STM32 Custom Bootloader

by

Raunak
[redacted]
rraunak@asu.edu

Viraj Savaliya
[redacted]
vsavaliy@asu.edu

Srihari Danduri
[redacted]
ddanduri@asu.edu

Project 2 Report for EEE 598: Mobile Systems Architecture

Master of Science in Computer Engineering (Electrical Engineering)

Course Instructor:

Dr. Robert LiKamWa

ARIZONA STATE UNIVERSITY

December 2020

ACKNOWLEDGMENTS

We would like to thank Dr. Robert LiKamWa for providing us the opportunity to work on a project in our domain of interest which will helped us grow and explore new things and also is a valuable addition to our profile. Also, we would like to thank him again for conducting lectures on important topics in Mobile Systems like Computing, Mobile Security, Virtual Reality, Instructions Set Architecture, Memory Management, Power Management, Android Runtime, Sensor Processing, hierarchy and abstractions in systems, etc. These topics have helped us to get insight on various aspects of Mobile Systems which is currently on a rise and dominating in most sectors of Industry.

TABLE OF CONTENTS

Declaration of Honor Code

We, the students of Arizona State University, adopt this code as an affirmation of our commitment to academic integrity and our participation in ethical education.

We embrace our duty to uphold ASU's Honor Code, and in light of that duty,

We promise to refrain from academic dishonesty.

We pledge to act with integrity and honesty and to promote these values among our peers. We agree to always abide by the Sun Devil Way and uphold the values of the New American University."

The undersigned acknowledge and certify that all the tasks completed for the current stage of the project are within the ASU honor code and are our own work and with equal contributions from each team member.

Raunak	Viraj Savaliya	Danduri Srihari
Member 1	Member 2	Member 3

PREFACE

This report is about the implementation of a custom bootloader for stm32 micro-controllers. It is implemented in C programming language and is evaluated based on the expected features of a general bootloader for any firmware. The report is written for the second Project/Report submission for EEE 598: Mobile Systems Architecture in Fall 2020 at Arizona State University. The report structure is based on the target goals, implementation as per the requirements of the project and results that we achieved in our implementation. Various online resources, textbooks as well as external course notes were referred for understanding the subject matter and to form the detailed report for the topic. All the references which were used are mentioned at the end of the document and any content which is exactly taken from a source is cited as well.

Chapter 1: GOALS

1.1 High-level Objectives

The following are the high-level objectives of the project we sought to learn and accomplish for the project:

- The significance of the bootloader.
- How the bootloader works and how it is implemented in the firmware of an ARM based micro-controllers (MCUs).
- In-application programming of a firmware on an external source.
- Finding out the system bottlenecks to performance.
- To understand the linker setting of the application firmware.
- How the firmware works and handles the bootloader on startup in ARM based MCUs.
- Vector table addressing and modification.
- Memory protection unit(MPU) of ARM MCUs.
- Different memory regions, memory management and their allocation.
- How to flash an application on MCUs.
- Different protocols for communicating with the MCU for different types of message packets to be transferred.
- Necessary modifications required inside the MCU's memory in order to accept a new application.
- How switching between various applications and between bootloader and an application is implemented.
- Necessary initializations, deinitializations and configurations required for MCU to run an application.
- Establish a robust communication between the user and MCU.
- Achieve acceptable performance with low latency in servicing requests for the firmware.
- Careful handling of memory sectors when erasing and allocating memory.

1.2 Project Description

The following is the re-stated description of the assignment statement from our point of view:

- To implement a customizable bootloader for STM32 microcontrollers.
- Triggering appropriate boot pins for different boot modes of the MCU.
- Finding out different techniques to make the bootloader more efficient.
- Communicating messages between the user and MCU to make service requests.
- Using appropriate protocols for different types of message packets.
- Perform necessary checksum verifications to every new message packets.
- Proper handling of memory erasing and allocating space to flash new applications.
- To implement necessary flash protection to disable writing for an application to critical memory sectors.
- Develop smooth communication to service requests for any firmware upgrade and application handling.
- Make necessary modifications to linker scripts and vector table addresses for bootloader and applications.
- Make correct configurations, initializations and deinitializations for the MCU to setup environment for functionality of bootloader and applications.
- Develop well-structured modules which are easy to understand and which can be easily customized and imported to different MCUs.
- Implementing an error handler to make the design fail-safe.
- Using Serial tracing over SWO or other software modules for debugging purposes.

Chapter 2: DESIGN IMPLEMENTATION

2.1 Resources requirements:

1. STM32F3 Series Board – ARM® 32-bit Cortex -M4

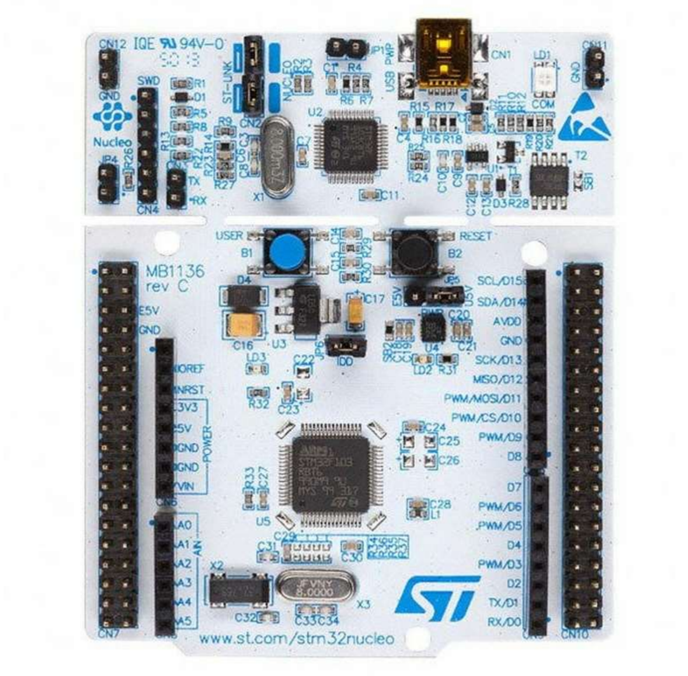


Fig . STM32 Nucleo-F303RE

2. ARM Cortex IDE (STM32CubeIDE)
3. Tera Term – Open-Source Terminal Emulator
4. STM32 ST-Link Utility – Software interface for programming/debugging.
5. STM32 ST-Link Server – Application to share debug interface.
6. Serial to USB Converter
7. Jumper wires

2.2 Program flow:

The working of the bootloader implemented for updating firmware and executing multiple applications is as follows:

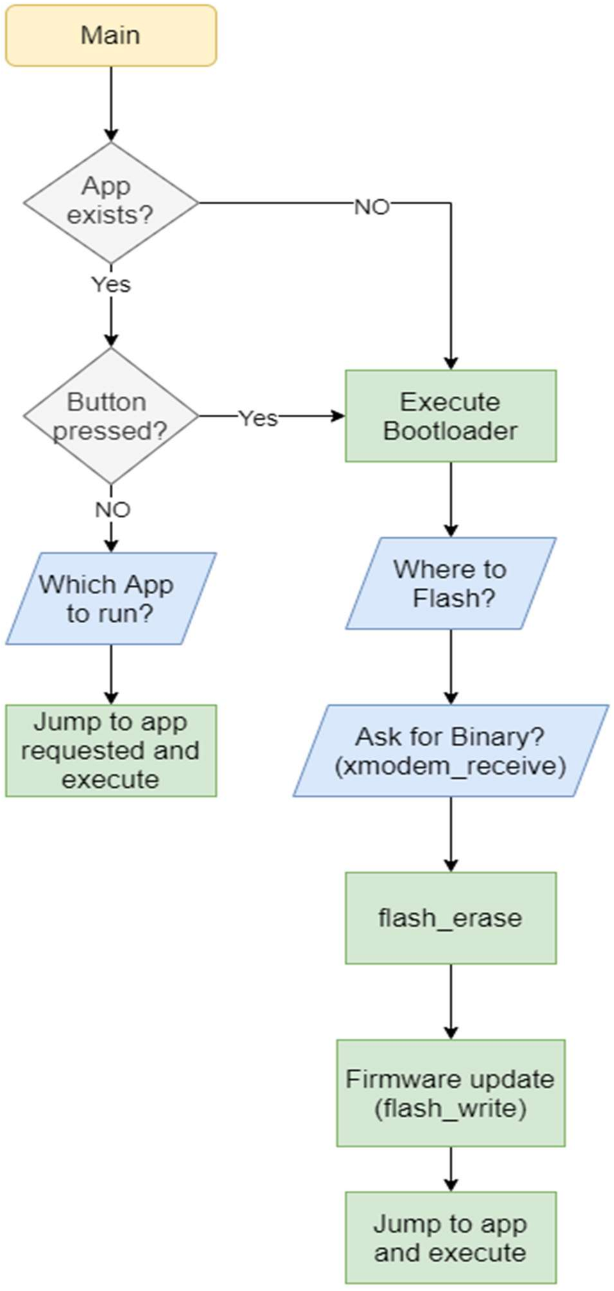


Fig. Custom Bootloader Flowchart

In the Implementation if the reset button is pressed then the program executes from the start and if the user button is also pressed when reset is pressed then the user is requesting an upgrade to the firmware. The track of all the applications present and its total count is stored in the flash memory and the program runs any applications requested by the user.

2.3 Apis in the algorithm:

Flash Apis:

1. erase_flash()

   ```
   flash_status erase_flash(uint32_t addr)
   ```

   This function takes address as argument. To erase the flash we have to unlock it first using **HAL_FLASH_Unlock**. This address specifies the starting address to erase the flash memory. It uses the HAL(Hardware abstraction layer) Api, **HAL_FLASHEx_Erase** to erase the flash memory. We must specify number of pages to erase, erase type and page address to erase that part of flash memory. And at the end, we should lock the flash using **HAL_FLASH_Lock.**

2. write_flash()

   ```
   flash_status write_flash(uint32_t addr, uint32_t *data, uint32_t len)
   ```

   This function takes address, data to write and length as arguments. To write to the flash we have to unlock the flash using **HAL_FLASH_Unlock** and we are checking if the specified address is greater than the flash end address and then setting the flash status to error. After this, we are programming the flash memory using **HAL_FLASH_Program**. And at the end we should lock the flash using **HAL_FLASH_Lock**.

3. flash_jump_userapp()

   ```
   void flash_jump_userapp(void)
   ```

   This function is of void type and takes no arguments. It is used to jump to the location of the application flashed in the memory by setting the main stack pointer to the address of the application.

UART Apis:

4. uart_rcv()

   ```
   uart_status uart_rcv(uint8_t *data, uint16_t len)
   ```

   This function takes the data and length of the data as arguments and uses **HAL_UART_Receive** to receive the data from the serial device.

5. uart_trnsmt_str()

```c
uart_status uart_trnsmt_str(uint8_t *data)
```

This function takes the data to be sent as arguments and uses **HAL_UART_Transmit** to transmit the data to the serial device. It checks for the string input and transfers the whole string value.

6. uart_trnsmt_ch()

```c
uart_status uart_trnsmt_ch(uint8_t data)
```

This function also takes the data to be sent as arguments and uses **HAL_UART_Transmit** to transmit the data character by character.

XMODEM Apis:

7. xmodem_rcv()

```c
void xmodem_rcv(void)
```

This function is of void type and takes no arguments. In this function we are using xmodem protocol to receive the binary through uart to update our firmware. In this function we will be calling our wrapper function, **xmodem_packet_handler** to erase the flash and then write our application in the flash to update the firmware. We are also checking the incoming data through CRC-16 by calling the wrapper function **xmodem_calculate_crc**. And if there are any errors, we are handling those through our function, **xmodem_error_handler.**

Main Apis:

8. bootloader_uart_read_data()

```c
void bootloader_uart_read_data(void)
```

This function is of void type and takes no arguments. In this function we would be taking the start address of the application where we want our application to be flashed in the flash memory. In this function we are using our xmodem_rcv to receive the binary file of the application and after updating the firmware it jumps to the user application.

9. printmsg()

```
void printmsg(char *format,...)
```

This api prints messages to uart3

10. wait_for_EnterKey()

```
void Wait_For_EnterKey(void)
```

It will wait till the user enters the enter key. If any invalid key is entered, it will prompt the user to try again.

11. My_flash_erase()

```
flash_status  My_flash_erase(uint32_t address)
```

This api will take address to be erased as an input and returns success, if it was able to clear the flash and fail if it doesn't. In total it erases 4 bytes of data.

12. My_flash_write()

```
flash_status  My_flash_write(uint32_t  address,  uint32_t  *data,  uint32_t
length)
```

It takes in the data pointer buffer address and address where we need to flash the data and length of the data buffer. This api flashes the data buffer data into the particular flash address.

Chapter 3: WHAT WE LEARNT

1. External Reset:

Every MCU has an external reset pin, which is used to reset the microcontroller board. The external reset button located on the stm32f303re board is shown in the picture below,

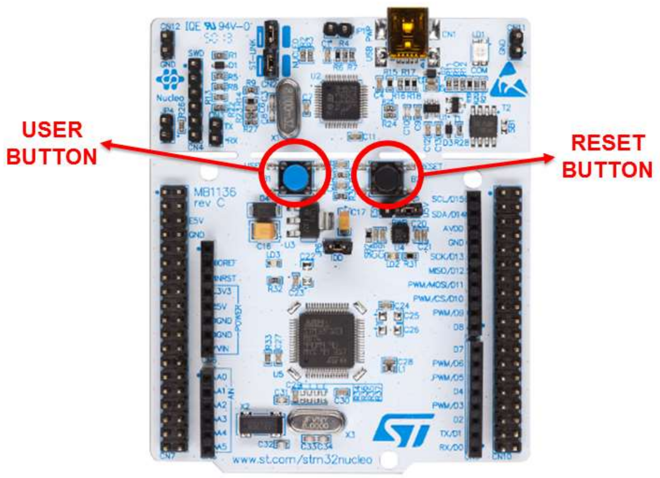


Fig. Reset and User button on board

The circuitry behind the reset button is shown in the figure below, whenever the reset button is not pressed, the capacitor gets charged, pulling the reset pin high. When the button is pressed, the capacitor gets discharged and the reset pin is pulled to a low value. The capacitor improves the electromagnetic susceptibilty of the system. Electromagnetic susceptibility is defined as the ability of the system to operate without fault under electrical disturbances and noise. The capacitor can remove high frequencies voltage signal from NRST pin and thus reduces the transient demands on the system and power supply unit. Sytem reset can also be generated through the software, where instead of the button we use NRST pin of the microcontroller, which is pulled low, internally inside the processor to reset the system.

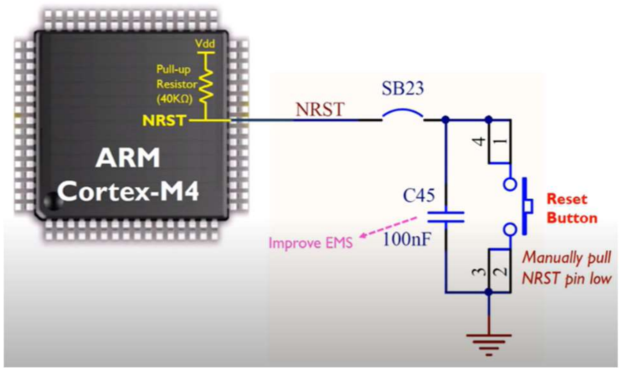


Fig. Reset pin circuitry

2. Booting Process:

The first word in the memory contains the value to initialize the main stack pointer. The second word in the memory is the pointer to the function reset handler, which is the starting memory address or is the entry point of the function reset handler. Interrupt vector table describing the memory locations is depicted below,

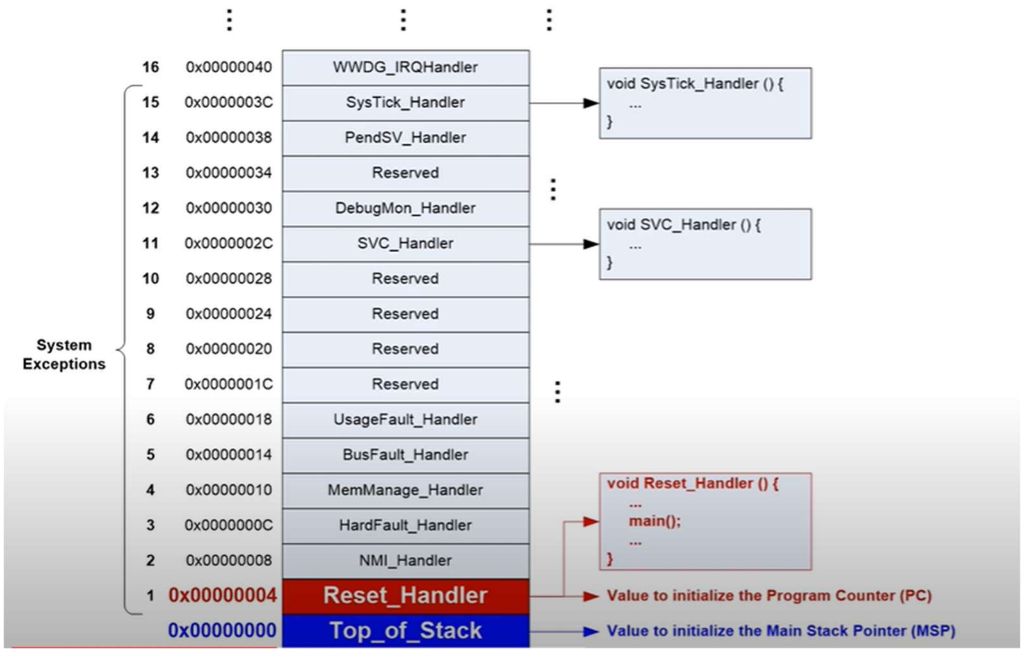

16 0x00000040 WWDG_IRQHandler
15 0x0000003C SysTick_Handler
14 0x00000038 PendSV_Handler
13 0x00000034 Reserved
12 0x00000030 DebugMon_Handler
11 0x0000002C SVC_Handler
10 0x00000028 Reserved
9 0x00000024 Reserved
8 0x00000020 Reserved
7 0x0000001C Reserved
6 0x00000018 UsageFault_Handler
5 0x00000014 BusFault_Handler
4 0x00000010 MemManage_Handler
3 0x0000000C HardFault_Handler
2 0x00000008 NMI_Handler
1 0x00000004 Reset_Handler
0x00000000 Top_of_Stack
System Exceptions
void SysTick_Handler () { ... }
void SVC_Handler () { ... }
void Reset_Handler () { ... main(); ... }
Value to initialize the Program Counter (PC)
Value to initialize the Main Stack Pointer (MSP)

Fig. Interrupt Vector Table

Booting steps:

a. Read pins BOOT0 and BOOT1 and determine the boot mode.
b. Fetch Main stack pointer (MSP) from address 0x00000000.
c. Fetch Program counter (PC) from address 0x00000004.

In the first step, when the processor boots, it first reads BOOT0 and BOOT1 pins to determine the boot mode. In the second step, the processor copies the value stored at the memory address 0x00000000 to the main stack pointer, basically MSP is initialized in this step. In the third step, the processor copies the value stored at memory address 0x00000004 to the program counter (PC), program counter always holds the memory address of the next instruction to be executed by the processor. Hence, immediately after the processor boots, the processor will start to execute the function reset handler. Reset handler is used to perform some hardware initialization, such as initialization of data segment and bss segment, after that the reset handler calls the main() function and transfers the control to the main() function.

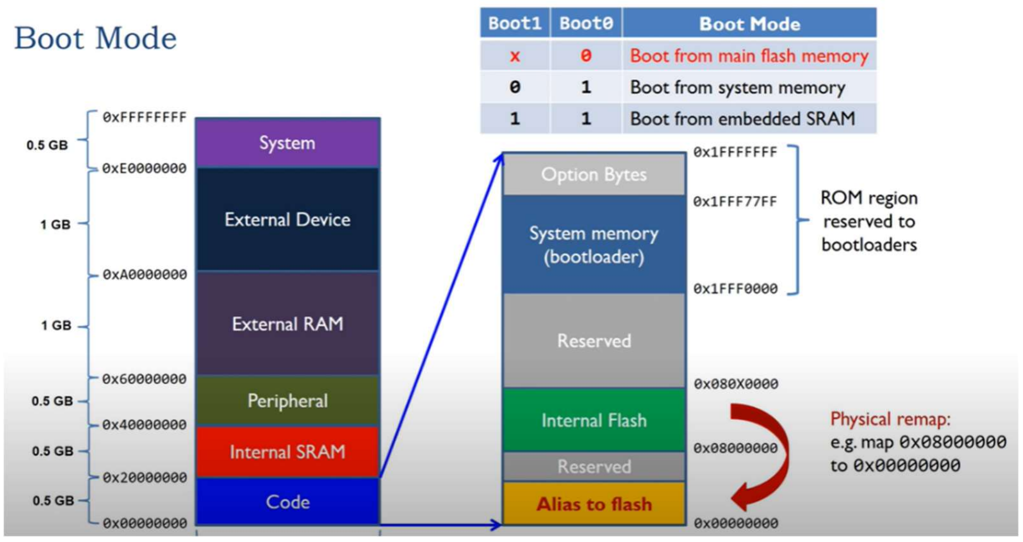
3. Boot Modes:

There are 3 types of boot mode in STM32 based microcontroller board,

Boot1	Boot0	Boot Mode
X	0	Boot from main flash memory
0	1	Boot from system memory
1	1	Boot from embedded SRAM

On the left side, memory map of a ARM-Cortex based microcontroller is shown. The address range of each memory region is fixed. The code region ranges from 0x00000000 to 0x1FFFFFFF in hex. The top area is the ROM region and is reserved to store bootloaders. The middle area is the in-chip flash memory and the bottom area is an area that can be mapped to the internal flash, system memory and internal SRAM. The starting address of the internal flash, system memory and internal SRAM is also fixed.

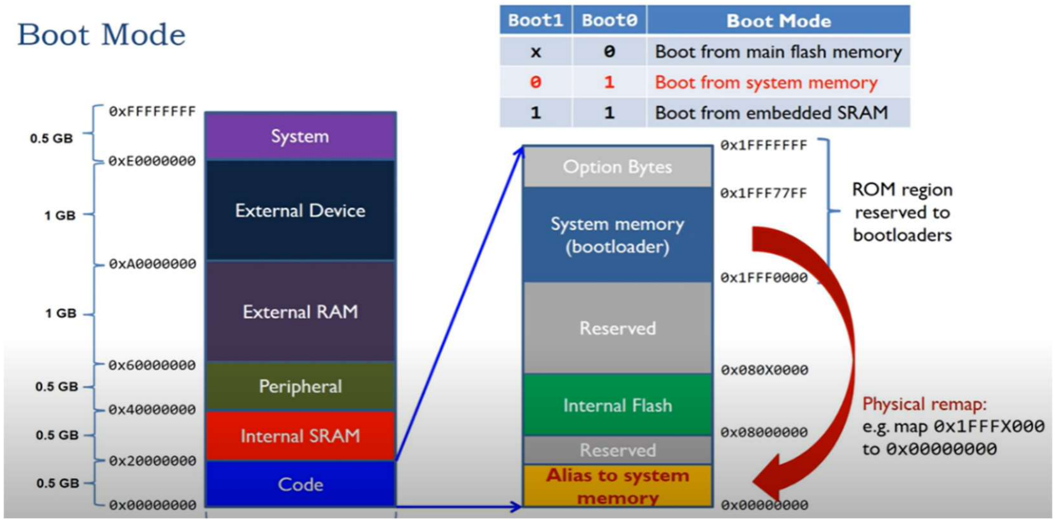
The boot modes are determined by the voltage applied to the boot0 and boot1 pins. If the boot0 is connected to the ground the processor will boot from the internal flash memory. The processor will physically map the internal flash memory to 0x00000000. In the figure below, 0x80000000 is mapped to 0x00000000. In short, flash memory contents can be accessed from 0x00000000 or 0x80000000. So, it will fetch the stack pointer and program counter starting from the address of the internal flash memory i.e. 0x80000000.



Boot1	Boot0	Boot Mode
x	0	Boot from main flash memory
0	1	Boot from system memory
1	1	Boot from embedded SRAM

Fig. Boot mode - main flash memory

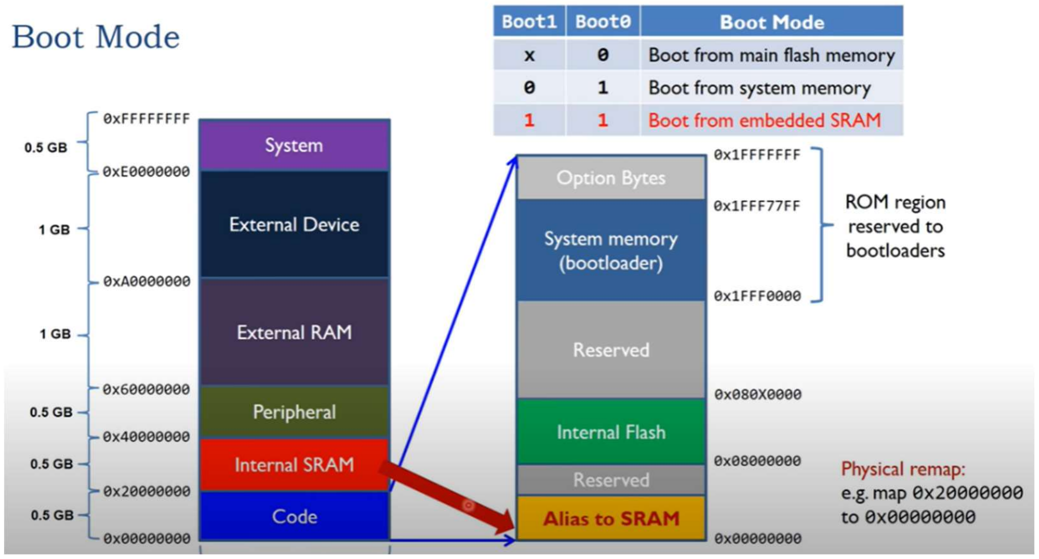
When the boot 1 pin is low and boot0 pin is high, then the system memory is physically mapped to the address 0x00000000. In this boot mode, the processor can reprogram the flash memory or perform the device firmware update. System memory contains the STM32 default non-erasable bootloader, which is embedded into the device at manufacturing. We have used this mode to write our bootloader to the flash and then we are booting from the flash.



Boot1	Boot0	Boot Mode
x	0	Boot from main flash memory
0	1	Boot from system memory
1	1	Boot from embedded SRAM

Fig. Boot mode – system memory

When boot1 and boot0 pins are both high, the internal SRAM is mapped to the bottom region i.e address 0x20000000 is mapped to 0x00000000. Therefore, the processor boots from SRAM.

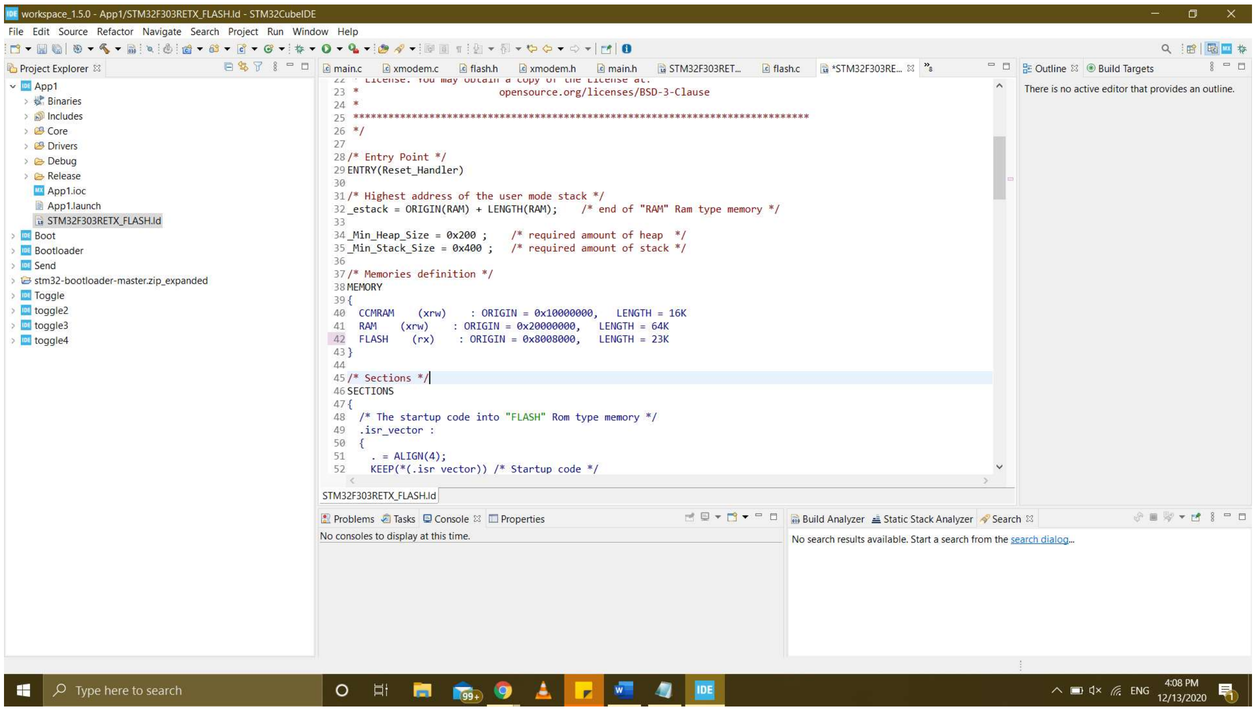


Boot1	Boot0	Boot Mode
x	0	Boot from main flash memory
0	1	Boot from system memory
1	1	Boot from embedded SRAM

Fig. Boot mode – embedded SRAM

4. Linker Script:

Linker is a program that, links/joins one or more object files generated by the compiler together into an executable program. Which means that it creates one executable program from multiple object files. In STM32, we use .ld file (linker script) to modify the linker configuration. Since, our bootloader starts from 0x80000000 and ends at 0x08008000, our application program should start from 0x08008000, so as we can see in the given figure, we have modified the starting address for our App1 to 0x8008000 and changed the size that will be visible to the App1 in the flash to 23 kB. In order for our App1 to flash it to correct memory location, we need to change the vector table offset address too.



```
 * License. You may obtain a copy of the License at:
 *                       opensource.org/licenses/BSD-3-Clause
 *
 ******************************************************************************
 */

/* Entry Point */
ENTRY(Reset_Handler)

/* Highest address of the user mode stack */
_estack = ORIGIN(RAM) + LENGTH(RAM);    /* end of "RAM" Ram type memory */

_Min_Heap_Size = 0x200 ;    /* required amount of heap  */
_Min_Stack_Size = 0x400 ;   /* required amount of stack */

/* Memories definition */
MEMORY
{
  CCMRAM    (xrw)    : ORIGIN = 0x10000000,   LENGTH = 16K
  RAM    (xrw)    : ORIGIN = 0x20000000,   LENGTH = 64K
  FLASH    (rx)    : ORIGIN = 0x8008000,   LENGTH = 23K
}

/* Sections */
SECTIONS
{
  /* The startup code into "FLASH" Rom type memory */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
```

There is no active editor that provides an outline.

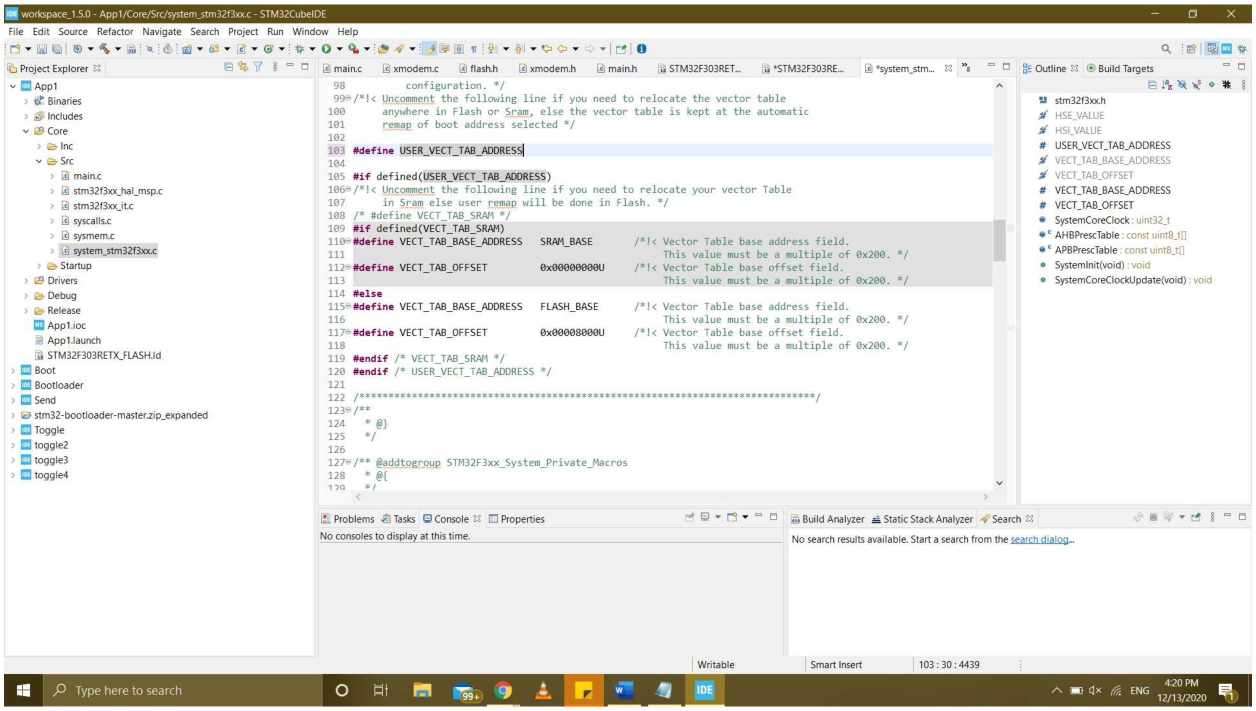
No consoles to display at this time.

No search results available. Start a search from the search dialog...

Fig. STM32 linker script snapshot

5. Interrupt Vector Table:

An interrupt vector table is a data structure, which stores the addresses of the interrupt handler for each and every type of interrupt. As seen earlier, we take the starting address of the flash to store our bootloader and as the app is flashed it should use it’s vector table offset to jump to the correct starting address location of the flash memory, to the region after the bootloader from where the app region starts. In STM32, to define a custom vector table offset, we can use system_stm32f3xx.c file, as we can see from the figure, we have defined the VECT_TAB_OFFSET address as 0x00008000 so that, our App1 is flashed from 0x08008000 memory location in the flash.



```
           configuration. */
/*!< Uncomment the following line if you need to relocate the vector table
     anywhere in Flash or Sram, else the vector table is kept at the automatic
     remap of boot address selected */

#define USER_VECT_TAB_ADDRESS

#if defined(USER_VECT_TAB_ADDRESS)
/*!< Uncomment the following line if you need to relocate your vector Table
     in Sram else user remap will be done in Flash. */
/* #define VECT_TAB_SRAM */
#if defined(VECT_TAB_SRAM)
#define VECT_TAB_BASE_ADDRESS   SRAM_BASE       /*!< Vector Table base address field.
                                                     This value must be a multiple of 0x200. */
#define VECT_TAB_OFFSET         0x00000000U     /*!< Vector Table base offset field.
                                                     This value must be a multiple of 0x200. */
#else
#define VECT_TAB_BASE_ADDRESS   FLASH_BASE      /*!< Vector Table base address field.
                                                     This value must be a multiple of 0x200. */
#define VECT_TAB_OFFSET         0x00008000U     /*!< Vector Table base offset field.
                                                     This value must be a multiple of 0x200. */
#endif /* VECT_TAB_SRAM */
#endif /* USER_VECT_TAB_ADDRESS */

/******************************************************************************/
/**
  * @}
  */

/** @addtogroup STM32F3xx_System_Private_Macros
  * @{
  */
```

No consoles to display at this time.

No search results available. Start a search from the search dialog...

Fig. STM32 Interrupt Vector Table address snapshot

6. UART:

UART stands for Universal asynchronous receiver/transmitter. It is used for serial data communication. For serial communication through UART we have to connect Transmit (Tx) and Receive (Rx) pins opposite to one another in both the devices. The data is sent serially one by one from the least significant bit to the most significant one. In this project, we have used UART to communicate and send the binary to the microcontroller board from our system to update the firmware.

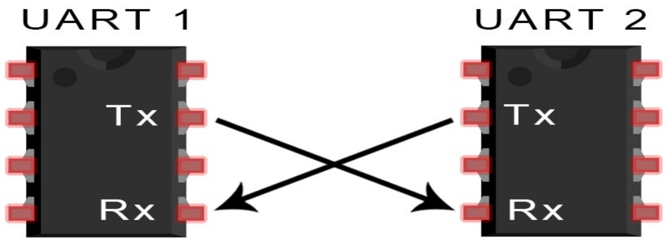


Fig. UART communication

7. XMODEM:

It is a file transfer protocol to transmit files between different machines. The basic functioning of the protocol is to break up the files into number of packets and send them in series. Along with the packets, additional information about the correctness of the packet is sent to the receiver which can verify it for correct reception. If the packet is incorrect the receiver can ask the transmitter to re-send the packet or abort the communication if multiple packets are tampered. Here the transfer process if driven by the receiver i.e. the transmitter only starts sending the data when receiver asks for it. At first, the transmitter is waiting for a NAK byte by the receiver to start. NAK is also sent as not acknowledge signal, informing the transmitter that the packet was not received and re-send it. The receiver keeps waiting for the packets till EOT (end of transmission) is sent by the transmitter. For all the packets received, the receiver checks for the packet number, using 1's complement of previous number to make sure if any packet missed and also checks the checksum for correct transmission of the packet. In any event of incorrect reception in all the three checks, it will send transmitter a CAN (Cancel) signal for the transmission. Also, at the start for the very first packet, SOH (Start of Header) is sent to the receiver. The communication signals between the transmitter and receiver are shown in the figure below.

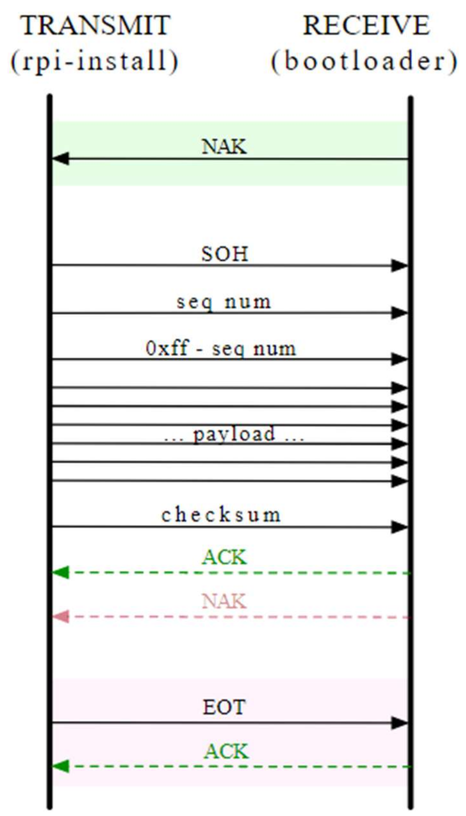


Fig. XMODEM File transfer protocol

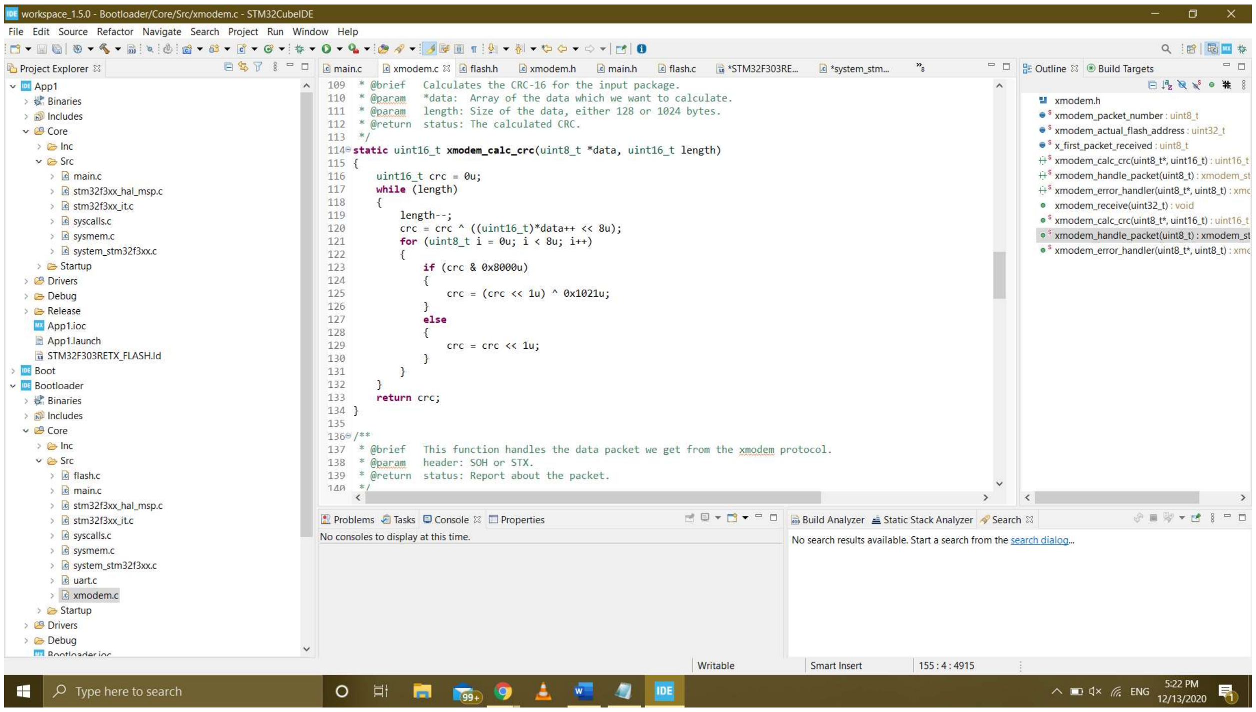
The packet structure to be transmitted is as shown below:

SOH	Packet Number	1's Compliment of Packet Number	The Packet	Checksum
1 Byte	1 Byte	1 Byte	128 Byte	1 Byte

Fig. XMODEM Packet structure

8. Cycle Redundancy Check (CRC) Check:

The cycle redundancy check is used to validate the received data, it helps us track if there is an accidental change in the transmitted data. At the receiving end, data is divided by a check value and if the remainder comes out to be 0 then, then data is considered valid and is accepted else, the data is considered to be corrupted and is rejected. We have used CRC when receiving the binary through XMODEM receive. To divide we xor the divisor with the 8 bytes of the input starting from the most significant bit. In our code, we have used 0x1021 as the divisor to check the validity of the received data.


workspace_1.5.0 - Bootloader/Core/Src/xmodem.c - STM32CubeIDE
File Edit Source Refactor Navigate Search Project Run Window Help
main.c xmodem.c flash.h xmodem.h main.h flash.c *STM32F303RE... *system_stm...
Project Explorer
Outline Build Targets

```c
 * @brief   Calculates the CRC-16 for the input package.
 * @param   *data:  Array of the data which we want to calculate.
 * @param   length: Size of the data, either 128 or 1024 bytes.
 * @return  status: The calculated CRC.
 */
static uint16_t xmodem_calc_crc(uint8_t *data, uint16_t length)
{
    uint16_t crc = 0u;
    while (length)
    {
        length--;
        crc = crc ^ ((uint16_t)*data++ << 8u);
        for (uint8_t i = 0u; i < 8u; i++)
        {
            if (crc & 0x8000u)
            {
                crc = (crc << 1u) ^ 0x1021u;
            }
            else
            {
                crc = crc << 1u;
            }
        }
    }
    return crc;
}

/**
 * @brief   This function handles the data packet we get from the xmodem protocol.
 * @param   header: SOH or STX.
 * @return  status: Report about the packet.
 */
```

No consoles to display at this time.

No search results available. Start a search from the search dialog...

Fig. XMODEM CRC checking code snapshot

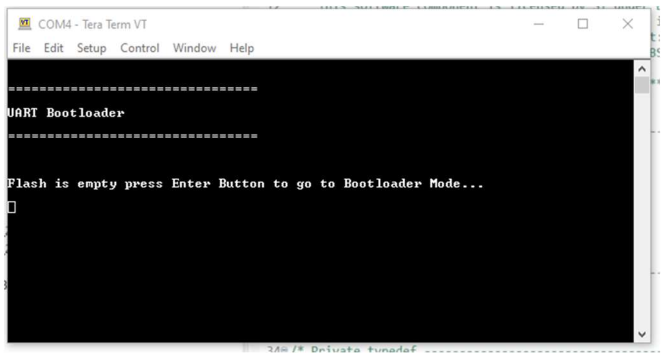
AREA	CRC
Purpose	Detection of data transmission errors between the transmitter and receiver
Error Detection	Double-digit error detection capacity
Complexity	Usage of complex functions to detect errors
Reliability	More reliable due to mathematical formula used to calculate CRC
Existence	Recent technology with verified results

Chapter 4: WHAT WE ACHIEVED

The following are the notable achievements of our project:

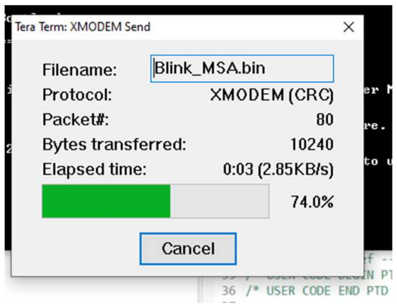
- Implemented a custom bootloader for STM32 microcontrollers.
- Achieved a smooth communication between the STM32 ARM based MCU and the System through UART communication and XMODEM protocol.
- Made a streamlined process bootloader to update the firmware and switch to various application based on the user input.
- Understood the critical steps in flashing the ARM based MCUs.
- Did in-application programming of a firmware form an external source.
- Understood how the bootloader works and how it is implemented in the ARM based MCUs
- Implemented CRC based checking mechanism to verify the validity of the data received through XMODEM protocol and UART.

The screenshots of important outputs achieved for bootloader and the applications flashed into the STM32 board's memory as per the functionality and the working flow developed for the program are as follows:



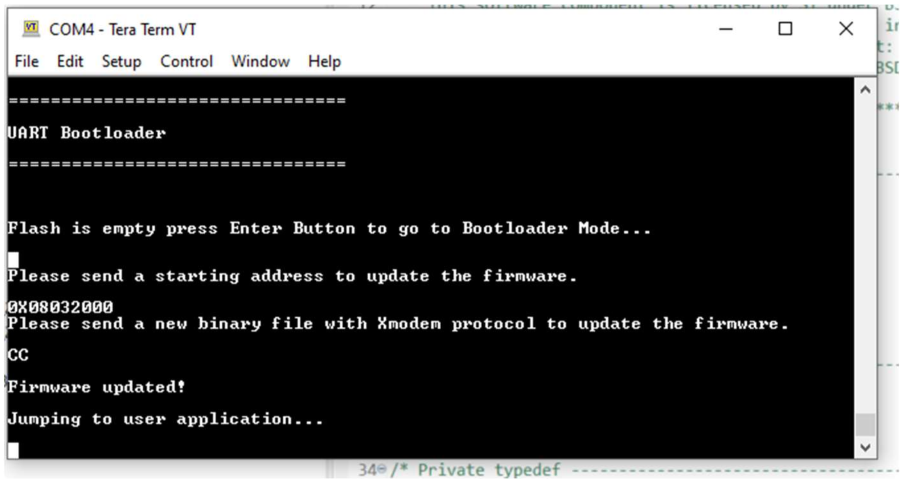
```
================================
UART Bootloader
================================

Flash is empty press Enter Button to go to Bootloader Mode...
```

Fig. Initial screen indicating no applications present



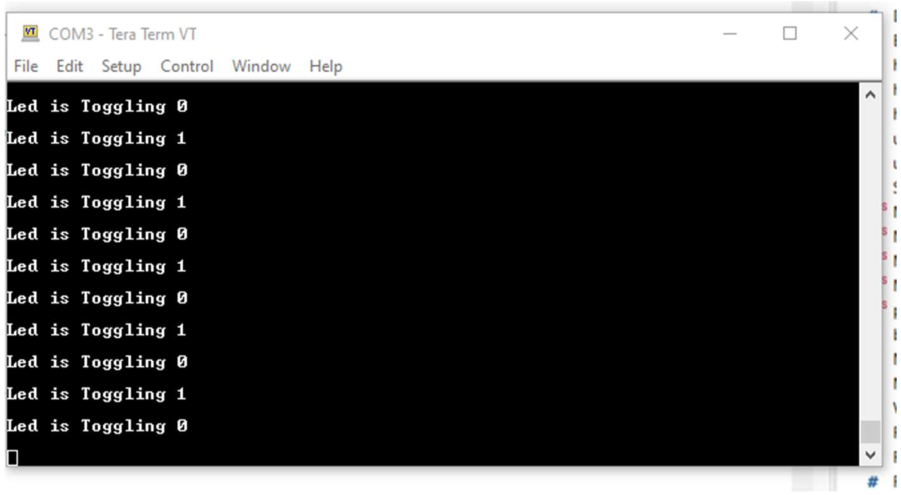
Filename: Blink_MSA.bin
Protocol: XMODEM (CRC)
Packet#: 80
Bytes transferred: 10240
Elapsed time: 0:03 (2.85KB/s)
74.0%

Fig. Flashing of application binary using XMODEM send



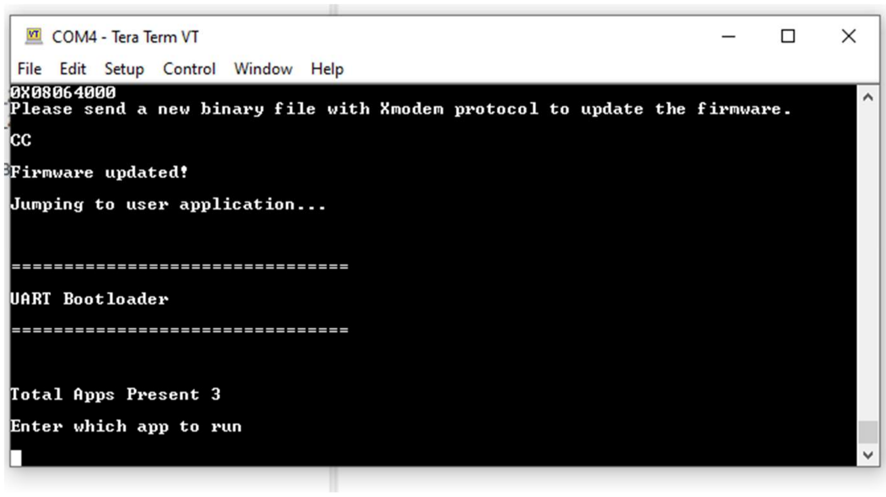
```
================================
UART Bootloader
================================


Flash is empty press Enter Button to go to Bootloader Mode...

Please send a starting address to update the firmware.

0X08032000
Please send a new binary file with Xmodem protocol to update the firmware.

CC

Firmware updated!

Jumping to user application...
```

Fig. Successful loading for Application 1 at its requested address



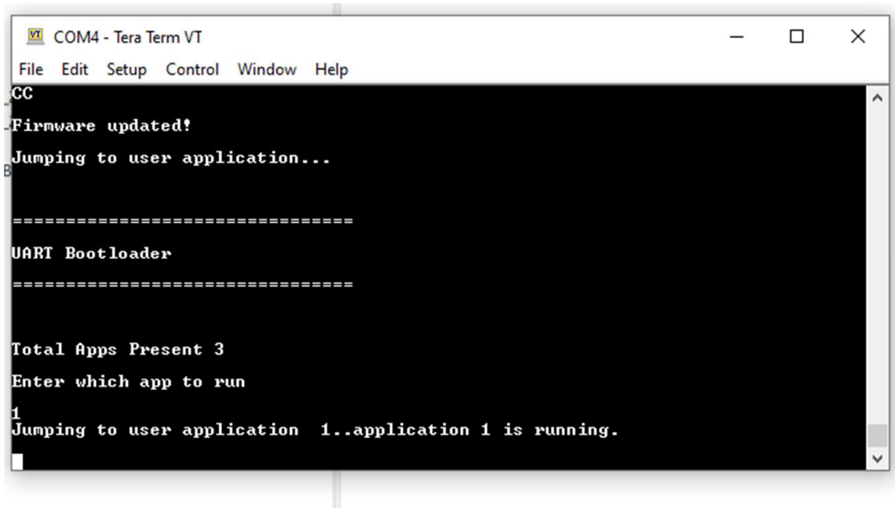
```
Led is Toggling 0
Led is Toggling 1
Led is Toggling 0
Led is Toggling 1
Led is Toggling 0
Led is Toggling 1
Led is Toggling 0
Led is Toggling 1
Led is Toggling 0
Led is Toggling 1
Led is Toggling 0
```

Fig. Successful execution of Application 1 – Toggling LED



```
0X08064000
Please send a new binary file with Xmodem protocol to update the firmware.
CC
Firmware updated!
Jumping to user application...


================================
UART Bootloader
================================


Total Apps Present 3
Enter which app to run
```

Fig. Identification of total apps currently present when reset button pressed



```
CC
Firmware updated!
Jumping to user application...


================================
UART Bootloader
================================


Total Apps Present 3
Enter which app to run
1
Jumping to user application  1..application 1 is running.
```

Fig. Successfully jumping to requested application from all the apps currently present

4.1 Project Tutorial:

The GitHub repository for the project is: https://github.com/rraunak/eee598-stm32Bootloader-raunak-savaliya-danduri

The video tutorial explaining all the goals of the project, important concepts involved which we learnt while implementing the project, explanation of the code and demonstration can be found here:

https://drive.google.com/file/d/1PGlNR6MICHNjIIZR72r1yAr60wT6A0E2/view?usp=sharing

Time	Section name
00:00	Project Introduction
02:02	Boot mode
03:33	XMODEM
07:12	CRC
09:51	Code Overview & Bootloader Explanation
26:01	App1 Explanation
27:57	App2 Explanation
29:20	App3 Explanation
31:35	ST-LINK Utility
32:02	STM32 System Bootloader Explanation
35:10	Program Flowchart
36:23	Output Demonstration

Chapter 5: CHALLENGES FOR THE PROJECT

- In initial stage of the project, in order to make connections we followed online resources for selecting GPIO pins and ports on the board and make necessary initializations for in code. Here the issue was the selected pins and ports weren't correct as they will defer for each stm32 board and the resources might be using a different one. Then we corrected it by digging in to the STM32-nucleo-f303re's pin schematic and datasheet to find the correct pins numbers and ports numbers with respect to the name of the pin on the board. We also cross verified it's correctness by checking the associated pins labels in the STM32CubeIDE.

- There are various in-built HAL driver modules for each board of STM32 which we can use as per our requirements. While using these careful attention needs to be paid at the arguments to the APIs. Especially the format type of the arguments as we will need to type cast our variables to it and then pass.

- Another issue while using the in-built HAL APIs was incorrect functionality. This was resolved by reading through the modules and understanding the functionality of each and then selecting appropriate API if present or at times developing our own API.

- When passing the memory address location for a new application sent by user to flash it, some conversions were necessary from decimal to hex or vice versa when using the user input in our code. As incorrect format created issues like reading incorrect memory location, erasing out the entire flash memory or even segmentation fault at times.

- When a new application is sent by the user, a sufficient amount of memory is required to be freed so that the application binary can be dumped into that space. The calculation of the number of pages was required to be done here correctly as at times depending on the application size, weird behavior of our MCU was observed. This issue was because of the fact that a some MCUs have different flash erase struct members. There were no banks for the board we were using so the calculation for number of pages to be erased needed modifications.

- One minor error we faced was with setting correct baud rate between the UART peripheral and the host machine. The host machine(Tera Term – Serial Communication terminal) should match with the baud rate set for the UART on board or else the terminals might display incorrect values.

- When using our own build APIs, at times in the main function the MCU was not waiting for user input and was proceeding onto the next line of code. Necessary blocking (or waiting) for a given period of time was required here to receive input and then accordingly act on it.

- At times, the message packets to be transferred for application were getting dropped due to some reason. We fixed this issue by using a CRC checksum for packets which will detect if there is any tampering of data.

- When taking the address for the application to be flashed at from the user, the value of address was not getting passed correctly between multiple functions located in different c files. To correct this a double pointer was used so that each access of the variable will be directly from its value stored in the memory.

- In our implementation we are flashing the application at an address specified by user and then run that application by jumping to that address. Here, one limitation found was that the application cannot be flashed at any location given by user. The application's vector table address should be modified and the offset value in the vector table address must be known to the user to ask the bootloader to flash the application. If the location to be flashed at needs to be changed by the user, then first the vector table offset address must be changed in the application.

- When flashing two applications, the program was able to flash the first app and execute it but after that the bootloader was only able to flash the second application and not able to jump to its address. This issue was solved by using the ST-Link debugger to trace the value of memory address variable and if that was updating for the second application coming. We found that that the second time the address was getting corrupted, so we made sure each time when updating a sector in flash we erased the sector and then did write so that we get the latest value without any tampering to it.

- When updating the firmware each time maintaining a total count of the applications currently present and its locations was difficult. The approach used to solve this was that we did write directly into flash as it is non-volatile and will store the values present even when the board is reset. To correctly write into the flash for keeping track of all applications we carefully studied the HAL APIs and selected an appropriate one suitable for our needs.

REFERENCES

1. F. Baldassari, “From zero to main(): How to write a bootloader from scratch,” Aug 2019. [Online]. Available: https://interrupt.memfault.com/blog/how-to-write-a-bootloader-from-scratch

2. [Online]. Available: https://api.riot-os.org/group__boards__nucleo-f303re.html

3. ViktorXP, “Stm32 tutorial - bootloader that can update and jump to multiple applications,” Oct 2020. [Online]. Available: https://www.youtube.com/watch?v=S0s69xNE1dE

4. Ferenc-Nemeth, “ferenc-nemeth/stm32-bootloader.” [Online]. Available:https://github.com/ferenc-nemeth/stm32-bootloader

5. Niekiran, “niekiran/bootloaderprojectstm32.” [Online]. Available: https://github.com/niekiran/BootloaderProjectSTM32

6. Akospasztor, “akospasztor/stm32-bootloader.” [Online]. Available: https://github.com/akospasztor/stm32-bootloader

7. “Lecture 15: Booting process,” Feb 2017. [Online]. Available:https://www.youtube.com/watch?v=3brOzLJmeek&t=10s