Grey Matter Tech

Software Architecture Document

USB Bootloader

Harsh Dave

Version 1.0

28/03/2021

Revision History

Version	Description of Versions / Changes	Responsible Party	Date
1.0	Initial version	Harsh Dave	28/03/21

Approval Block

Version	Comments	Responsible Party	Date

Table of Contents

1. Introduction

This document provides a high level overview and explains the architecture of the USB Bootloader for ARM M series based controllers, explicitly targeting STM controllers.

The document defines goals of the architecture, the use cases supported by the system, architectural styles and components that have been selected. The document provides a rationale for the architecture and design decisions made from the conceptual idea to its implementation.

1.1. Purpose

The Software Architecture Document (SAD) provides a comprehensive architectural overview of the USB Bootloader.

1.2. Scope

The scope of this SAD is to explain the architecture of the USB Bootloader.

This document describes the various aspects of the USB Bootloader design that is considered to be architecturally significant.

1.3. Definitions, Acronyms, and Abbreviations

- **SAD -** Software Architecture Document
- **User -** This is any user who is going to utilize USB Bootloader for their custom application update.

1.4. References

For [STM32F76xxx] : https://www.st.com/resource/en/reference_manual/dm00224583-stm32f76xxx-and-stm32f77xxx-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf

For [STM32F72xxx] : https://www.st.com/resource/en/reference_manual/dm00305990-stm32f72xxx-and-stm32f73xxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

1.5. Overview

In order to fully document all the aspects of the architecture, the Software Architecture Document contains the following subsections.

Section 2: describes the architectural goals and constraints of the system.

Section 3: describes logical view of the system including interface and operation definitions.

2. Architectural Goals and Constraints

There are some key requirements and system constraints that have a significant bearing on the architecture. They are:

1. The USB Bootloader architecture is meant as a proof of concept for a more complete project prediction system to be built in the future. Therefore one of the primary stakeholders in this document and the system as a whole are future architects and designers, not necessarily users as is normally the case. As a result, one goal of this document is to be useful to future architects and designers.

2. The USB Bootloader will be implemented for STM32F767Zi EVK, STM32L4R5Zi EVK as these EVKs have significantly large flash memory with additional features of dual boot, read/write flash protection and most important one is dual bank feature. Secure firmware validation, encryption/decryption are special deployment requirements that require additional consideration in the development of the architecture.

3. The USB Bootloader must be able to do:

    - Initial hardware initialization
    - Before jumping to already existing firmware in flash memory, it must trigger a sequence to detect USB key and to load the firmware from USB key into flash memory if USB key is connected to the hardware. It must boot the new loaded firmware from USB key.
    - If no USB key is detected, it must jump to already existing firmware in flash memory and start executing it.
    - If no firmware is found in flash at all and also no USB key is detected, it must throw fatal error log over UART and wait for further user commands.

3. Archietcture Implementation

3.1. Overview

USB Bootloader architecture is meant to either load the new firmware from USB stick, if it is been inserted within defined wait period after the CPU is powered on, or It jumps to already existing firmware in flash memory and starts executing.

As soon as the CPU is powered up, USB Bootloader starts executing. It sets up user execution environment, initialize the CPU clock, GPIOs, USB peripheral in OTG mode, and FAT file system. After initialization USB Bootloader sits in tight loop, polling for USB stick to get detected for "`USB_WAIT_PERIOD`" which is user configurable.

If USB stick is detected within the wait period, USB Bootloader looks for "`DOWNLOAD_FILENAME`" which is user configurable. If the "`DOWNLOAD_FILENAME`" is found in USB drive, USB Bootloader will read the file info and if the file size is valid(depends upon flash memory available in MCU

selected for implementing USB Bootloader), USB Bootloader erases the sectors starting from "`START_ADDRESS_OF_USER_FIRMWARE`" and it erases "`TOTAL_SECTORS_OF_USER_FIRMWARE`".

Once flash is emptied for placing the new firmware from USB stick, USB Bootloader reads out the binary file and copies it into the flash memory at address "`START_ADDRESS_OF_USER_FIRMWARE`".
After the firmware has been written into flash memory USB Bootloader calls an API "`BootAppImage()`". This API is core of USB Bootloader, which takes care of preparing CPU for firmware jump within flash memory and starts executing the new firmware.

On the other hand if the USB stick is not inserted within wait period, USB Bootloader simply jumps to "`START_ADDRESS_OF_USER_FIRMWARE`", assuming an default firmware already exist at the address and starts executing.

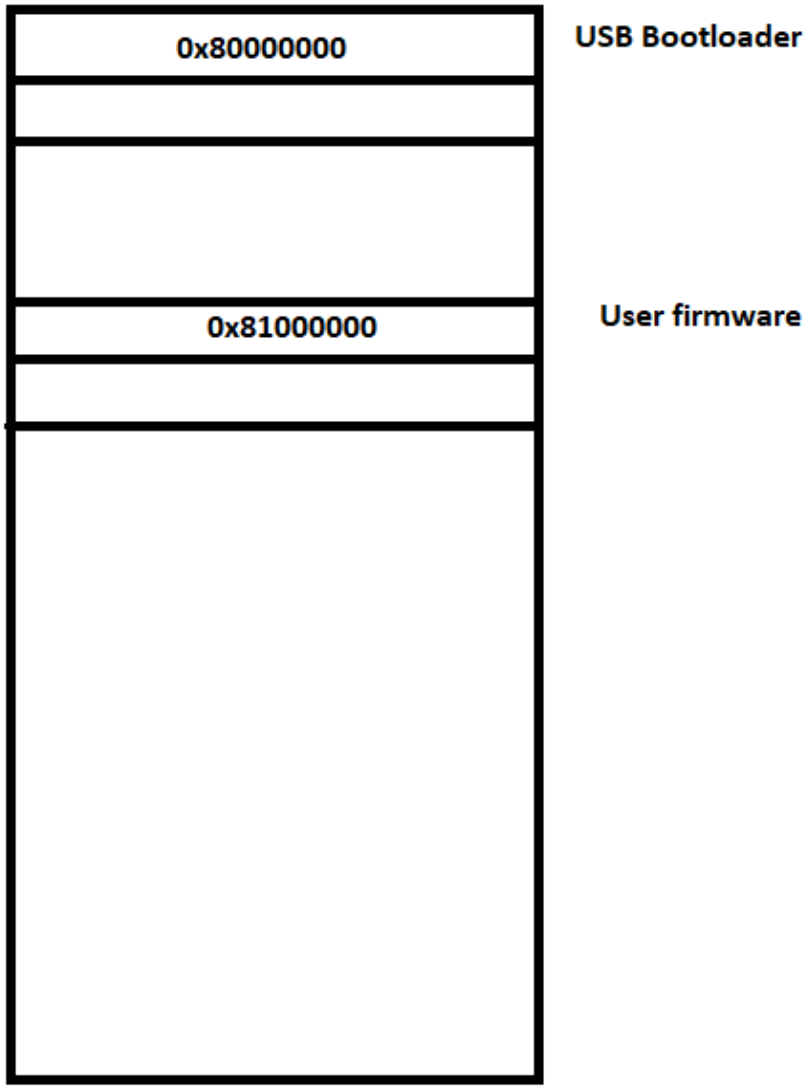
Figure 3.1 USB Bootloader Sequence Diagram.
Figure 3.2 USB Bootloader Flowchart

3.2. USB Bootloader Memory Map

Since the USB Bootloader is the first piece of code that should executes right away after the CPU is powered on even though firmware was updated previously via USB stick, MCU's flash memory must be configured to meet this requirement.

To address the challenge, simple arrangement should be made in flash memory as per the following memory map figure.

Figure 3.3 USB Bootloader Memory Map



Here the USB Bootloader is configured to get loaded at the start address of flash memory which is 0x80000000. This address may vary from MCU to MCU. As this initial architecture version is implemented for STM32 family MCU, flash starting address is 0x80000000. So as soon as the CPU is powered on, ROM Bootloader will bring the CPU to the address 0x80000000 and will instruct it to start executing. Since USB Bootloader is the one which can read the new firmware from USB stick and boot the newly loaded firmware, it must located at address 0x80000000.

Following snippet shows the linker file flash configuration for USB Bootloader.

```
/* Memories definition */
MEMORY
{
      RAM    (xrw)    : ORIGIN = 0x20000000,   LENGTH = 512K
      FLASH    (rx)    : ORIGIN = 0x8000000,   LENGTH = 2048K
```

```
}
```

For application firmware the address 0x8100000 is picked in the initial architecture draft. There is specific technical reason for selecting such address. MCU selected for initial architecture implementation were having flash memory of 2MB. So leaving 1MB for USB bootloader, the user firmware was placed in the 2nd half flash memory at the address 0x8100000. That means other 1MB was allocated for user firmware.

3.3. USB Application Firmware Configuration

For the application firmware which is to be loaded from USB stick, to be compatible with USB Bootloader, so that USB Bootloader correctly writes it into the flash memory, jumps to that firmware and start its execution, some important configuration needs to be made. If configurations are not as per the mentioned points below, USB Bootloader won't be able to boot the newly loaded firmware.

- **Flash start address for application firmware**
  As it is shown in above memory map diagram the application image needs to be placed in flash after the USB Bootloader image and considering some padding. Since MCU picked for initial implementation was having 2 MB flash memory, so leaving first half flash of 1 MB for USB Bootloader, other half flash's start address 0x8100000 was selected. But user can select any address, even immediate next sector of flash where the USB Bootloader image ends, if the memory is too small for selected MCU.

```
/* Memories definition */
MEMORY
{
        RAM    (xrw)    : ORIGIN = 0x20000000,   LENGTH = 512K
        FLASH    (rx)    : ORIGIN = 0x8100000,   LENGTH = 1024K
}
```

- **Vector table offset**
  ARM MCU always places vector table at top of memory. Vector table is nothing but the list of interrupt handler which instruct MCU to how to handle received interrupt. Since application firmware which is to be loaded from USB stick into flash memory has its starting address aligned with USB Bootloader such that both the firmware can live in same flash memory. So with respect to start address of application firmware vector table offset also needs to be reflected.

```
#define VECT_TAB_OFFSET  0x10000
```

3.4. Core of USB Bootloader (BootAppImage API)

This API does two major thing, one is setting back the stack pointer to default stack address and setting up the jump to reset vector address of new application firmware.

Vector table's first entry is stack address and the immediate next entry is reset handler's address. BootAppImage simply sets up stack pointer to very start address of new application firmware and jump pointer to next entry of vector table which gives the address of reset handler of application firmware.

```
uint32_t msp_value = *(volatile uint32_t *)0x8010000;
if(msp_value == 0xFFFFFFFF)
{
	return;
}
uint32_t resest_address = *(volatile uint32_t *)0x8010004;
__set_MSP(msp_value);
AppEntry Jump = (void *)resest_address;
```