CSE-3211: Operating Systems Lab

Assignment 02: Design and Implement Boot Loader Program for DUOS

Submitted By:

Ahaj Mahhin (Roll - 01)

Md. Mahmudul Hasan (Roll -10)

Abrar Eyasir (Roll - 12)

Submitted to:

Dr. Mosaddek Tushar, Professor

Computer Science and Engineering, University of Dhaka,

Submission Date:

18 November 2024

Bootloader Documentation

How it Operates

Bootloader Initialization & Switching

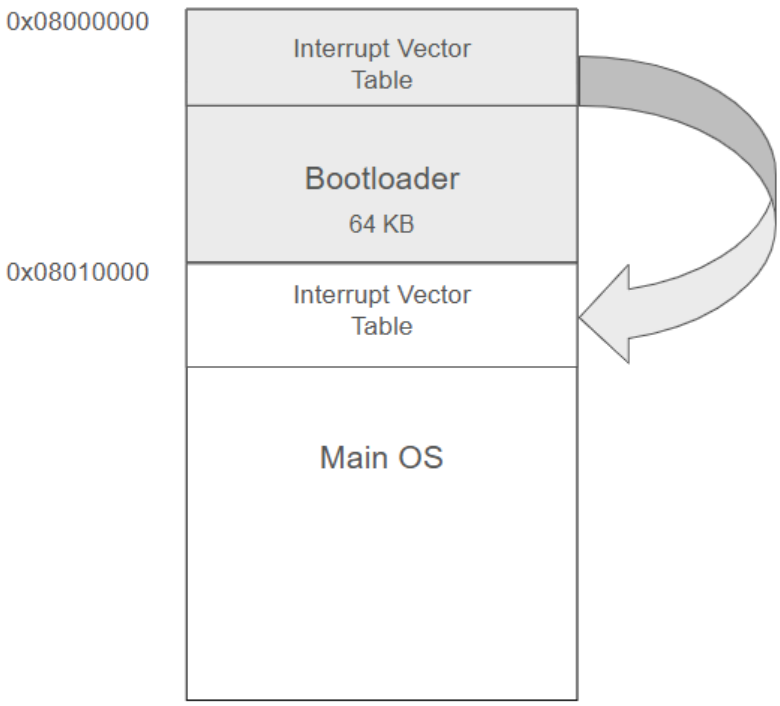
The bootloader is the first piece of code executed by the operating system. It resides in the Flash Memory, starting at address `0x08000000`. At the beginning of the Flash Memory, the **Interrupt Vector Table** is located.

The first bytes of this table contain the address of the **Main Stack Pointer**, followed by 4 bytes that store the address of the **Reset Vector**. The Reset Vector is the initial code executed on an STM32 microcontroller. This function is used to transfer control of the MCU from the bootloader to the operating system by jumping to the OS's starting address.

The starting address of the operating system is located immediately after the end of the bootloader in the Flash Memory.

Flash Program Memory (512KiB)

Bootloader (64KiB) Address: **0x0800_0000 to 0x0801_0000**
Main Firmware (448KiB) Address: **0x0801_0000 to 0x0880_0000**



Adjust the sizes of the Bootloader and Main Firmware according to your system's requirements. Note that the Bootloader file may exceed 64 KiB, as the binary file configuration can vary depending on your operating system. For example, Linux and macOS may compile the binary file differently.

Setting the Vector Table Offset Register (SCB->VTOR)

Before transferring control from the bootloader to the operating system (OS), it is essential to configure the **Vector Table Offset Register (SCB->VTOR)**. This step ensures that the Interrupt Vector Table points to the correct memory location for the OS.

The vector table address is updated as follows:

```
SCB->VTOR = BOOTLOADER_SIZE;
```

`BOOTLOADER_SIZE` represents the memory offset where the OS starts, immediately following the bootloader. This configuration redirects the processor's interrupt handling to the OS's vector table. Without this adjustment, the system could continue to use the bootloader's interrupt table, leading to undefined behavior.

UART Communication

UART communication protocol was used to establish communication between the Bootloader and ***The SERVER*** (essentially a python code that provides OS update information and sends the OS as chunks of binary bits). At first, the bootloader asks the SERVER for the latest OS version and matches it with the current OS version. It requests the SERVER to send the OS executable as a stream of binary bits and stores it in an array. The Bootloader accepts a chunk of bytes and sends an acknowledgement along with a CRC value calculated from that chunk. If the CRC value matches that at the SERVER side, the bits are received by the Bootoader and the SERVER prompts to send the next chunk.

As all the bytes of the have been received , the Bootloader cuts off the communication and writes the data bytes in the ***FLASH*** at the appropriate location. The Bootloader’s Reset Vector function calls the OS’s Reset Vector to allow it to switch and start running.

Flash Memory Management

1) Unlocking and Locking Flash Memory

2) Erasing Flash Memory for the OS
   Memory sectors erased:

- Sector 4: `0x0801_0000` to `0x0801_FFFF` (64 KB)
- Sector 5: `0x0802_0000` to `0x0803_FFFF` (128 KB)
- Sector 6: `0x0804_0000` to `0x0805_FFFF` (128 KB)
- Sector 7: `0x0806_0000` to `0x0807_FFFF` (128 KB)

The important part to notice is that we only delete the memory from Sector 4 to Sector 7 because our Bootloader size is 64 kB. So, our main application starts from the `0x0801_0000` address. If you change the Bootloader size, you have to adjust this part. We can adjust the part according to the picture given below.

Table 4. Flash module organization

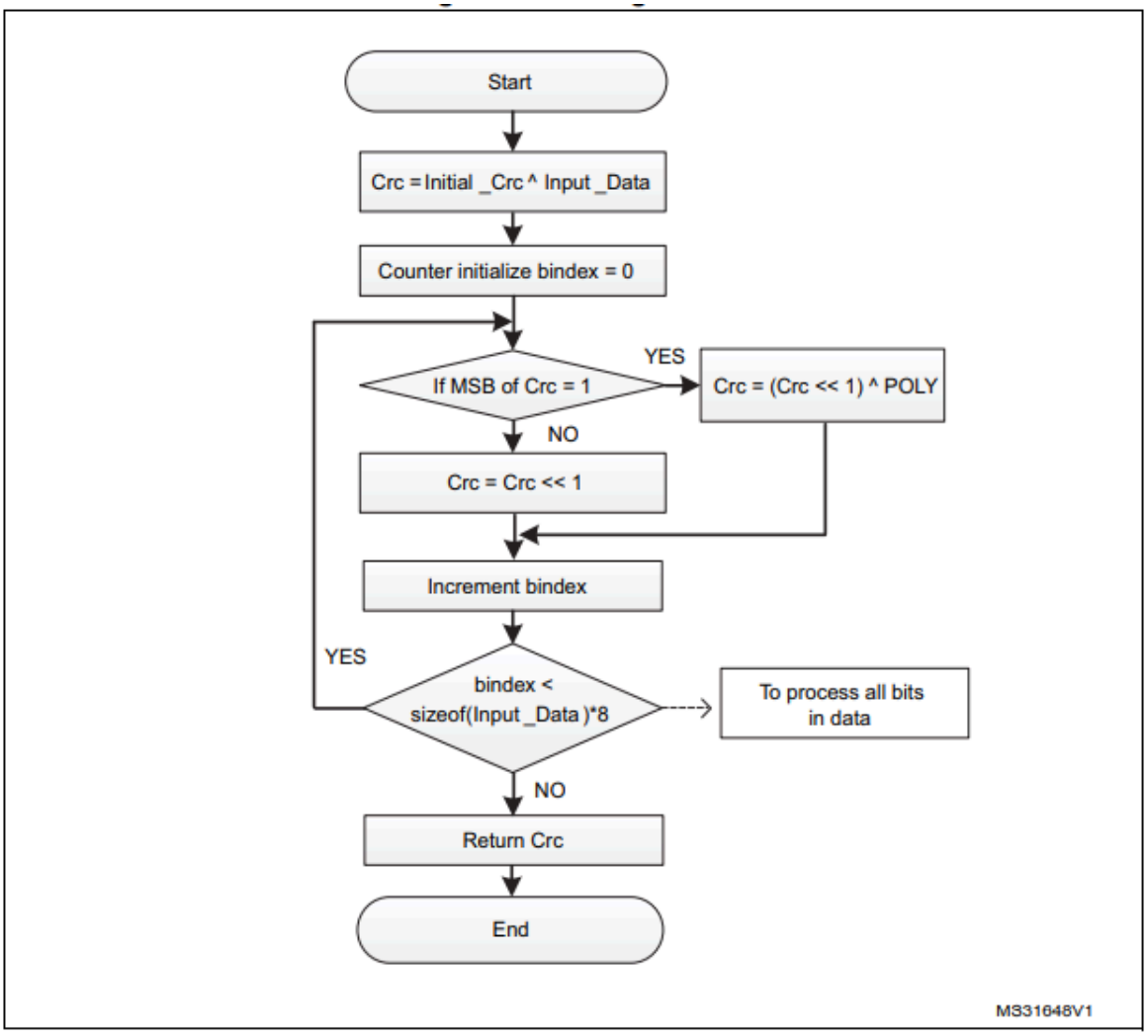
Block	Name	Block base addresses	Size
Main memory	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
	Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
	Sector 7	0x0806 0000 - 0x0807 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

CRC Calculation

CRC values are calculated using the —- Ethernet polynomial. It uses a division mechanism with XOR-ing technique on the data bytes written to the CRC_DR register. Here's a flowchart to demonstrate how the algorithm works:

This function, `CRC_Config`, is a typical setup function to configure and initialize the CRC (Cyclic Redundancy Check) calculation unit in an embedded system (STM32F44RE)

```
void CRC_Config(void)
{
    RCC->AHB1ENR |= RCC_AHB1ENR_CRCEN;
     CRC->CR = CRC_CR_RESET; // reset the CRC calculation
unit to        0xFFFF FFFF
}
```



The input parameters are:
• the dividend, also called input data, abbreviated to *"Input_Data."*
• the divisor: a generator polynomial, abbreviated to *"POLY."*
• an initial CRC value: abbreviated to *"Initial_Crc"*

An example is given below:

bindex	Execution step	Binary format	Hex
	Crc = *Initial_Crc* ^ *Input_Data*	1 1 1 1 1 1 1 1 (*Initial_Crc*)	0xFF
		^ 1 1 0 0 0 0 0 1 (*Input_Data*)	0xC1
0	Crc << 1	[0] 0 1 1 1 1 1 0	0x3E
1	Crc << 1	[0] 1 1 1 1 1 0 0	0x7C
2	Crc << 1	[1] 1 1 1 1 0 0 0	0xF8
		[1] 1 1 1 0 0 0 0	0xF0
	Crc = Crc ^ POLY	^ 1 1 0 0 1 0 1 1 (*POLY*)	0xCB
3	Crc << 1	[0] 0 1 1 1 0 1 1	0x3B
4	Crc << 1	0 1 1 1 0 1 1 0	0x76
5	Crc << 1	[1] 1 1 0 1 1 0 0	0xEC
		[1] 1 0 1 1 0 0 0	0xD8
	Crc = Crc ^ POLY	^ 1 1 0 0 1 0 1 1 (*POLY*)	0xCB
6	Crc << 1	[0] 0 0 1 0 0 1 1	0x13
7	Crc << 1	[0] 0 1 0 0 1 1 0	0x26
	Crc (Returned value)	0 1 0 0 1 1 0 0	0x4C

Terminal Command

make clean: Cleans up the project directory by removing files generated during previous builds.

make all: Compiles the entire project and builds all required targets. Runs the build process as defined in the **Makefile.**

make load: Loads the compiled program onto a target device (STM32F44RE)

How to Use

1. Download the zip file from this link : [-----] and unzip it in a preferred location.
2. Go to the Bootloader compile folder and open a terminal and then run the following command:

   make clean && make all && make load

3. Open another terminal in the directory where du_store.py is located. Run the python script i.e, the DU_SERVER with the following command:

   python du_store.py

4. Push the reset button in the MCU if it the update operation does not start instantly.

Important things to consider

1) In python du_store.py, /dev/cu.usbmodem11303 needs to be changed according to your laptop port.

```
serial_instance = serial.Serial(port="/dev/cu.usbmodem11303", baudrate=115200)
```

To find the appropriate device path (like `/dev/cu.usbmodem11303`) for serial communication you can use the following commands:

Mac command : **ls /dev/cu.***

```
Last login: Mon Nov 18 21:17:20 on console
(base) eyasir2047@Abrars-MacBook-Pro ~ %  ls /dev/cu.*

/dev/cu.Bluetooth-Incoming-Port /dev/cu.usbmodem11303
/dev/cu.QCY-T5
(base) eyasir2047@Abrars-MacBook-Pro ~ %
```

This will list all connected serial devices. Look for something like `/dev/cu.usbmodemXXXX` or `/dev/cu.usbserial-XXXX`.

On Linux: **ls /dev/ttyUSB* /dev/ttyACM***

This lists USB serial devices and Arduino-like devices. Look for `/dev/ttyUSB0` or `/dev/ttyACM0`.

2) Using OpenOCD to Load Firmware

The `Makefile` contains commands for loading firmware onto the STM32F4 Discovery board using OpenOCD. Depending on your operating system, use the appropriate command to ensure compatibility with your system's file paths.

For macOS:

Use the following command:

```
load:

   openocd -f /opt/homebrew/share/openocd/scripts/board/stm32f4discovery.cfg -f
/opt/homebrew/share/openocd/scripts/interface/stlink.cfg       -c       "program
target/duos verify reset exit"
```

For Linux:

Use the following command:

```
load:
```

```
   openocd -f /usr/share/openocd/scripts/board/stm32f4discovery.cfg \
-f   /usr/share/openocd/scripts/interface/stlink.cfg   \-c   "program
target/duos verify reset exit"
```

On **macOS**, OpenOCD is typically installed via Homebrew, so its scripts are located under `/opt/homebrew/share/openocd/scripts/`.

On **Linux**, OpenOCD scripts are usually found under `/usr/share/openocd/scripts/`.