Software architecture document for project “STM32-bootloader”

0. Authors

- Tarasov Artem
- Vasilev Pavel
- Patrushev Boris

1. Goals and limitations

1.1. Key functional requirements

- Use-case “Firmware update” - to update the device’s firmware
- Use-case “Bootloader stopped working” - to restore the correct operation of the program if the bootloader crashed
- Use-case “Upload firmware to developers” - to get the program data
- Use-case “Modificate the build” - to make it so that firmware built from the project could be used with the flasher and the bootloader.
- Use-case “Get memory layout”- to check the memory layout with other memory

1.2. Non-functional requirements

Environment:

- Any STM32 chip with any interaction interface (e.g. UART or I2C), which is specified by the developer.
- PC with Windows 10

Performance:

The bootloader takes up a small amount of flash memory on the chip and transfers control to the firmware in a short amount of time.

Reliability:

- If the program is not completed, then we restore the previous version of the firmware and execute it
- Bootloader will control checksums to be confident that program was downloaded successfully
- Nobody can read the code starting from the build process to the installation process on the device
- No one can accidentally remove any data from the device by downloading a new version.

Extensibility:

The flasher will support plugins for compatibility with different types of project files, so it can be used with different IDEs.

Other:

- Data format that contains the device name, ID, and other data to adjust the installation process
- The firmware is encrypted at all times between the build process and getting on the device’s flash thus protecting it from being copied and reverse engineered.

1.3. Architectural goals

- STM32 support: most of the chips
- Security: the firmware is encrypted at all times between the build process and getting on the device's flash
- OS support: Flasher must operate on Windows 10

1.4. Additional goals, restrictions and preferences

- Using C# to write flasher, due to the fact that a lot of developing software for STM32 is written in C# so the flasher could be made a plugin for that software later
- Using AES encryption, due to the fact that it is approved by US government, very efficient and also easy to implement
- Make a simple to understand UI
- The bootloader itself has to be small in size: the limit is soft around X KB

2. Goals analysis

2.1. Security

Security is very important in our project and this is one of the key components of the STM32 bootloader. So we decided to use **AES256** encryption. Every STM-32 chip will have its own unique key before it is sent to the user and the developer of the firmware for each instance. So the developer can safely send firmware to the end user using the same key.

This type of encryption has following advantages:

1. It has high speed
2. Some libraries use small amount of resources (about 1k of flash memory, one of the most important resources)
3. It's secure

But also AES has several disadvantages:

1. This method of encryption is symmetric and only one key will exist at all times for the stm32 chip. That means that if someone stole the key from the specific ship, he will be able to decrypt the program and data for this controller or upload illegal firmware. And even you can simply lose the key and you will not have access to the board So for every chip you need to keep the secret key in a safe.
2. Despite the fact that a small amount of memory is used, additional memory is still used (it is common for all encryptions)

Also we need to think about how and who of the developers will have access to the key from a specific device.

2.2. Bootloader

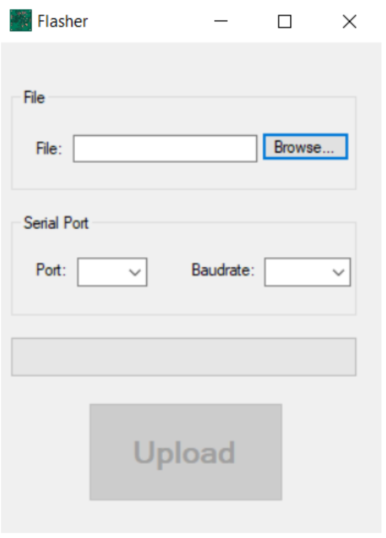
Bootloader is the program on the STM-32 chip. So its main goal is to receive a program by UART, check its integrity, decrypt it and store it in flash. And after all, transfer control to the main program. Also, if required, encrypt the program and data and send it to the PC, so that developers can use this information to improve next versions of the program.

2.3. Flasher

Using C# to write flasher, due to the fact that a lot of developing software for STM32 is written in C# so the flasher could be made a plugin for that software later. Due to this fact we don't use Python, Java and others.

Application with GUI(For end users):

- GUI should be simple to understand
- GUI itself has typical design (like in every program that can fork with text files for example). It has only must-have buttons on normal places. There is no massive or extra stuff to give the user any reason to be confused. GUI is written using WFA as a good tool for building small, simple and familiar designs. We choose WFA, because we don't need a cross-platform and also it's basic utility for Visual Studio.
- Example of Simple GUI:



Console Application(For developers):

- This is the same as app with GUI, but with more configurable parameters especially for developers, who know what they need and what they do

2.4. Modifier

Bootloader might have a different size for a different chip, which means that the program has to be adjusted for being placed at another address in flash memory. This can be fixed by changing project files with the relevant variables at build time.

Modifier will allow automatic modification of project files for every specific STM32 chip and memory layout. So it will be easy to upload firmware on other devices without many additional actions.

2.5. Bootloader File Generator

The Flasher will require a binary file which contains encrypted firmware that is to be uploaded to the chip and a checksum. The file also should contain other relevant information like unique hardware ID, firmware version, firmware size, etc, which will be used to test whether the firmware is supposed to be uploaded to the device and is compatible with it.

3. Solution description

3.1. Modules and subsystems

Bootloader

Bootloader is a software on stm32 board. It communicates with the flasher to receive or send any data. Communication between them is very simple: the bootloader sends by UART the message to the flasher and receives an answer. Specific elements of protocol are described in section 4.1.

Bootloader receives data by UART.

Bootloader must verify data integrity, decrypt data and handle situations when new data cannot be correctly received from the program (i.e. someone accidentally pulled out a UART-USB connector).

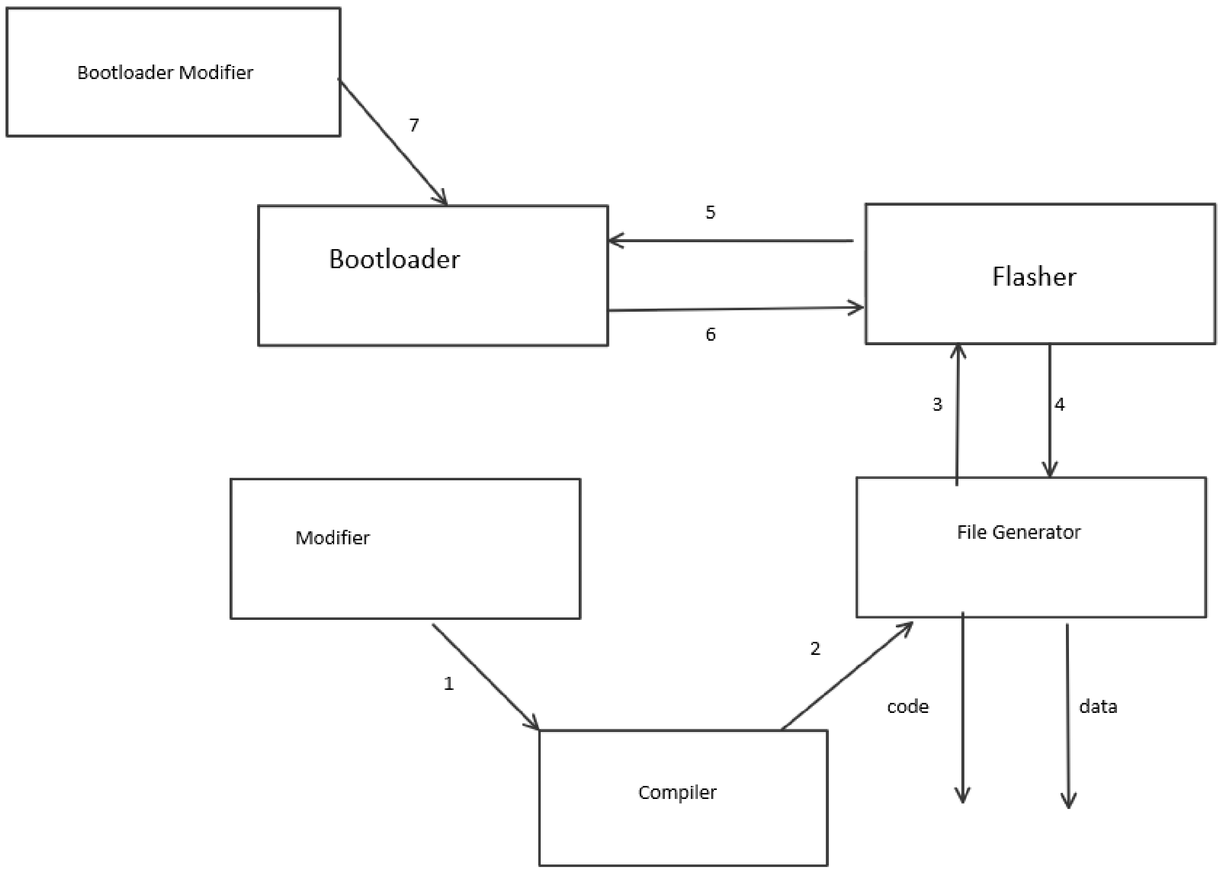
Bootloader is written in C using library HAL, but should be overwritten without it to reduce the amount of memory used.

Flasher

Flasher is a console application and application with GUI designated to communicate with stm32 via protocol, which is described in section 4.1. Flasher is written in C#, due to the fact that a lot of developing software for STM32 is written in C# so the flasher could be made a plugin for that software later. And also we use WFA to create simple graphical user interfaces for end users on Windows 10. End users can choose baud rate, serial port, firmware project file to transmit it on stm32 bootloader.

Modifier

Modifier is a console application designated to change firmware project files so that the built firmware is compatible with the bootloader. Modifier is written in C# because most of the embedded development software is also written in C# which means that it is likely that no additional software will be required to use it.



1. Modifier change project to be compatible with bootloader chip
2. We compile the project
3. Then file generator encrypt code
4. Also file generate can decrypt data received by flasher
5. flasher send new code to the bootloader by specific protocol
6. bootloader sends some data to flasher by specific protocol
7. before uploading bootloader on the chip, we need to modify it (i.e. add encryption key)

4. Key architectural elements

[Identify and define key system-wide (i.e. affecting multiple modules from section 3) behavioral elements like protocols, algorithms, computational mechanisms. Usually such elements are corresponded to at least one goal from section 1. Each element must be described in terms of modules identified in section 3 and their behavior. Pictures are welcome.

This section could be placed before section 3 if necessary.

The rest part of this section including subsections are examples]

4.1. Protocol of data transmission BOOTLOADER-FLASHER

Segments	Transmission code	Length of message body	Header CRC	Message Body	CRC

Size in bytes	4	4	4	Length of message body	4

0-3 bytes. Transmission code. What type of data are going to be received/transmitted

1 - String message (printf output for example)
2 - Error message
3 - Binary code(firmware) - parts of code
4 - Request block. Required number of block
5 - Acknowledge
6 - Ready to receive next block of code
7 - Request to change baud rate to new value
8 - Request to change timeout to new value
9 - Release firmware to start it on bootloader

4-7 bytes. Length of message body (see below)
8 - Length bytes. Body of message, containing data that must be transmitted/received
Length bytes - (Length bytes + 4). Cyclic Redundancy Check to check integrity of received data

4.2 Flasher file format

Flasher requires a specially formatted file (which is generated by BootloaderFileGenerator). The format is specified as follows (**[n]** - the segment takes up n bytes):

[6] - "HEADER" in ASCII

[2] - an unsigned integer specifying the size of the manufacturer's name. (max = 64)

[?] - manufacturer's name in ASCII (? = previous segment)

[8] - version, stored as an array of 4 unsigned 16-bit integers

[8] - time of creation (UNIX time)

[4] - "DATA" in ASCII

[16] - initialization vector for encryption

[4] - encrypted firmware size

[?] - encrypted firmware (? = previous segment)

Note: integers are stored in LSB order.

Segment	"HEADER"	n	Name	Version	Creation time	"DATA"	IV	Size	Encrypted Firmware
Size in bytes	6	2	*n*	8	8	4	16	4	*Size*

4.3. Encryption

Tiny AES. AES encryption was described earlier.

4.4. [ex.]Modules API

All the modules' external APIs should follow Command-Query Separation [<reference>] pattern excluding …, where atomic types should be used and, thus, they must violate this pattern.

All the modules excluding … must implement their APIs in two forms: 1) regular Java API in terms of interfaces, and 2) RESTful API using Jetty [<reference>]as embedded servlet container.

Regular Java API must be used in most situations due to better performance and reliability. RESTful APIs should be used mostly for automatic testing purposes.

4.5. [ex.]External plug-in API

For modules … the external API for 3rd party video processing plugins must be implemented in a reliable way. A plug-in must be run as a separate process restricted by RAM and CPU usage by … So, the main application shall tolerate the situations where plug-ins are killed due to critical error (segmentation fault or so), overusing memory or CPU, spawning threads.

Interaction with plug-in on lower level must be implemented via STDIN/STDOUT/STDERR. On a higher level, the sandbox for plug-ins must be implemented (see 3.NN) and provide the following operations: …

To transmit objects through the raw stream JSON serialization/deserialization must be used. Jackson [<reference>] library is suggested for this. There is the requirement to possess bandwidth of … MB/s for plugins, so Jackson must be checked first to match this.

5. Platform

Users' system requirements:

- Personal computer with Windows.
- Processor: any.
- Graphic adapter: any.
- X Mb of free space on a disc.
- .Net Core 5.0 is required.
- STM32: any.
- Interaction interface: UART/(mb) I2C

Language: C# for flasher and modifier, C for bootloader