Design and Implement Boot Loader Program for DUOS

Abdullah Al Mahmud
Roll: 15
Sarowar Jahan Rafin
Roll: 09
Md. Sadman Sakib
Roll: 49

Submitted to:
Dr. Mosaddek Hossain
Professor
University of Dhaka
Submitted on: 2024-11-18

Contents

Chapter 1

Introduction

1.1 Overview

Maintaining and updating firmware is essential for the performance and security of embedded systems. This project focuses on developing a custom bootloader for the STM32F4xx microcontroller series, which will manage firmware updates for the DUOS operating system.

1.2 Objectives

The key objectives are to:

- Implementing a bootloader to handle firmware updates securely and efficiently.
- Using UART protocol for communication between the microcontroller and a host system.
- Understanding memory management, flash programming, CRC implementation, and error handling in arm cortex processors.

Chapter 2

System Architecture and Design

2.1 Bootloader Architecture

A bootloader is a critical component in embedded systems and microcon trollers. It is responsible for initializing the system and loading the main application (firmware) into the device's memory during the boot process. Below is a description of the core functionality of a bootloader, including its memory mapping, initialization process, and control flow.

Core Functionality of a Bootloader

- **Memory Mapping:** The bootloader must be aware of the system's memory layout.
  - Bootloader Code: Stored in a special memory region starting from 0x08000000.
  - Application Code: Stored at a separate memory address following the bootloader code in flash memory.
  - Memory Segments: First one is "Flash Memory" where both the bootloader and application are stored. Second one is "RAM" where variables and temporary data are held during execution
- **Initialization:** The bootloader sets up the processor, peripherals, and memory.
- **Control Flow:** The bootloader's control flow begins with the startup phase, where it is the first code executed after the micro controller is powered on or reset, with execution starting from a predefined reset vector pointing to the bootloader's entry point. The bootloader then checks for the presence of a new firmware im age, which can be initiated either by

user interaction (e.g., pressing a button during reset) or by a signal (e.g., a special UART com mand). If a new firmware is detected, the bootloader verifies its integrity using checksums or cryptographic methods to ensure it is not corrupted. Upon successful verification, the bootloader proceeds to load the firmware into the appropriate memory region, such as flash memory, and may include writing and verifying the firmware. After this, or if no update is needed, the bootloader jumps to the application code by setting the program counter to the start of the application and hands over control. If the firmware fails verification, the bootloader may attempt to reload the firmware or enter a recovery mode as a fallback mechanism.

2.2 Communication Protocol

In this bootloader implementation, the UART (Universal Asynchronous Receiver-Transmitter) protocol is chosen for binary file transfer. UART is widely used in embedded systems for serial communication because it is simple, requires minimal hardware, and offers straightforward error detection mechanisms. The details of the protocol design, frame format, and error checking strategy are as follows:

2.2.1 UART Protocol

The bootloader uses UART2 for communication, configuring it with parame ters that support reliable data transfer, particularly at a baud rate of 115200, which provides a balance between speed and stability. The bootloader han dles both transmission and reception using the functions USART2Config, USART2SendChar, USART2SendString, and UART GetString1. These func tions enable interactive commands, such as checking for firmware updates, initiating erasure, or writing a new application

2.2.2 Frame Format for File Transfer

The binary file is received in chunks with a frame format that includes:

- Data Frame: A fixed-size chunk of bytes, defined by CHUNK (here, 32 bytes) is sent in each frame..
- Checksum Byte: After each data frame, a single checksum byte is sent, allowing for verification of the data integrity of each frame.

  The function USART2ReceiveBinary1 implements this frame-based trans fer. It retrieves each chunk of CHUNK SIZE bytes, followed by a single byte representing the checksum.

2.2.3 Error-Checking Mechanism: Checksum

This mechanism verifies data integrity using CRC32 checksum.

- Calculate Checksum on Send: On the sender side, a checksum is calculated by XORing each byte in the data frame. This 32 bits checksum value is then appended to the data frame.
- Verify Checksum on Receive: On the receiver (bootloader) side, the checksum is recalculated upon receiving the data frame. If the recalculated checksum matches the checksum from the sender, it indicates that the data frame is likely error-free.
- Handle Errors: If the checksums do not match, the bootloader re quests a retransmission of the frame by sending "RESEND". If the checksum matches, it acknowledges successful receipt by sending "OK".

Chapter 3

Implementation

3.1 Flash Memory Management

Flash Memory Unlocking

STM32 microcontrollers have a lock mechanism to protect flash memory from unintended changes. Before making any modifications, we need to unlock the flash by writing two unique keys (FLASH_KEY1 and FLASH_KEY2) to the KEYR register in sequence. This process prevents unauthorized access or accidental writes. The flash unlock function in the code checks if the flash memory is locked by examining a specific bit (FLASH_CR_LOCK) and, if locked, performs the unlocking sequence. This unlocks the flash memory, making it ready for erasing or writing.

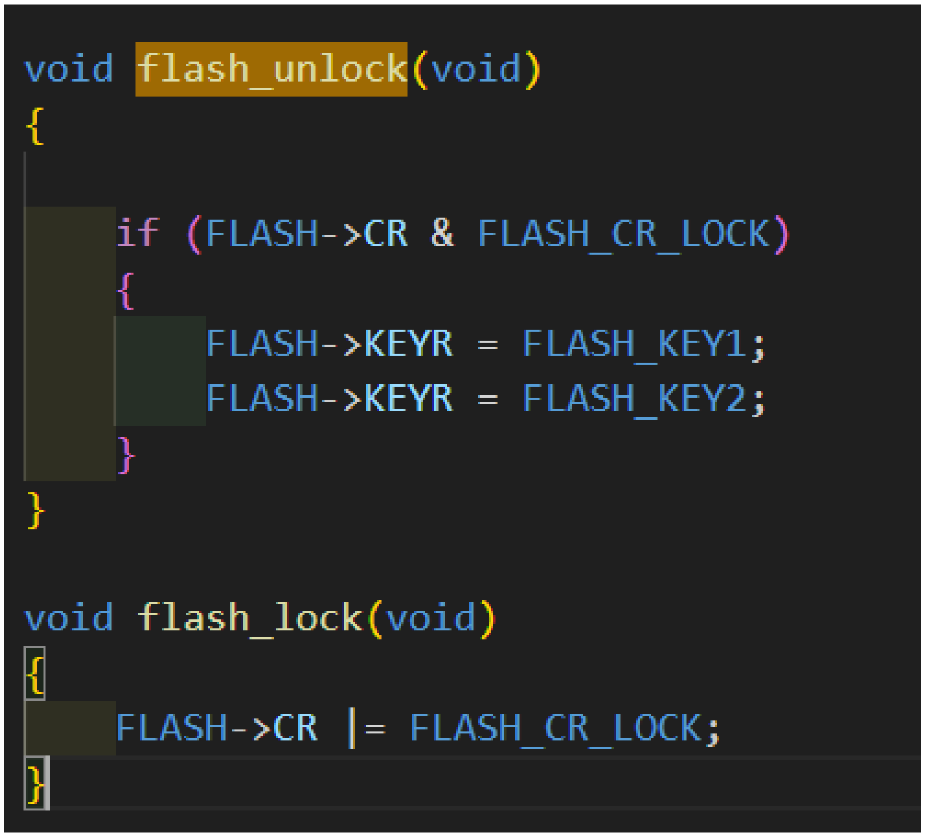
Erasing Flash Memory

Flash memory must be erased before new data can be written, as STM32 flash cells require an erased state (all bits set to 1) to store new information accurately. Our code erases memory one sector at a time, specifically from sectors 4 to 7, which are designated memory blocks in flash.

The erase memory in flash function first unlocks the flash and then erases each sector sequentially. It waits for the flash to be ready before starting each erase operation and checks for any errors that might arise during the process. Once all selected sectors are erased, the function locks the flash again to prevent unintended writes.

Writing to Flash Memory

After erasing, the flash is ready for new data. Flash writing in STM32 is done word-by-word or byte-by-byte, depending on the memory alignment and write size needed.



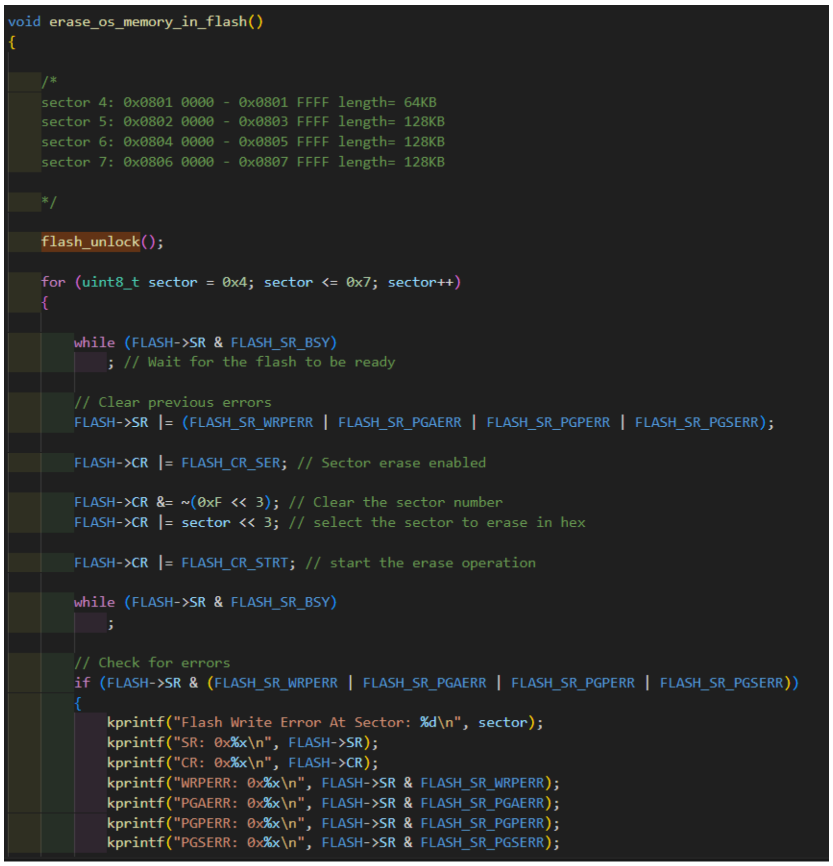
```c
void flash_unlock(void)
{

    if (FLASH->CR & FLASH_CR_LOCK)
    {
        FLASH->KEYR = FLASH_KEY1;
        FLASH->KEYR = FLASH_KEY2;
    }
}

void flash_lock(void)
{
    FLASH->CR |= FLASH_CR_LOCK;
}
```

Figure 3.1: Flash lock and unlock

The flash write function in our code writes data to flash memory one byte at a time, verifying each byte after it is written. The function starts by erasing the memory region and unlocking the flash, then enables programming mode. For each byte in the data array, it writes to the corresponding flash address, waits for the flash operation to complete, and verifies that the data was written correctly. If there are any errors, the function prints an error message, exits, and locks the flash.

Relocking Flash Memory

After each operation, the flash is relocked to protect it from further unintended modifications. This locking mechanism, controlled by setting the FLASH_CR_LOCK bit, ensures that flash remains secure until another intentional unlock command is issued.



```c
void erase_os_memory_in_flash()
{

    /*
    sector 4: 0x0801 0000 - 0x0801 FFFF length= 64KB
    sector 5: 0x0802 0000 - 0x0803 FFFF length= 128KB
    sector 6: 0x0804 0000 - 0x0805 FFFF length= 128KB
    sector 7: 0x0806 0000 - 0x0807 FFFF length= 128KB

    */

    flash_unlock();

    for (uint8_t sector = 0x4; sector <= 0x7; sector++)
    {

        while (FLASH->SR & FLASH_SR_BSY)
            ; // Wait for the flash to be ready

        // Clear previous errors
        FLASH->SR |= (FLASH_SR_WRPERR | FLASH_SR_PGAERR | FLASH_SR_PGPERR | FLASH_SR_PGSERR);

        FLASH->CR |= FLASH_CR_SER; // Sector erase enabled

        FLASH->CR &= ~(0xF << 3); // Clear the sector number
        FLASH->CR |= sector << 3; // select the sector to erase in hex

        FLASH->CR |= FLASH_CR_STRT; // start the erase operation

        while (FLASH->SR & FLASH_SR_BSY)
            ;

        // Check for errors
        if (FLASH->SR & (FLASH_SR_WRPERR | FLASH_SR_PGAERR | FLASH_SR_PGPERR | FLASH_SR_PGSERR))
        {
            kprintf("Flash Write Error At Sector: %d\n", sector);
            kprintf("SR: 0x%x\n", FLASH->SR);
            kprintf("CR: 0x%x\n", FLASH->CR);
            kprintf("WRPERR: 0x%x\n", FLASH->SR & FLASH_SR_WRPERR);
            kprintf("PGAERR: 0x%x\n", FLASH->SR & FLASH_SR_PGAERR);
            kprintf("PGPERR: 0x%x\n", FLASH->SR & FLASH_SR_PGPERR);
            kprintf("PGSERR: 0x%x\n", FLASH->SR & FLASH_SR_PGSERR);
```

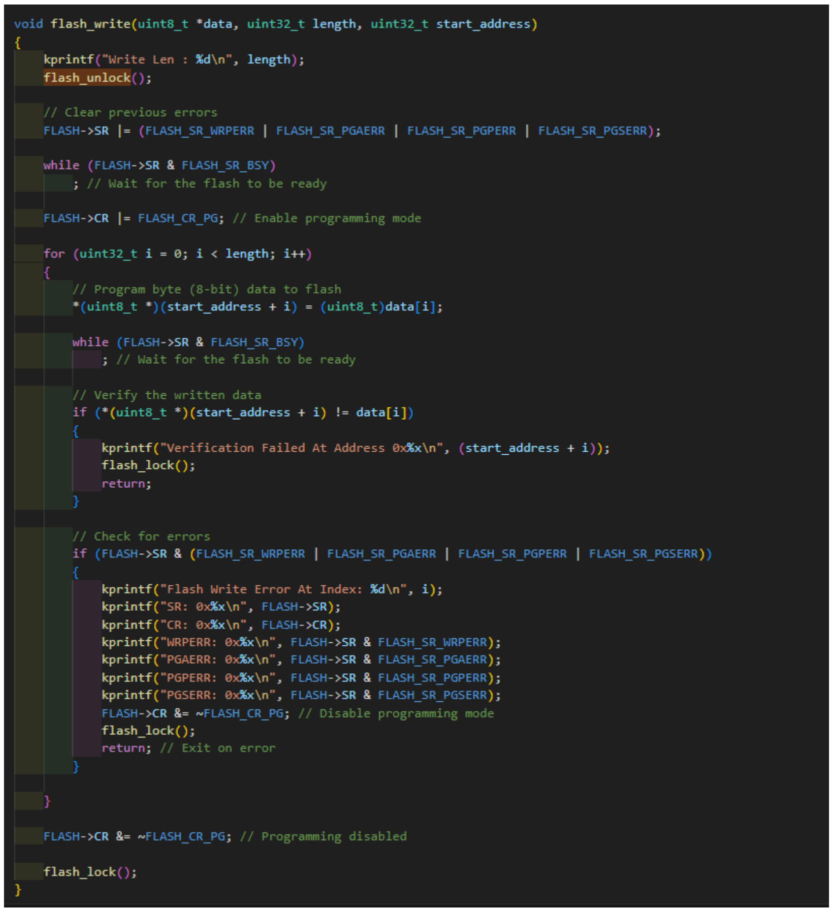
Figure 3.2: Flash erase

3.2 Firmware Transfer and Validation

The firmware update process involves a serial communication protocol where the STM32 microcontroller receives firmware data reliably through a structured procedure. This is achieved via a Python script (Terminal server) on the host system and the corresponding `kmain` function on the STM32 microcontroller. Below is an outline of the key steps for receiving firmware data, packet format, and ensuring data integrity with CRC-based validation.

Initial Setup and Serial Communication

The Python script (Terminal server) initializes the serial connection at 115200 baud on COM3, with a delay to stabilize the connection. This setup ensures reliable communication between the host and the microcontroller.



```c
void flash_write(uint8_t *data, uint32_t length, uint32_t start_address)
{
    kprintf("Write Len : %d\n", length);
    flash_unlock();

    // Clear previous errors
    FLASH->SR |= (FLASH_SR_WRPERR | FLASH_SR_PGAERR | FLASH_SR_PGPERR | FLASH_SR_PGSERR);

    while (FLASH->SR & FLASH_SR_BSY)
        ; // Wait for the flash to be ready

    FLASH->CR |= FLASH_CR_PG; // Enable programming mode

    for (uint32_t i = 0; i < length; i++)
    {
        // Program byte (8-bit) data to flash
        *(uint8_t *)(start_address + i) = (uint8_t)data[i];

        while (FLASH->SR & FLASH_SR_BSY)
            ; // Wait for the flash to be ready

        // Verify the written data
        if (*(uint8_t *)(start_address + i) != data[i])
        {
            kprintf("Verification Failed At Address 0x%x\n", (start_address + i));
            flash_lock();
            return;
        }

        // Check for errors
        if (FLASH->SR & (FLASH_SR_WRPERR | FLASH_SR_PGAERR | FLASH_SR_PGPERR | FLASH_SR_PGSERR))
        {
            kprintf("Flash Write Error At Index: %d\n", i);
            kprintf("SR: 0x%x\n", FLASH->SR);
            kprintf("CR: 0x%x\n", FLASH->CR);
            kprintf("WRPERR: 0x%x\n", FLASH->SR & FLASH_SR_WRPERR);
            kprintf("PGAERR: 0x%x\n", FLASH->SR & FLASH_SR_PGAERR);
            kprintf("PGPERR: 0x%x\n", FLASH->SR & FLASH_SR_PGPERR);
            kprintf("PGSERR: 0x%x\n", FLASH->SR & FLASH_SR_PGSERR);
            FLASH->CR &= ~FLASH_CR_PG; // Disable programming mode
            flash_lock();
            return; // Exit on error
        }

    }

    FLASH->CR &= ~FLASH_CR_PG; // Programming disabled

    flash_lock();
}
```

Figure 3.3: Flash write

Firmware Conversion and Preparation

The firmware update process begins by converting the firmware from ELF to binary format using the `objcopy` command, as the STM32 expects binary data. This conversion, executed by the `convert_elf_to_bin` function within the Python script (Terminal server), prepares the firmware file for direct memory programming.

Packet Transmission Structure

The firmware binary data is transmitted in segments or packets, allowing for organized communication:

- **File Size Transmission**: Before sending the actual firmware data, the total file size is sent as a 5-character string. This informs the STM32 of the total data to expect, aiding in memory allocation and transfer preparation.

- **Data Packet Structure**: The firmware data is divided into packets, each containing:
  - A data chunk of up to 32 bytes.
  - A single-byte checksum calculated by XOR-ing all bytes in the chunk, ensuring data integrity.

The segmentation is controlled within the Python script (Terminal server), with each chunk sent sequentially to the microcontroller.

Checksum Calculation and CRC-Based Validation

On the Host Side: Within the Python script (Terminal server), each packet's checksum is calculated by XOR-ing all bytes in the data chunk (via the `calculate_checksum` function). The checksum is then appended to the data and transmitted to the STM32.

On the STM32 Side (`kmain`): The `kmain` function receives each packet, calculates its own checksum for the received data chunk, and compares this with the transmitted checksum. This is critical for detecting data errors. If the checksums match, the STM32 acknowledges with an "OK"; otherwise, it requests a "RESEND" of the packet.

Acknowledgment and Error Handling

For each transmitted packet, the Python script (Terminal server) waits for acknowledgment from the STM32:

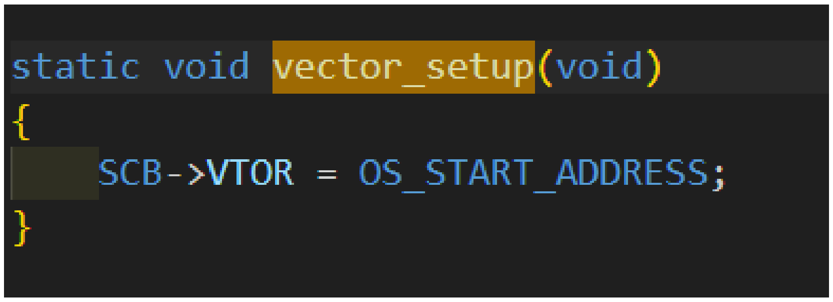
- If "OK" is received, indicating successful verification, the host proceeds to the next packet.
- If "RESEND" is received, the packet is resent to correct any transmission error.
- Unexpected responses trigger a halt in the transfer to prevent data corruption.

Completion

Once all packets are transmitted and verified, the Python script (Terminal server) concludes the transfer, marking the firmware update as successful. This structured approach, using segmented packets and CRC validation, ensures the STM32 reliably receives and verifies each firmware data chunk. This process minimizes the risk of corrupted firmware installation and highlights the robustness of the update protocol established between the host and STM32 in the Python script (Terminal server) and `kmain`.

3.3 Interrupt Vector Table Relocation

First, in bootloader the `SCB->VTOR` register is configured to point to the starting address of the bootloader's vector table.



```
static void vector_setup(void)
{
    SCB->VTOR = OS_START_ADDRESS;
}
```

Figure 3.4: Bootloader vector table address allocation

For main app `SCB->VTOR` register is configured to point to the starting address of the OS.

The bootloader size is defined to calculate the starting address of the main application (OS vector table address). The bootloader size is defined as:

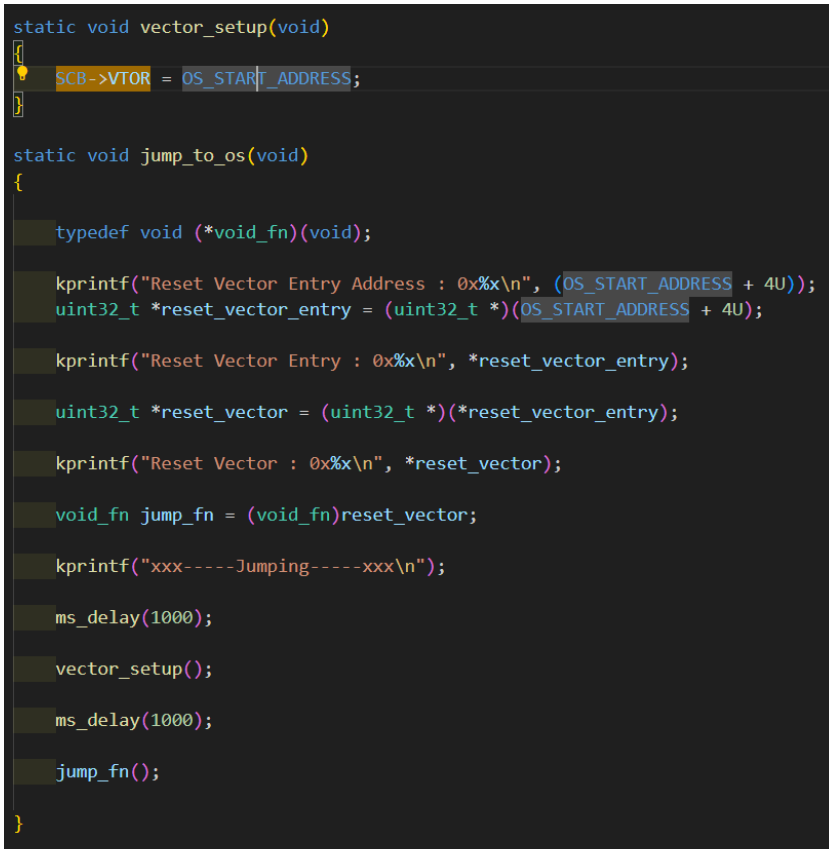
```
#define BOOTLOADER_SIZE (0x24000U) % 144kB
```

This means the bootloader occupies 144 kB of flash memory.

The main application starts at the address calculated by adding the bootloader size to the base address of the flash memory. The starting address of the main application is defined as:

```
#define MAIN_APP_START_ADDRESS (0x08024000U) % main app address
```

This value, 0x08024000, represents the start address of the main application after the bootloader section in the flash memory.



```c
static void vector_setup(void)
{
    SCB->VTOR = OS_START_ADDRESS;
}

static void jump_to_os(void)
{

    typedef void (*void_fn)(void);

    kprintf("Reset Vector Entry Address : 0x%x\n", (OS_START_ADDRESS + 4U));
    uint32_t *reset_vector_entry = (uint32_t *)(OS_START_ADDRESS + 4U);

    kprintf("Reset Vector Entry : 0x%x\n", *reset_vector_entry);

    uint32_t *reset_vector = (uint32_t *)(*reset_vector_entry);

    kprintf("Reset Vector : 0x%x\n", *reset_vector);

    void_fn jump_fn = (void_fn)reset_vector;

    kprintf("xxx-----Jumping-----xxx\n");

    ms_delay(1000);

    vector_setup();

    ms_delay(1000);

    jump_fn();

}
```

Figure 3.5: Main app vector table address allocation

Bibliography

[1] Youtube Tutorial: Writing a Simple Bootloader for STM32, `https://www.youtube.com/playlist?list=PLArwqFvBIlwHRgPtsQAhgZavlp42qpkiG`

[2] STMicroelectronics, *STM32F4xx Reference Manual.*

[3] Youtube Tutorial: Simple Bootloader for STM32, `https://www.youtube.com/playlist?list=PLnMKNibPkDnEb1sphpdFJ3bR9dNy7S6mO`