering course

&

Dipartimento di Ingegneria e Scienza dell'Informazione

Ingegneria dell'Informazione e delle Comunicazioni

Embedded System Project

FOTA via LoRa

Firmware Over The Air via LoRa implementation

Professors
Davide Brunelli
(Matteo Nardello)

Students
Spadetto Matteo [214352]
Damiano Tessaro [193716]

Accademic year 2019/2020

Contents

Summary

The goal of this project is to design a system able to update an STM32L476RG firmware via LoRa communication.
Technologies are moving to the IoT concept, that means smaller, more reliable and long battery life devices. In this context devices are connected in a network and positioned, not only far from each other, but, sometimes, also far from the master. This means that if an update is needed could be not possible or not convenient to reach all the devices to manually update them. LoRa protocol, as the name suggests, is a long range communication so it is a good solution for this kind of application. Moreover, used with an STM32L4 device the power consumption decreases significantly.
Starting from the simplest "Ping-Pong" LoRa example, a new firmware with the sender and receiver functionalities were designed. After some tests with this protocol, the best communication parameters for the application were set. Then, leaving aside LoRa, a study on STM32L4 memory and update procedures was performed in order to understand which operations the new softwares must implement for:

- sending the new firmware after a build;
- receiving the new firmware on the device for the update;
- writing the new firmware into the receiver memory;
- executing the new firmware stored in the memory.

1 Prerequesites

The project was developed using two Nucleo **STM32L476RG Rev 4** by STMicroelectronics. They are the most suitable boards due to their stock bootloader with useful built-in functionalities. Not all bootloaders by STMicroelectronics support so many protocols for the firmware update. Moreover, the L4 series is a low power microcontroller. This feature is important for this kind of applications. LoRa can send packages very far away so it is preferable to have less power consumption on the receiver device giving him more battery life. Even if the first part of the project needs particular features, as SPI initialized in the bootloader, the final system does not depend on which STM32 bootloader is used.
The two boards, the transmitter and the receiver, were connected to a **RFMxx** LoRa module, one for each of them, with relative antennas.
Also jumper wires and usb cables were necessary for a faster prototyping process and firmware update via UART. In order to implement this application, the software needed were:

1. **SystemWorkbanch** for STM32: due to the fact that the "PingPong" LoRa example is built inside this IDE, it is easier to continue the work here. Moreover, it is not possible, for this example to find the ".ioc" file to easily open it in STM32CubeMx and re-build the project for another IDE.

2. **STM32CubeMx**: with this application is possible to generate new STM32 projects from scratch with all the configurations needed. The new firmware to send is initialized with this software before importing it in a comfortable IDE.

3. **VSCode**: thanks to STM32CubeMx it is possible to import the projects in different IDE, in this case VSCode is a good solution. Here it is possible to do almost everything that SystemWorkbanch by STMicroelectronics does but in a more user-friendly way. To be able to build and run the firmware in VSCode, the "stm32-for-vscode" extension must be installed. VSCode, in this application is used also to implement some C++ and Python scripts that will be described in the next pages.

4. **STM32CubeProgrammer**: this tool by STMicroelectronics gives the possibility to:

   - erase the chip memory;
   - read memory at specific addresses;
   - write into memory at specific address.

   This tool is necessary to debug the bootloader operations before sending the firmware directly via LoRa.

5. Other softwares and dependencies:

   - Java: almost all STMicroelectronics applications need it to run;
   - Opengdb: for firmware update and debug;
   - Github: for documentations, demo videos and possibility to clone this project (GitHub project). Other github projects useful to understand some processes are: [7], [3], [2], [12], [11].

2 LoRa

2.1 Introduction

LoRa (Long Range) is a low-power wide-area network (LPWAN) protocol developed by Semtech. It is based on **spread spectrum modulation** techniques derived from chirp spread spectrum (CSS) technology, so the symbols are encoded with increasing (or decreasing) frequency. This technique allows the signal to propagate far away. It is possible to see an example of LoRA communication in figure 2.1.

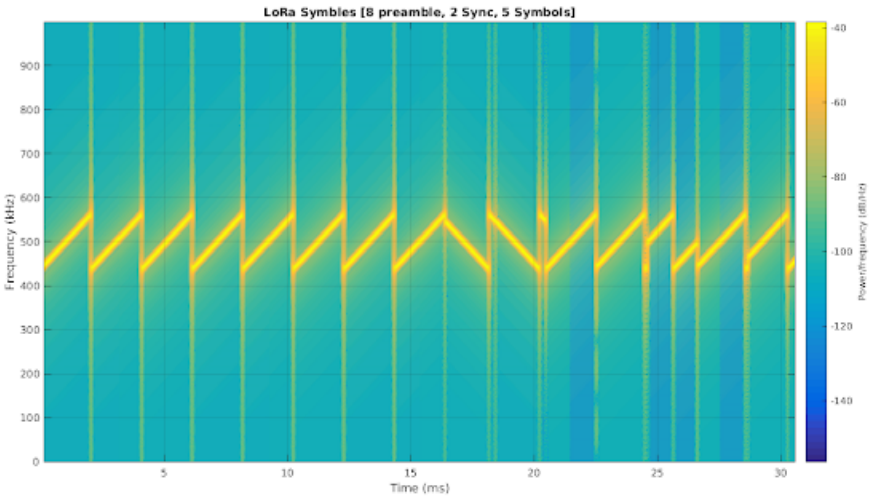

Figure 2.1: LoRa protocol

A LoRa device can be configured to use different Transmission Power (TP), Carrier Frequency (CF), Spreading Factor (SF), Bandwidth (BW) and Coding Rate (CR) to tune link performances and energy consumption.

Transmission Power: TP on a LoRa radio can be adjusted from −4 dBm to 20 dBm, in 1 dB steps, but because of hardware implementation limits, the range is often limited to 2 dBm to 20 dBm;

Carrier Frequency: CF is the centre frequency that can be programmed in steps of 61 Hz between 137 MHz to 1020 MHz. Depending on the particular LoRa chip, this range may be limited from 860 MHz to 1020 MHz.

Spreading Factor: SF is the ratio between the symbol rate and chip rate. A higher spreading factor increases the Signal to Noise Ratio (SNR), and thus sensitivity and range, but also increases the air time of the packet. Spreading factor can be selected from 6 to 12. The number of chips per symbol is calculated as 2^{SF} chips/symbol.

Bandwidth: BW is the width of frequencies in the transmission band. Higher BW gives a higher data rate (thus shorter time on air), but a lower sensitivity (because of integration of additional noise). Data are sent at a chip rate; a bandwidth of 125 kHz corresponds to a chip rate of 125 kcps. Although the bandwidth can be selected in a range of 7.8 kHz to 500 kHz, a typical LoRa network operates at either 500 kHz, 250 kHz or 125 kHz.

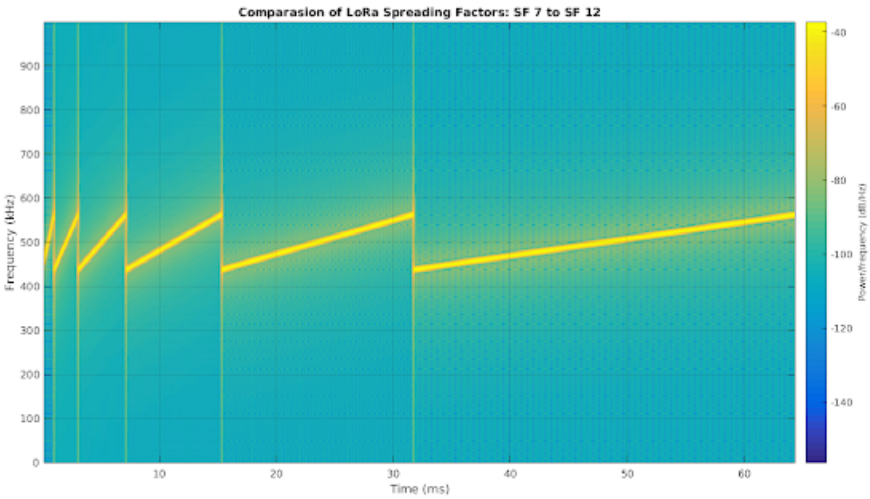

Figure 2.2: Spreading Factor comparison

Coding Rate: CR is the Forward Error Correction (FEC) rate used by the LoRa modem that offers protection against bursts of interference, and can be set to either [1...4].A higher CR offers more protection, but increases time on air.¡

With this parameter is possible to calculate the Nominal Bit Rate [6].

$$R_b = SF \cdot \frac{\left[\frac{4}{4+CR}\right]}{\left[\frac{2^{SF}}{BW}\right]} \quad [bit/s] \tag{2.1}$$

However only with (2.1) is not possible to determinate the time of whole communication because there are other bits over the payload.

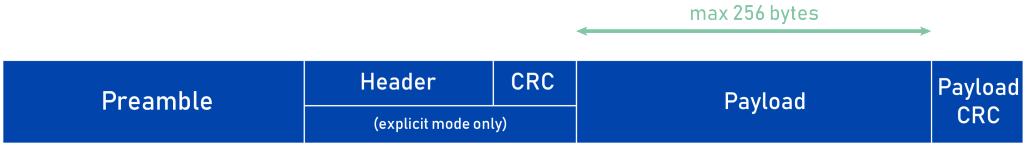

Figure 2.3: LoRa packet formatting [5]

2.2 LoRa tests

2.2.1 Hardware

In this test has been used some hardware devices.

- Two Nucleo-L476RG (Figure 2.4);
- Two LoRa RFM69HCW Radio module (Figure 2.5);

2.2.2 Implementation

The LoRa communication protocol is not so easy to implement from scratch, for this reason the project started from the "**Ping-Pong**" example where the functions needed were already implemented. Nevertheless, the final goal aimed by the firmware update was different, so changes were done as explained in the next paragraphs. In the "Ping-Pong" example, the **sender** is set as master and the **receiver** as slave. When the receiver get

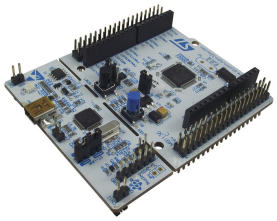

Figure 2.4: Nucleo-L476RG

Figure 2.5: Lora RFM69HCW

a “PING” message, it replays with a “PONG” to the sender. If the initial sender receive a “PONG”, it replays with a “PING” and the loop goes on. This project was used to understand the implementation of the LoRa protocol and its parameters.

After this initial step it was also used to tune the settings of the communication with the goal of establish a very reliable communication. To achieve these settings some **tests**, in different conditions, were performed. To reduce the number of tests CR and CF has been define at value CF=868 MHz and CR=4/5. When the connection established between the two devices was good enough, the example code was modified to perform the following test. The master sends some packages to the slave and, if the message is accepted from the receiver, it is stored in an array on the second device and ”messagerx” counter is incremented. At the end the difference between the sent and received messages was calculated for each different parameters configuration.

In Table 2.1 there are the results of this test and the best result is highlighted in green.

BW	SF	Power	Time [s]	Messages Rx	Messages Rx %
125	12	20	–	0	0,00%
125	11	20	–	0	0,00%
125	10	20	230	20032	100,00%
125	8	20	76	20032	100,00%
250	12	20	–	0	0,00%
250	10	20	120	19968	99,68%
250	8	20	43	19840	99,04%
500	11	20	112	19904	99,36%
500	10	20	63	19840	99,04%
500	8	20	25	20032	100,00%
125	11	18	–	0	0,00%
125	10	18	227	20032	100,00%
125	8	18	76	19968	99,68%
250	11	18	–	0	0,00%
250	10	18	118	20032	100,00%
250	8	18	42	19968	99,68%
500	11	18	–	0	0,00%
500	10	18	63	20032	100,00%
500	8	18	25	19796	98,82%
125	11	8	–	0	0,00%
125	10	8	227	19968	99,68%

Table 2.1: Parameters tested

Other consideration to choose the parameters is the energy consumption. Tough the power transmission is lower, the time of transmission is very influential. To determinate the best parameter has been used data in Figure 2.6 where the best setting is: SF=7, BW=500 kHz, CR=4/5, TP=14 dBm [4].

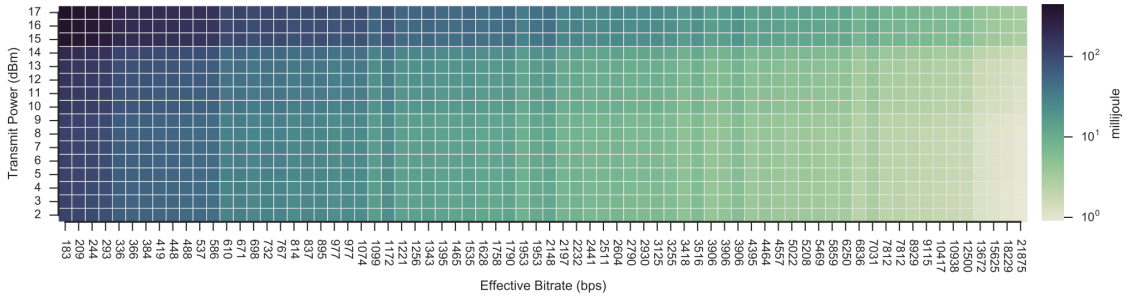

Figure 2.6: Energy consumption

The choice was made in order to not lost any message but with best energy saving. The tests were performed with a very short distance due to the spaces limitations during the test but the same procedure is valid also for long range tests. The final chosen parameters are reported in Table 2.2.

Parameter	Value
Transmition Power	14 dBm
BandWidth	500 kHz
Spreading Factor	7
Coding Rate	4/5
Carrier Frequency	868 MHz

Table 2.2: Parameters chosen

Once the LoRa parameters were chosen, other tests on the **time spent for the update** were performed sending 16000 bytes. As it is possible to see in the table 2.3 and its plot 2.7, using the configuration in table 2.2, the communication behaviour is dependent to the bytes sent in the payload. In particular there is a strong influence if the bytes sent are few. The reason is that a LoRa message is not only composed by the payload but also from other parts, before and after it, as it was shown in 2.3. Sending only few bytes means that the message is mainly storing bytes that are not the payload itself making the communication not efficient. Filling all the payload is possible to have a faster system but it reaches a limit over the 200 bytes. This is due to the fact that the payload size becomes comparable with the length of the other parts of the message so some bytes more or some less are not influencing so much the final time for the update.
Nevertheless for this project application 11 bytes of payload were more simple to handle on the receiver board, so this configuration was used. Changing the settings for a more reach message can decrease the update time of the system.

Payload bytes	Update Time [s]
11	74
23	50
43	39
63	35
123	31
243	29

Table 2.3: Payload bytes vs update time

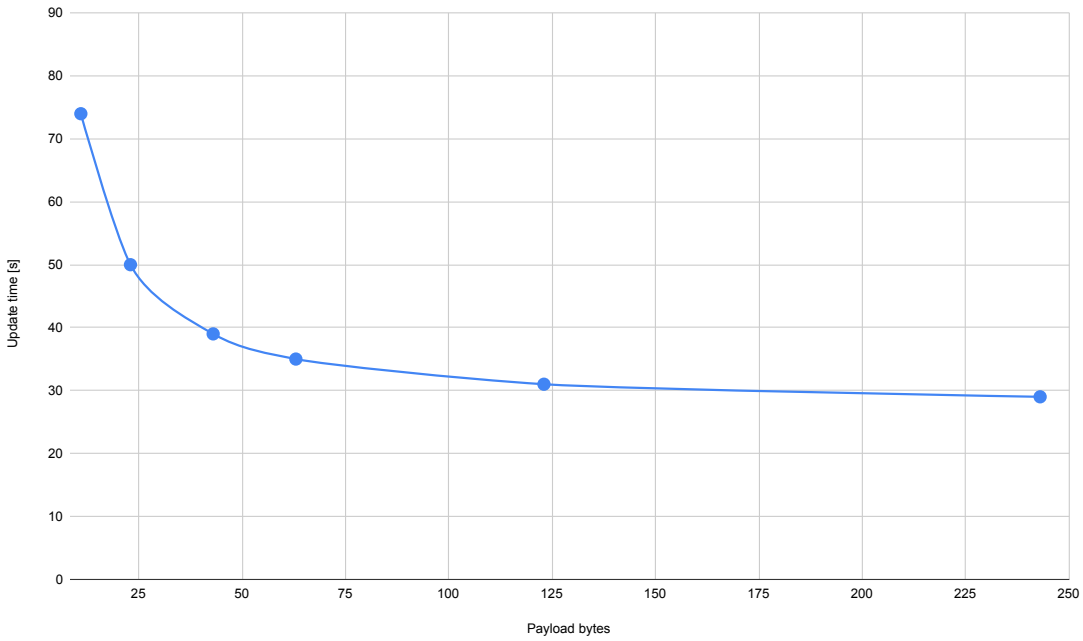

Figure 2.7: Payload bytes vs update time plot

3 Hex File and Parser

3.1 Firmware formats

How explained in the previous sections, the communication was established but the messages to send are not yet defined.
Starting from the compiler output a study of the generated files was performed. After a build request is possible to find in the "Project/build/" folder three new files:

- **project.elf**: this file stores all the firmware instructions to be written in the device memory but with additional ones. In fact this kind of file gives the possibility to debug the firmware uploaded with SystemWorkbech. These features make the file bigger than the other kind of files: for example for a simple blink it contains 265.7 kB;

- **project.bin**: this file stores all and only the firmware instructions to be written in the device memory. It is small, for the blink code it is a 10.2 kB, but difficult to handle. Text editors are not able to open this kind of file and, due to the fact that it is necessary to understand the structure of the firmware, this file is good to make the code running but not to study it;

- **project.hex**: this kind of file stores all the firmware instructions but also the addresses in which the bytes must be written to and some other information. Its size, for the blink test, is 28.9 kB so it is a good trade-off between complexity and size.

3.2 Hex File

After the introduction to the kind of files used to run the code on a STM32 system, a more detailed overview to the ".hex" file is required. This is the base of the system described in this project. The size of the file to be sent stays for the bytes to be sent to the second device and so it is directly proportional to the update time required by the system: more bytes means more time. Furthermore the ".hex" file is formatted in a known way, as described next, giving the possibility to extrapolate from it all the data needed. As it is possible to see at the link [1], the ".hex" file is readable line by line. For example, studying an HEX line it is possible to **decompose** it in such a way:

$$: LLAAAATT[DD..DD]CC$$

Where:

: is the colon that starts every Intel HEX record;

LL is the record-length field that represents the number of data bytes (DD) in the record;

AAAA is the **address** field that represents the starting address for subsequent data in the record. It must be concatenated with the address of the first line. For example if the first line gives an address of 0x0800 and the n-line 0x1234, the address in which the data has to be written is 0x08001234;

TT is the field that represents the HEX record type, which may be one of the following:

- **00** Data record;
- **01** End-of-file record;
- **02** Extended segment address record;
- **04** Extended linear address record;
- **05** Start linear address record (MDK-ARM only).

DD is a **data** field that represents one byte of data. A record may have multiple data bytes. The number of data bytes in the record must match the number specified by the LL field;

CC is the checksum field that represents the checksum of the record. The checksum is calculated by summing the values of all hexadecimal digit pairs in the record modulo 256 and taking the two's complement.

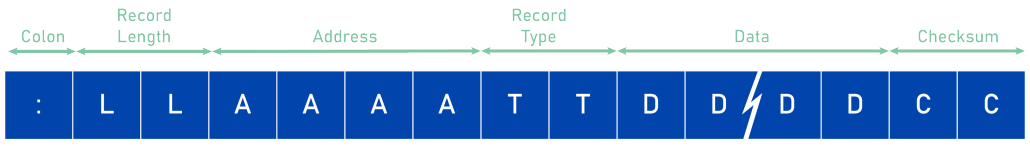

Figure 3.1: Hex format

3.3 Parser

After the ".hex" structure study, it was possible to extrapolate the addresses and data to send via LoRa to the slave in order to give to it all the necessary information to write the memory as a normal bootloader by STMicroelectronics does. To do this a **software tool** was developed in a first stage in C++ and then, for simplicity, also in Python. Passing as argument the ".hex" file to parse to the software, it opens it in read mode and line by line it decomposes the file in different arrays, one for each field of the ".hex" structure. The second part of the code is an **auto-generator** code which write an **header** file, easy to import in the STM32 C embedded language. In this header file are stored two arrays:

- ordered array of address values AAAA;
- ordered array of data values DD.

Each data value is stored in the same position of its respective address but in the DD array. With this format it is easy to handle operations as writing on memory. Moreover, just calling the Python script it is possible to have all data stored in comfortable way for different purpose.

```
#ifndef _PARSED_HEX_H
#define _PARSED_HEX_H
int arr_aaaa[735] = {0x0, 0x8, 0x10, 0x18, ..., 0x16e8, 0x16f0};
int arr_dd[1470] = {0x00800120, 0x31100008, ..., 0xFFFFFFFF, 0xFFFFFFFF};
#endif
```

4 Write memory via SPI

4.1 STM32 memory

In this project two STM32L476RG have been used. They have 1MB of flash memory where is possible to write the user code from 0x08000000 to 0x08100000.

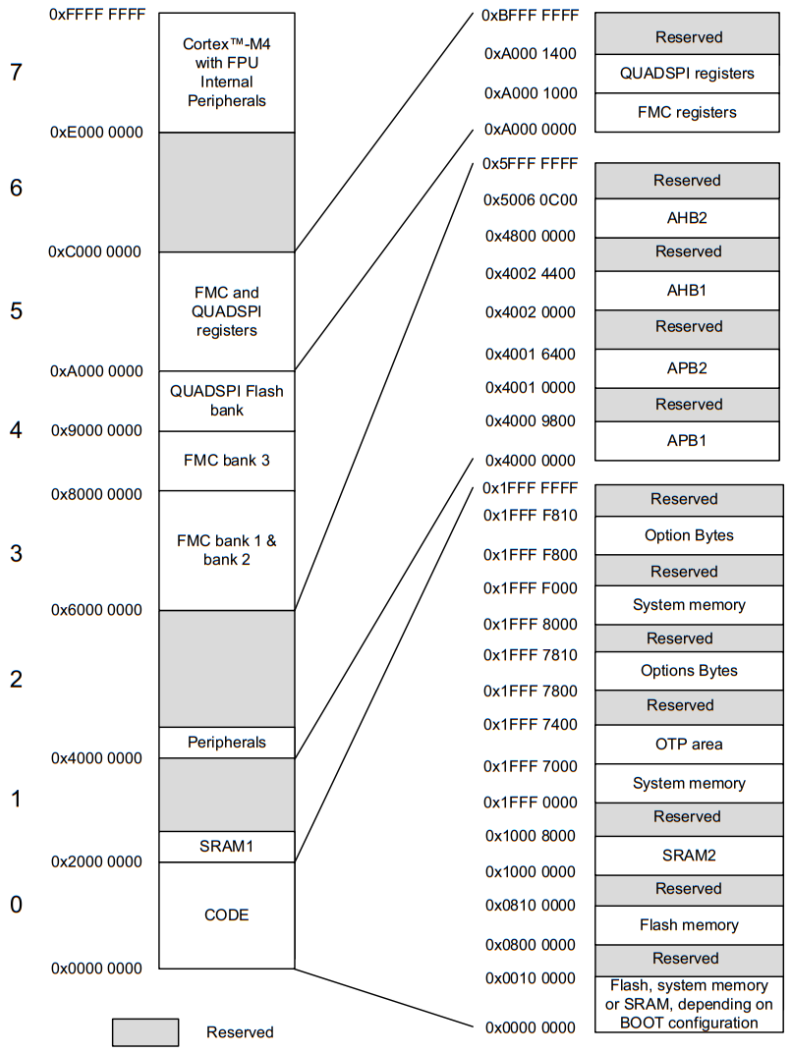

Figure 4.1: Memory mapping [10] [9]

4.2 SPI implementation

The previous paragraphs described the bases needed to implement the firmware update via LoRa but before explaining the final application, another step is needed. Due to the complexity of the system, an SPI project was designed. In this case, instead of using LoRa as communication protocol, SPI was used. The sender, with all data stored in the ".h" file generated by the Python parser, writes the information to the receiver memory via **SPI**. This protocol was chosen because it is the same that LoRa modules use to communicate with the STM32 microcontrollers, so it is easier to convert it, in a second moment, with the definitive protocol. With this method, all the problems due to the wireless communication and lost of data are not present anymore. Furthermore it is also possible to program the

memory with specific commands and understand all the functions needed to manage the memory of the STM32 based devices [8] . The SPI application has the goal to write an entire firmware to the receiver memory. After this process a reset of the receiver has to be performed to run the new firmware stored. The application is divided in different tasks:

1. first of all a **synchronization** routine must be called to establish a communication between the two devices. This is done by sending to the receiver an SPI byte:

$$0x5A$$

2. then the code sends an **ACK request** to the receiver:

$$0x00 \;\; 0xFF \;\; 0x79$$

3. if the synchronization procedure return a positive status, it is possible to **erase** all the receiver memory. This is necessary because is possible to write bytes only on the empty memory (0xFF). If this process is not done, the final firmware will be corrupted and it can not run at all. To be able to erase the memory, a specific routine must be performed:

   (a) sending the **CMD erase** frame that enable the erase capability on the receiver;

   $$0x5A \;\; 0x44 \;\; 0xBB$$

   Where 0x5A is the start of frame, 0x44 is the erase command request and 0xBB is the checksum of the message;

   (b) sending the **ACK request** of the CMD erase frame:

   $$0x00 \;\; 0x00 \;\; 0x79$$

   (c) waiting for the **ACK response** from the receiver:

   $$0xXX \;\; 0x79 \;\; 0xXX$$

   Where 0xXX is a dummy byte.

   (d) if the ACK is a positive response, then the **erase options** are sent to the receiver:

   $$0xFF \;\; 0xFF \;\; 0x00$$

   this command means that the erase will be a global mass erase (0xFFFF) and 0x00 is the checksum of the first two bytes;

   (e) sending the **ACK request** of the erase request:

   $$0x00 \;\; 0xFF \;\; 0x79$$

   (f) waiting for the **ACK response** from the receiver:

   $$0xXX \;\; 0x79 \;\; 0xXX$$

4. Now all memory is set to 0xFF, that means it is empty, and it is possible to write bytes into it. This procedure is performed in a loop where, for each cycle, **16 bytes** are written to the receiver memory. The sequence of SPI messages to be sent is the following:

   (a) sending the **CMD write** frame that enable the writing capability on the receiver:

   $$0x5A \;\; 0x31 \;\; 0xCE$$

Where 0x5A is the start of frame, 0x31 is the write command request and 0xCE is the checksum of the message;

(b) sending the **ACK request** of the CMD write frame:

$$0x00 \;\; 0x00 \;\; 0x79$$

(c) waiting for the **ACK response** from the receiver:

$$0xXX \;\; 0x79 \;\; 0xXX$$

(d) if the ACK is a positive response, then the **address** where the data will be stored is sent to the receiver:

$$0x08 \;\; 0x00 \;\; 0xB1 \;\; 0xB2 \;\; 0xCC$$

The first 4 bytes identify the address, in particular, 0x0800 is the common part of the address for all the data and it is followed by the first and second byte, stored in the ".h" file generated by the parser. 0xCC represents the checksum of the 4 first bytes;

(e) sending the **ACK request** of the address request:

$$0x00 \;\; 0x00 \;\; 0x79$$

(f) once the address is set on the target, it is possible to send the **bytes to store**. In this case 16 bytes are sent for each SPI message. The message structure is the following:

$$0x0F \;\; 0xD0 \;\; 0xD1 \;\; ... \;\; 0xD15$$

Where 0x0F is the payload size and the other 16 bytes the data. It is also necessary to send another frame with the **checksum** of all bytes of the data message, including also the frame size, in this case 0x0F:

$$0xCC$$

Where 0xCC is the checksum of the previous message sent;

(g) sending the ACK request of the data request:

$$0x00 \;\; 0x00 \;\; 0x79$$

(h) if the ACK is positive, than the loop continues with another cycle, overwise, if it is a negative response, the firmware try again to write the same address and data until it receives a positive ACK.

Thanks to this routine it was possible to successfully write an entire simple blink firmware on the receiver device memory. After an hard reset of the receiver, the firmware starts to run as if it was updated by UART as default. Mainly due to the use of jumper wires, this application is not really stable and it could be implemented to be safer, easier to use and faster. Just for stability reasons a delay was used to not risk messages overlapping but it is possible to reduce it or delete it. In fact, functions to handle the SPI were used. These functions check if the SPI is or not busy before using the protocol, if it is busy the code wait until they are not before making another request. Time tests were performed: in 50.8 seconds an entire blink firmware (16000 bytes) was written into receiver memory.

5 Overall System

The bases of the overall system are now defined and it will be more clear to understand it. As mentioned in the introduction, the goal of the project is to update a STM32 based microcontroller over the air via LoRa communication protocol. At this point the sender is able to program another STM32 but via SPI. To implement the possibility to receive firmwares via LoRa a new or modified bootloader had to be designed. In Fact the stock bootloader has different peripheral initialized for the update (SPI, CAN-bus, UART, I2C) but LoRa is not from these. Different solutions could be implemented to achieve the goal but two of them were more deeply evaluated:

- **designing a new bootloader with the LoRa extensions to replace to the stock one by STMicroelectronics**.
  The new bootloader, receiving all the addresses and data will be able to store them in the user memory and then run the firmware as a normal bootloader. This solution has different issues to overcame. The most important problem is that the STMicroelectronics bootloader is difficult to find and the memory location, in which it is stored, is writable just one time. So, also if it would be possible to change the STMicroelectronics bootloader with a custom one, all the stock functionalities would be lost and it would be not possible to restore them flashing again the stock bootloader.

- **designing a new custom bootloader with LoRa extensions to store in the user memory application**.
  With this approach all the functionalities of the stock bootloader by STMicroelectronics are preserved and it is possible, in every moment, to reuse the microcontroller as a normal one. In addition, when the custom bootloader is flashed with the normal UART method, also the LoRa functionalities are enabled. This firmware was built starting from the "Ping-Pong" example, adding to it the functionalities required by a bootloader.

For the reasons explained the choice was to use the second method described. As it is possible to see from the modified code, both the sender and receiver are implemented in the same project and the use of one or the other is done just by setting a flag.

Starting to examine the process, the master STM32, or also called the sender, reads, one by one, all the elements of the address and data arrays stored in the auto-generated header file and send them via LoRa in a specific kind of package. In this case the package is **11 bytes** long and composed like in Figure 5.1.

Where **K** is the **key** of the message. If the key, compared in the receiver as control, is not the right one, the message is discard. At the moment the key is set to 0xFF, just for prototyping purpose, but it could be, for example, the checksum of all sent bytes of the package. The important thing is that the key values in the sender and in the receiver, for that specific message, are the same. The second and third bytes are the 2 LSBytes of the address to be added to the first part 0x0802 to have a working 32 bit address in which the

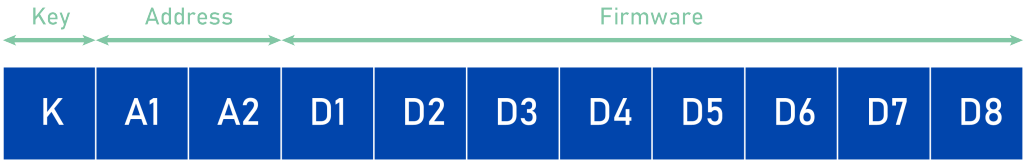

Figure 5.1: LoRa Payload

next 8 bytes of data are written. The last 8 bytes are the data to write in the memory.

On the other side, the slave, also called the receiver, reads the packages from LoRa buffer and performs some actions. The receiver uses a loop to run and for each cycle it check if there is an update and, if it's true, receives and stores **8 bytes** for cycle in memory. First of all, if an update is detected, the chip memory where the firmware will be stored is **erased**. If the key is valid, the buffer sent by the master is stored in an array. After this first step the array is parsed and the address and data are used for the next steps. As it is possible to see, LoRa packages contains just 8 bytes of data instead of 16 as in the SPI case. This choice was made because the receiver, can write in memory only 8 bytes at once. To reduce the update time it is possible to send more bytes just changing few parameters but, in any case, it is not possible to change the write in memory function that accepts maximum 8 bytes. Due to the fact that in the default user memory (starting at the address 0x08000000) the custom bootloader itself is stored, the bytes are **written** in an empty location some addresses forward. The writing procedures require around **300-400 ms** for 16000 bytes so it is an irrelevant time compared with the transmission period required by the LoRa system (around 30 s). When all firmware is received and stored, a flag is set to high and the bootloader **jumps** into the start address of new firmware and executes it. In case the bootloader understands that there is not an update sequence it jumps directly to the new address. In this way if the firmware is under update the storing procedures are performed, otherwise it means that the firmware is already stored from previous updates and it jumps to the firmware location. The jump function performs delicate and low level commands. In fact, before jumping to another address it is necessary to disable all interrupts, set the right entry point and vector table for the new firmware. If this is not done a system crash occurs.

6 Conclusion

This project was very useful not only for academic purpose but it is also innovative and has good possibility of implementation in the IoT world (mainly for long distances). During researches and studies it was possible to see that almost all the elements needed have a good documentation but very few communities tried to implement a firmware update via LoRa communication protocol. LoRa is mainly used for information transferring from a device to another and not for firmware updates.
After some studies on different LoRa parameters was possible to understand how it works and how to toggle the settings to reach the most desirable behaviours. The Python parser is a good tool to extrapolate from ".hex" files something not so readable by human and not so good to handle by machine if there is not a proper function that read them. This tool could be used not only for this project purpose but also for different applications (for example for low level debugging). The SPI application was used as necessary step to understand more deeply how the STM32 memories work and how to fully control them. Even if this software is not directly related to the final goal, it could be a good tool to update STM32 firmware via SPI, useful in many wired systems. Nevertheless some debug is needed on this application to make it more safe and reliable.
Finally the overall system was tested with good results:

- **the sender** is able to write all the desired packages in the right order to the LoRa module;
- **the receiver** is able to read all the packages with not lost of data from LoRa communication (due to the parameter set as explained in the LoRa section). All data are correctly written in the right memory addresses by the receiver;
- **the jumping function** works properly. It was tested jumping from the custom bootloader to the STMicroelectronics stock one. This causes the stock bootloader to reinitialize all the system and jump again to the custom bootloader generating a controlled loop.

The main problem that has to be resolved is to manage properly the settings of the new firmware to send for the update. Due to the fact that the STM32 firmwares are set to work in a **precise memory location** and with particular configurations, it is necessary to modify the linker files, memory partitions, addresses starting point, vector table and other configurations to make all the firmware runnable in a different memory location. Unfortunately the description of these parameters and settings is not so easy to find and for this reason it is very difficult to change them. Firmware is one of the most delicate components in a microcontroller and just one wrong byte means that anything will not run at all. Due to this problem it was not possible to modify these parameters without all the information to do it. Nevertheless some configurations of these files was tested but with bad results causing crashes of the system after the jump into the firmware location. Finally, it is possible to say that this system works as expected but with the problem of the new firmware configurations.

6.1 Future Implementation

Besides the conclusions explained above, it is possible to identify some future implementations:

- the LoRa protocol settings are set by hand before building the project. It is possible to recall the LoRa initialisation run-time for the update of the settings of the protocol itself. In this case, if the communication is not stable enough, the software can try to reach the best configurations to handle the firmware transmission;
- the LoRa firmware transmission could be more safe if some control to the integrity of the messages are performed. Due to the fact that this project aimed to design a working prototype, using the right LoRa configurations this was not necessary. In the case the settings change for speed up the system it could be better to check the packages in order to avoid lost of bytes or corrupted messages;
- it is possible, as explained before, to use a different method of encryption of the packages to avoid external and undesired messages interference;
- if the update time is not fast enough, sending more bytes a time should be a good way to speed up the overall system;
- Even if all commands to manage the memory are executed in a safe way, the final state of the memory is not checked. For example it is possible to develop a function that automates the check of memory emptiness before storing the bytes and the correct writing after data are stored;
- this project wants to be a prototype but it was developed with all the necessary components to design a tool that can make the update via LoRa easier. For example with just a line in the terminal, passing as argument the ".hex" file and the starting address, the tool could perform all the operations to auto-generate the ".h" file, store it in the sender and launch the master for the update.

Bibliography

[1] General: Intel hex file format. http://www.keil.com/support/docs/1584/#:~:text=The%20Intel%20HEX%20file%20is,code%20and%2For%20constant%20data.

[2] akospasztor. stm32 bootloader. https://github.com/akospasztor/stm32-bootloader.

[3] glegrain. Stm32 spi bootloader host. https://github.com/glegrain/STM32-SPI-Bootloader-host.

[4] Utz Roedig Martin Bor. Lora transmission parameter selection. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8271941.

[5] Semtec. Lora designer's guide, an1200.13 rev. 1. https://www.rs-online.com/designspark/rel-assets/ds-assets/uploads/knowledge-items/application-notes-for-the-internet-of-things/LoRa%20Design%20Guide.pdf.

[6] Semtec. Lora modulation basics, an1200.22 rev. 2. https://semtech.my.salesforce.com/sfc/p/#E0000000JelG/a/2R0000001OJu/xvKUc5w9yjG1q5Pb2IIkpolW54YYqGb.frOZ7HQBcRc.

[7] STMicroelectronics. Iap binary template. https://github.com/STMicroelectronics/STM32CubeH7/tree/master/Projects/STM32H743I-EVAL/Applications/IAP/IAP_Binary_Template/SW4STM32.

[8] STMicroelectronics. Spi protocol used in the stm32 bootloader. https://www.st.com/resource/en/application_note/dm00081379-spi-protocol-used-in-the-stm32-bootloader-stmicroelectronics.pdf.

[9] STMicroelectronics. Stm32 microcontroller system memory. https://www.st.com/resource/en/application_note/cd00167594-stm32-microcontroller-system-memory-boot-mode-stmicroelectronics.pdf.

[10] STMicroelectronics. Stm32l476 datasheet. https://www.st.com/resource/en/datasheet/stm32l476je.pdf.

[11] viktorvano. stm32 flash operations. https://github.com/viktorvano/STM32F3Discovery_internal_FLASH.

[12] weifengdq. stm32 flash. https://github.com/weifengdq/STM32.