Designing And Implementing An Embedded Bootloader For Secure Initialization And Update of Microcontroller Applications

Barbu Paul - Gheorghe
"Lucian Blaga" University of Sibiu
10 Victoriei Bd., Sibiu 550024
Romania
barbu.paul.gheorghe@gmail.com

ABSTRACT: *This paper aims to describe a bootloader for embedded devices. The purpose of the bootloader is to be the first to run when the microcontroller starts and to provide a way of controlling what application runs next, similarly to how bootloaders work on the desktop systems. The difference here being that the bootloader's purpose is to allow the embedded device to be much more flexible, since the application running on the microcontroller can be changed without using specialized hardware. This flexibility and the fact that no specialized hardware programmers are needed also leads to a reduction of costs for deploying and maintaining the embedded systems. These two goals, flexibility and cost reduction must be accompanied by a third one: security. The recent growth of embedded systems in the IoT (Internet of Things) domain demands that the device deployed be more secure than in the past.*

Keywords: Bootloader, Embedded device, Microcontroller, Internet of Things, IoT

Received: 14 May 2017, Revised 20 June 2017, Accepted 23 June 2017

1. Introduction

This paper extends the previous work that was presented at the Ninth International Conference on Applied Informatics, Imagination, Creativity, Design, Development - ICDD [1], by presenting along the main modules of the bootloader, also the auxiliary ones and the high level view of the system. This extended version also includes a summary of how the reusable modules are shared between the applications that compose the system together with a brief description of the security features of the microcontrollers used for developing the system described in this paper.

Working in an embedded area we have to think about how we can achieve the goals that we've set for the bootloader application.

First of all, any microcontroller that has to be programmed must be connected to a desktop system using a specialized piece of hardware, called most commonly a “programmer device”, also known as a JTAG connector [2]. Through this connector the developer of the application can perform the flashing of the application. This is the first point that the application developed in this article will tackle, the fact that an expensive hardware tool is needed for flashing most industrial microcontrollers. This issue will be solved by allowing the bootloader application to communicate through the microcontroller’s RS232 port. Hence we will develop a serial protocol for writing applications on the embedded device. According to the goals set, this protocol also has to be secure, when flashing an application on the microcontroller the flashed application’s machine code should not be available to the outside world.

The bootloader’s role is to allow the developer to flash a new application on an MCU without having to connect a JTAG device to it. This also allows for a simpler flashing procedure that the client can also perform when there is a need for a firmware upgrade. The bootloader loads applications only via RS232 and only in a specific, encrypted file format. Hence, the bootloader makes sure that the flashed application is not tampered with by third parties, in which case it should refuse to boot the application. For security reasons in order to upgrade the bootloader a classical flashing process needs to be used, eg. JTAG. The classical use case for the bootloader is when a client has devices all over the world and a firmware upgrade is necessary, then the client’s technicians can perform the upgrade by themselves, using the bootloader and the new firmware file received from the developer. This is done in order to avoid bringing in all the devices and flashing them at headquarters, then shipping them back to their client’s locations. In order for the bootloader to work properly it should receive via RS232 a valid firmware file (.fw extension). This has to be created by an application, which should also be responsible for encrypting it, we’ll call this application “Bundler”, since it creates an encrypted file that will contain the application that the bootloader will load on a microcontroller. In order for the client to be able to program the microcontroller and to use the bootloader he will need an application that will send the firmware file to the MCU, via the serial port, we will call this tool the “Flasher”, since it does the flashing of the application, similar to the classical process of programming a microcontroller using a JTAG connector.

The following chapter of the paper will describe the general way a bootloader works, then in the third section the implementation details will be presented, touching the most important parts of the designed system with its characteristic architecture as well as the auxiliary modules. The fourth section provides a high level overview of the whole system, while the last section provides the conclusions of the paper and the pitfalls of the application that can be improved in future versions.

2. How does a bootloader work?

A bootloader can be found on any system that needs to load an operating system or, in our case, a third party application. The etimology of this word stems from the word “to bootstrap”, which means “to pull oneself up by one’s bootstraps” [3].

A bootloader is, without exception, the first application that runs on a system. It is loaded in memory (in RAM - Random Access Memory) from non-volatile storage, such as hard-disks or, in the case of microcontrollers, flash memory. After loading, the bootloader (on x86 systems) generally asks the user what operating systems he wants to load. On embedded devices, this usually cannot happen because a microcontroller, by default has reduced input and output capabilities. Also a key difference on the embedded side of things is the fact that a microcontroller has a smaller non-volatile memory than a personal computer has and this allows only one application to be stored at a time on the microcontroller, so the bootloader, after start-up, will automatically choose just this application. But since one of our goals is to allow the user to change the application, so that the embedded system becomes more flexible, we will have to first erase the old application and then write the new one on the microcontroller. This way we still maintain flexibility, but also overcome the small flash memory space embedded devices have.

With this in mind, the initialization sequence of a Cortex M7 ARM microcontroller starts with reading the first two memory locations of the flash memory, which store the Stack Pointer and the Program Counter. The Stack pointer allows the MCU to know where the stack memory area starts and the Program Counter points to the next memory location of the program that needs to be executed. The following program code shows the first few lines of the the Interrupt Vector Table, the location on flash that holds the Stack Pointer, Program Counter and the Interrupt Service Routines, which are called when an interrupt occurs on the microcontroller.

_ _isr_vector:

- long _ _StackTop

- long Reset_Handler
- long NMI_Handler
- long HardFault_Handler
- long MemManage_Handler
- long BusFault_Handler
- long UsageFault_Handler

The **_ _StackTop** entry represents the Stack Pointer and the **Reset_Handler** represents the first Program Counter that needs to be loaded when the microcontroller starts, this entry also represents the Interrupt Service Routine for the "reset" signal of the microcontroller [4].

The **Reset_Handler** routine does nothing else other than call the function **SystemInit**, which is responsible with cache initialization and the Floating Point Unit initialization. Next, the **Reset_Handler** will just initialize the heap memory area and will call the **_ _libc_init_array** which will then prepare the microcontroller for running the C environment. Lastly, the main function of the programmed application will be called. After the main function is called, the initialization sequence has ended.

As we can see there is no bootloader involved in the process by default, the microcontroller just initializes the necessary memory areas and structures and then proceeds with calling the loaded application. In order to introduce a bootloader into this initialization sequence we will make the bootloader the application that will be called when calling the main function, and only then, the bootloader may call the main function of another application, possibly after receiving this application form the serial port. The whole initialization sequence of the microcontroller can be seen in the figure 1.

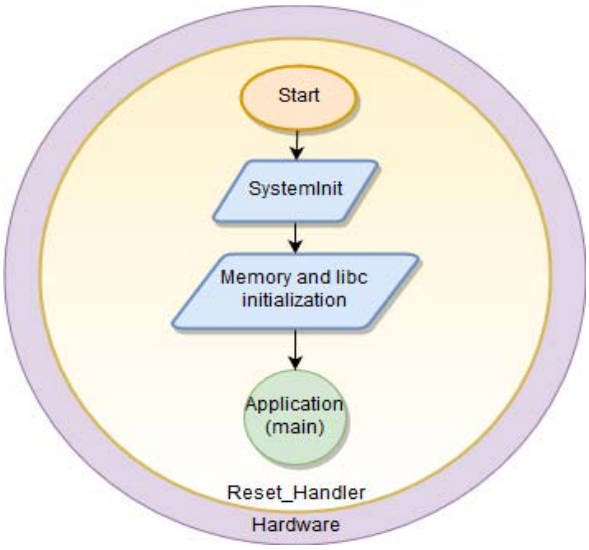

Figure 1. The initialization sequence of an MCU

3. Implementing the bootloader

In the following part we're going to present the technical details of how the bootloader and the helper tools, such as the Bundler and the Flasher are implemented.

As stated briefly in the previous part of this paper, there are three main components:

• **The bootloader:** an embedded application, which is responsible with managing the microcontroller and receiving the third party firmware file from the serial port. After the firmware is decrypted and loaded, it will be executed.

• **The Flasher tool:** a PC application, that will be run by the client whenever he wants to change the application on the microcontroller. This tool is responsible with sending the bundled application (the firmware file) to the microcontroller using the serial port.

• **The Bundler tool:** a PC application (available only to developers) that allows one to create firmware files for the microcontroller. The firmware files are encrypted and only the bootloader may decrypt them. The reason behind this decision was the fact that a potential cracker may intercept a firmware file and try to reverse engineer its mechanisms and thus, compromising the security of the embedded device.

So in order to maintain security we're going to need to encrypt and decrypt the firmware file. For this job, the symmetric key algorithm Rijndael was chosen, due to its unbroken security [5]. This algorithm is also known as the Advanced Encryption Standard, since it is used by the United States as a standard for encrypting communications [6]. The Rijndael algorithm requires the use of an encryption and decryption key. In order to maintain the security of the firmware file, this key should only be known to the Bundler tool and the bootloader, hence we're going to present how the bootloader and the keys will be laid out in the flash memory of the microcontroller.

3.1 Memory Map

The memory map of the microcontroller represents how the application code and data is laid out in memory. What starting address do they use, and how much space do they occupy.

In order to accomplish all of our goals, the memory will be split in three main sections, the bootloader's code which is the main part of this paper, the encryption keys, which will satisfy the security requirement and finally the user's application which will be decrypted and executed by the bootloader.

For exemplification and testing purposes we chose the MKV58F1M0VLQ24 microcontroller, based on the Cortex M7 ARM core. The microcontroller MKV58F1M0VLQ24 defines its memory map starting with address 0x10000000, this is also the address the bootloader is located at [4].

The memory is divided into the three main sections as the following table shows:

Address	Description
0x10000000	Bootloader's start address
0x10008000	Encryption keys start address
0x1000a000	Application start address

Table 1. The microcontroller's memory map

So the bootloader's size is limited to maximum 32KB (including interrupt vector table and flash config), while the keys may span at most 8KB. The application may span the rest of the space up to the end of the flash memory (1MB – 40KB). The keys are required to span a whole sector on the flash since we need to delete it when the keys are changed. The linker scripts or scatter files of the application have to be modified according to the memory map required by the bootloader, ie. the applications should be placed in memory at 0x1000a000, otherwise they will not be loadable by the bootloader.

Also applications should not write the flash memory between address 0x10000000 and 0x1000a000, otherwise the device is recoverable only via a traditional re-flash, using a JTAG connector.

Please note that if the microcontroller is changed, the memory map and possibly the sector size will change too. That will lead to different starting positions and sizes of the application and the keys.

Some could argue that the fact that the encryption keys are stored on the microcontoller's flash might pose a security risk

because after the embedded device is sent to the client, the client can read the keys from memory by physically opening the device and attaching a JTAG connector. Still, this is not the case here, since both of the above-mentioned microcontrollers have a feature called “flash security”. This feature, once activated, disables all external activity with the microcontroller on the JTAG connector, thus disabling the possibility to read flash memory after the devices have been shipped. It should be mentioned that the “flash security” feature of the microcontrollers has to be activated before a device is shipped to a client, the reason is the fact that the activation of this feature should be done only with a JTAG connector. In order not to lock a single application on a microcontroller, without the means to write another one, after the “flash security” feature has been activated, the memory of the microcontroller may still be erased, but that’s just it, the whole memory is wiped. This, for the purposes of the bootloader is not a problem, since the application and the keys are still safe even if the memory is erased, in spite of the microcontroller not being usable anymore.

3.2 The Bundler tool

The Bundler tool is responsible with converting the application executable file (the result of the compilation process) to the firmware file that can be used by the Flasher tool to send to the bootloader. Basically it takes the output of the compilation process (the .bin file), encrypts it using an encryption key and writes it back in the .fw file format. The resulting .fw file is the one that the flasher will use to feed the bootloader. Generally the embedded compilers generate the executable file in the .hex format [8]. In order to generate a .bin file from a .hex file you can use srec_cat [7]:

```
srec_cat file.hex -intel -offset - -minimum-addr file.hex -intel -o file.bin -binary
```

If you have the binary file and a key file, an example command for running the Bundler is:

```
bundler.exe keys\xyz.key app\Debug\app.bin
```

After this command runs, we should end up with a app.fw file in the app\Debug\ directory that is suitable for usage with the Flasher tool. Please note that the Bundler only takes two arguments and the order is not relevant. One argument should be the path to the .key file and the other one should be the path to the application’s . bin file.

3.3 The Key files

If you need to generate a new key file, the process is simple (on Linux):

```
head -c 16 /dev/urandom > keys/xyz.key
```

The key files are simple binary files, of size 16 bytes. The 16 bytes represent the key that the Bundler will use to encrypt the .bin application with and then store it in the firmware file. It is important that the key used by the Bundler is also the key the bootloader will also use to decrypt the firmware, otherwise the flashing process will fail.

3.4 The Flasher tool

The Flasher tool is used to load a new application on the bootloader enabled MCUs. This is done by sending a firmware bundle to the microcontroller by means of serial communication. The firmware should be received by the client from the developers.

A prerequisite of using the Flasher tool is to first connect the microcontroller and the personal computer using a serial cable and determining the serial port on the PC the cable is connected to. On Windows based machines this can be done by inspecting the Ports (COM & LPT) section of the Device Manager. The port should look like: COM4.

After the bootloader is started, the client has approximately 20 seconds to properly start the Flasher tool. For the loading to be successful the user needs to use the COM port determined earlier and the path to the firmware file received from the application’s developers. In order to run the Flasher tool a command window has to be opened, then the following command has to be run:

```
flasher.exe -p COM4 path\to\app.fw
```

The Flasher takes exactly two arguments, a named one and a positional one:

- -p COM4: represents the communication port

• path\to\app.fw: this is the path to the firmware file the client received from the developers. This file was created using the Bundler tool.

3.5 The Bootloader

Having to deal with reading a firmware file from the serial port, decrypting it, writing it to the flash memory and loading it, the bootloader is divided into several functional modules that each handle some independent responsibility. The modules will then be combined together in order to achieve the desired functionality: loading a third party application in a secured way.

The architecture of the bootloader may be seen in figure 2.

In this figure we can distinguish the following modules and responsibilities:

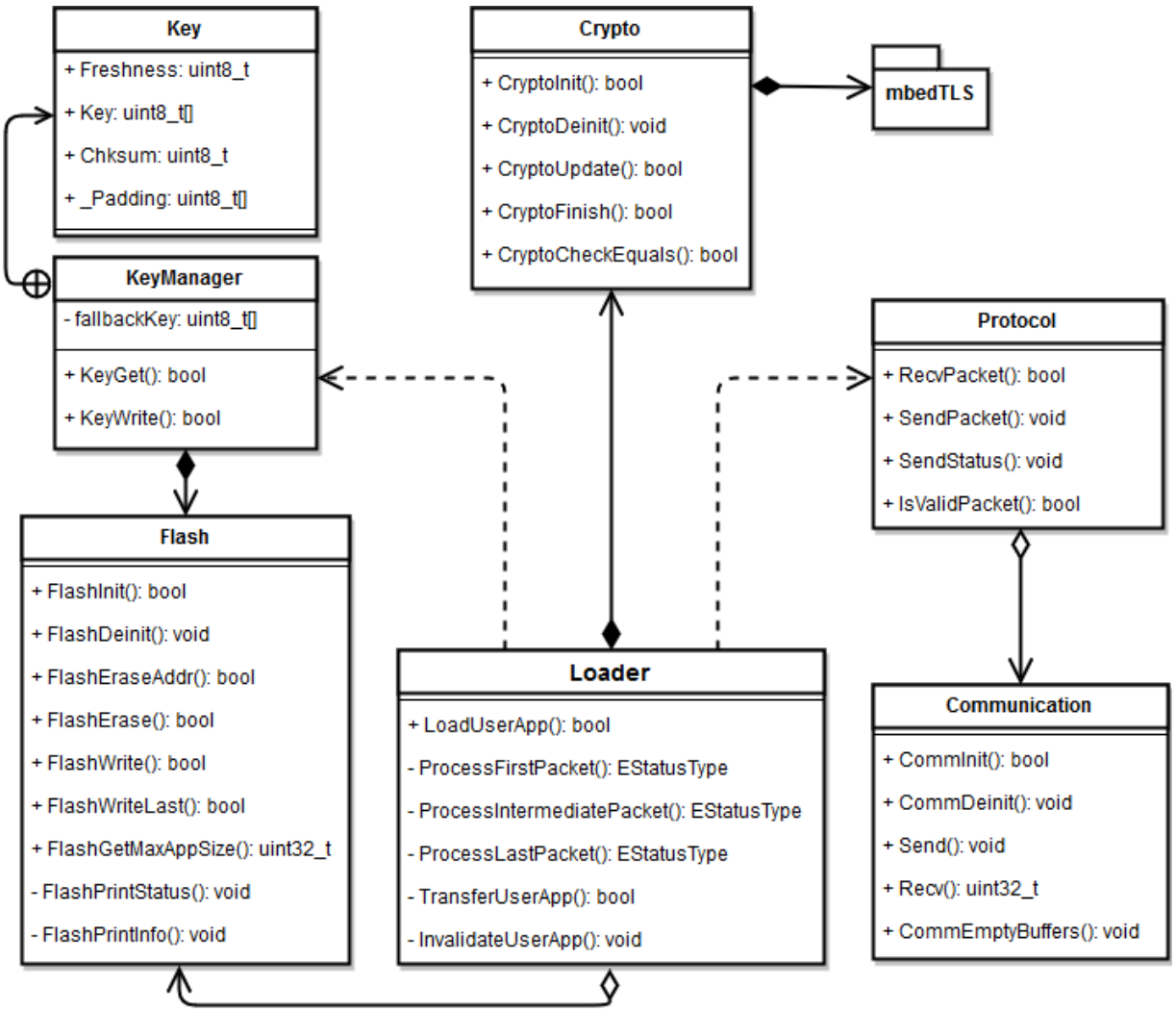

Figure 2. The bootloader's modular architecture

• **Flash:** This module is used for writing, reading and erasing the flash memory of the microcontroller.

• **Crypto:** Its role is to manage the cryptograpy library and to provide an abstraction layer over it. The library used here is called mbedTLS and it implements the Rijndael algorithm used to encrypt and decrypt the user application.

• **Communication:** Implements the communication channel, it is responsible with opening the serial port, sending and receiving

of data and of course is also responsible of properly closing the port.

• **Protocol:** It packages and formats the data in packets in order to facilitate the communication.

• **KeyManager:** Responsible for writing and reading the encryption keys in their reserved memory areas.

• **Loader:** This is the module that brings all the other functionality together and performs the actual application loading on the microcontroller.

The most complex module of all is the Loader module, which combines all the other functionality in order to transfer the application on the serial port, decrypt and execute it. Figure 3 shows how the module works on a high level.

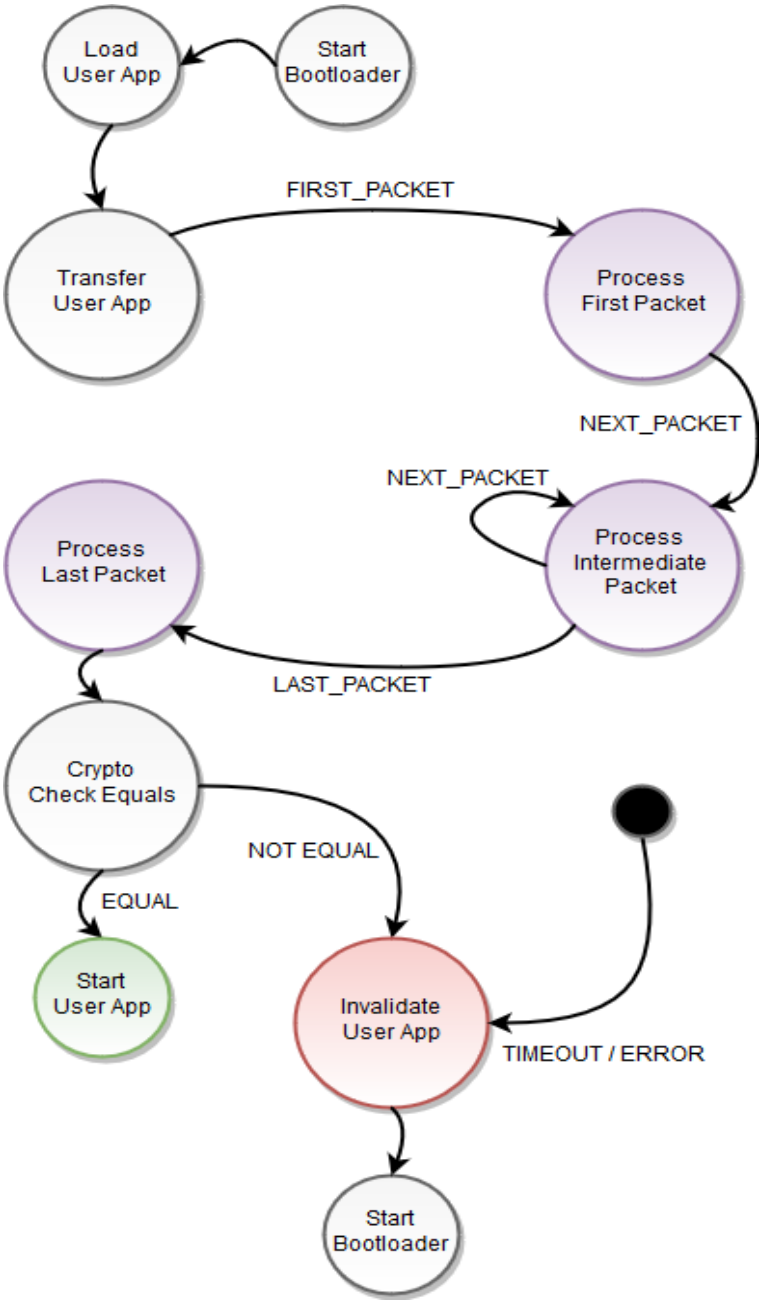

Figure 3. The data flow in the Loader module

This data flow is based on the protocol developed specially for the bootloader, the transitions between states are determined by the arrival of different packet types:

• **FIRST_PACKET:** The first packet sent from the Flasher to the bootloader. After this packet only packets of type NEXT_PACKET will follow.

• **NEXT_PACKET:** An intermediary packet neither the first, nor the last. The number of these packets sent by the Flasher and received by the bootloader vary with the application's size, the bigger the application, the more packets sent to the bootloader.

• **LAST_PACKET:** This type of packet always follows a NEXT_PACKET packet and is surely the last packet that the bootloader will receive.

The states of the Loader module represent functions inside the code, these do the processing required after a state transition and hopefully will end up in the StartUserApp state, which means that the application was successfully transferred and decrypted and lastly it can be executed.

The main modules presented here are shared between the applications of the system. Both the Bundler and the Flasher use the some of the modules to achieve their respective functionality. This is possible because of the way the code is written, namely in a portable fashion, because the Flasher and the Bundler are applications designed for desktop whereas the bootloader is designed for embedded devices.

How the modules are shared between applications can be seen in figure 4, this figure also lists the "Firmware" module, which is not a standalone module, but more of a representation of the fact that the firmware file is being handled in that application as well. Also the "Communication" module is named here "UART" in order to better illustrate its nature.

Bundler	Bootloader	Flasher
Firmware	Firmware	Firmware
Crypto (AES)	Crypto (AES)	
	Flash	
	UART	UART
	Protocol	Protocol
	Loader	

Figure 4. How modules are shared between applications in the system

It can be seen that only the bootloader and the Bundler use the cryptography module, since these are the only place we trust the environment enough in order to allow usage of encryption and decryption keys (which in AES's case the same key is used for both decryption and encryption).

The above modules are just the main modules comprising the bootloader, there are also a couple of auxiliary modules, that are succinctly presented below:

• **Hardware:** Responsible with managing the button that starts the bootloader. The bootloader developed here can be started like any other, that is: by rebooting the system and pressing a button while it reboots.

• **Retarget:** The bootloader is targeted at embedded systems, as such this module redirects the printf function's output on a serial port reserved for debugging the applications that are running on the microcontroller. The reason behind this output redirection is the fact that a microcontroller hardly ever has any display connected or any operating system to handle visual graphics or ASCII text, thus the output is redirected via a serial port to a fully fledged PC, where it can be inspected.

• **Crc:** Here lay the functions meant to compute the cyclic redundancy check codes used by the serial protocol for transferring

the application to the microcontroller.

• **Timer:** This module helps detect if some packets fail to arrive at the bootloader in a specified amount of time, hence avoiding any lock-ups.

• **RingBuffer:** This module implements a ring buffer (also known as circular buffer). A ring buffer is used by the communication module and filled with data whenever a packet arrives on the serial lines. It's purpose is to avoid allocating heap memory for the arriving packets, allowing the bootloader to run even on the most restrictive embedded devices.

These auxiliary modules, and their connections with one-another can be inspected in figure 5.

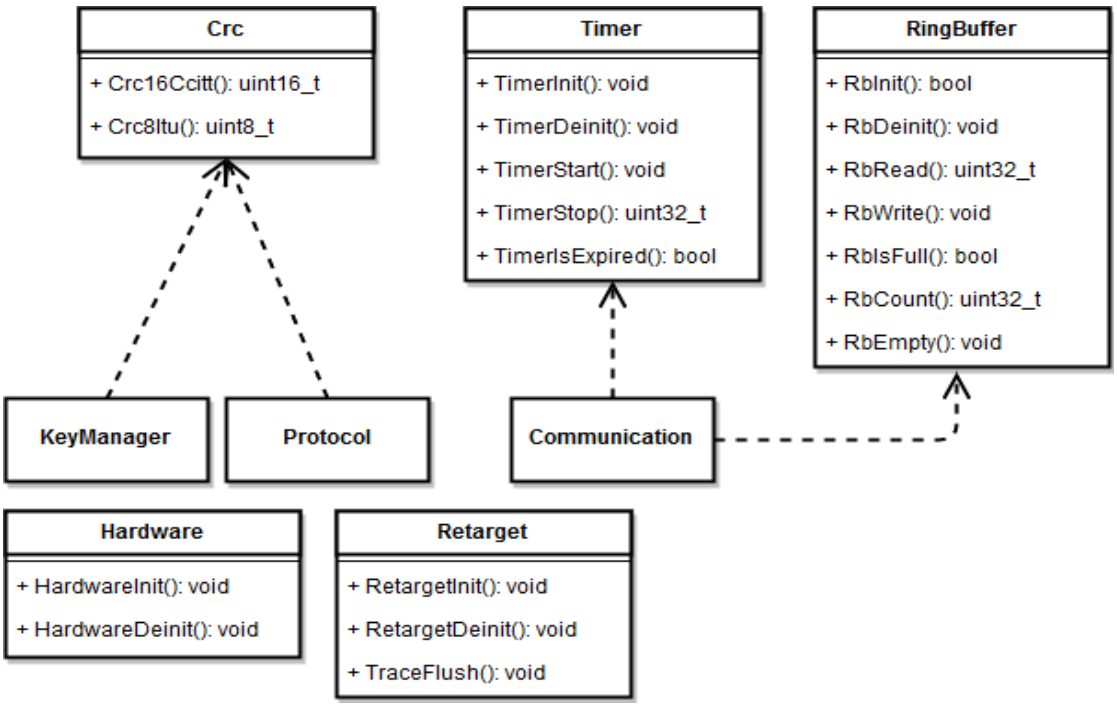

Figure 5. The bootloader's auxiliary modules

It should be noted that all of these modules are implemented in a source file, with the .c extension and a header file, with .h extension.

3.6 The communication protocol

Zone	Address	Description	Size (bytes)
Header	0x00	ID (fixed at 0xA5A5)	2
	0x02	Type of packet	1
	0x03	Length	1
Data	0x04	Content of packet	variable
CRC	-	Polynomial code	2

Table 2. Packet structure for the communication protocol

As stated previously, there needs to be a standard way of communicating between the Flasher tool and the bootloader running on the microcontroller. This standard way is implemented by the means of a communication protocol. This protocol is

implemented by the Protocol module of the bootloader, which is shared by the Flasher tool, too.

The packet structure of the protocol may be seen in the table 2.

Apart from the fixed ID that starts a packet, the type of packet field may take several values. These values were presented in the last section of this document and in addition to those, there is one more packet type, STATUS_PACKET that has the role of communicating the status of the transfer up to this point in time. After the type of the packet, the length field follows, which indicates how many data bytes of the application will be sent from the Flasher to the bootloader in the current packet. After the actual data, a 16 bit CRC is appended to the end of the packet in order to make sure the transmission of the packet is done without errors.

The message sequence diagram in figure 6 illustrates the process of successfully sending a firmware file from the Flasher tool to the bootloader.

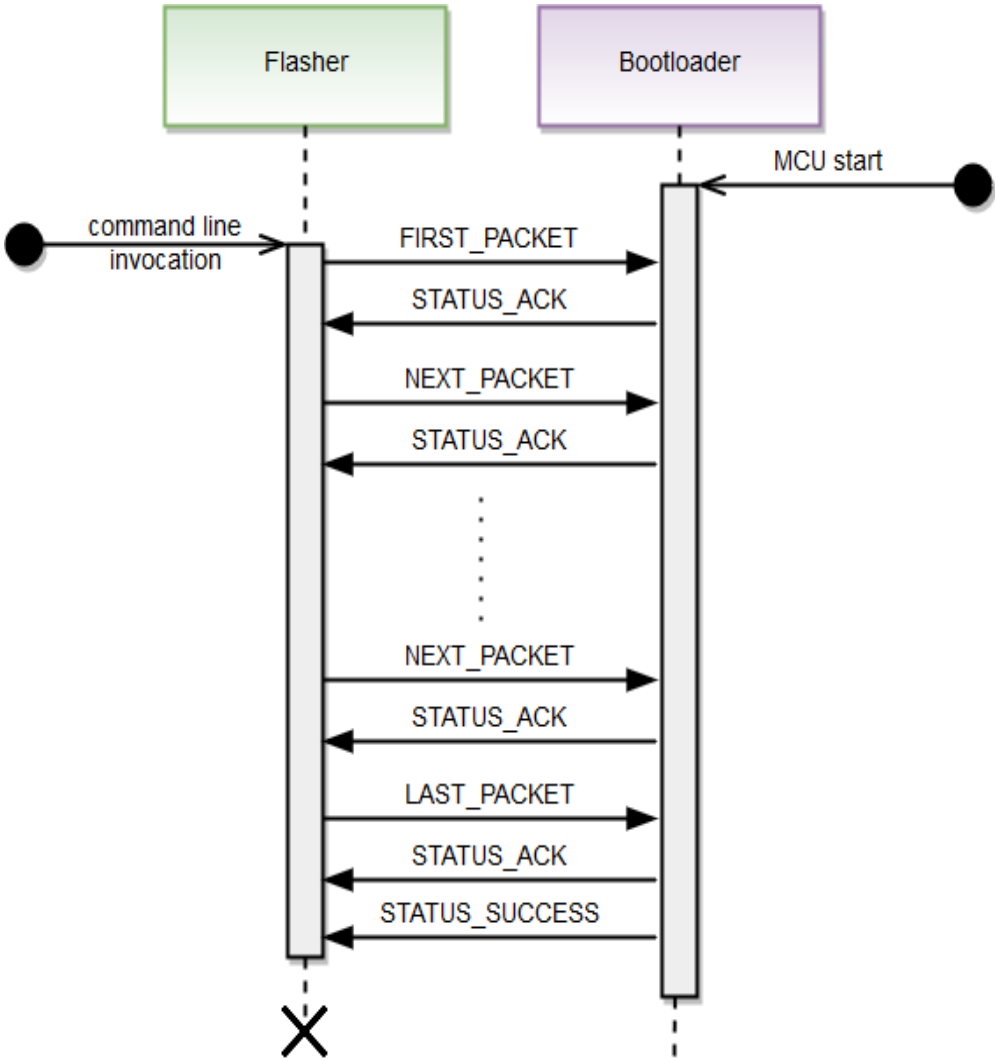

Figure 6. Protocol message sequence diagram

It can be seen in this figure that the transfer starts when both the Flasher and the bootloader are ready to send and receive data and the fact that every packet is followed by a STATUS_PACKET indicating that the transfer up to a certain point was successful.

The last two packets are sent by the bootloader, acknowledging the receipt of the last packet and indicating the success of the whole transmission of the firmware file. After the firmware file was successfully transmitted, the bootloader proceeds with executing the third party application.

4. High level view of the system

This section aims to provide a high level view of the system. By system, here I mean the bootloader together with the Bundler and the Flasher tool.

Figure 7 shows how the system works starting from the application that the developer wants to use on a bootloader-enabled embedded device and ending with the final step of the process, the execution of this application on the target medium.

Suppose the following use-case: the bootloader has been flashed on the microcontroller with a JTAG adapter by the developer, together with the needed application. The embedded device has been shipped to the client. After a period of usage the client discovered that the application is not functioning properly anymore due to a software bug. Given this scenario and the fact that the client has a bootloader on the faulty microcontroller, the firmware on it may be easily changed as follows:

First of all the developer has to fix and recompile his code for the target architecture, where the updated application will be loaded by the bootloader. Knowing which device is causing problems, the developer has to run the Bundler tool with the key present on the device's flash and the new application binary as command line arguments.As a result of running the Bundler tool,the firmware file is generated. This firmware file is an encrypted form of the application binary.

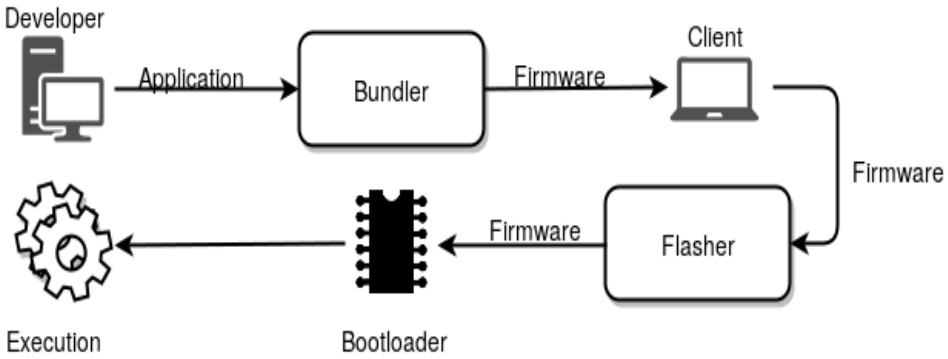

Figure 7. The usage of the auxiliary tools and of the bootloader itself

After the completion of the last step the firmware file is now ready to be sent to the client, by whatever means necessary, via email or web upload.

The client then, has to connect his PC and the embedded device together using a serial cable. This will be used by the Flasher in order to send the firmware file to the bootloader. At this point the bootloader has to be started, this can be done by rebooting it and pressing a button on the microcontroller. Now the Flasher can be started, by specifying the firmware file and the correct serial port as command line arguments. At the end of the process, if everything went well, the bootloader should have received the firmware, decrypted it and executed it.

Of course apart from the success case there are also three cases where the flashing process may fail:

1. If the Flasher appears to be unresponsive, then this indicates the fact that maybe the specified COM port is not the correct one. If this is the case, then the Flasher tool can be stopped and restarted with the correct serial port where the serial cable is connected to the PC.

2. If the user doesn't start the Flasher tool within 25 seconds of starting the bootloader on the microcontroller and there is already an application on the flash memory, then the bootloader times out and proceeds with executing the exiting application. In this scenario, the flashing process should be restarted and the Flasher tool should be started sooner.

3. It is possible that the transfer of the new firmware between the Flasher tool and the bootloader to fail. This could be due to a number of reasons, such as transmission error or maybe the application has been modified by an unauthorized third party, before the client received it. These error cases are reported by the Flasher tool with a uggestive message like "*ERR* Failed to flash firmware file app.fw" and the flashing procedure needs to be started all over again. If the situation persists, there is a possibility that the firmware file was modified during the download from the web, then a simple re-download of the file might solve the issue, otherwise the developer of the application should be contacted since the application was either modified by an unauthorized third party or maybe it was mistakenly encrypted with the wrong key by the developer.

5. Conclusions and future work

This paper successfully presented the design and implementation of a bootloader for embedded devices. The aim of the bootloader was to allow for cost reductions security and flexibility of embedded applications deployment.

The cost reductions are achieved by allowing the users to program applications on the microcontrollers without the necessity for special programmer hardware. The flashing, when using the bootloader, can only be done using a serial cable connected to a personal computser. The flexibility comes from the fact that the bootloader allows the client to change the application on the microcontroller whenever there is an update available, without the need to ship the device to the developer so he can program it securely using the special JTAG connectors.

Although the practical implementation on the MKV58F1M0VLQ24 microcontroller was a success, there are still open points that may be improved. One such point would be to allow the bootloader to write multiple applications into the flash memory, given the fact that enough storage space is available. Also the usability of the Bundler and Flasher tools can be improved by implementing a Graphical User Interface, since now they are console based programs. Finally, the portability of the bootloader may be enhanced to other microcontrollers in the future, this task is made simple by the fact that all the modules are written independently. Partly this feature was demonstrated by the fact that the Flasher tool shares the same protocol code with the bootloader, no line of code had to be changed for it to work.

Acknowledgement: This work is based on the idea of Mr. Razvan Marcus, , who also offered technical guidance and advice from time to time regarding some of the implementation details.

Part of this text was edited as suggested by conf. dr. ing. *Morariu Daniel, from "Lucian Blaga" University of Sibiu.*

References

[1] Barbu, Paul - Gheorghe., Secured, Bootloader. (2017). Application for initializing and updating microcontroller applications ICDD, Sibiu.

[2] Wikipedia, (2017). JTAG *https://en.wikipedia.org/wiki/JTAG.*

[3] Wikipedia, (2017). Bootstrapping *https://en.wikipedia.org/wiki/Bootstrapping#Etymology.*

[4] NXP Semiconductors, (2016). *KV5x Sub-Family Reference Manual,* Revision 4.

[5] Joan, Daemen., and Vincent, Rijmen. (2002). *The design of Rijndael: AES — the Advanced Encryption Standard,* Springer-Verlag.

[6] Wikipedia, (2017). Advanced encryption standard *https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.*

[7] Peter Miller, Srecord. (2017). Binary Files *http://srecord.sourceforge.net/man/man1/srec_examples.html#BINARY%20FILES.*

[8] ARM Group. (2017). General: Intel Hex File Format *http://www.keil.com/support/docs/1584/.*