TALLINN UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Department of Computer Engineering

IAY70LT

Kumar Amit Mehta

FAIL-PROOF OVER THE AIR FIRMWARE UPGRADE FOR EMBEDDED SYSTEMS

Master thesis

Mairo Leier

M.Sc.

Early stage researcher

Uljana Reinsalu

PhD.

Research scientist

Tallinn 2016

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Kumar Amit Mehta

25.05.2016

Abstract

The internet of things (IoT) domain is poised to grow multifold in the near future. This will lead to wider deployments of connected embedded devices. System manufactures would like to upgrade the firmware in field to improve system functionality, provide bug fixes or respond to security threats. During the development, one might use Join Test Action Group (JTAG) or Serial Wire Debug (SWD) interface to program system memory, however due to cost-effectiveness reasons, development hardware is generally not shipped with the end product. The ability to upgrade the firmware in the field is even more challenging for itinerant embedded systems or, for those having large quantities of deployments.

This work concentrates on design considerations and implementation of highly portable bootloader for fail-proof firmware upgrade over the air for resource constrained embedded devices. In this thesis, a bootloader was developed for STM32 series system on chip that allows firmware upgrade over GSM/GPRS and Wi-Fi network. Though the implementation is for STM32, generic design considerations for bootloaders for fail-proof firmware update is also presented here. Hence, I believe that the ideas presented here can be readily adopted to similar Embedded systems project.

At first, over the air firmware upgrade process, its challenges and associated system components are explained in brief. Then the implementation specific details are explained. Based on the System architecture, various individual components are discussed in detail. One of the main focus of the thesis is discussion about the dependability and fault tolerance of the firmware upgrade process. The implementation is validated by performing number of experiments. Thesis outcome is a successful implementation on a commercial embedded systems product.

This thesis is written in English and is 67 pages long, including 6 chapters, 39 Figures and 6 Tables.

Annotatsioon

Eksimiskindel eetri kaudu tehtav püsivara uuendus sardsüsteemidele

Seadmete Interneti (IoT) domeen laieneb lähitulevikus mitmetesse suundadesse. See juhatab meid erinevatesse sardsüsteemide kasutusvaldkondadesse. Süsteemi funktsionaalsuse parandamise, vigade kõrvaldamise ja turvalisuse tõstmise eesmärgil sooviksid tootjad uuendada seadme püsivara üle eetri ja hoides seda installeerituna seadme asukohas. Arenduse käigus on võimalik süsteemi mälu programmeerimiseks kasutada Ühendatud Testimisrühma (JTAG) või Seerialiidese Siluri (SWD) liidest. Toote omahinna alandamise eesmärgil ei anta tavaliselt arendusriistvara lõpptootega kaasa. Süsteemide puhul, mis on pidevalt liikumises, või suurte hulkade seadmete puhul on püsivara uuendamise teostamine veelgi suurem väljakutse.

Käesolevas lõputöös keskendutakse portatiivse alglaaduri disaini valikutele ja implementeerimisele teostamaks piiratud võimalustega seadmetes töökindlat püsivara uuendust üle eetri. Töö käigus arendati välja alglaadur STM32 tüüpi mikrokontrollerile, mis võimaldab püsivara uuendust üle GSM/GPRS ja WIFI võrgu. Samuti on käsitletud üldisi alglaaduri disaini võimalusi töökindla püsivara uuenduse teostamiseks. Seetõttu usun, et siin esitatud ideid on võimalik kasutusele võtta teistes taolistes sardsüsteemide projektides.

Esiteks esitletakse üle eetri teostatava püsivara uuenduse protsessi, väljakutseid ning sellega seotud süsteemi komponente. Järgmisena selgitatakse implementatsiooniga seotud detaile ning seadmete üksikosadega seotud detaile, mis on setud süsteemi arhitektuuriga. Üks peamisi fookuseid käesolevas töös on töökindluse ja veakindluse tagamine püsivara uuenduse käigus. Implementatsiooni käigus teostati erinevaid eksperimente demonstreerimaks püsivara uuenduse töö- ja veakindlust. Töö käigus arendati välja kommertsiaalsele tootele sobiv püsivara uuendamise lahendus.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 67 leheküljel, 6 peatükki, 39 joonist, 6 tabelit.

Table of abbreviations and terms

IAP	In-Application Programming. It is the ability of the application to erase and program code memory.
ISP	In-System Programming. It is the ability of dedicated hardware circuitry to program the microcontroller or similar programmable logic device after it’s been soldered on a PCB.
JTAG	Join Test Action Group. It is a IEEE 1149.1 standard that specifies the specification for performing boundary-scan hardware testing at the IC level.
SoC	System on Chip. A system on a chip or system on chip is an integrated circuit (IC).
SWD	Serial Wire Debug. It is a 2 pin alternative to a traditional IEEE 1149.1 compliant (JTAG) interface.
HAL	Hardware Abstraction Layer is software abstraction layer, consisting, application programming interface to interact with underlaying hardware.
TE	Terminal Equipment.
TA	Terminal Adapater
MT	Mobile Termination
CR	Carriage Return. ASCII code in hexadecimal format is 0xD
LF	Line Feed, ASCII code in hexadecimal format is 0xA
OTP	One Tme Programmable.
MCU	Microcontroller Unit
USART	Universal Synchrononus Asynchronous Receiver Transmitter

UART	Universal Asynchronous Receiver/TSransmitter. A serial communication protocol
I2C	I-Squared-C. A serial communication protocol
ROM	Read Only Memory.
SRAM	Static Random Access Memory
ASCII	American standard code for information interchange
OTA	Over The Air
CMSIS	Cortex Microcontroller Software Interface Standard is a vendor-independent hardware abstraction layer for the Cortex-M processor series and defines generic tool interfaces [19]
HTTP	Hypertext trasfer protocol
HTTPS	Hypertext trasfer protocol secure
SD	Secure digital. A non-volatile memory card format.
RTOS	Real Time Operating System
CAN	Controller Area Network.
IRQ	Interrupt Request
ISM	Industrial, Scientific and Medical

Table of contents

List of Figures

List of tables

1. Introduction

According to joint research by IT research firm IDC, Intel and United nations [1], 200 billion connected things will be in use by 2020. The driver to the extraordinary growth of IoT domain is the service offerings by the end user organizations and vendors. In their already deployed IoT systems, device firmware update over the air is one of the key enabler for the system manufacturers to offer new features or cater to the new feature requirements from their customer.

The impressive growth of IoT has seen various organizations and vendors compete for the similar product/service offerings. To stay competitive, the time to market should be kept as low as possible. The sooner, one integrates its offerings, better are its chances on capitalizing the market. However, this could lead to partially tested systems, resulting in critical software defects in field. System manufacturers would like to provide patches to fix those software defects, however applying software update on tiny embedded systems is rather complex for three main reasons:

- The programming tools for microcontrollers such as those popular in IoT domain are generally not shipped with the product. Typically, it requires dedicated circuitry such as Join Test Action Group (JTAG) or Serial Wire Debug (SWD) interface to program the main memory of microcontroller.
- Embedded systems are generally deployed in very large numbers and hence it makes it difficult to manually update the software on each.
- Many a times, programmable logic devices could be deeply embedded in the system, requiring a lot of effort in physically accessing the system.

For these reasons, the ability to update the software in field in timely and cost effective manner is immensely attractive to system manufacturers.

Due to the connected technology, a system flaw is easier than before to exploit. Security breach can have serious consequences and at the very least may render the system useless for significant amount of time. Over the air update could potentially enable the vendors to avert cyber-attacks or respond quickly to security threats.

1.1. Motivation

Over the air update is not a new topic; this area has been researched and different forms of implementation already exist at the market. For example, the firmware update of Android or iOS-based cellphones or update of mobile applications. However, the major difference between these forms of over the air update with the target System on Chips (SoCs) of this thesis is that the latter is extremely resource constrained, is typically more deeply embedded, have budgetary restrictions and could operate in critical applications.

A fail-proof, over the air firmware update of resource constrained embedded systems is a challenging task and is of more relevance for critical applications. It is surprisingly easy to render the system useless, if the software update process is not handled properly. In most cases, it could lead to revenue loss, but it can have dire consequences as well.

Technological advances are bringing myriad of applications in large number of system deployments. System manufacturers would like to bring the turnaround time and time to market to a minimum, while not overlooking the safety and functionality requirements. This is more of an engineering challenge and is implementation specific, but nonetheless a portable software is hugely appreciated.

Therefore, a portable solution for over the air upgrade of firmware for resource constrained embedded systems is an interesting area of research.

1.2. Scope

Firmware upgrade over the air is quite an extensive topic, so the following goals were specified and accordingly work flow was planned:

- Study In-Application Programming (IAP) technique.
- Implement IAP technique to program STM32F051 Microcontorller Unit (MCU). STM32 series MCUs are one of the most popular MCU [2] in the IoT market today.
- Develop a bootloader.
- Extend bootloader to support firmware update over GSM/GPRS network.
- Develop Hardware Abstraction Layer (HAL).

- Using this HAL, port bootloader and application on different MCUs (STM32F407VG and STM32F072CB)
- Extend bootloader to support firmware update over Wi-Fi.

1.3. Thesis organization

Thesis work is divided into 6 chapters. The problem statement is introduced in chapter 1. Basic concepts of the studied area are presented. A short overview on the complexity of the problem giving explanations why current problem is valuable. After that, goals of thesis are identified.

Chapter 2 covers the basic definitions and concepts of major components of this thesis work.

In chapter 3, implementation specific, systems architecture is discussed. It lays foundatation for the later chapters.

Various system components are covered in length in chapter 4. This chapter starts with generic concepts behind bootloader and the design considerations for a fail-safe system. The system requirements and implementation details for the co-existance of bootloader and application firmware in the main flash memory of the MCU is then discussed. The firmware file format is then discussed in brief. The rest of the chapter covers data communication details and software portability and reusability.

Chapter 5 summarizes the result of the thesis.

Finally, Chapter 6 addresses unsolved problems and possible future studies.

2. Theoratical backgorund

2.1. Device firmware update over the air

Device Firmware Update Over The Air (DFU OTA) refers to various methods of distributing software updates. Though, a software update could be as simple as some simple configuration changes that does not require reprogramming the main flash memory or halting the normal operation of the system, this thesis focuses on more engaging firmware update process, that would typically halt the normal execution of the system and repogram certain areas of main flash memory.

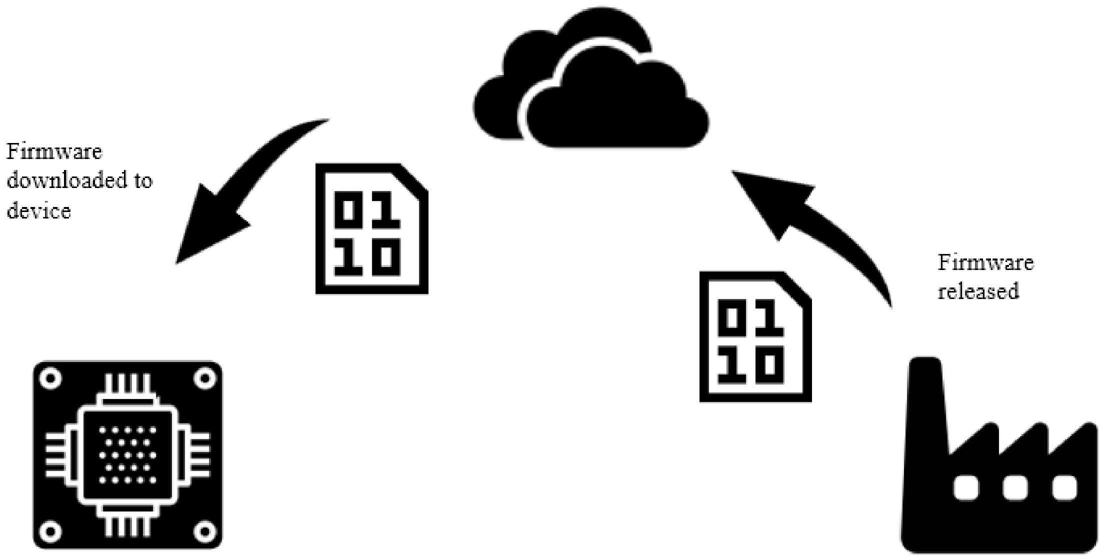


Figure 1. Device firmware update over the air.

A sample firmware update process in its most basic form is shown in Figure 1. As shown in Figure, software update is released from the factory and uploaded to cloud. The target system gets notification of the software update availability, downloads the software image/firmware and updates the software running on itself. However, during this entire process, many things can go wrong. Some of the most common problems are shown in Figure 2, 3 and 4.

It is assumed that the over the air firmware upgrade process can can get interrupted due to intermittent network failure. Similarly, various other network issues can be seen and are summarized in Figure 2.

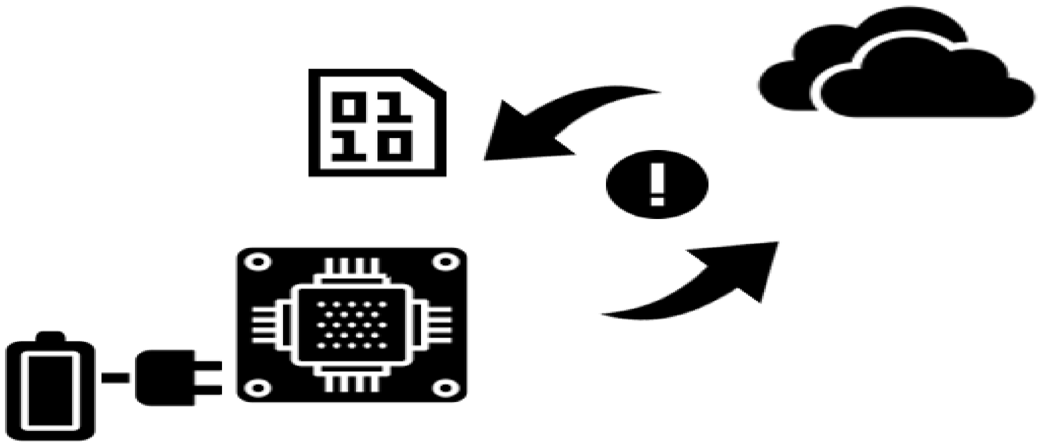


Figure 2. Network error during firmware update.

As shown in Figure 3, It is possible that a power reset or a system shutdown during the upgrade process is applied. It will also interrupt the fimware upgrade process.

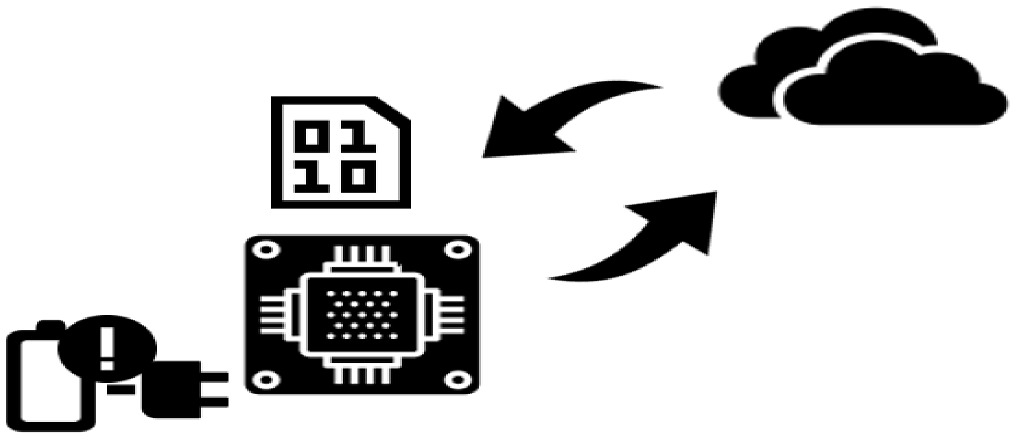


Figure 3. Power outage during firmware update.

Another software upgrade failure scenario is the firmware file corruption. It can happen due to multiple reasons; the firmware file could be corrupted at the source itself or could get corrupted at the destination due to either of the issues shown in Figure 2 and 3. This scenario is shown in Figure 4.

In the presence of such errors, an incorrectly designed system, may report undefined behvaior or in worst-case, could have catastrophic consequences. Therefore these issues and other failure issues should be handled properly by the overall system.

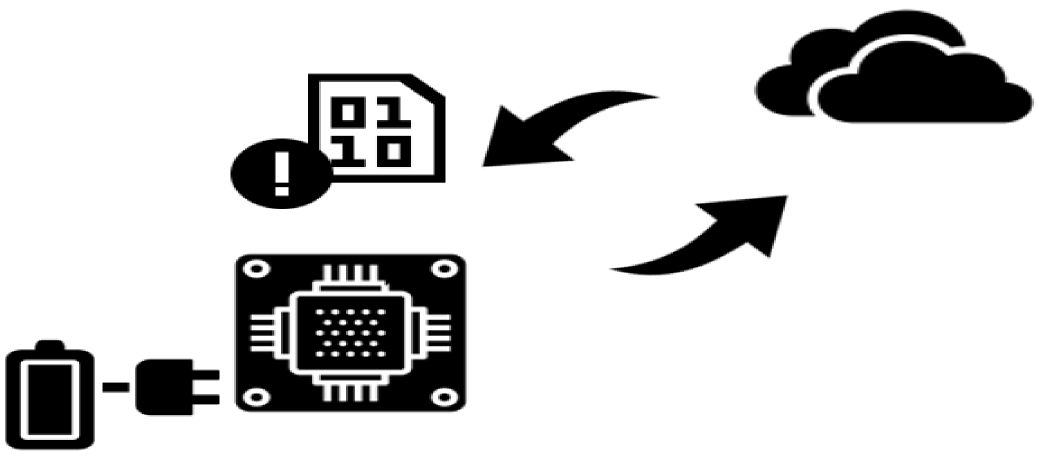


Figure 4. Firmware image corruption during network download.

2.2. Embedded systems

An embedded system is a special-purpose system in which the computer is completely encapsulated by the device it controls. Unlike a general purpose computer, an embedded system performs pre-defined tasks, usually with very specific requirements. Card readers, Calculators, Smoke detectors etc. are all examples of Embedded Systems. Since the system is dedicated to specific tasks, such systems are highly optimized for size and cost.

2.3. In-application programming

Flash memory is a special type of non-volatile memory that stores the application. It can be either integrated into the MCU or is soldered on the Printed circuit board (PCB) externally. There are two programming methods to reprogram this memory area; In-System Programming (ISP) and IAP. With ISP, the device can be re-programmed in the circuit by using specialized hardware such as JTAG or SWD interface. The re-programming process is started manually, during which processor is halted. It requires special circuitry. IAP on the other hand allows the running application to re-program the on-chip Flash memory. During the IAP process, the application continues to run. With IAP, it is possible to implement applications that can be updated remotely without the need of physical presence of a technician. IAP allows a cost-effective method to perform a software enhancement, once the system has been already shipped. The central theme of this thesis is to use IAP techniques to build a robust and portable bootloader that facilitates re-programming the on-chip Flash memory over different communication channel such as GSM/GPRS network and Wi-Fi.

2.4. Firmware

Firmware is a software program that typically runs on Embedded devices. It is responsible for overall Embedded Systems functioning. It is often stored on the on-chip Flash memory. The ability to upgrade the Firmware over the air, in the field, is the central theme of this thesis.

2.5. Bootloader

Bootloader is an application whose primary purpose is to allow a system software to be updated without the use of specialized hardware such as a JTAG programmer or SWD interface. The boot-loader manages the systems images. There are many different sizes and flavors to embedded boot-loaders. They can communicate over a variety of protocols such as Universal Synchrononus Asynchronous Receiver Transmitter (USART), Controller Area Network (CAN), I-Squared-C (I2C), Ethernet, USB etc. Systems with boot-loaders have at least two program images coexisting on the same micro-controller and must include branch code that performs a check to see if an attempt to update software is in progress.

2.6. Flash memory

Flash memory is an electronic non-volatile computer memory storage medium that can be electronically reprogrammed. Embedded devices, such as those that are used during this thesis work have on-chip Flash memory, which stores the bootloader and the application firmware. Storing data in Flash memory is quite different from Static Random Access memory (SRAM) and has its own nuances. Since, Flash memory is a non-volatile memory, it retains the data even after the power supply has been removed. Typically, an erase operation is needed before writes can be perfored. Based on the type of Flash memory, the erase block varies. For example, on NAND-type Flash memory, data may be written in blocks, whereas on NOR-type Flash memory, a machine word size data can written to an erased location or can be read after. Erase, read and write granularity of MCUs used in this thesis is shown in Table 2 in page 27.

2.7. SRAM

Static Random Access Memory is a volatile memory that retains data bits in memory as long as the power is being supplied. Unlike dynamic RAM (DRAM), which stores bits in cells consisting of a capacitor and a transistor, SRAM does not have to be periodically refreshed. Static RAM provides faster access to data and is more expensive than DRAM. Typical applications of SRAMs are low capacity memory products such as on-chip memory on microcontrollers and L1, L2, L3 caches on computer.

2.8. GSM/GPRS

Global System for Mobile (GSM) Communications is a standard developed by the European Telecommunications Standards Institute (ETSI) to describe the protocols for second-generation (2G) digital cellular networks used by mobile phones. As of 2014 it has become the default global standard for mobile communications. General Packet Radio Services (GPRS) is a packet-based wireless communication service that promises data rates from 56 up to 114 Kbps and continuous connection to the Internet for mobile phone and computer users. GPRS is based on Global System for Mobile (GSM) communication and complements existing services such circuit-switched cellular phone connections and the Short Message Service (SMS). For this thesis, A GSM/GPRS hardware module (modem) was used for downloading Firmware over the air.

2.9. AT+ command

AT commands are instructions used to control a modem. Since each command starts with "AT", such commands in general are called as AT+ command sets. The ETSI GSM 07.07 (3GPP TS 27.007) specifies AT style commands for controlling a GSM phone or modem and The ETSI GSM 07.05 (3GPP TS 27.005) specifies AT style commands for managing the Short Message Service (SMS) feature of GSM.

2.10. Wi-Fi

Wi-Fi is a wireless technology that uses radio waves to provide network connectivity. It is a wireless local area network (WLAN) technology, based on the IEEE 802.11 standards. Generally, it operates in 2.4Ghz Industrial, Scientific and Medical (ISM) band.

3. System architecture

As part of thesis work, an implementation was developed. This chapter covers the overall system architecture of the implementation.

3.1. System overview

The overall system architecture is shown in Figure 5. The center theme of this thesis work is to perform firmware upgrade on MCUs that do not have the ability to directly access the remote firmware repository. However, after interfacing a Radio Frequency (RF) module over serial communication interface, over the air firmware upgrade can be achieved on such MCUs.

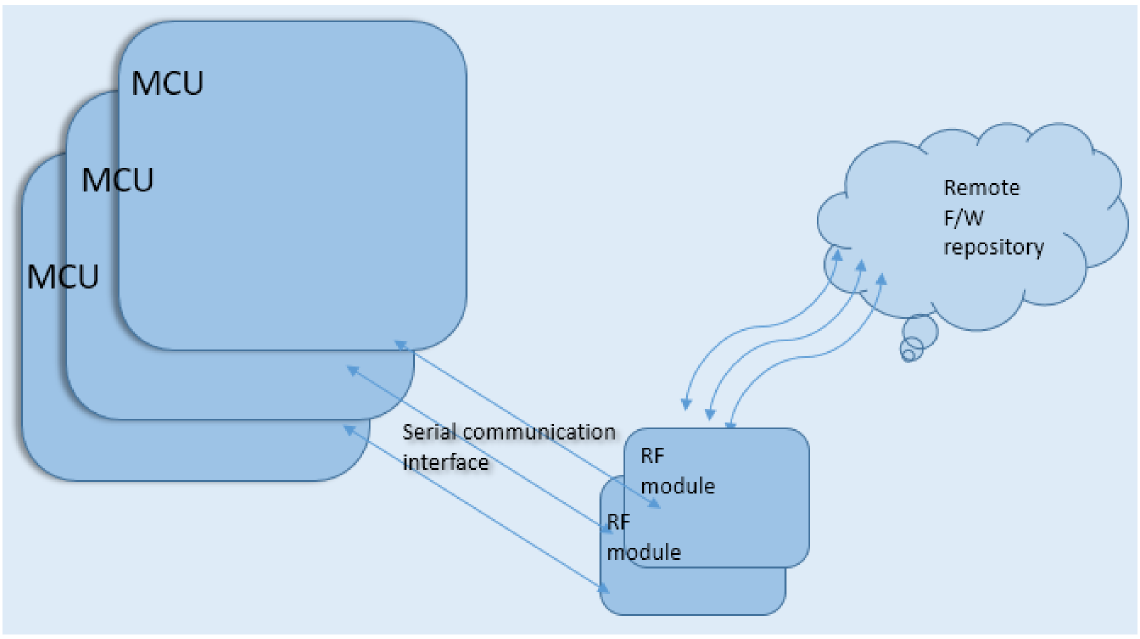


Figure 5. Overview of system architecture

There are several different types of Radio Frequency (RF) modules that could be interfaced with the target MCU. Broadly they are classified as tramistter, receiver and transreceiver. The RF module should be selected, based on the the area of system application. Since, bi-directional communication between the remote firmware repository and RF module is required, transreceivers are selected.

Remote firmware repository stores the firmware files and associated checksum. A client-server architecture is used for communication between the RF module and the remote firmware repository. A fully functional server has content management, authentication techniques, high availability, disaster recovery, and security features, however these topics are outside the scope of this thesis. As a bare minimal requiremet for this thesis, the remote server should be capable of responding to Hypertext Trasfer Protocol (HTTP) request and should be capable of storing multiple files that could be individually accesssed.

As part of this thesis work, software update was performed on three different target MCUs over GSM/GPRS network. Software update over Wi-Fi support is planned for future work. Various system blocks are covered next.

3.2. Memory overview

Each microcontroller has on chip flash memory to store the program. This area of main flash memory is divided into four different regions; bootloader area, application area, backup application (for dual mode firmware update) area and reserved area (for storing few parameters). Since the bootloader is in-charge of programming the application in the flash memory, it is of utmost importance to understand the flash memory organization.

On ARM-Cortex version 6 and 7 based SOC, the linear address space is 4GB. For example, Figure 6 shows the memory map on STM32F051 SOC which is based on ARM cortex M0 series processor. The addressable memory is divided in 8 main blocks, each of 512 MB. Program memory data memory, registers and I/O ports are organized within the same linear address space.

Flash is broken into divisible sections. The smallest section of flash is called page. Pages are grouped together into a larger structure called sectors. Sectors are combined to form blocks. Each microprocessor is different as to how these sections of flash can be manipulated. The microprocessor, selected for this thesis allow word level (32 bit) write and sector level write protection. In most cases the smallest section of Flash that can be erased is a sector, however on some, the erase granularity is page level. Flash memory size, alignment and access details for SoCs selected for this thesis are described in Table 2. It should be noted that one has to erase a sector before it can be programmed.

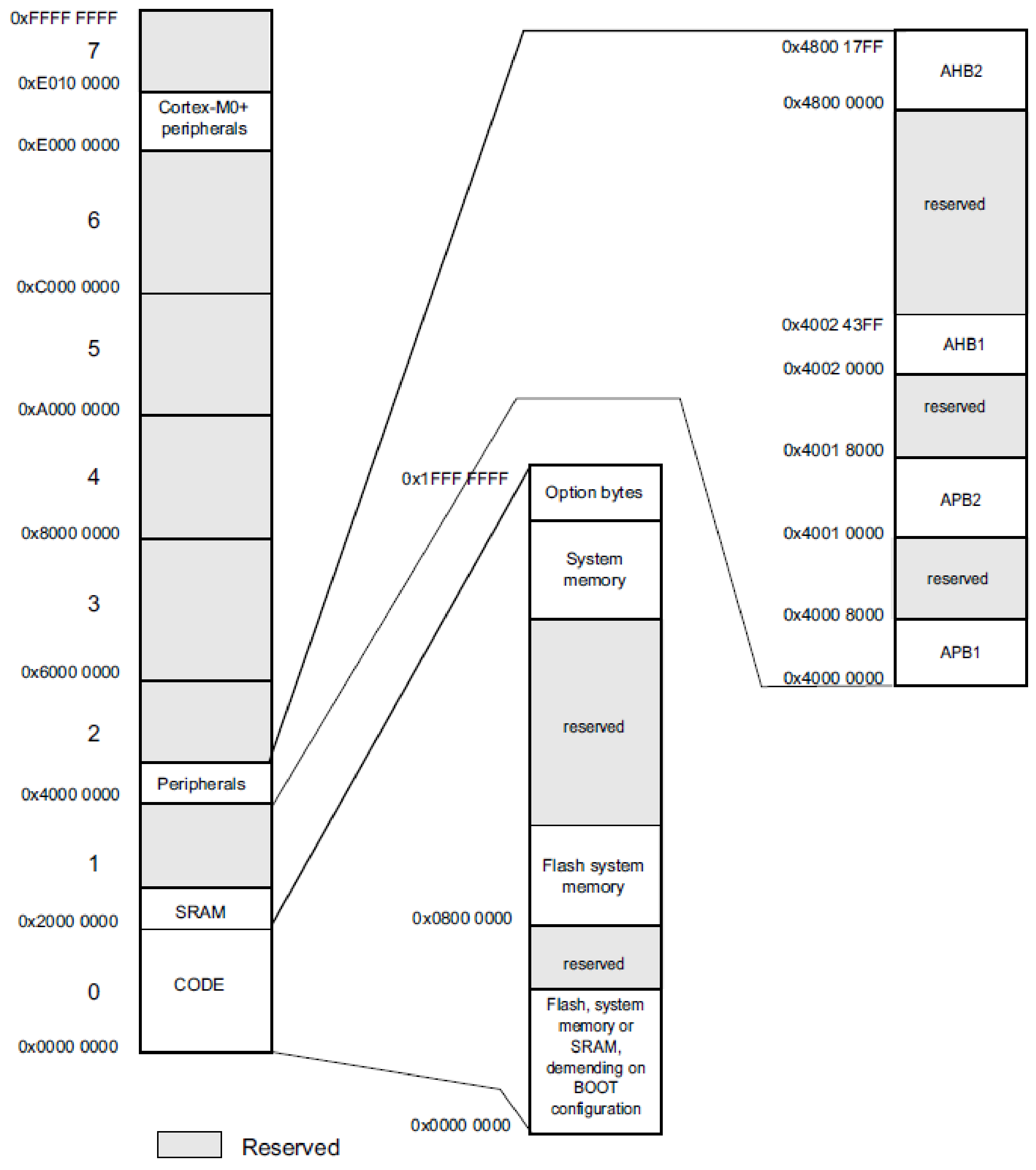


Figure 6. Memory map of STM32F0xx SoC[4].

Table 1. Main flash memory details

Component	**MCU**		
	STM32F072CB	**STM32F051R8**	**STM32F407VG**
Start address	0x08000000	0x08000000	0x08000000
End address	0x08020000	0x08010000	0x08100000
Total flash memory	128 KB	64 KB	1024 KB
Page size	2 KB	1 KB	N/A
Number of pages	64	64	N/A
Erase sector size	4 KB	4 KB	Hybrid
Initial content of the memory	0xFF	0xFF	0xFF
Memory Endianess	Little	Little	Little
Processor address space	4 GB	4 GB	4 GB
Flash write protection granularity	Sector	Sector	Sector
Erase granularity	Page	Page	Sector
Flash memory access alignment requirement	Word (32 bits)	Word (32 bits)	Word (32 bits)

3.3. Serial data communication

Serial communication over UART interface is used for communication between the RF module and the MCU. The STM32 MCUs used for implementation purposes have two or more UARTs, hence the other UART interface is used for printing debugging messages on serial console.

The universal synchrononus asynchronous receiver transmitter (USART) offers a flexible method of full-duplex data exchange with external equipment. The communication can be either synchronous (by means of using a clock) or can be asynchronous (by using special signal along with data). The serial data is transferred one bit at a time. Because, the serial interfaces are relatively cheap, they are implemented on almost all MCUs.

Any USART bidirectional communication requires a minimum of two pins (Receive data In (Rx) and Transmit data Out (Tx). Serial data are transmitted and received through certain pins (refer pinout configuration in appendix 3) on STM32 MCU. The frames are comprised of

- An Idle Line prior to transmission or reception
- A start bit
- A data word (7, 8 or 9 bits) least significant bit first
- 1, 1.5, 2 stop bits indicating that the frame is complete
- The USART interface uses a baud rate generator
- A status register (USARTx_ISR) [8]
- Receive and transmit data registers (USARTx_RDR, USARTx_TDR) [8]
- A baud rate register (USARTx_BRR) [8]
- A guard-time register (USARTx_GTPR) in case of smartcard mode. [8]

The 8 bit word length is selected for this thesis and the frame format is shown in Figure 7.

MCU uses USART transmit pin (USART_Tx) to send AT+ command to the RF module. The response to the AT+ command arrive on the USART receive pin (USART_Rx). MCU uses interrupt mechanism for communication with the RF module. Any data sent from the RF module to MCU on USART receive pin generates an interrupt on MCU. Upon receieving an interrupt, USART specific interrupt handler gets invoked and

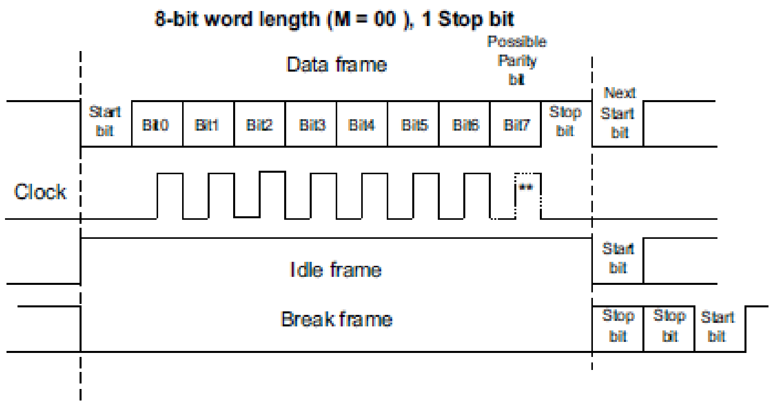


Figure 7. 8-bit frame format for USART communication.

appropriate action is then taken on the recieved data. STM32 MCU uses on-chip Nested Vectored Interrupt Controller (NVIC) and peripheral specific interrupt register for interrupt handling. Refer, Appendix 1 for more information on NVIC and ARM processors.

4. Software architecture

This chapter covers the bulk of the thesis work. The components necessary to develop a system, capable of over the air firmware upgrade are broken into separate modules for ease of understanding.

4.1. Bootloaders

The first part of this chapter discusses Bootloader requirements. The later part of this section covers the bootloader itself and its behavior.

4.1.1. Bootloader requirements

Each Embedded Systems project will have its own requirements for the bootloader design, however there are few requirements that are common to all bootloaders. They can be grouped into seven fundamental requirements that are common to all boot-loaders. They are:

Table 2: Bootloader requirements

Requirement	**Sub-requirement**
Ability to switch or select the operating mode (Application or bootloader)	Bootloader has the intelligence to locate the existence of valid application in the Flash memory and transfer control to application for normal operation.
	Bootloader has to distinguish between the normal operation and application firmware update message.
Communication interface (USB, CAN, I2C, USART, etc)	Application firmware can be transferred over any of the serial interface but for thisis purpose, UART was selected as an interface to RF module (GSM/GPRS or WIFI).
Firmware file format (binary, S-Record, hex, intel, toeff, etc)	The application image shall be sent in binary format. Benefit of using a binary file stems from the fact that it is significantly smaller in size than the Intel hex format or absolute and executable object file (.axf) generated by the armlink linker and hence it will save network bandwidth, when downloaded over GSM/GPRS network or over Wireless network. Although the Intel hex file format is straight forward, it requires additional

Requirement	Sub-requirement
	parsing.
Flash system (read, write, erase)	The boot-loader shall be capable of erasing application section of flash. Application section will be further divided into two regions; current application and backup application. Keeping a backup of current application firmware provides the possibility of rollback. However, the rollback will be completely transparent and will not require any explicit instruction.
	The boot-loader shall write application image records to flash.
	The boot-loader shall be located in the first few sectors of main flash memory.
SRAM (read, write, erase)	On-chip SRAM shall be used for sharing data between the bootloader and application.
Application checksum	Bootloader shall be capable of calculating the application checksum.
Code security	Bootloader will protect itself from accidental modification by the application.

4.1.2. Bootloader behavioral models

A bootloader is not much different than the other application. But, this is the first application that executes, on power-on-reset and is in charge of transferring system control to other application. Bootloader has the capability to erase and program a new application in place. On a typical embedded system, resources are scarce, especially the on chip memory (flash memory and SRAM) and hence bootloader designer has to be very judicious in deciding the bootloader features. It should use the minimum amount of peripherals in order to maximize the amount of flash memory space that will be available for the application code.

Historically, there are two behavioral models that describe how a boot-loader can behave. In the first model, the boot-loading process is completely automated and self-contained within the system. An example of this would be an Secure Digital (SD) card boot-loader. The boot-loader would automatically detect the new firmware and manage its own flashing process. Commands from an external source would not be required to successfully carry out the boot-loading process.

In the second model, the system does not automatically handle the boot-loading process itself. Instead, the boot-loader initializes into an idle state and awaits instructions from an outside source. This source would typically be a Personal Computer (PC) based software application that commands the boot-loader into the different states necessary to flash a new image onto the system. The primary reason for the external software application to command the process is that in most applications without an SD card, there is not sufficient space to retrieve the whole software image. Instead, an external source with the image acts as the master of the boot-loading process Here the application on the embedded systems and the PC software operate in client server architecture. For this thesis, the first model is selected.

4.1.3. Bootloader classification

Figure 8 shows three different methods to upload the code.

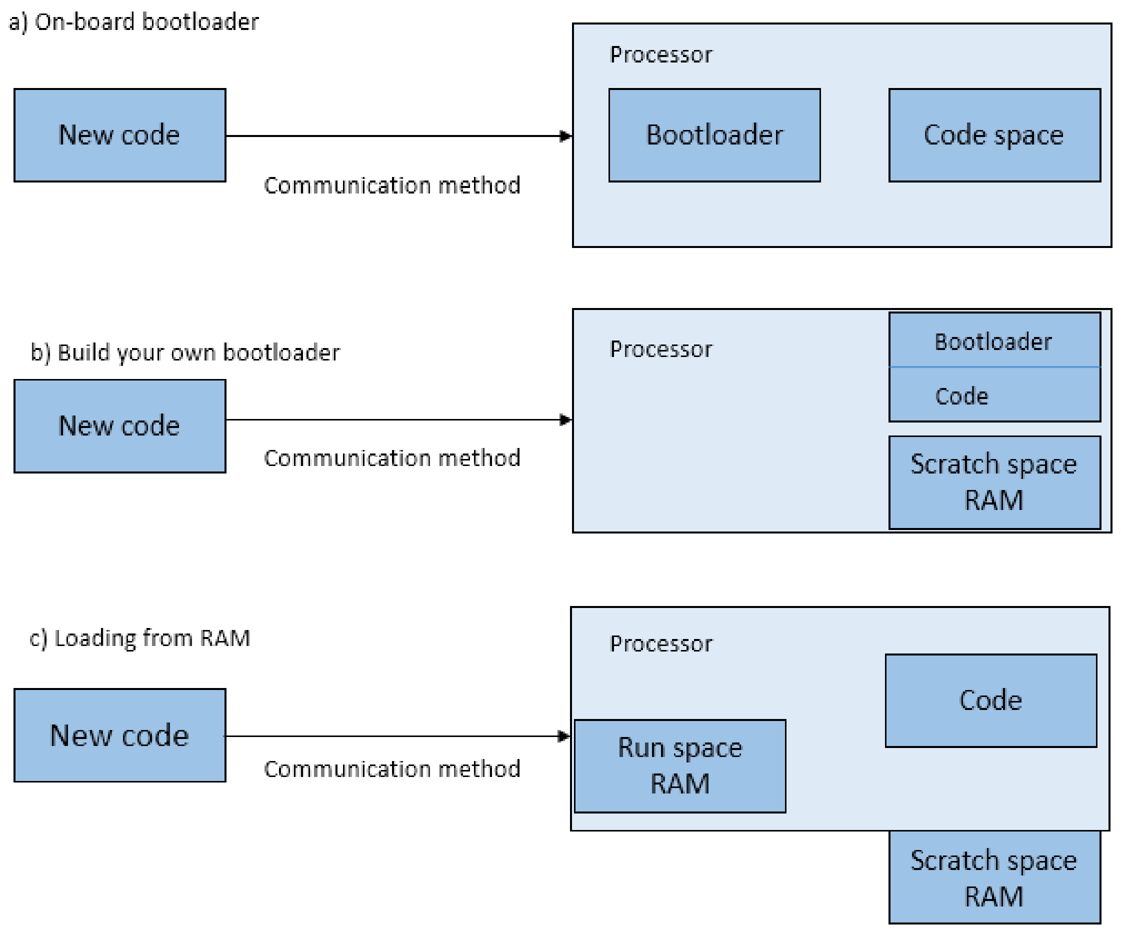


Figure 8. Code upload method [3].

4.1.4. On board bootloader

Some processors such as NXP Kinetis K8x [5] have an internal bootloader that will load code from an external source if the right I/O pins are set. In an ideal world, one would just set the I/O pin to upload the new code to the system. This could be generally triggered when the system powers up. The bootloader would automatically load the data into the code space. The new code is transferred to the processor using one of the communication methods (typically some serial interface such as USART). The bootloader code inside the chip reads the data and transfers the code in the code space. This is the simplest bootloader among the three.

4.1.5. Custom bootloader

A custom bootloader shares the code space with the application. The benefit of custom bootloader is the flexibility. This is the bootloader of choice for this thesis. Programming Flash memory requires two step process; erase and program. Typically, Flash memory is divided into sectors and most of the flash memories have sector level erase granularity and word level program granularity. This means, that to program a word (4 bytes) at the same address, one has to erase an entire page (typically 2KB) first. A primitve bootloader, shown in Figure 9, would erase the application area from the flash memory and then program the new application image.

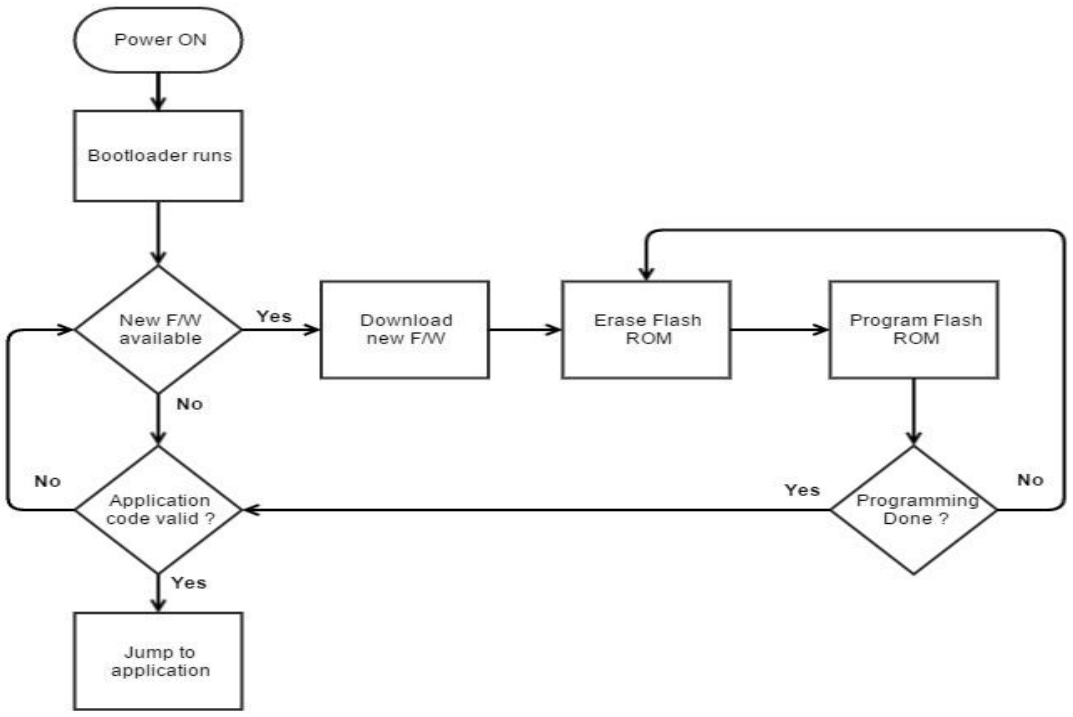


Figure 9. A primitive bootloader.

This approach has multiple issues.

- The new application image could be corrupted from the source.
- Application image could get corrupted due to communication errors.
- Partial erase.
- Programming error could happen, during the process of flashing the new application image.
- Programming process could get interrupted.

These faults will lead to system failure. System will exhibit different behavior based on the fault scenario. There are multiple approaches to solve these problems. For example, a checksum could detect the corrupt image, a copy on write approach will make sure that the old application is preserved until the new application is correctly written into the flash memory. The bootloader developed as part of this thesis takes care of such failure scenarios in consideration and implements tiered approach for a failsafe firmware upgrade. A copy on write approach is shown in Figure 10 below. The bootloader needs to make a copy of current application in a different region of flash memory area. It then erases the application area and starts to program the new application. Bootloader compares the data written on the flash memory with the application buffer in every pass and in case, there is a programming error, bootloader can recognize it and then it has to initiate the recovery process by programming the older application from the backup area to application area and notify the application that the firmware update process has failed. A sample procedure that copies the current application in the backup area is shown in Figure 10.

The application firmware backup routine starts with unlocking the flash controller to enable write access to flash memory. On a power reset, the flash memory is locked to prevent accidental modification to program memory. Since the flash memory requires erase operation before it can be programmed, the backup routine first performs erase operation on backup area (the memory partitioning sceheme is shown in Table 4) of the main flash memory. The erase granularity depends on the target MCU. Most of the MCUs allow sector level erase operation, however some MCUs such as STM32F051 have page level erase granularity (more information in Table 1). After a successful erase operation, a successive read operation on the same memory location returns all 0xFFs, i.e. all bits are set to ’1’. Application firmware is then programmed in backup area in word-sized chunks. The word-sized program requirement is imposed by the underlaying flash memory hardware. Once the programming is done and varified by reading back the content, flash controller is locked. At this point, flash memory has two identical copies of application firmware and bootloader.

```
#define BACKUP_ADDRESS
(uint32_t)0x0800B000
#define APPLICATION_ADDRESS
(uint32_t)0x08007000
#define APPLICATION_SIZE        0x4000
int copy_image(void)
{
    __IO uint32_t dst = BACKUP_ADDRESS;
    __IO uint32_t src = APPLICATION_ADDRESS;
    __IO int i;
    /* Unlock the Flash Program Erase controller */
    FLASH_Unlock ();
    /* page-wide erase of backup area */
    if (FLASH_If_Erase(dst))·
    {
        FLASH_Lock();
        return ERROR;
    }
    /* word-wise program */
    /* start copying */
    for (i = 0;  i < (APPLICATION_SIZE/4); ++i)
    {
        while (FLASH_ProgramWord(dst, *(uint32_t
*)src) != FLASH_COMPLETE) {}
            if ((*(uint32_t *)dst) != (*(uint32_t
*)src))
            {
                /* Lock flash */
                FLASH_Lock();
                return ERROR;
            }
        src += 4;
        dst += 4;

    }
    /* Lock flash */
    FLASH_Lock();
    return SUCCESS;
}
```

Figure 10. Backup of current application.

The corresponding, improved bootloader is shown in Figure 11. All blocks except those highlighted, are same as before.

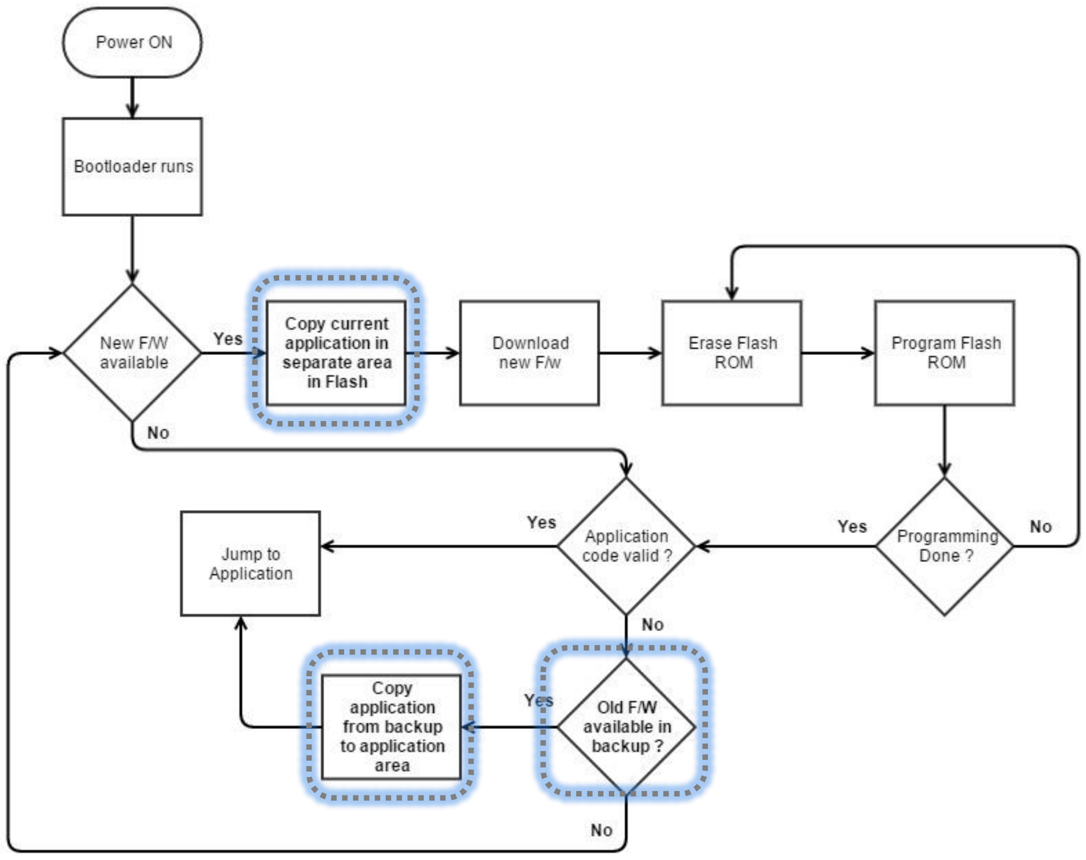


Figure 11. An improved dual mode bootloader.

The major benefit of dual mode bootloader is that we always have a applicaiton as a backup and if something goes wrong during or after the update process, bootloader could be instructed to initate a rollback. During a successful firmware upgrade, the main flash memory contents over a period of time is shown in Figure 12. Unless programmed, all bits in flash memory are set to ’1’. Therefore, a read operation on empty block or an erased block returns 0xFF, i.e. all bits set as ’1’ and is shown in the Figure as 0xFF..FF.

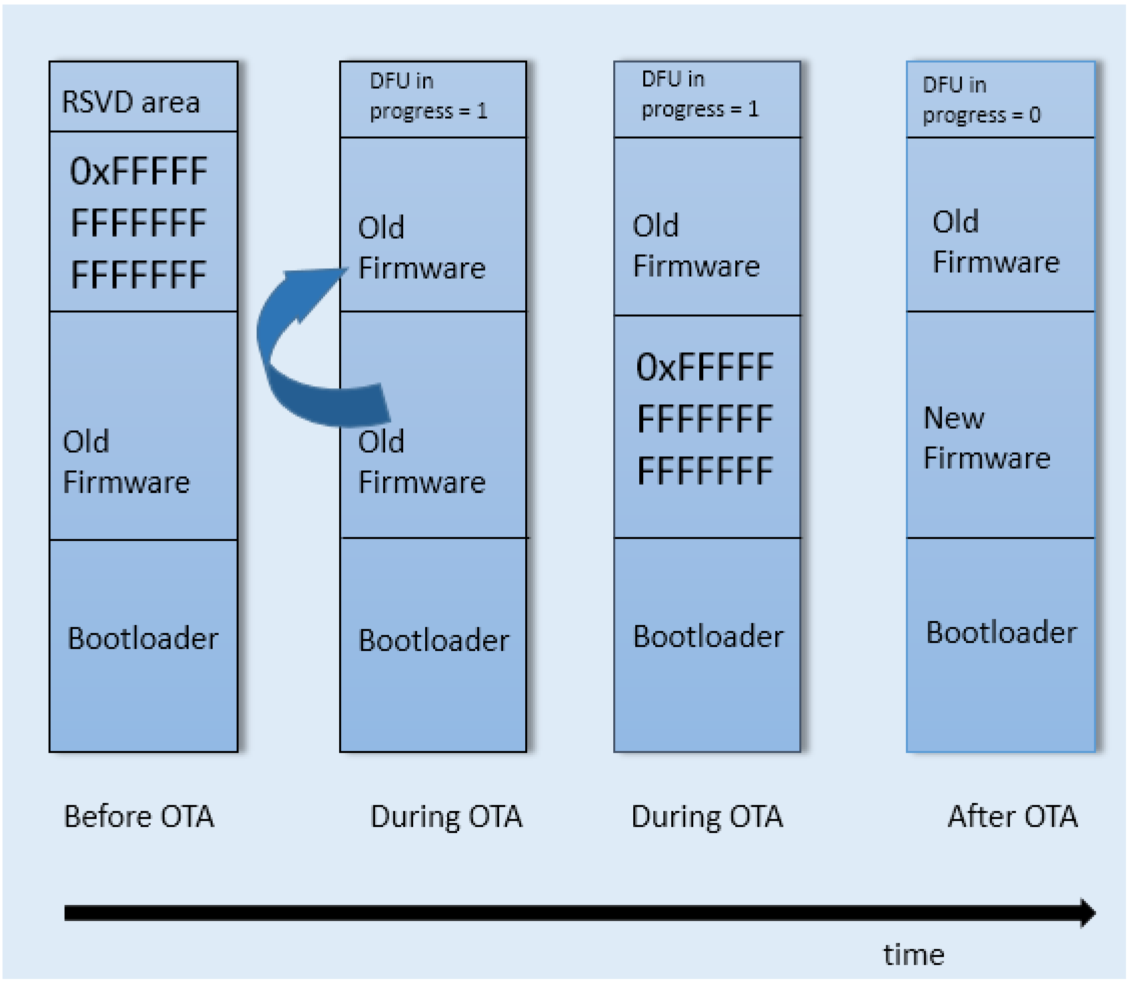


Figure 12. Flash memory contents vs time chart.

To overcome, transient Flash programming errors, retries can also be used. This type of dual mode firmware update process reduces the effective area of Flash memory for application, but improves the overall reliability of the system. Custom bootloader is the focus of this thesis and is discussed in more details in this chapters.

4.1.6. Boot from SRAM

Sometimes, it is needed to update the bootloader itself. Recall that a bootloader is nothing but an application with some specific requirements. To update the resident bootloader, the old bootloader first download the new bootloader in SRAM, and then passes control to new bootloader. The new bootloader then erases the resident bootloader in flash memory and then program the new resident bootloader. Once the new bootloader is programmed, normal operation can resume. This type of bootloader is more complex and this type of bootloader has the possibility of making the system useless and unable to ever load a valid code. For example, if the power reset happens after the old resident bootloader has been erased. Since updating the bootloader itself is not the topic of interest for this thesis, therefore, this type of bootloader will not be discussed further.

4.2. Application firmware behavior

The behavior of the application image is mostly not of any interest to the boot-loader designer except in few aspect; the application needs to be capable of receiving a command to enter the boot-loader, pass some parameters, such as retry counter and be able to run from another load address than the default load address (since the default load address is occupied by the bootloader). This means that the application needs to have following capabilities:

- Instruct bootloader to start software update process.
- Reset the system to initiate branching decision by bootloader.
- Pass few parameters to bootloader
- Inform the server (source of firmware image) about the firmware upgrade result
- Be able to share the program memory with bootloader and a backup image
- Be able to run from a different memory location.

The best place for the application to store a value that can be detected by the branching code is on chip SRAM. SRAM in most cases exists in a memory space that can be shared by both the boot-loader and the application. When the application receives a request to enter the boot-loader, the application can write a value to SRAM and then perform the second function which is to reset the system.

The reset of the system is performed by doing a soft reset. Before performing a soft reset, it is made sure that no I/O operations, such as access to Flash memory is in progress, interrupts are disabled and clock domains are shut down. There is a hazard in sharing information between bootloader and application in SRAM. If after setting certain information in SRAM by the bootloader, a power reset (power lost to the system), happens, that information in SRAM will get lost and hence bootloader will not take appropriate action. Application overcomes this problem by:

- Keeping the time interval between updating SRAM and performing a soft reset to a minimum.

- Comparing current application firmware revision with the expected firmware revision. It is done by consulting with the server.

In ARMv6-M, the Application Interrupt and Reset Control Register (AIRCR) provides a mechanism for a system reset. Setting the AIRCR.SYSRESETREQ [25] control bit to 1 requests a reset by an external system resource. Setting the SYSRESETREQ bit to 1 does not guarantee that the reset takes place immediately. A typical assembly code sequence to synchronize reset following a write to the relevant control bit is shown in Figure 13.

```
        DSB
Loop    B Loop;
```

Figure 13. Assembly code sequence to synchronize reset.

The Data Synchronization Barrier (DSB) shown in Figure 13 is an instruction in ARM processors [26]. It is a special kind of memory barrier. It makes sure that no instruction in program order after this instruction executes until this instruction completes.

System components that are reset by this request are implementation defined, hence another method to soft reset the system can be performed by entering into an infinite loop, allowing the watchdog timer to reset the system.

4.3. Start-up branching

Start-up process is shown in Figure 14 Reset vector is the first thing that runs at startup. It is a hardware specific code. It performs simple functions like setting up the processor into a pre-defined steady state by configuring registers etc. Then it will jump to the startup code.

Startup code is the first software-specific code that runs. Its job is basically to set up the software environment so that C code can run on top. For example, C code assumes that there is a region of memory defined as stack and heap. These are usually software constructs instead of hardware. Therefore, this piece of start-up code will define the stack pointers and heap pointers Interrupt Request Lines (IRQs) and such. Variables that need to be initialized and also certain parts of memory that need clearing are done here. Basically, everything that is needed to move things into a 'known state'. At the end of the routine, it will execute main function

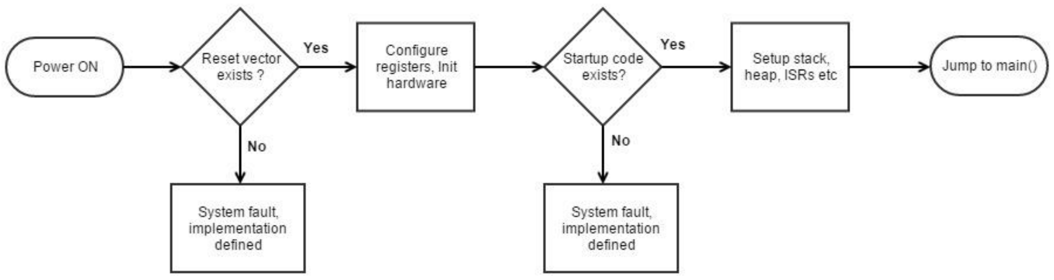

Power ON
Reset vector exists ?
Yes
No
Configure registers, Init hardware
Startup code exists?
Yes
No
Setup stack, heap, ISRs etc
Jump to main()
System fault, implementation defined
System fault, implementation defined

Figure 14. MCU start-up sequence.

At this point, there are at least two different software images that can be loaded and executed by the MCU, the boot-loader, application and possibly a back-up application image. For the purpose of this thesis, it was decided that the botloader will be the first C application that will execute. This application is mostly hardware agnostic as such things are taken care by the reset vector, startup code, and compiler. For portability, the application is statically linked to vendor provided libraries and uses hardware abstraction layer. As part of the boot-loader image code, a branching algorithm is included that handles the decision making process of loading the application image or handling device firmware update request.

Assuming that the application start address is always fixed, and if there was no firmware update request, a primitve bootloader will simply jump to this location. But what if there is no application at the application start address. There are couple of reasons, that this can happen.

- Bootloader exists in Flash memory, but there is no application yet and a power cycle was performed.
- Application firmware was erased as part of firmware update process but before the new application could be programmed, some error or power cycle occured. It lead to a blank application area.

In such cases, a primitive bootloader will jump to the application start address without doing any validation and the system might spin forever in an infinite loop, doing nothing. Therefore a verification that the application stack pointer exists is imperative. The check is hardware specific and on ARM cortex based SoC, it can be done as show in Figure 15.

```
#define APPLICATION1_ADDRESS
(uint32_t)0x08006000
#define SP_RANGE                    0x2FFE0000
#define SRAM_BASE_ADDR              0x20000000
#define IS_VALID_SP(address)        (((*(volatile
uint32_t*)address) & SP_RANGE) == SRAM_BASE_ADDR)
typedef  void (*pFunction)(void);
pFunction Jump_To_Application;
uint32_t JumpAddress;
if (IS_VALID_SP(APPLICATION1_ADDRESS))
{
    /* Application found, Jump to user application
*/
    JumpAddress = *(__IO uint32_t*)
(APPLICATION1_ADDRESS + 4);
    Jump_To_Application = (pFunction) JumpAddress;
    /* Initialize user application's Stack Pointer
*/
    __set_MSP(*(__IO uint32_t*)
APPLICATION1_ADDRESS);
    /* Jump to application */
    Jump_To_Application();
}
```

Figure 15. Validate stack pointer and perform a jump to application.

It is the most basic test and works in most of the scenario, however, there is a problem with this approach. The above check does not validate if the application is indeed correct. If the application image was corrupted from the source or while programming (as shown in Figure 4), then the jump to application will lead to undefined behavior. One easy solution is to perform a checksum validation before jumping to application. Once the application development is finished, its checksum is calculated and is stored at the remote server in plain text format. Once the new application has been programmed in memory, bootloader uses the same algorithm that was used before to generate the application checksum. It then compares the calculated checksum with the checksum value stored at the remote server. If the checksum do not match, it indicates, application firmware corruption, either at the source or during the download process. In that case, bootloader will not jump to a corrupted application firmware. It improves reliability of the system. Figure 16 shows the checksum validation routine in the bootloader. While developing the bootloader, mobile embedded systems, operating in lossy network with firmware upgrade over the air requirement, were kept in mind and hence, the two pass application validation was implemented.

```
int calc_checksum(void)
{
    uint32_t cksum;
    /* calculate checksum of downloaded firmware */
    crc_init();
    cksum = CRC_CalcBlockCRC((uint32_t
*)(APPLICATION1_ADDRESS), (APPLICATION_SIZE/4));
    /* compare calculated checksum with the known
checksum */
    if (cksum != orig_cksum)
    {
        /* erase the application area, so that the
         * main routine will not jump into a
partially
         * programmed firmware.
         */
        FLASH_If_Init();
        if
(FLASH_If_EraseAppArea(APPLICATION1_ADDRESS))
        {
            FLASH_Lock();
            return ERROR;
        }
        FLASH_Lock();
        return ERROR;
    }
    return SUCCESS;
}
```

Figure 16. Checksum validation before jumping to the application.

After introducing the checksum validation, the system becomes more robust to faults, however this failsafe technique introduces two problems:

- Every time the system boots, it needs to connect to the remote server over GSM network, download checksum value, re-calculate checksum of application in the Flash memory and only after validating the checksum, jump to the application. Imagine, the system being power cycled just ten times a day or a mobile system, that frequently changes it's geographical position; The incurred latency from bootup to an actual functional system is unacceptable in most of the cases.
- During the system development process, typically, one would be using ISP progrmaming method such as dedicated SWD or JTAG interface to progam the application in the main Flash memory. On a system hard reset bootloader will find

correct stack pointer but the checksum validation will fail as bootloader has no information about the application's checksum, which was programmed using In-circuit programming tool. Therefore despite having a correct application in the application area of main flash memory, bootloader will fail to jump to application.

Problem 1 can be solved simply by storing the known checksum in non-volatile memory, however it still doesn't solve the second problem at hand. Essentially, we would like to achieve a one time checksum validation and also support ISP methods during the development process. The bootloader, developed as part of this thesis solves these problems by using a flag in non-volatile memory, that represents device firmware update in progress status. This flag is set, just before the DFU process begins (application download) and reset, immediately after new application firmware has been downloaded, programmed and its checksum has been validated. On a system reset, bootloader checks the status of this flag and only after it finds that the flag is not set, it proceeds further with the stack pointer validation, before jumping to the application. Bootloader, no longer needs to perform checksum validation on every reset; It's done only once, during the DFU process. Also, ISP is not an issue anymore as, the only time, device-firmware update-in-progress flag is set, is during DFU over the air process only. Another big advantage of this approach is that this check solves prospective problems that could occur due to power reset during the DFU process. This improvement is shown in Figure 17.

```
#define RSVD_FLASH_PAGE_ADDRESS
(uint32_t)0x0800f000
#define DFU_IN_PROGRESS         0x4B1D
#define IS_DFU_IN_PROGRESS      ((*(uint32_t
*)RSVD_FLASH_PAGE_ADDRESS) == DFU_IN_PROGRESS)
if (IS_VALID_SP(APPLICATION1_ADDRESS) &&
!(IS_DFU_IN_PROGRESS))
{
    prep_jump_to_application();
    /* Jump to user application */
    JumpAddress = *(__IO uint32_t*)
(APPLICATION1_ADDRESS + 4);
    Jump_To_Application = (pFunction) JumpAddress;
    /* Initialize user application's Stack Pointer
*/
    __set_MSP(*(__IO uint32_t*)
APPLICATION1_ADDRESS);
    /* Jump to application */
    Jump_To_Application();
}
```

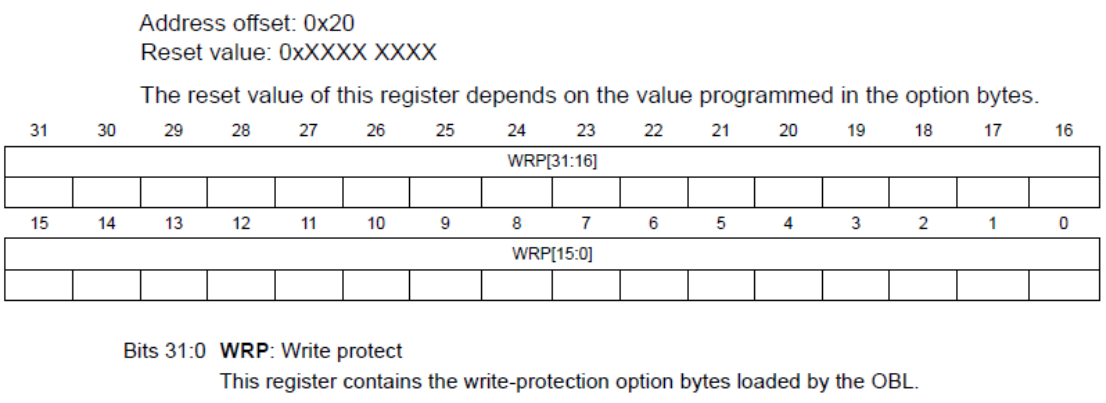
Figure 17. Improved branching to application.

In this Figure, prep_jump_to_application() is a small helper function, that disables all interrupts that were enabled before and shuts down all clock domains. It is needed for smooth transition from bootloader to application. Refer Appedix 4 for more information on bootloader entry function.

4.4. Memory partitioning

Application may try to overwrite the bootloader area and hence areas occupied by bootloader must be protected. It can be achieved in number of ways. On STM32 MCU, the write protection of certain areas (In this case, area occupied by bootloader) of memory can be achieved by memory protection feature. It is activeted by configuring the WPRx option bytes (shown in Figure 18) and then by reloading them by setting the OBL_Launch bit in the FLASH_CR register (shown in Figure 19).

If a program or an erase operation is performed on a protected sector, the Flash memory returns a WRPRTERR protection error flag in the Flash memory Status Register (FLASH_SR).


Address offset: 0x20
Reset value: 0xXXXX XXXX

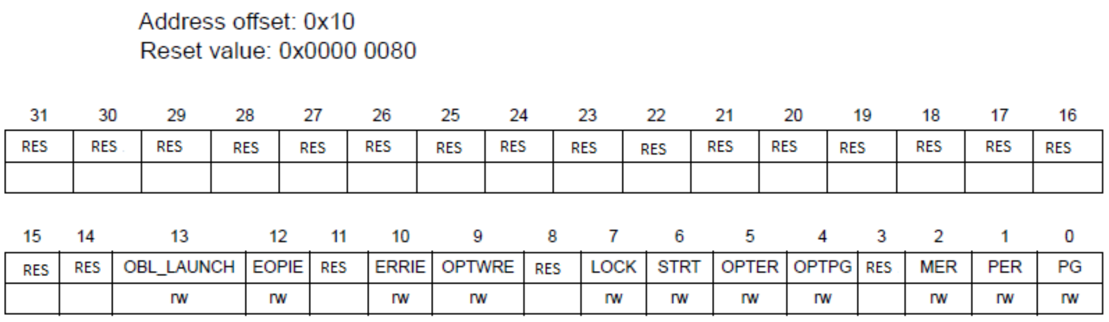
The reset value of this register depends on the value programmed in the option bytes.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
WRP[31:16]															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
WRP[15:0]															

Bits 31:0 **WRP**: Write protect
This register contains the write-protection option bytes loaded by the OBL.

Figure 18. STM32F051xx, Write protection register [4].

As shown in Figure 18, the reset value of this register depends on option bytes. On every system reset, the Option Byte Loader (OBL) [28] reads the information block and stores the data into the Option byte register (FLASH_OBR) [4] and the Write protection register (FLASH_WRPR).


Address offset: 0x10
Reset value: 0x0000 0080

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES	RES	RES	RES	RES	RES	RES	RES	RES	RES	RES	RES	RES	RES	RES	RES
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
RES	RES	OBL_LAUNCH	EOPIE	RES	ERRIE	OPTWRE	RES	LOCK	STRT	OPTER	OPTPG	RES	MER	PER	PG
		rw	rw		rw	rw		rw	rw	rw	rw		rw	rw	rw

Figure 19. STM32F051xx, Flash control register [4].

RES bits in Figure 19 represent reserved bits.

Another method is to use the one time programmable feature of MCU, However this method cannot be used during the system development process. In general, the bootloader and application area need to be kept separate.

There are a number of factors that should be taken into consideration when selecting where to locate the boot-loader; these are:

- Size of the bootloader
- Size of the application
- Location of the vector table
- Write protection and erase granularity

Depending on the features, code optimization level in the compiler, the bootloader size will vary, and hence during the bootloader development, it is difficult to guess a most optimal bootloader size. However, the rule of thumb is to pick bootloader size in the multiples of sector size and once everything is working, memory map can be easily manipulated by Linkers. Since the erase granularity on some microprocessors (e.g. STM32F051 selected for this thesis) is page level, a common mistake is to select bootloader size in multiples of pages, however this may lead to overlapping sectors. One reason for such behavior is that the write protection granularity could be different than the erase granularity. For example, on STM32F051 SOC, selecting the memory map as shown in Table 3 will corrupt the bootloader or lead to a reset, arising due to incorrect memory access.

Table 3: Example of incorrect memory partitioning

MCU	**Componenet**	**Start**	**Size**	**Size (in KB)**	**Page**	**Sector number**
STM32F051R8	Bootloader	0x08000000	0x4800	18	0 - 17	0 - 4
	Application	0x08004800	0x5800	22	18 - 39	4 - 9
	Backup area	0x0800A000	0x5800	22	40 - 61	10 - 15
	Reserved area	0x0800F800	0x800	2	62 - 63	15

Recall from the Table 2, that the erase granularity on STM32F051R8 is page level. So at, the first glance, the memory partitioning scheme used in Table 2 seem correct, however, the bootloader and the application have overlapping sectors (sector 4) and as the write protection granularity for this processor is sector level, unless the bootloader performs write protection manipulation (which is also not safe in this particular scenario). erase operation on page number 18 by the application will fail (Since this sector is occupied by the bootloader). It is even more complicated on processors that have hybrid sectors (for example, STM32F407VG). Therefore, one has to be very careful while partitioning the memory. The memory partitioning scheme used for this thesis is shown in Table 4.

Table 4: Memory partitioning scheme for three different MCU's used in this thesis

MCU	**Componenet**	**Start**	**Size**	**Page**	**Sector number**
STM32F051R8	Bootloader	0x08000000	0x7000	0 - 27	0 - 6
	Application	0x08007000	0x4000	28 - 43	7 - 10
	Backup area	0x0800B000	0x4000	44 - 59	11 - 14
	Reserved area	0x0800F000	0x1000	60 - 63	15
STM32F072CB	Bootloader	0x08000000	0x5000	0 - 9	0 – 4
	Application	0x08005000	0xD000	10 - 35	5 - 17
	Backup area	0x08012000	0xD000	36 - 61	18 - 30
	Reserved area	0x0801F000	0x1000	62 - 63	31
STM32F407VG	Bootloader	0x08000000	0x8000	N/A	0 - 1
	Application	0x08008000	0x8000	N/A	2 - 3
	Backup area	0x08010000	0x10000	N/A	4
	Reserved area	0x080E0000	0x20000	N/A	11

The configurable write protection is the most primitive method to protect the bootloader from accidental write attempt by the application, however an intentional overwrite by the application (e.g. a malicious application) can still be achieved by programming the option bytes on STM32xx series SOCs. Therefore, many processors (e.g. STM32F4xx) provide OTP area for the soul purpose of storing bootloader and protecting it from unwanted writes. Though, one is not able to use OTP during bootloader development but once the bootloader works, is tested and validated the linker can be adjusted to use OTP.

In either of these cases (write protection using option bytes or by using OTP features), once the flash areas have been selected, it is important to update the linker for the application to exclude area occupied by bootloader.

4.5. Reset and Interrupt vector

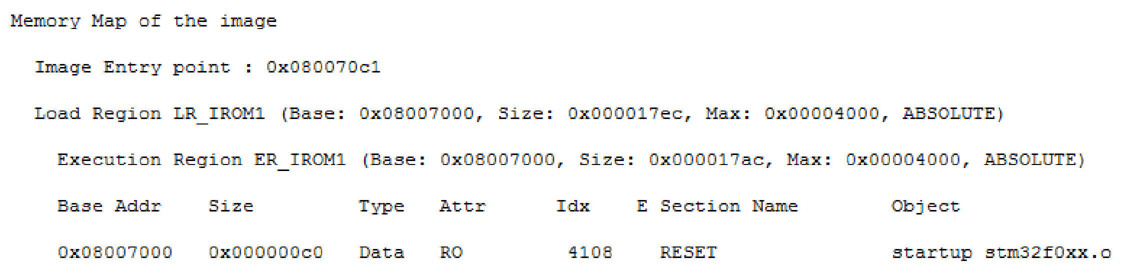
Reset vector is the location of memory where the first instruction of the application is located. During the boot-up process, processor begins program execution from the address stored in the reset vector. On a system with a bootloader (as on system developed for this thesis), this address is the entry into the bootloader and not to application reset vector. Therefore:

- The application reset vector is needed to be stored somewhere in the processor address space.
- A mechanism is needed in the bootloader to relocate the reset vector of the application.
- A mechanism is needed in the bootloader to perform a jump to application start address.

Relocation technique of the reset vector varies from one microprocessor to another. Unlike ARM Cortex M4 [23] processor, on, ARM cortex M0 [22], there is no dedicated register to relocate the reset vector to CODE or SRAM region, and instead have a memory remap feature on its memory system that allow vector table accesses to be redirected to SRAM. The question then arise is the amount of SRAM needed for the vector relocation. The answer lies in the startup file and the memory map of the application image. Information about the reset vector is shown in Figure 20. Notice the RESET identifier in Figure 20. Now, look for same identifier (RESET) in the memory map generated by the linker. It is show in the last line in Figure 21. The assoicated size information is mentioned under size column as 0x000000C0. Thus the reset vector table size needed for STM32F051R8 is 0xC0 (48 words)

```
; Vector Table Mapped to Address 0 at Reset
AREA RESET, DATA, READONLY
EXPORT __Vectors
EXPORT __Vectors_End
EXPORT __Vectors_Size
```

Figure 20. ARM assembly code snip of STM32F051R8 from startup.S .



```
Memory Map of the image

  Image Entry point : 0x080070c1

  Load Region LR_IROM1 (Base: 0x08007000, Size: 0x000017ec, Max: 0x00004000, ABSOLUTE)

    Execution Region ER_IROM1 (Base: 0x08007000, Size: 0x000017ac, Max: 0x00004000, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x08007000   0x000000c0   Data   RO          4108    RESET                 startup stm32f0xx.o
```

Figure 21. Snip from Toolchain generated memory map for STM32F051R8.

Once the application reset vector size calculation is done, the next step is to relocate the application reset vector. The bootloader accomplishes this by utilizing Linker specific directive and memory remap feature.

```
#define APPLICATION1_ADDRESS (uint32_t)0x08007000
#if (defined ( __CC_ARM ))
__IO uint32_t VectorTable[48]
__attribute__((at(0x20000000)));
#elif (defined (__ICCARM__))
#pragma location = 0x20000000
__no_init __IO uint32_t VectorTable[48];
#elif defined ( __GNUC__ )
__IO uint32_t VectorTable[48]
__attribute__((section(".RAMVectorTable")));
#elif defined ( __TASKING__ )
__IO uint32_t VectorTable[48] __at(0x20000000);
#endif

int i;
/* Copy the vector table from the Flash (mapped at
the base
 * of the application load address 0x08004000) to
the base
 * address of the SRAM at 0x20000000.
 */
for (i = 0; i < 48; i++)
{
    VectorTable[i] = *(__IO
uint32_t*)(APPLICATION1_ADDRESS + (i<<2));
}
/* Enable the SYSCFG peripheral clock*/
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG,
ENABLE);
/* Remap SRAM at 0x00000000 */
SYSCFG_MemoryRemapConfig(SYSCFG_MemoryRemap_SRAM);
```

Figure 22. Application reset vector relocation to SRAM in ARM Cortex M0 microprocessor

Figure 22 shows the vector relocation method on Cortex M0 based SOC (STM32F0xx). As shown in Figure 22, first, 48 words (0xC0 bytes) are copied from the application image and allocated in a particular area of SRAM by using toolchain specific Linker command. Recall from Figure 6, SRAM is mapped to 0x20000000 in the processor address space. Once the reset vector of application is copied to SRAM, it is remapped to 0x000000000. Note that the default start address 0x08000000 is an alias to 0x00000000, hence accessing 0x00000000 is same as accessing the default start address.

The final glue logic to make the application load at a physically different address is done by modifying the linker. It is toolchain specific and on KEIL MDK5, it is done by scatter loading. By using the scatter loading, one can fully control the grouping and placement of image components. To achieve scatter loading, armlink [24] linker uses scatter file. Scatter file format is shown in Figure 23, however, instead of writing scatter file from scratch, an easy mehtod on KEIL MDK5 [27] is to modify IROM1 and IRAM1 field and the toolchain automatically generates the scatter file. IROM1 values for bootloader and application for target MCU is selected accroding to memory partitioning scheme shown in Table 4, whereas, IRAM1 value is adjusted (show in Table 5) to accommodate first 42 words starting from 0x20000000 for application reset vector and also to pass some parameters between bootloader and application. Note that, on STM32F407VG, reset vector is relocated to code area (main flash memory) instead.

Table 5: SRAM partitioning for reset vector relocation in SRAM and passing parameters between bootloader and application

MCU	**Componenet**	**Start**	**Size**
STM32F051R8	Bootloader	0x20000100	0x1F00
	Application	0x20000100	0x1F00
STM32F072CB	Bootloader	0x20000100	0x3F00
	Application	0x20000100	0x3F00
STM32F407VG	Bootloader	0x20000100	0x1FF00
	Application	0x20000100	0x1FF00

Scatter file generated by the toolchain for bootloader and application for STM32F051R8 MCU is shown in Figure 24 and 25 resepectively.

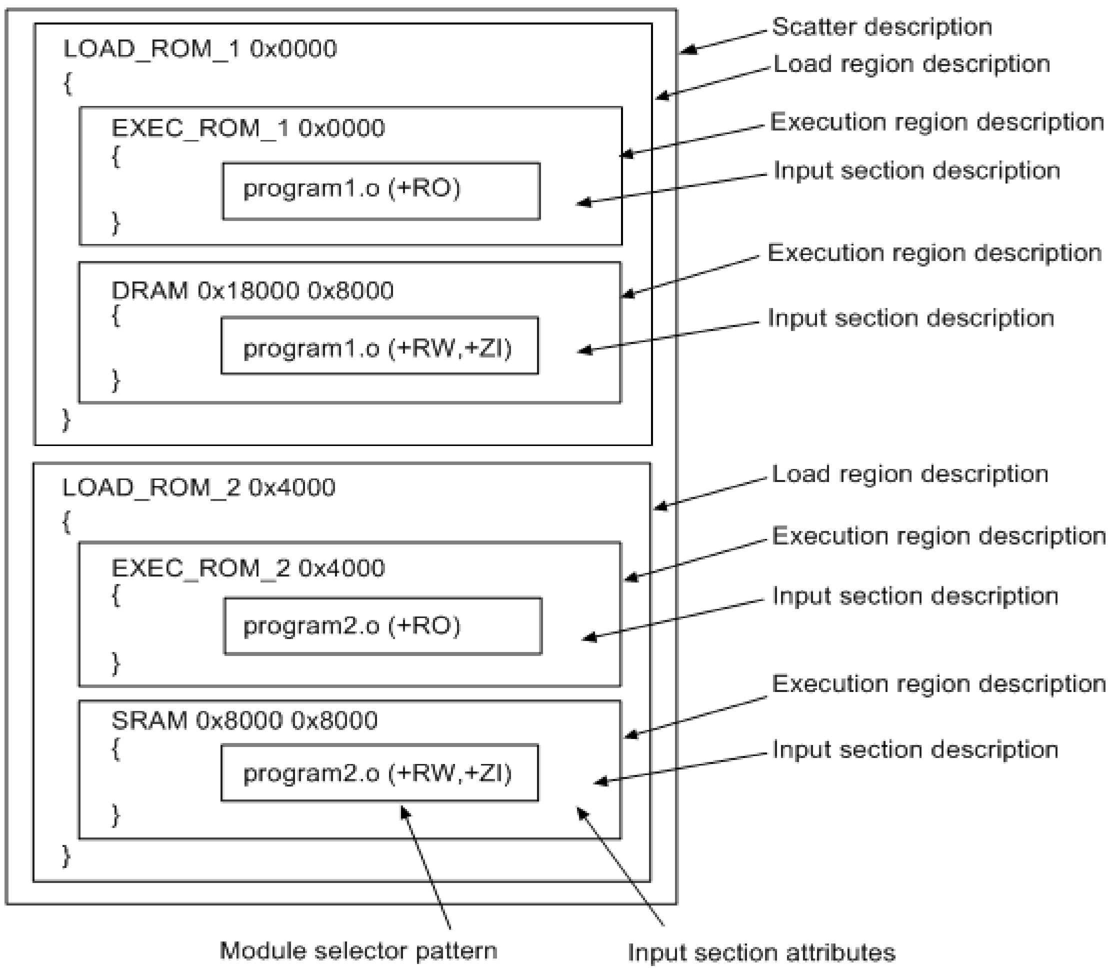


Figure 23. Components of a scatter file [20].

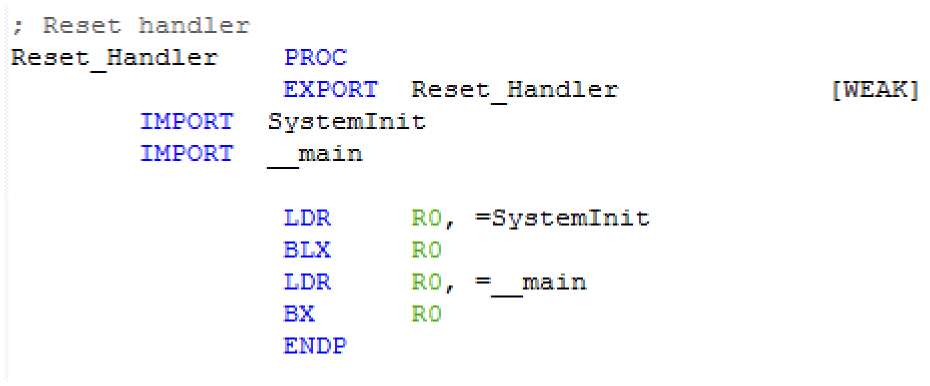
```
LR_IROM1 0x08000000 0x00007000  {    ; load region size_region
  ER_IROM1 0x08000000 0x00007000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }
  RW_IRAM1 0x20000100 0x00001F00  {  ; RW data
   .ANY (+RW +ZI)
  }
}
```

Figure 24. Scatter file for bootloader.

```
LR_IROM1 0x08007000 0x00004000  {    ; load region size_region
  ER_IROM1 0x08007000 0x00004000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }
  RW_IRAM1 0x20000100 0x00001F00  {  ; RW data
   .ANY (+RW +ZI)
  }
}
```

Figure 25. Scatter file for application.

Reset vector relocation is far more easier on ARM Cortex M4 (STM32F407VG SOC) as it has a dedicated register, VTOR (Vector Table Offset Register), which can be used to relocate vector table to CODE/Flash region or to SRAM. The application code only need to set appropriate value in the VTOR register and update it's load address by using scatter loading as mentioned before. A sample VTOR configuration during the system initialization process (just before the entry into the main routine) is shown in Figure 26 and 27.



```
; Reset handler
Reset_Handler    PROC
                 EXPORT  Reset_Handler             [WEAK]
        IMPORT  SystemInit
        IMPORT  __main

                 LDR     R0, =SystemInit
                 BLX     R0
                 LDR     R0, =__main
                 BX      R0
                 ENDP
```

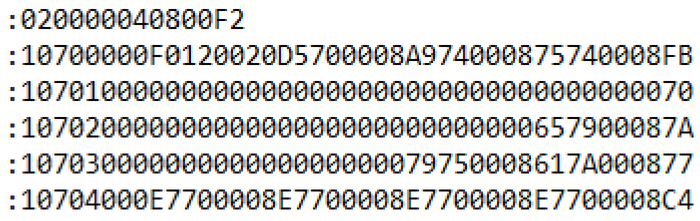
Figure 26. File startup_stm32f4xx.s

```
#define FLASH_BASE      ((uint32_t)0x08000000)
#define VECT_TAB_OFFSET 0x8000 /* User defined,
must be multiple of 512 bytes */

void SystemInit(void)
{
    /* Set HSION bit */
    RCC->CR |= (uint32_t)0x00000001;
    /*
     * some other initialization code here
     */
    SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; /*
Vector Table Relocation in Internal FLASH */
}
```

Figure 27. File: system_stm32f4xx.c.

4.6. Firmware file format

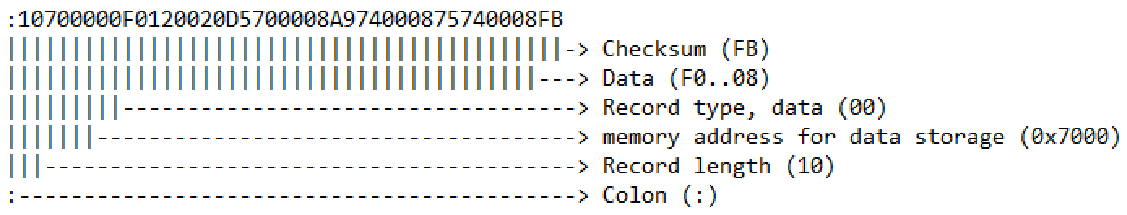
Firmware which is to be received over serial interface can be in different format, depending on the toolchain, compiler, target hardware. Some of the most famous file formats are Intel hex, binary, arm executable format etc. The toolchain used for firmware development is KEIL's ARM-MDK, which by default generates an object file that has both object code and the debug information. ISP tools such as SWD or JTAG are intelligent enough to strip the debug information and and load only the object code on the target. Bootloader needs to achieve something similar. Parsing a hex file is straight forward and KEIL'S ARM-MDK toolchain can be instructed to generate a hex file. It's an ASCII text file with lines of text that follow Intel HEX file format [6]. Each line in an Intel HEX file contains one HEX record. These records are made up of hexadecimal numbers that represent machine language code and/or constant data. These Intel Hex files are often used by EPmemory/Flash programmers to transfer program that would be stored in code region of main Flash memory. A snip form hex file generated by KEIL MDK5 is shown in Figure 28. The file is composed of any number of records, where each



```
:020000040800F2
:10700000F0120020D5700008A974000875740008FB
:10701000000000000000000000000000000000000070
:107020000000000000000000000000006579000087A
:1070300000000000000000000079750008617A000877
:10704000E7700008E7700008E7700008E7700008C4
```

Figure 28. Snip from Intel Hex file.

record is terminated with a carriage return and line-feed. For example, the record format of second record is shown in Figure 29 below. The Flash programmer can parse this record and perform Flash progrmaming and also perform validation by referring the



```
:10700000F0120020D5700008A974000875740008FB
|||||||||||||||||||||||||||||||||||||||||||-> Checksum (FB)
|||||||||||||||||||||||||||||||||||||||||---> Data (F0..08)
|||||||||---------------------------------> Record type, data (00)
|||||||-----------------------------------> memory address for data storage (0x7000)
|||---------------------------------------> Record length (10)
:-----------------------------------------> Colon (:)
```

Figure 29. Record format in Intel hex file.

checksum field associated with each record. This is a straight formware process, however while receiving a hex file over serial interface, the bootloader would have to implement parsing logic. For simplicity reasons, the bootloader developed as part of this thesis, expects a plain binary file. There are two benefits of this approach:

- The plain binary file is significantly smaller (about 60% smaller in size) than the hex version, hence it reduces the GSM/GPRS data traffic usage during the firmware download over GSM network.

- No need to parse the received image, thus reducing overall system downtime. (During the firmware upgrade process, system is not useable).

4.7. Data communication

Recall from the chapter on system architecture, transreceivers are selected for bi-directional communincation with the remote firmware repository. Different transreceivers operate in different frequency band. For this thesis, GSM/GPRS and Wi-Fi RF modules are used.

4.7.1. GSM/GPRS

One of the data communication link, chosen for this thesis is GSM/GPRS network. It is selected because one of the target application area of this thesis is IoT devices that are mobile; those which are constantly changing their physical location, such as electric bicycles. To download firmware over GSM/GPRS network, a GSM/GPRS module is used which is connected to target MCU over serial (UART) communication interface. The MCU sends AT+ command to interact with GSM/GPRS module.

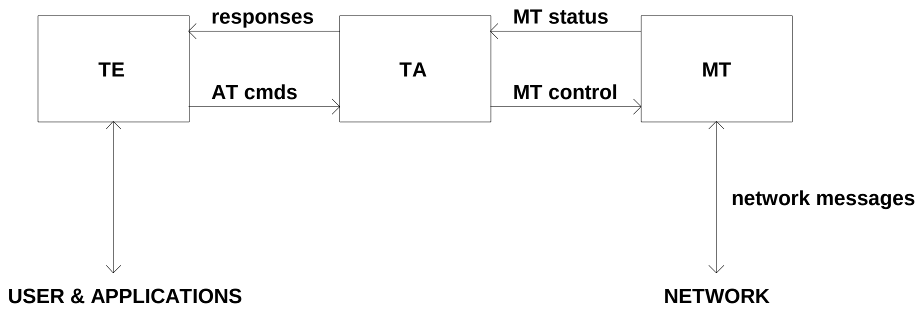
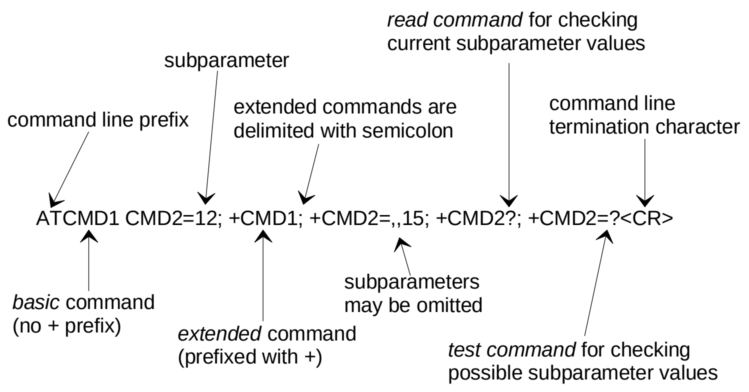


Figure 30. Abstract GSM/GPRS setup [7].

The abastract architecture, comprising Terminal equipment (TE) such as a computer and a Mobile terminal (MT) interfaced by a Terminal adapter (TA) is shown in Figure 30. The basic structure of a command line is shown in Figure 31. GSM/UMTS commands use syntax rules of extended commands. Every extended command has a test command (trailing =?) to test the existence of the command and to give information about the type of its subparameters. Parameter type commands also have a read command (trailing ?) to check the current values of subparameters. Action type commands do not store the values of any of their possible subparameters, and therefore do not have a read command.



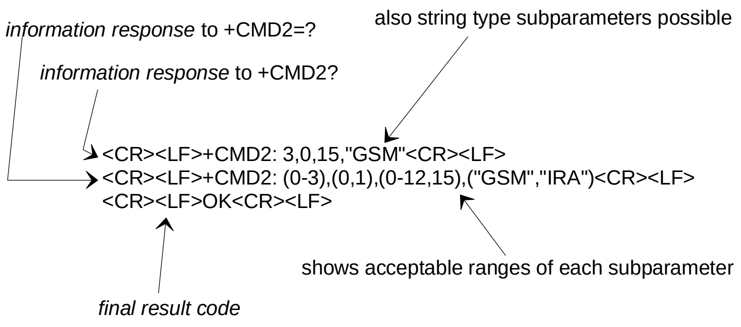
read command for checking current subparameter values
subparameter
command line prefix
extended commands are delimited with semicolon
command line termination character
ATCMD1 CMD2=12; +CMD1; +CMD2=,,15; +CMD2?; +CMD2=?<CR>
basic command (no + prefix)
extended command (prefixed with +)
subparameters may be omitted
test command for checking possible subparameter values

Figure 31. Basic structure of a command line [7].

If verbose responses are enabled with command V1 [7] and all commands in a command line has been performed successfully, result code <CR><LF>OK<CR><LF> is sent from the TA to the TE. If numeric responses are enabled with command V0, result code 0<CR> is sent instead.

If verbose responses are enabled with command V1 and sub-parameter values of a command are not accepted by the TA (or command itself is invalid, or command cannot be performed for some reason), result code <CR><LF>ERROR<CR><LF> is sent to the TE and no subsequent commands in the command line are processed. If numeric responses are enabled with command V0, result code 4<CR> is sent instead. ERROR (or 4) response may be replaced by +CME ERROR: <err> (refer clause 9) [7] when command was not processed due to an error related to MT operation.

The terminal adapter response for the command line of Figure 31 is shown in Figure 32.



information response to +CMD2=?
also string type subparameters possible
information response to +CMD2?
<CR><LF>+CMD2: 3,0,15,"GSM"<CR><LF>
<CR><LF>+CMD2: (0-3),(0,1),(0-12,15),("GSM","IRA")<CR><LF>
<CR><LF>OK<CR><LF>
shows acceptable ranges of each subparameter
final result code

Figure 32. Response to a command line [7].

On a power reset, before downloading the application firmware image, the GSM/GPRS Terminal adapter goes through an initialization steps. During these initialization steps, certain AT+ command are sent in a particular sequence to peform network discovery, operator selection and GPRS attach. The initialization process is shown in Figure 33.

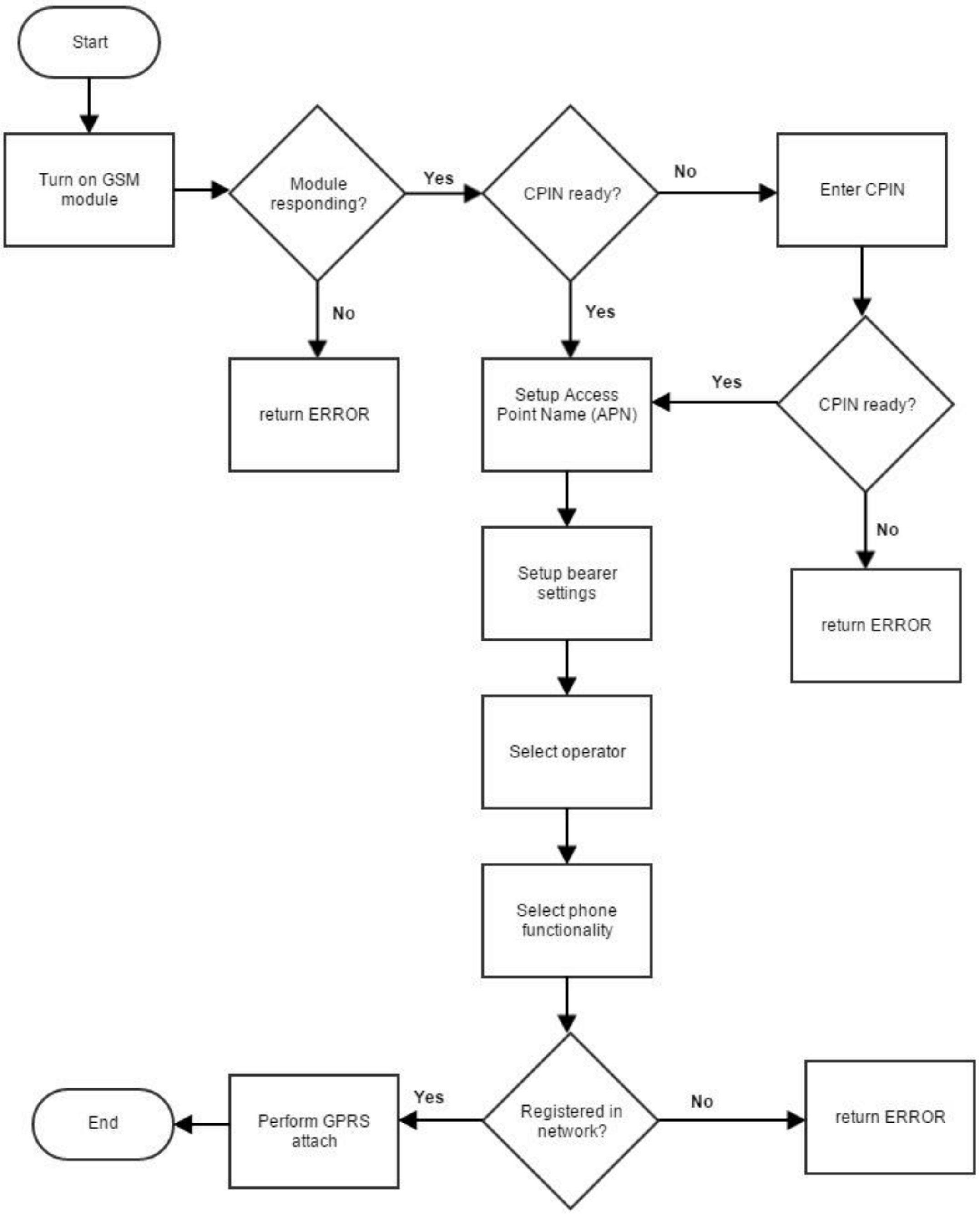


Figure 33. GSM/GPRS Terminal equipment initialization and network registration.

Once the GPRS attach is completed, Software update in the form of firmware image file can be download over HTTP/HTTPS. Typically, the firmware image would be larger than the maximum stack size on target MCU, therefore it is downloaded in multiples of smaller chunks. It is achieved by advancing the read pointer by the amount of data read. Figure 34 describes the firmware download sequence.

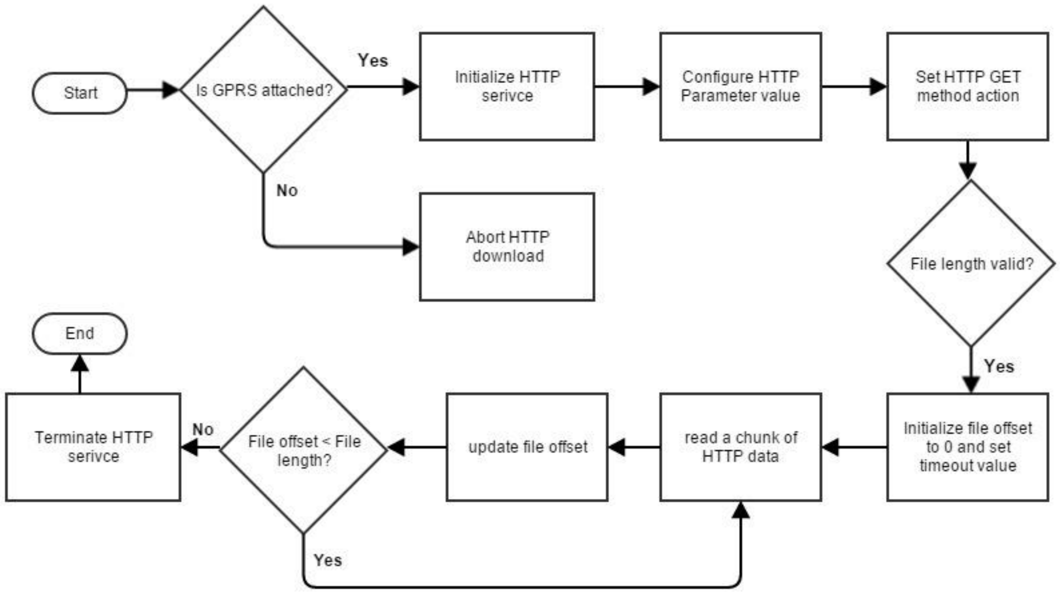

Start
Is GPRS attached?
Yes
No
Initialize HTTP serivce
Configure HTTP Parameter value
Set HTTP GET method action
Abort HTTP download
File length valid?
Yes
End
Terminate HTTP serivce
No
File offset < File length?
Yes
update file offset
read a chunk of HTTP data
Initialize file offset to 0 and set timeout value

Figure 34. Firmware file download over GSM/GPRS network.

The corresponding ’C’ routine is shown in Figure 35 below.

```
    /* Unlock the Flash Program Erase controller */
    FLASH_If_Init();
    while ((offset < len ) && timeout_ms > 0)
    {
        bytes_to_read = ((len - offset) >
GSM_HTTP_DATA_SIZE) ? GSM_HTTP_DATA_SIZE : (len -
offset);
        ret = gsm_http_read(offset, bytes_to_read);
        if (ret != GSM_SUCCESS)
        {
            FLASH_Lock();
            break;
        }
        if (line_valid)
        {
            /* Wait for data arrival */
            while
(USART_GetITStatus(gsm_uart.USARTx, USART_IT_RXNE)
== SET);
            /* disable GSM UART Receive IRQ */
            USART_ITConfig(gsm_uart.USARTx,
USART_IT_RXNE, DISABLE);
            if (FLASH_If_Write(&flashaddress,
(uint32_t *)memory_buffer, (bytes_read/4)) != 0)
            {
                FLASH_Lock();
                line_valid = 0;
                USART_ITConfig(gsm_uart.USARTx,
USART_IT_RXNE, ENABLE);
                ret = ERROR;
                break;
            }
            /* all consumed */
            line_valid = 0;
            /* Re-enable GSM UART Receive IRQ */
            USART_ITConfig(gsm_uart.USARTx,
USART_IT_RXNE, ENABLE);
        }
        offset += bytes_read;
        --timeout_ms;
    }
/* lock the Flash Program Erase controller */
FLASH_Lock();
```

Figure 35. Code snip of firmware file download in chunks over GSM/GPRS network.

4.7.2. Wi-Fi

Wi-Fi allows electronic devices to conenct to Wireless Local Area Network (LAN). It operates in 2.4 GHz band. The RF module selected for this thesis is ESP8266 [11] SoC. It has an in-built transreceiver that operates in the same frequency band as the Wi-Fi and has integrated TCP/IP protocol. The selected RF module also has pre-programmed AT+ command firmware, hence, interfacing this module with the MCU brings Wi-Fi capability to MCU and is easy to control.

The Wi-Fi module initialization and firmware file download sequence over TCP/IP is shown in Figure 36.

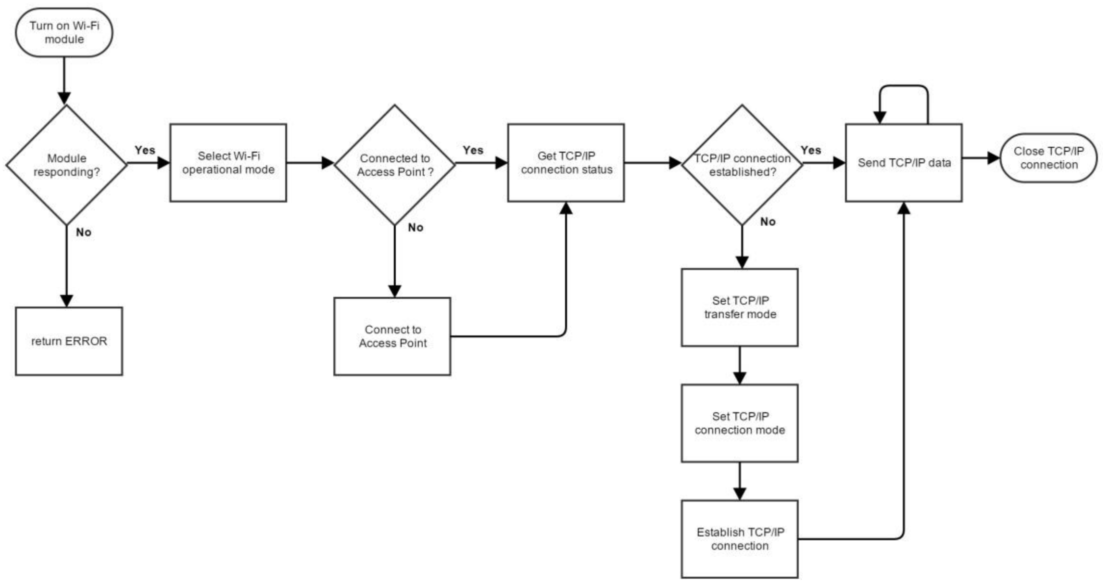

Turn on Wi-Fi module
Module responding?
Yes
No
return ERROR
Select Wi-Fi operational mode
Connected to Access Point ?
Yes
No
Connect to Access Point
Get TCP/IP connection status
TCP/IP connection established?
Yes
No
Set TCP/IP transfer mode
Set TCP/IP connection mode
Establish TCP/IP connection
Send TCP/IP data
Close TCP/IP connection

Figure 36. Wi-Fi module initialization and TCP/IP connection establishment

4.8. Hardware abstraction layer

Software portability and time to market are two very important factors in today's rapidly changing market. While software portability is important, it should not come at the expense of increased time to market or with increased complexity. The target devices used for this project are from the STM32 family of 32-bit Flash microcontrollers, based on ARM Cortex M processor. There are broad set of options at different levels of software, such as board support package, compiler, linker, static library, assembler, debugger etc. Therfore a small survery on available options was conducted and following decisions were made for the implementation:

- STM's Cube library [13] is highly portable static library, however, it is newer than the STD Peripheral library [14] and I had no prior experience with Cube library, hence it was not selected.
- There are many Real Time Operating System (RTOS) such as FreeRTOS [15] and RIOT-OS [16], which have support for the selected target devices, however since there are no real time requirements for this project, RTOS were not used.
- STM's STDPeripheral library allows one to write extremely portable application for a particular family (e.g. STM32F0xx or STM32F4xx) of SOCs from STM. These libraries have been thoroughly tested by the vendor and by users.

Code portability and reuseability was achieved by developing a semi-platform agnostic library, shown as one of the Component code block in Figure 37. The static library is compiled individually and is linked to the bootloader code at build time. The application uses 'C' language's pre-processor symbol and pre-processor directives to appropriately select the statically built library.

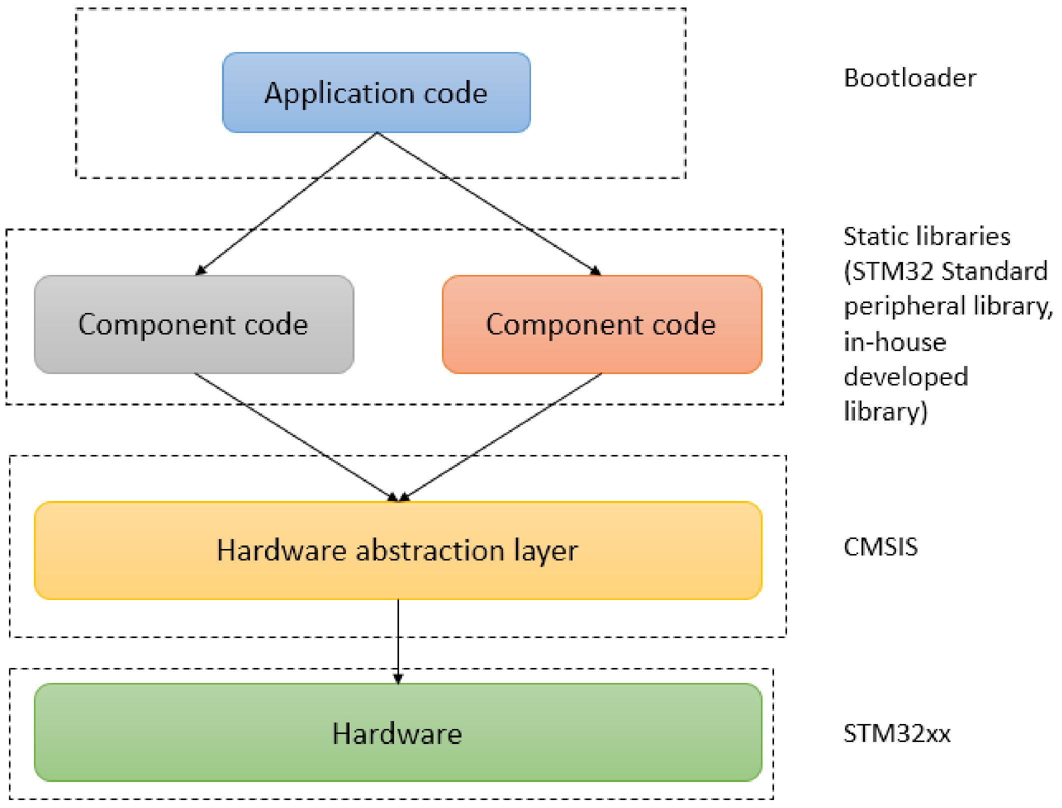

Application code
Bootloader
Component code
Component code
Static libraries (STM32 Standard peripheral library, in-house developed library)
Hardware abstraction layer
CMSIS
Hardware
STM32xx

Figure 37. Software abstraction.

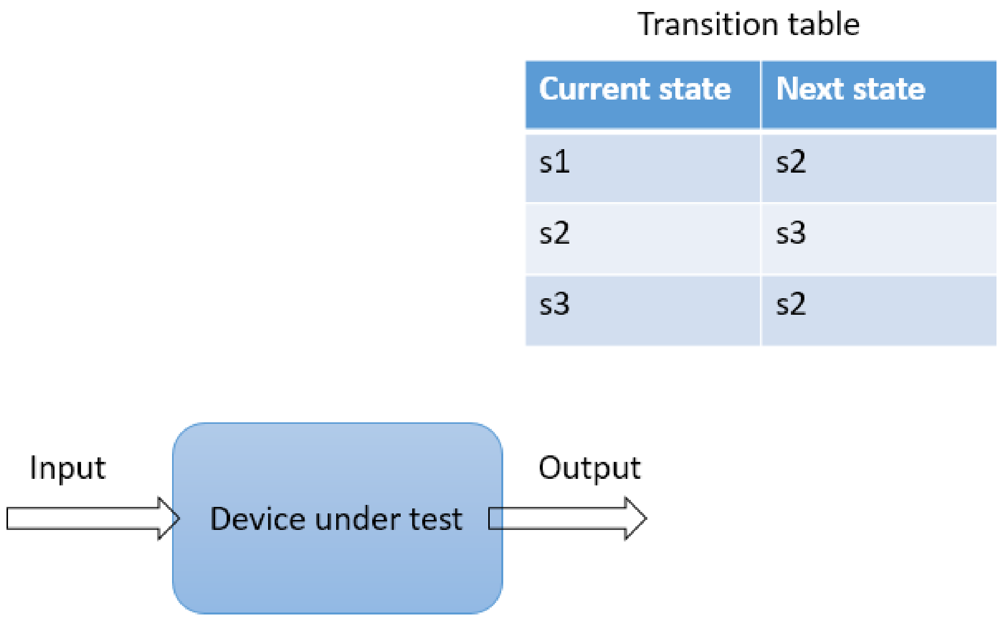
Figure 38 shows a sample code for clock domain initialization for two different target SoC.

```
#if defined (STM32F0xx)
    #define RCC_GPIO_GSM_PERIPH
RCC_AHBPeriph_GPIOA
    #define RCC_CRC_PERIPH
RCC_AHBPeriph_CRC
#elif defined (STM32F4xx)
    #define RCC_GPIO_GSM_PERIPH
RCC_AHB1Periph_GPIOB
    #define RCC_CRC_PERIPH
RCC_AHB1Periph_CRC
#else
    #error "Target not supported, refer toolchain
preprocessor symbols"
#endif

void rcc_init(void)
{
#if defined (STM32F0xx)
    RCC_AHBPeriphClockCmd(RCC_GPIO_GSM_PERIPH,
ENABLE);
    RCC_AHBPeriphClockCmd(RCC_CRC_PERIPH, ENABLE);
#endif

#if defined (STM32F4xx)
    RCC_AHB1PeriphClockCmd(RCC_GPIO_GSM_PERIPH,
ENABLE);
    RCC_AHB1PeriphClockCmd(RCC_CRC_PERIPH, ENABLE);
#endif
}
```

Figure 38. Clock domain initialization.

5. Experiments

An abstract black box test [18] setup is shown in Figure 39. The device under test is supposed to change it's state based on the input parameters and it's current state. The state transition is known a priori. The system is said to exhibit correct (PASS) behavior, if the output matches with the expected behavior, otherwise its deeemed incorrect (FAIL). Verification is done by consulting the transition table.



Transition table

Current state	Next state
s1	s2
s2	s3
s3	s2

Figure 39. Abstract black box test setup.

Similar approach is used to validate the over the air firmware upgrade implementation. Functional test cases were written by assuming different operational, upgrade and failure scenarios and accordingly tests were carried out on all the three target SoCs (refer Appendix 2). Test case description and actual result is shown in Table 6.

As, the test results are used to characterize a system's correctness, it is very important to judiciously identify the test cases. It is even more important when the device under test is in large quantity.

Table 6: Test description and results

Test case	**Expected behavior**	**Actual behavior**		
		Custom board (STM32F072CB)	**STM32F051-Discovery**	**STM32F407VG-Discovery**
Flash memory contains bootloader and factory firmware. Power on the board.	After Power-ON, Bootloader should find the factory firmware in the flash and normal operation of factory firmware should follow.	PASS	PASS	PASS
Factory firmware initiates f/w upgrade over GSM by setting a flag in SRAM and issues soft reset.	Board goes through a soft reset and Bootloader should acknowledge the f/w upgrade process, by issuing AT+ command to download the firmware.	PASS	PASS	PASS
No connection while trying to contact HTTP server the Bootloader.	Retry once more by issuing a soft reset. Once retries are exhausted, abort firmware upgrade process and boot from current factory firmware.	PASS	PASS	PASS
Remove the GSM antenna from the board.	Retry once more by issuing a soft reset. Once retries are exhausted, abort firmware upgrade process and boot from current factory firmware.	PASS	PASS	PASS
Board is connected to GSM network and Firmware upgrade is issued (in other words, No network issue at the beginning of f/w upgrade process)	Factory firmware is copied to a separate region in flash and the new firmware is downloaded and programmed into application region of the flash. This is followed by soft reset. After a soft reset, bootloader finds the new firmware in the flash, Transfers control to the new application and then normal operation of the board resumes.	PASS	PASS	PASS
No firmware at the HTTP server.	Retry once more by issuing a soft reset. Once retries are exhausted,	PASS	PASS	PASS

Test case	**Expected behavior**	**Actual behavior**		
		Custom board (STM32F072CB)	**STM32F051-Discovery**	**STM32F407VG-Discovery**
	abort firmware upgrade process and boot from current factory firmware.	PASS	PASS	PASS
HTTP Server that stores the firmware is down.	Retry once more by issuing a soft reset. Once retries are exhausted, abort firmware upgrade process and boot from current factory firmware.	PASS	PASS	PASS
Firmware file size bigger than the area reserved in the flash for the application.	Retry once more by issuing a soft reset. Once retries are exhausted, abort firmware upgrade process and boot from current factory firmware.	PASS	PASS	PASS
Remove the GSM antenna from the board during the firmware upgrade process.	Retry once more by issuing a soft reset. Once retries are exhausted, abort firmware upgrade process, copy back the older firmware from the backup area and boot from older firmware.	PASS	PASS	PASS
Network error before the firmware upgrade process.	Retry once more by issuing a soft reset. Once retries are exhausted, abort firmware upgrade process and boot from current factory firmware.	PASS	PASS	PASS
Power cycle the board during the firmware upgrade process.	Bootloader will retry software update until retries are exausted. After which a rollback would be initiated to boot from older application.	PASS	PASS	PASS

6. Conclusion and future work

The ability to perform firmware upgrade, after the system has been already shipped is very attractive to system manufacturers. It has become of more relevance in recent times, due to impressive growth of IoT domain. A cost effective and scalable solution is to ship bootloader along with the actual product, however this is rather a challenging task for following reasons:

- In IoT world, where power consumption is of prime concern, ARM based products are vastly used. The licensing model of ARM and various family of ARM processors has allowed hardware vendors to manufacture ARM based SoC in variety of packaging, gate count and density.
- Ultra-low power embedded systems have very limited resource.
- Different deployment environments.

For these reasons, bootloaders are also application centric, but since the generic ideas behind every bootloader are same, a good design pattern could be developed while spending engineering effort on developing bootloader. Though bootloader is not the main end product, but is potentially the most important part of the product and hence bootloader design should be considered at the very early stage of the project.

As part of the thesis work, bootloader was developed and was successfully deployed on a commercial IoT product [12] to facilitate firmware update over the GSM/GPRS network. An in-house test bench was also developed to validate different software update scenario. These test cases were conducted to test the correctness and robustness of the system, more specifically in the event of firmware upgrade process. Various fault scenarios were simulated to test the fail-proof behavior of device under test (STM32 MCU). These failure scenarios and the test results are mentioned in Table 6.

A static ‘C’ library was developed. It drastically reduced the porting effort of bootloader from one MCU to another. Total lines of ‘C’ code written for portable, fail-proof bootloader developed as part of this thesis is about 4000. Bootloader binary size is about 20 KB (with compiler optimization level set to O3)

According to current Intel’s head, security is the third pillar of computing (performance and connectivity being the other two). An unsecured software update could open a

Pandora’s box. Therefore, effort must be made to make the software update as secured as possible. As part of the thesis work, due to time constraints, security aspects of software update could not be explored, therefore the author plans to enhance the current bootloader by incorporating security features. Author believes that a layered approach for security features should be investigated and secured over the air, firmware update process can be achieved by implementing, authentication, securing the download data stream (HTTPS instead of HTTP), data encryption and memory read out protection.

Firmware upgrade over Wi-Fi network is the other planned future work. The work has already started to interface Wi-Fi module [11] with the target MCU and currently a library is being developed.

The long term goal is to add support for ARM GNU Toolchain [17] and release the Bootloader code in public domain as open source project.

References

[1] A guide to internet of things. (http://www.intel.com/content/www/us/en/internet-of-things/infographics/guide-to-iot.html) (25.05.2016)

[2] STM32, the most popular MCU ever. (http://www.edn.com/electronics-blogs/other/4419922/8/Slideshow--The-most-popular-MCUs-ever) (25.05.2016)

[3] Elicia white (2011). Making Embedded Systems, Design pattern for Great Software, Publisher: O'Reilly Media (Book)

[4] RM0091 reference manual. (http://www2.st.com/content/ccc/resource/technical/document/reference_manual/c2/f8/8a/f2/18/e6/43/96/DM00031936.pdf/files/DM00031936.pdf/jcr:content/translations/en.DM00031936.pdf) (25.05.2016)

[5] NXP Kinetis Kx8 Microcontroller. (http://cache.nxp.com/files/32bit/doc/data_sheet/K80P121M150SF5.pdf?fpsp=1&WT_TYPE=Data%20Sheets&WT_VENDOR=FREESCALE&WT_FILE_FORMAT=pdf&WT_ASSET=Documentation&fileExt=.pdf) (25.05.2016)

[6] Intel HEX File format. (http://www.keil.com/support/docs/1584/) (25.05.2016)

[7] 3GPP TS 27.007 V13.4.0.

[8] STM32F051 Discovery board. (http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/PF253215?sc=internet/evalboard/product/253215.jsp) (25.05.2016)

[9] STM32F407VG Discovery board. (http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/PF252419?sc=internet/evalboard/product/252419.jsp) (25.05.2016)

[10] Seeeduion GPRS IoT panel. (http://www.seeedstudio.com/wiki/Seeeduino_GPRS) (25.05.2016)

[11] ESP8266, Wi-Fi Serial transreceiver module. (http://www.seeedstudio.com/depot/WiFi-Serial-Transceiver-Module-w-ESP8266-p-1994.html) (25.05.2016)

[12] Gloabal leader in bicycle and scooter connectivity. (www.comodule.com) (25.05.2016)

[13] STM cube library (http://www.st.com/content/st_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32cube-embedded-software.html?querycriteria=productId=LN1897) (25.05.2016)

[14] STM Standard Peripheral library (http://www.st.com/content/st_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32-standard-peripheral-libraries.html?querycriteria=productId=LN1939) (25.05.2016)

[15] FreeRTOS. (http://www.freertos.org/) (25.05.2016)

[16] RIOT-OS, The friendly operating system for the Internet of things. (http://riot-os.org/) (25.05.2016)

[17] GNU Toolchain for ARM processors. (https://launchpad.net/gcc-arm-embedded) (25.05.2016)

[18] Black box testing. (https://en.wikipedia.org/wiki/Black-box_testing) (25.05.2016)

[19] Cortex Microcontorller Interface standard. (http://www.keil.com/pack/doc/CMSIS/General/html/index.html) (25.05.2016)

[20] ARM KEIL, syntax of a scatter file. (http://www.keil.com/support/man/docs/armclang_link/armclang_link_pge1362075656353.htm) (25.05.2016)

[21] J Yiu (2011). The definitive guide to ARM Cortex-M0, Publisher: Newnes, (Book)

[22] ARM Cortex M0. (http://www.arm.com/products/processors/cortex-m/cortex-m0.php) (25.05.2016)

[23] ARM Cortex M4. (http://www.arm.com/products/processors/cortex-m/cortex-m4-processor.php) (25.05.2016)

[24] ARMLINK linker. (http://infocenter.arm.com/help/index.jsp) (25.05.2016)

[25] Cortex-M0 Devices generic user guide. (http://infocenter.arm.com/help/topic/com.arm.doc.dui0497a/DUI0497A_cortex_m0_r0p0_generic_ug.pdf) (25.05.2016)

[26] ARM compiler version v5.06 for uVision, armasm user guide. (25.05.2016)

[27] ARM KEIL Microcontroller development kit (http://www2.keil.com/mdk5/) (25.05.2016)

[28] STM32F051xx Data sheet. (http://www.st.com/content/ccc/resource/technical/document/datasheet/55/53/3e/86/29/61/41/d9/DM00039193.pdf/files/DM00039193.pdf/jcr:content/translations/en.DM00039193.pdf) (25.05.2015)

Appendix 1 – ARM architecture basics

Since the target SOC is based on ARM architecture, hence this chapter provides brief information on ARM architecture and an introduction to ARM series.

The ARM architecture is similar to a Reduced Instruction Set Computer (RISC) architecture, as it incorporates these typical RISC architecture features:

- A uniform register file load/store architecture, where data processing operates only on register contents, not directly on memory contents.
- Simple addressing modes with all load/store address deteremined from register contents and instruction field only.

Overall, ARM has three distinct lines of ARM architectures, These are:

- Cortex A – used in performance centric applications.
- Cortex M – used in low power applications.
- Cortex R – used in real time constrained applications.

The focus of this thesis is Internet of things domain and Cortex M series is the most popular in electronic industry and hence only ARM Cortex M series processors will be discussed further. The existing Cortex-M processors are based on two architecture versions:

- Cortex-M3, Cortex-M4 and Cortex-M7 are based on ARMv7-M architecture
- Cortex M0, Cortex-M0+ and Cortex-M1 are based on ARMv6-M architecture

The architecture specifications define the behavior of the processors from both software and debug points of view. For example, the instruction set, programmers' model, exception model, and debug registers, which are visible to debug tools, are all defined by the architecture specifications. Each architecture can result in multiple processor implementations, which in turn can be used in multiple SoC products. As part of this thesis Bootloader has been developed for supporting both ARMv6-M (STM32F0 SOC) and ARMv7-M (STM32F4 SOC) architecture. Although the processor architecture specific details are abstracted by the compiler and the high level programming language, it is nonetheless important to know the compatibility between the processor revision, before hand for ease of developing portable applications. According to the ARM compatiblity matrix, shown in Table below, ARMv6-M is fully compatible with ARMv7-M.

Table. Compatibility matrix.

Software developed for	**Is compatible with these implementations**
ARM v6-M	ARMv6-M with Unprivileged/Privileged Extension.
	ARMv6-M with Unprivileged/Privileged Extension and PMSAv6-M
	All ARMv7-M
ARMv6-M with Unprivileged/Privileged Extension	ARMv6-M with Unprivileged/Privileged Extension and PMSAv6
	All ARMv7-M
ARMv6-M with Unprivileged/Privileged Extension and PMSAv6	ARMv7-M with PMSAv7

The following text in this appendix on ARM Cortex M0 is taken from a textbook [21] . Cortex-M0 processor is a 32-bit Reduced Instruction Set Computing (RISC) processor with a von Neumann architecture (single bus interface). It uses an instruction set called Thumb, which was first supported in the ARM7TDMI processor; however, several newer instruction from the ARMv6 architecture and a few instructions from the Thumb-2 technology are also included. Thumb-2 technology extended the previous Thumb instruction set to allow all operations to be carried out in one CPU state. The instruction set in Thumb-2 included both 16-bit and 32-bit 24 instructions; most instructions generated by the C compiler use the 16-bit instructions, and the 32-bit instructions are used when the 16-bit version cannot carry out the required operations. This results in high code density and avoids the overhead of switching between two instruction sets. In total, the Cortex-MO processor supports only 56 base instructions, although some instructions can have more than one form.

A simplified block diagram of the Cortex-MO is shown in Figure below.

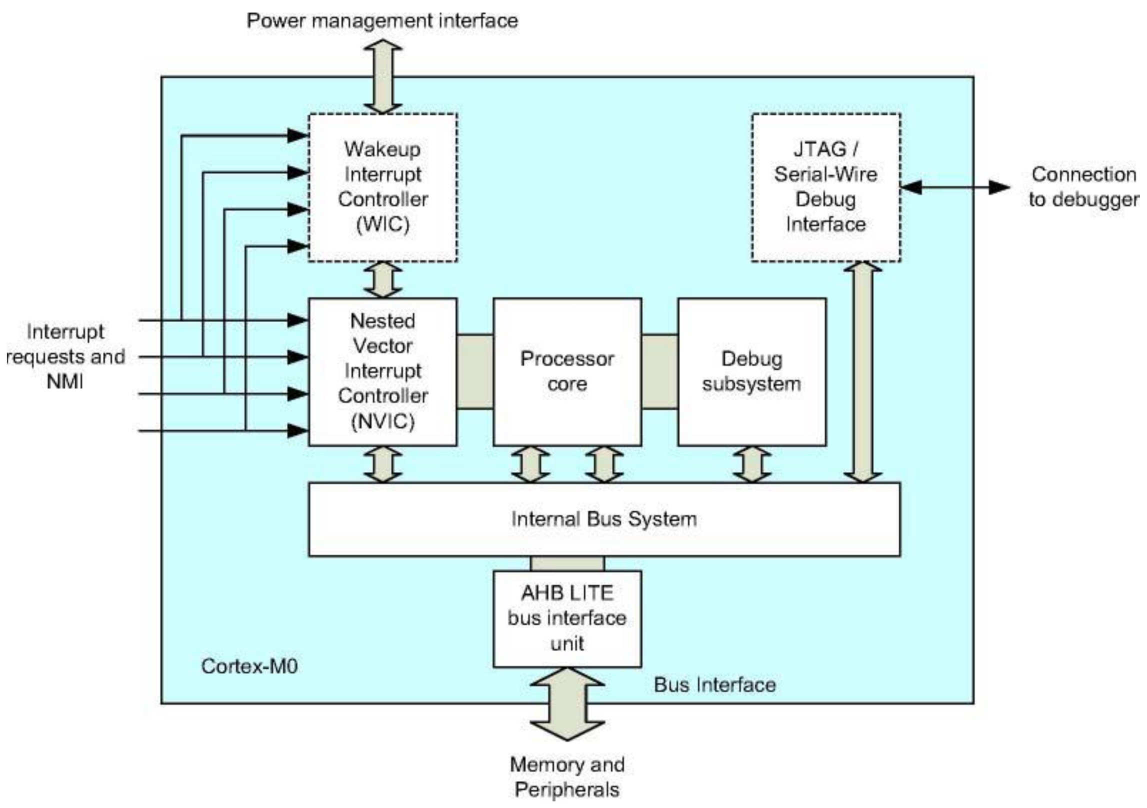
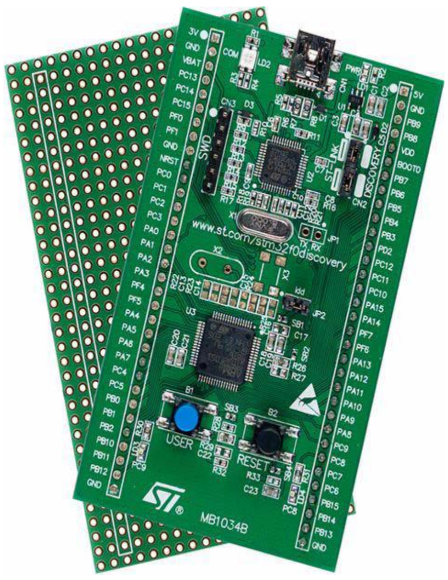

Power management interface
Wakeup Interrupt Controller (WIC)
JTAG / Serial-Wire Debug Interface
Connection to debugger
Interrupt requests and NMI
Nested Vector Interrupt Controller (NVIC)
Processor core
Debug subsystem
Internal Bus System
AHB LITE bus interface unit
Cortex-M0
Bus Interface
Memory and Peripherals

Figure. Simplified block diagram of Cortex M0 processor [21]

- The processor core contains the register banks, ALU, data path, and control logic. It is a three stage pipeline design with fetch stage, decode stage, and execution stage. The register bank has sixteen 32-bit registers. A few registers have special usages.
- The Nested Vectored Interrupt Controller (NVIC) accepts up to 32 interrupt request signals and a nonmaskable interrupt (NMI) input. It contains the functionality required for comparing priority between interrupt requests and the current priority level so that nested interrupts can be handled automatically. If an interrupt is accepted, it communicates with the processor so that the processor can execute the correct interrupt handler.
- The Wakeup Interrupt Controller (WIC) is an optional unit. In low-power applications, the microcontroller can enter standby state with most of the processor powered down. In this situation, the WIC can perform the function of interrupt masking while the NVIC and the processor core are inactive. When an interrupt request is detected, the WIC informs the power management to power up the system so that the NVIC and the processor core can then handle the rest of the interrupt processing.

- The debug subsystem contains various functional blocks to handle debug control, program breakpoints, and data watchpoints. When a debug event occurs, it can put the processor core in a halted state so that embedded developers can examine the status of the processor at that point.
- The JTAG or serial wire interface units provide access to the bus system and debugging functionalities. The JTAG protocol is a popular five-pin communication protocol commonly used for testing. The serial wire protocol is a newer communication protocol that only requires two wires, but it can handle the same debug functionalities as JTAG.
- The internal bus system, the data path in the processor core, and the AHB LITE bus interface are all 32 bits wide. AHB-Lite is an on-chip bus protocol used in many ARM processors. This bus protocol is part of the Advanced Microcontroller Bus Architecture (AMBA) specification, a bus architecture developed by ARM that is widely used in the IC design industry.

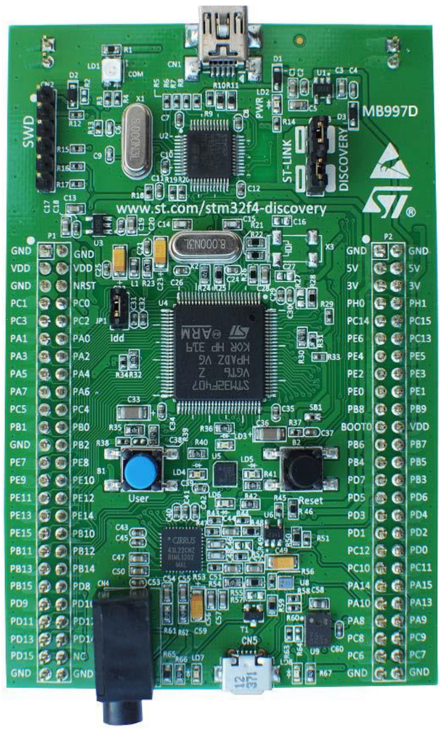
Appendix 2 – Development boards



STM32F051 Discovery board [8].

STM32F051-Discovery board feature summary:

- STM32F051R8T6 microcontroller featuring 64 KB Flash, 8 KB RAM in an LQFP64 package
- On-board ST-LINK/V2 with selection mode switch to use the kit as a standalone ST-LINK/V2 (with SWD connector for programming and debugging)
- Board power supply: through USB bus or from an external 5 V supply voltage
- External application power supply: 3 V and 5 V
- Four LEDs:
  - LD1 (red) for 3.3 V power on
  - LD2 (red/green) for USB communication
  - LD3 (green) for PC9 output
  - LD4 (blue) for PC8 output
- Two push buttons (user and reset)
- Extension header for LQFP64 I/Os for



STM32F407VG Discovery board [9].

STM32F407VG-Discovery board feature summary:

- STM32F407VGT6 microcontroller featuring 32-bit ARM Cortex®-M4 with FPU core, 1-Mbyte Flash memory, 192-Kbyte RAM in an LQFP100 package.
- On-board ST-LINK/V2 on STM32F4DISCOVERY or ST-LINK/V2-A on STM32F407G-DISC1.
- USB ST-LINK with re-enumeration capability and three different interfaces:
  - Virtual com port (with ST-LINK/V2-A only)
  - Mass storage (with ST-LINK/V2-A only)
  - Debug port
- Board power supply: through USB bus or from. an external 5 V supply voltage.
- External application power supply: 3 V and 5 V.
- LIS302DL or LIS3DSH ST MEMS 3-axis accelerometer.
- MP45DT02 ST MEMS audio sensor omni-directional digital microphone.
- CS43L22 audio DAC with integrated class D speaker driver.
- Eight LEDs:
  - LD1 (red/green) for USB communication
  - LD2 (red) for 3.3 V power on
  - Four user LEDs, LD3 (orange), LD4 (green), LD5 (red) and LD6 (blue)

- 2 USB OTG LEDs LD7 (green) VBUS and LD8 (red) over-current

▪ Two push buttons (user and reset).

▪ USB OTG FS with micro-AB connector

Seeeduino GPRS IoT panel [10].

Seeeduino GPRS IoT panel Specification:

▪ SIM Card Interface

▪ Battery CR1220

▪ Headset Interface :3.5mm headphones

▪ Micro USB:Port used to connect the board to your PC for programming

▪ Power Switch:Slide switch used to change the logic level and power output of the board to either 5V or 3.3V.

▪ DC Jack

▪ Power LED

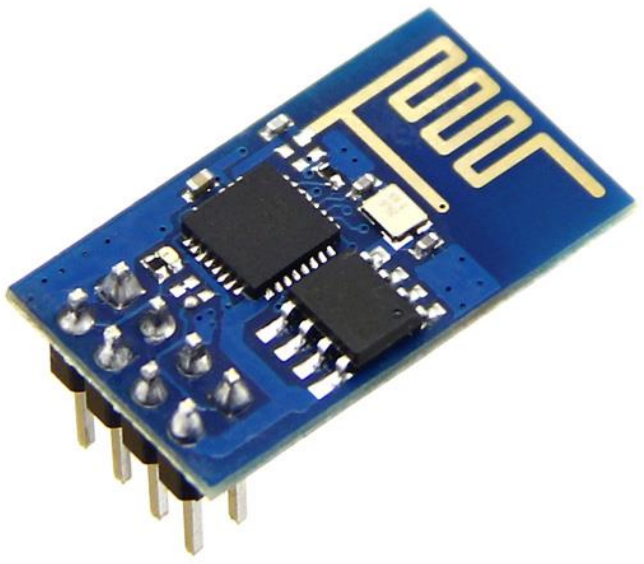
▪ Reset button :it can reset SIM800h and MCU

▪ Reset indicator LED

▪ MCU: The ATMEGA32U4-MUR chip

▪ GPRS Model: SIM800h

- Breakout for SIM800h :For debugging sim800h by this interface.
- LEDs indicator



WI-FI serial transreceiver module [11].

Seeduino WiFi serial transreceiver module with ESP8266 Specification:

- 802.11 b/g/n
- Wi-Fi Direct (P2P), soft-AP
- Integrated TCP/IP protocol stack
- Integrated TR switch, balun, LNA, power amplifier and matching network
- Integrated PLLs, regulators, DCXO and power management units
- +19.5dBm output power in 802.11b mode
- Power down leakage current of <10uA
- Integrated low power 32-bit CPU could be used as application processor
- SDIO 1.1/2.0, SPI, UART
- STBC, 1×1 MIMO, 2×1 MIMO
- A-MPDU & A-MSDU aggregation & 0.4ms guard interval
- Wake up and transmit packets in < 2ms
- Standby power consumption of < 1.0mW (DTIM3)

Appendix 3 – Pinout configuration

MCU	**Peripheral**	**Identifier**	**Pin**	**Mode/Baud rate/**	**Usage**
STM32F051 R8 Discovery board	GPIO	GPIOA	PA.2	AF_1	USART2_TX/ COM1 Rx
			PA.3	AF_1	USART2_RX/ COM1 Tx
			PA.9	AF_1	USART1_TX/ GSM_Rx
			PA.10	AF_1	USART1_RX/ GSM_Tx
		GPIOC	PC.9	OUT	On board LED
	Clock	RCC_AHBPeriph_GPIOA	N/A	N/A	Clock for GPIOA
		RCC_AHBPeriph_GPIOC	N/A	N/A	Clock for GPIOC
		RCC_APB2Periph_USART1	N/A	N/A	Clock for USART1
		RCC_APB1Periph_USART2	N/A	N/A	Clock for USART2
		RCC_AHBPeriph_CRC	N/A	N/A	Clock for CRC computation unit
	UART	USART1	N/A	9600	STM32-GSM serial communication
		USART2	N/A	9600	Serial loging
STM32F407 VG Disocvery board	GPIO	GPIOA	PA.2	AF_1	USART2_TX/ COM1 Rx
			PA.3	AF_1	USART2_RX/ COM1 Tx
		GPIOB	PA.6	AF_1	USART1_TX/ GSM_Rx
			PA.7	AF_1	USART1_RX/ GSM_Tx
		GPIOD	PD.13	OUT	On board LED

MCU	Peripheral	Identifier	Pin	Mode/Baud rate/	Usage
	Clock	RCC_AHB1Periph_GPIOA	N/A	N/A	Clock for GPIOA
		RCC_AHB1Periph_GPIOD	N/A	N/A	Clock for GPIOD
		RCC_AHB1Periph_GPIOB	N/A	N/A	Clock for GPIOB
		RCC_APB2Periph_USART1	N/A	N/A	Clock for USART1
		RCC_APB1Periph_USART2	N/A	N/A	Clock for USART2
		RCC_AHB1Periph_CRC	N/A	N/A	Clock for CRC computation unit
	UART	USART1	N/A	9600	STM32-GSM serial communication
		USART2	N/A	9600	Serial loging
STM32F07CB Custom board	GPIO	GPIOA	PA.2	AF_1	USART2_TX/ COM1 Rx
			PA.3	AF_1	USART2_RX/ COM1 Tx
			PA.9	AF_1	USART1_TX/ GSM_Rx
			PA.10	AF_1	USART1_RX/ GSM_Tx
		GPIOC	PC.14	OUT	GSM reset
		GPIOF	PF.13	OUT	GSM Power enable (Active low)
		GPIOB	PB.5	OUT	On-board Green LED
	Clock	RCC_AHBPeriph_GPIOA	N/A	N/A	Clock for GPIOA
		RCC_AHBPeriph_GPIOB	N/A	N/A	Clock for GPIOB
		RCC_AHBPeriph_GPIOF	N/A	N/A	Clock for GPIOF

MCU	**Peripheral**	**Identifier**	**Pin**	**Mode/Baud rate/**	**Usage**
	Clock	RCC_AHBPeriph_GPIOC	N/A	N/A	Clock for GPIOC
		RCC_APB2Periph_USART1	N/A	N/A	Clock for USART1
		RCC_APB1Periph_USART2	N/A	N/A	Clock for USART2
		RCC_AHBPeriph_CRC	N/A	N/A	Clock for CRC computation unit
		USART1	N/A	9600	STM32-GSM serial communication
		USART2	N/A	9600	Serial loging

Appendix 4 – Bootloader entry point

```
int main(void)
{
    if (SysTick_Config(SystemCoreClock/1000))
    {
        while (1);
    }
    int ret;
    /* Initialize Leds mounted on discovery board */
    STM_EVAL_LEDInit(LED3);
    STM_EVAL_LEDInit(LED4);
    /* Enable clocks */
    rcc_init();
    /* set up IRQs */
    nvic_configuration();
    /* Enable General purpose port and pins */
    gpio_init();
    /* Enable UART peripherals */
    uart_init();
    /* Enable CRC calculation unit */
    crc_init();
    /* Unlock the Flash Program Erase controller */
    FLASH_If_Init();
    /* OTA flag has been set, initiate firmware download procedure */
    if (*(uint32_t *)IAP_FLAG_ADDRESS == IAP_FLAG)
    {
        ret = ota_gsm();
        /* Everything went well, reset the IAP flag and boot from new
application */
        if (ret == SUCCESS)
        {
            /* Reset IAP flag */
            *(uint32_t *)IAP_FLAG_ADDRESS = 0x0;
            /* Take branching decission on next reboot */
            reboot();
        }
        /* DFU failed, retry until retries are exausted. The retry count is
set
         * by the application in an area reserved in SRAM. Typicall it is 1
         */
        else
        {
            if (*(uint32_t *)IAP_RETRY_ADDRESS == 0x1)
            {
                (*(uint32_t *)IAP_RETRY_ADDRESS)--;
                NVIC_SystemReset();
            }
            /* IAP flag is still set and we ran out of retries.
             * Note that, the application need to contact the remote server
and accordingly
             * take action on setting or resetting the DFU flag.
             */
            *(uint32_t *)IAP_FLAG_ADDRESS = 0x0;
            NVIC_SystemReset();
        }
    } /* end of OTA */
    /* A partially programmed firmware will lead to hardfault. DFU in
progress
     * flag is stored in non volatile memory to take better branching
decission.
     * Jump to application will happen iff
     *
     * a) application exists at the application load address.
     * b) device firmware upgrade has finished.
     */
    else if (IS_VALID_SP(APPLICATION1_ADDRESS) && !(IS_DFU_IN_PROGRESS))
    {
        prep_jump_to_application();
        /* Jump to user application */
        JumpAddress = *(__IO uint32_t*) (APPLICATION1_ADDRESS + 4);
        Jump_To_Application = (pFunction) JumpAddress;
        /* Initialize user application's Stack Pointer */
        __set_MSP(*(__IO uint32_t*) APPLICATION1_ADDRESS);
        /* Jump to application */
        Jump_To_Application();
    }
    /*
     * power reset during the device firmare update will leave the dfu in
progress
     * flag in set state. We will attempt DFU one last time
     */
    else if (IS_DFU_IN_PROGRESS)
    {
        /* try once more */
        ret = ota_gsm();
        /* Everything went well, reset the IAP flag and boot from new
application */
        if (ret == SUCCESS)
        {
            /* the return value of SUCCESS from ota_gsm means that the dfu in
progress
             * flag has been cleared. system reboot would bring us to normal
operation
             * state.
             */
            /* Reset IAP flag */
            *(uint32_t *)IAP_FLAG_ADDRESS = 0x0;
            /* Take branching decission on next reboot */
            reboot();
        }
        /* Erase the partially programmed application area, so that on
         * next reboot, bootloader doesn't attempt to jump to it
         */
        FLASH_If_Init();
        FLASH_If_EraseAppArea(APPLICATION1_ADDRESS);
        FLASH_Lock();
        dfu_in_progress_reset();
        reboot();
    }
    /* Neither the OTA flag is set, nor we have a valid application. Try to
boot
     * from older application, if a backup exists in memory, else prompt the
     * user to program the application firmware over STM-NRF UART
communication
     * link.
     */
    else
    {
        /* perform a rollback if there is a valid application in the backup
area */
        if (IS_VALID_SP(APPLICATION2_ADDRESS))··
        {
            if (rollback())
            {
                NVIC_SystemReset();
            }
            else
            {
                Main_Menu();
            }
        }
        Main_Menu();
    }

}
```

```
/* blink LED on board with 1 second delay to represet OTA failure */
void Main_Menu(void)
{
    while (1)
    {
        STM_EVAL_LEDOn(LED3);
        delay(1000);
        STM_EVAL_LEDOff(LED3);
        delay(1000);
    }
}
```