Custom Bootloader for STM32

Summer Internship Report

Submitted in Partial Fulfillment of the Requirements for the Degree

of

Bachelor of Technology

in

Electrical Engineering

Submitted By:

Nistha Kumari (18BEE063)

Under Guidance of:

Prof. Manisha Shah

Department of Electrical Engineering

Institute of Technology, Nirma University,

Ahmedabad – 382481

July - 2021

ACKNOWLEDGEMENT

I must recognise the power, energy, and patience that almighty GOD bestowed upon me in order to begin and complete this job with the help of those involved, a few of whom I am attempting to mention here.

I'd want to convey my heartfelt appreciation to *my Supervisor,* **Prof. Manisha Shah,** *Assistant professor, Electrical Engineering Department* for her invaluable advice and inspiration during my project.

I'd want to convey my deep admiration and thanks to **Prof. (Dr.) S. C. Vora**, *Professor & Head of Electrical Engineering Department* for assisting me and providing the necessary resources for my summer project work.

I'd also want to thank all of my friends who assisted me, either directly or indirectly, in finishing my summer project.

No words can describe how grateful I am to my parents for their blessings and well wishes. I bow with great regard to them.

Nistha Kumari

ABSTRACT

In the actual application of embedded systems in the industry and development standards, with the continuous development of technology and firmware, there is a constant need to update software. We often see some built-in applications with such features. But achieving the same thing requires an understanding of a concept called IAP. Therefore, in this project, we will implement application programming (IAP) on STM32 devices. We have another requirement. In embedded applications, the lower the system response, the better. Therefore, the final algorithm of the system must be optimized for embedded applications. Even after repeatedly optimizing the algorithm, it can cause a delay in the response. Generally, for practical applications, it is necessary to write / read data to / from memory and peripheral devices. In embedded applications, memory read / write consumes a lot of CPU time. That is why the performance of the system DMA controller introduced in the latest generation of microcontrollers must be improved. In this project, we developed a unique peripheral DMA algorithm for it. After DMA is implemented in the main application, the performance is improved to a considerable level. For the same data length, DMA is 60% more efficient than the usual data transfer algorithm.

LIST OF FIGURES

LIST OF TABLES

TABLE OF CONTENTS

CHAPTER 1: Prerequisites

1.1: Boot Modes:

In this project, STM32f407VG discovery board is used which have the following BOOT mode configuration as shown in Table 1.In boards as Arduino UNO, we have bootloader stored in memory that runs when the board is restarted, when we upload a code, the board gets restarted and waits for the command from the programmer and receive the uploaded code in binary and it programs the flash of the MCU and again goes to RESET and gives control to the application. This is called In-Application programming.

Whereas in this discovery board the bootloader doesn't automatically run when the MCU is restarted. Using BOOT pins we can change that to our liking. The debugger part of the board is used to upload the code so the boot options are present to use the already present bootloader which is in SRAM or write a custom bootloader in flash and use x0 in BOOT1 and BOOT 0 pin respectively.

Boot mode selection pins		Boot mode	Aliasing
BOOT1	BOOT0		
x	0	Main Flash memory	Main Flash memory is selected as the boot area
0	1	System memory	System memory is selected as the boot area
1	1	Embedded SRAM	Embedded SRAM is selected as the boot area

Table 1. Boot modes in STM32F407

1.2: Memory Aliasing and Flash:

STM32F407 have the following memory organization:

- ***Flash:*** *non volatile memory(512kB)* stores the application code and the read only data.
- ***SRAM1 : (****112kB)* application global data, static variables.
- ***SRAM2:*** *(16kB)* application global data, static variables.
- ***ROM or System memory:*** *Contain native Bootloader.(30kB)* Read only memory.
- ***OTP memory: (****528kB)*
- ***Option Byte Memory:*** ***(****16 Bytes)*
- ***Backup RAM:*** *(4kB) Backup RAM using Battery.*

The flash memory in the controller is divided into 8 sectors, sector 0- sector 7, we want to store the bootloader in sector 0-sector 1 and the user application in the area sector2 to sector 7. As we know that the vector table is stored at memory location 0x0000 0000 which is aliased with the memory location 0x0800 0000 which is the starting location of the flash memory, but in this case this memory location will contain the vector table of the bootloader code and the vector table of the user application will start from 0x0800 8000 thus we change the memory aliasing to alias 0x0800 8000 and 0x0000 0000 by using VTOR (Vector Table Relocation Register). After the

RESTART the bootloader program will run first and if the user button is not pressed, the bootloader will give control to the user application.by calling the RESET_HANDLER() address of the user application.

Table 5. Flash module organization (STM32F40x and STM32F41x)

Block	Name	Block base addresses	Size
Main memory	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
	Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
	. . .	. . .	. . .
	Sector 11	0x080E 0000 - 0x080F FFFF	128 Kbytes
System memory		0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP area		0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes		0x1FFF C000 - 0x1FFF C00F	16 bytes

Table 2: Memory Organization in STM32F407

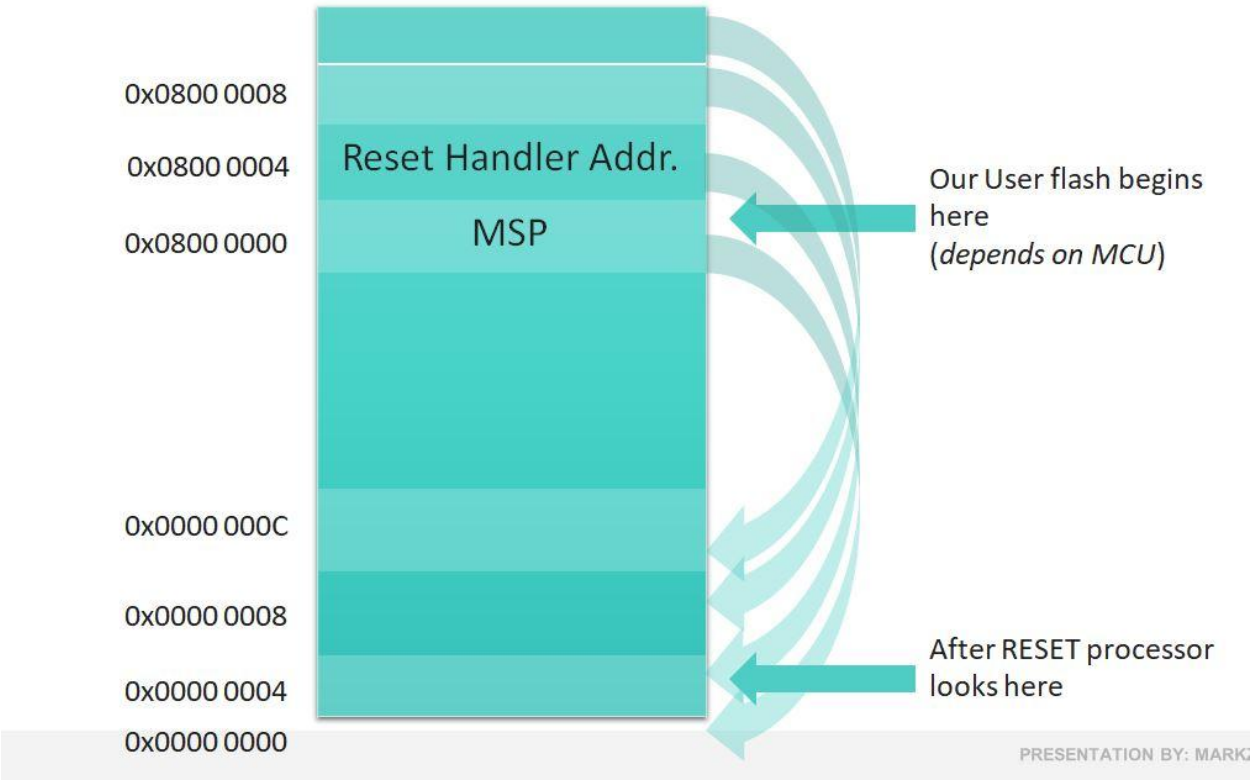

Figure 1. Memory Aliasing

Figure 2. Flash Code Placement

1.3: DMA:

1.3.1: Introduction of DMA:

In a microcontroller the peripheral takes data from the outside environment and then the data from the peripheral is transferred to the CPU and after the necessary computations it goes to the memory. As we know that the processor can only perform one operation at a time, this is the reason we have multi core processors, for the purpose of multitasking, we use schedulers that perform multiple tasks that appear to be acting simultaneously but are performing each task for a specific time and rotating control.

Such a system that uses processors for all the types of data transactions will not be good for practical applications that work in real-time. Thus DMA controller is used to reduce the load on the processor of performing data transfer from various peripherals to the memory. Thus relieving the CPU of the load to perform other operations.

This can overall accelerate the process and result in a more efficient system which is much more preferable for real time applications in industries.

1.3.2: Working of DMA:

When a peripheral wants to transfer data to memory or wants to receive data from memory first, it transmits the data that the peripheral wants to read or write. The read / write request sent from the read / write control line between the CPU and the DMA controller control logic unit.

When the peripheral processor sends the request, it includes the start address for DMA to begin reading or storing data This address is stored in the address register, which is also known as the beginning address register, via DMA. The processor also outputs the number of words, or how many bytes of data the DMA can store in the memory segment, as well as the length of the data, which will be placed in the data count register. Peripheral I/O addresses are saved in the data log for future use.

When a peripheral wishes to transfer data from memory, it sends a DMA request to the DMA controller, which approves the request and then sends a hold request to the CPU, which holds the CPU for many clock cycles. The CPU receives the controller's request, verifies if the bus is free, and sends a hold acknowledgment to the controller.

The DMA controller I / O device is activated after getting the held confirmation and data transfer are carried out, and the DMA assumes control of the system bus to move data to memory. DMA will produce an interrupt to transmit an acknowledgment to the CPU while transferring to memory.

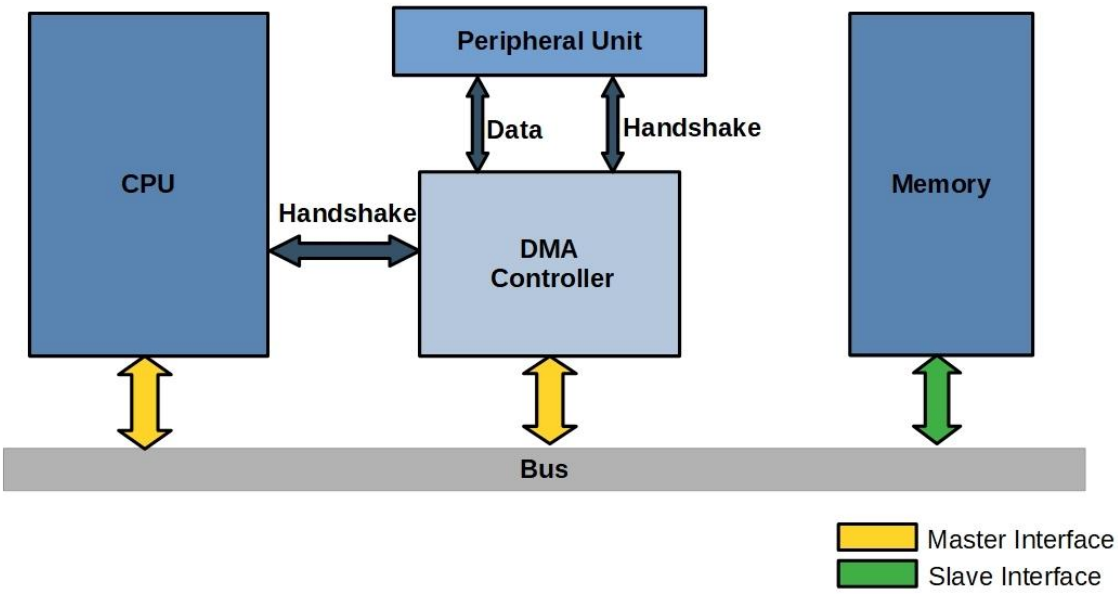

Figure 3. Data Transfer using DMA

CHAPTER 2: Implementation of IAP

2.1: Introduction to In Application Programming:

When implemented in the final product, one of the most essential requirements for most Flash memory-based systems is firmware update capability. This is referred to as In-Application programming (IAP). For example, an application may require the ability to retain calibration information or to download updated code parts. The end-capacity user's to delete and programme code memory application is “In-Application Programming” (IAP).

This functionality is provided by the majority of modern microcontrollers in the form of hardware circuits composed of this logic, which essentially performs IAP. However, older microcontrollers did not provide the same capability. Microcontrollers nowadays require user-specific firmware to conduct IAP on flash-memory-based microcontrollers. This functionality primarily makes advantage of the communication interface provided by the microcontroller and transfers the updated firmware data.

2.2: Principle of the project:

If a restart occurs in the microcontroller, the PC is configured to execute the IAP program. It is a code that checks a particular condition, for example, the mixing of the input button or the key is narrowed. When this condition is valid, the IAP code runs part of the code that updates the user application or directly (of course, of course, of course) this user application. Both user applications and IAP controllers are separated into different memory areas. A configuration is to place the user code for the next free flash memory square, area or page that starts the IAP controller code to start the memory of the program, which causes the safety of the autonomous memory, it is possible to design in two areas.

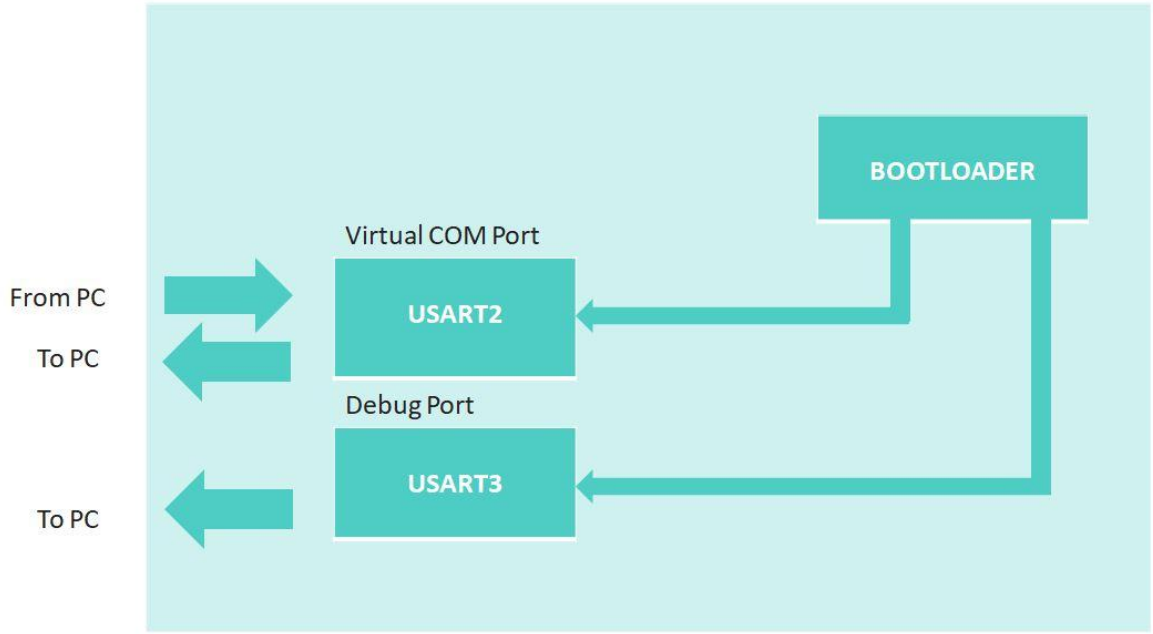

Figure 4. Python Program – Boot-loader Communication

2.3: Host-Bootloader Communication:

2.3.1:Jumping to user application

1. Declare a function pointer for the user request given the control.
2. Since the first value knows that the pointer value of the main battery holds, use CMSI to read content to set MSP values.
3. The following address of the first aggregate = 4 has an address of a user application (reset of the controller). Next, initialize the functional pointer of the user's application by reading that value and jumping to that resettleter
4. of the user application.
5. Change the value of the start file.

2.3.2:Bootloader read mode

1. We only read one byte, and it tells us the number of bytes remaining, because all different commands have different structures.
2. The second byte is always the command code. Based on the command code used by the switch instruction, we decode the sent command. We call different functions for different commands and pass the buffer as a parameter.
3. During processing, we first check the CRC and ACK the host if it is good.
4. If it is bad, it starts polling the next byte from the control function return.
5. If the crisis is good, send an ACK, we receive a response from HOST, then send a response to HOST, and then return to poll the next command.
6. In order to check the CRC, we have a function. If the function returns a non-zero value, it means that the CRC check fails and the bootloader must send a NACK to HOST. We also have a function.
7. If the CRC check passes, we send an ACK and get a response.
8. Then we send the response to HOST.
9. CRC verification process operation.
10. Get the bootloader version.

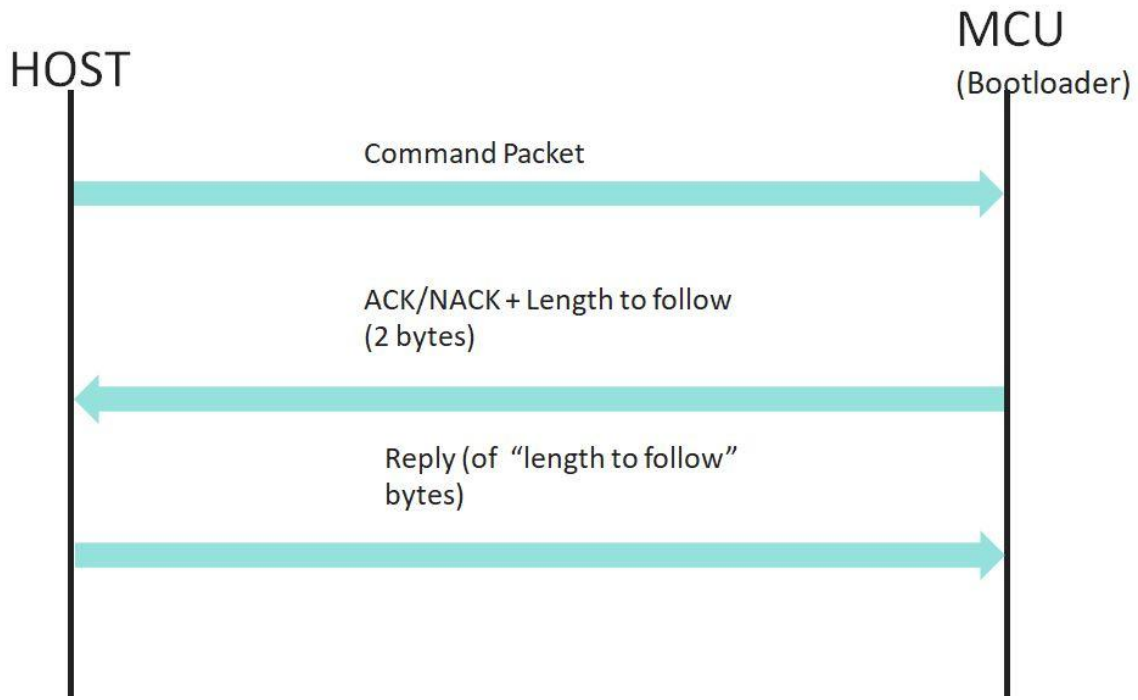

Figure 5. Host Bootloader communication

2.4: IAP Driver Functioning:

The IAP controller must be programmed through the JTAG / SWD interface, starting from the base address of the flash memory. This is done using the selected development toolchain or the factory integrated boot loader located in the system memory area. The IAP

controller contains the following source files:

• main.h - Sets USART initialization and RCC configuration. Then run the main menu from the menu.c. file.

• menu.c: contains the routines of the main menu. The main menu provides options to download a new binary file, load the internal flash memory, run the loaded binary file, and manage the write protection of the page where the user uploads the binary file.

• flash_if.c-Contains write, erase and write protection settings for internal flash memory functions.

• common.c:. Contains functions related to reading/writing USART peripherals from ./.

• STM32Cube hardware abstraction abstraction layer file.

The user selects to start the application or run the IAP code for reprogramming by pressing the button connected to the pin:

• If there is a user application in the memory, it will switch to the user application when restarting without pressing the button .

• Pressing the button during reset will display the IAP main menu. The

IAP controller uses the USART to:

• Load the contents of the internal STM32 flash memory into a binary file.

• Download the binary coded file from the terminal emulator to the internal flash memory of the STM32. The

IAP application flowchart is shown in Figure 5.

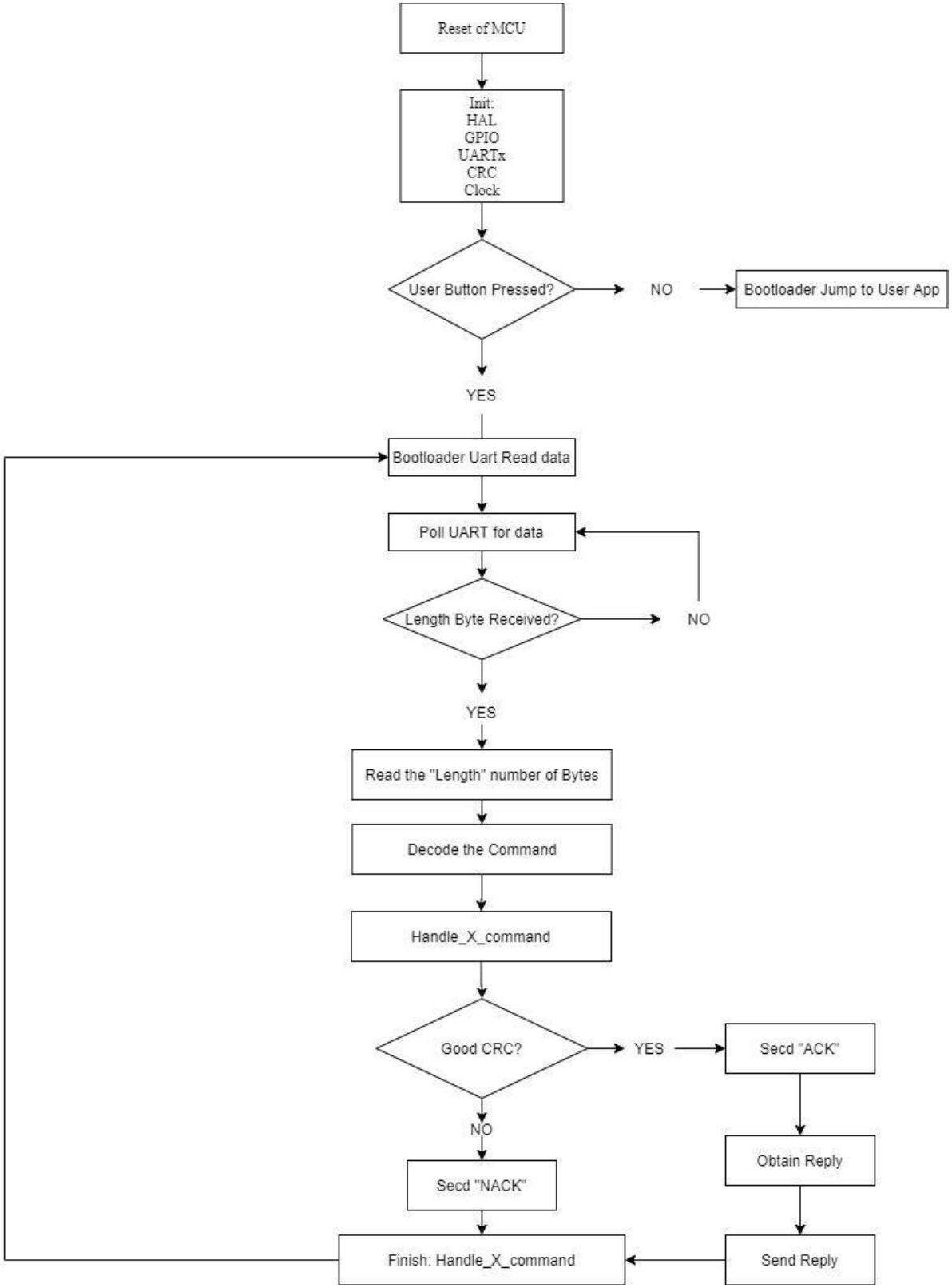

Figure 6. Flowchart of the Project

CHAPTER 3: CONCLUSION AND FUTURE SCOPE

From the previous chapters, we can say that IAP is nothing more than a small piece of code stored in the flash memory or ROM of the MCU, acting as an application loader and a mechanism to update the application when necessary. In addition, DMA is a circuit within the device that acts as a host and controls the bus to perform data transfer by reducing the load on the microprocessor.

Most microcontrollers today have this IAP (boot loader), which is provided by the chip supplier and stored in system memory. In addition, with the increase of microcontroller features and functions, the DMA master channel is also increasing.

Finally, IAP and DMA are two concepts that are practically applied in the embedded world. Both are useful in real-time systems. Therefore, these two concepts are very suitable for the latest equipment.

REFERENCES

[1]https://en.wikipedia.org/wiki/Direct_memory_access

[2]https://www.nxp.com/docs/en/application-note/AN461.pdf

[3]https://www.st.com/resource/en/application_note/dm00161366-stm32-inapplication-programming-iap-using-the-usart-stmicroelectronics.pdf

[4]https://www.udemy.com/course/stm32f4-arm-cortex-mx-custom-bootloader-development/

[5]http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka11265.html

[6]https://www.udemy.com/course/microcontroller-dma-programming-fundamentals-to-advanced/

[7]https://deepbluembedded.com/stm32-dma-tutorial-using-direct-memory-access-dma-in-stm32/

[8]https://www.st.com/resource/en/application_note/dm00161366-stm32-inapplication-programming-iap-using-the-usart-stmicroelectronics.pdf

[9]https://www.st.com/content/ccc/resource/technical/document/application_note/39/73/79/e7/27/87/40/a3/DM00036049.pdf/files/DM00036049.pdf/jcr:content/translations/en.DM00036049.pdf

[10]https://wenku.baidu.com/view/54bafa1c4028915f814dc25a.html?re=view

[11]https://max.book118.com/html/2019/0430/8053023052002021.shtm

[12]https://www.expatriate.cc/push-on-terminal-approved-by-the-user-factory_9902/

[13]https://deepbluembedded.com/stm32-dma-tutorial-using-direct-memory-access-dma-in-stm32/

[14]https://www.st.com/resource/en/application_note/dm00161366-stm32-inapplication-programming-iap-using-the-usart-stmicroelectronics.pdf

[15]https://open4tech.com/direct-memory-access-dma-in-embedded-systems/