ARM Cortex M Processor Custom Boot Loader

Chethan S K

Embedded Software Engineer, Sigma Microsystems India Pvt Ltd

Abstract: ***The developer faces the problem when they have two different applications (means two different source code) and they want to merge both of them to one application. At that point, the developer can place both the application separately in controller internal flash along with the custom boot loader on top of the applications. On power ON using the custom boot loader, the user can decide which application has to run. In this literature, I have used the NUCLEO F446RE STM development board.***

Keywords: Custom boot loader, STM32F446RE, Booting sequence, keil uVision5

1. Introduction

Over the years, The Embedded systems are playing a keen role in modernizing human life. Most of the human day to day work is automated using embedded systems. The embedded systems are existing in all kinds of industries like Automobile, Telecommunication, Medical, Agriculture, etc. Long years back Embedded systems were used to perform specific dedicated task but as time flies the embedded systems are also used for general applications. Example mobile phones, previously mobile phones were used only for phone calls, nowadays can do much more than phone calls. Microcontroller is a main working unit in Embedded systems which handles all the external events, process the event and reply to the event.

Booting is one of the first important process for the microcontroller. The booting sequence guides the microcontroller, that from where it has to read application code, where it has to write the application code and how the controller has to jump to the application part.

This literature explains :
1) The ARM booting sequence.
2) Memory architecture and memory aliasing.
3) How to load boot code along with two different application code into the internal flash.
4) How boot code decides to switch between the applications on power ON of the microcontroller.

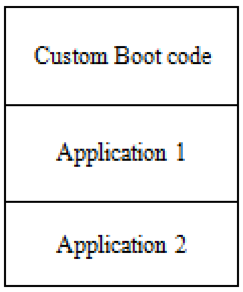


Figure 1: Block organization of internal flash

2. Arm Cortex M Booting Sequence

What happens when we reset the processor, the address pointer of the processor always starts with zero i.e. 0x00000000. The reset interrupt is the highest priority interrupt as per the vector table. Vector table provides the information about the initial stack pointer value and various exceptional handler addresses.

The sequence followed upon the reset of the ARM cortex M processor:
1) After reset, PC (Program counter) is loaded with the address 0x00000000.
2) The processor fetches the value at 0x00000000 into the MSP(Main Stack Pointer). The processor basically first initializes the main stack pointer.
3) The processor reads the address of the reset handler from the location 0x00000004 into the program counter. The reset handler is a normal function written in assembly/C language, which is get called whenever the processor resets. The reset handler code doesinitial device-specific initialization such as configuration clock, configuration hardware block, re-initializing the stack space, before calling the main function of the application source code.
4) The processor jumps to the updated reset handler address and start executing the first instruction written over there.
5) Finally, the reset handler function calls the main function. From there the processor executes all the user instructions.

3. Memory architecture and memory aliasing

To understand the concept of memory architecture, this literature consider the example of the microcontroller STM32F446RE. The understandability of memory architecture of microcontroller is very important for the developer because he has to be sure about the memory section where he is going to place the application code.
STM32F446RE memory architecture :
1) Internal flash memory – 512 KB
2) Internal SRAM1 and SRAM2 – 112KB and 16KB
3) ROM – 30KB
4) OTP memory – 528 bytes
5) Option byes memory – 16bytes
6) Backup RAM – 4KB

The ARM processor has different booting mode, in which booting from internal flash is also one of the option. The internal flash memory begins at 0x08000000 and ends at 0x0807FFFF, total 7 sectors each of 16Kbytes. This flash has main memory, System memory, OTP area and Option bytes. The main memory has seven sectors which all are used to store the user application code.

Now, let understand the concept of memory aliasing. Upon controller reset, the controller starts read data from the base address 0x00000000. But the developer has placed the code from the start location 0x08000000. So, to overcome this problem the chip manufactures use the concept called memory aliasing. Aliasing means the particular locations can be mapped to some the other locations.

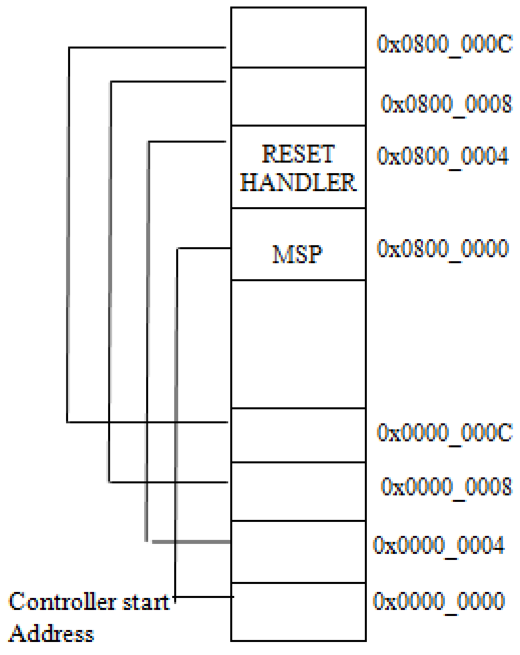


Figure 2: Memory aliasing

3.1 Flashing three different code to internal flash

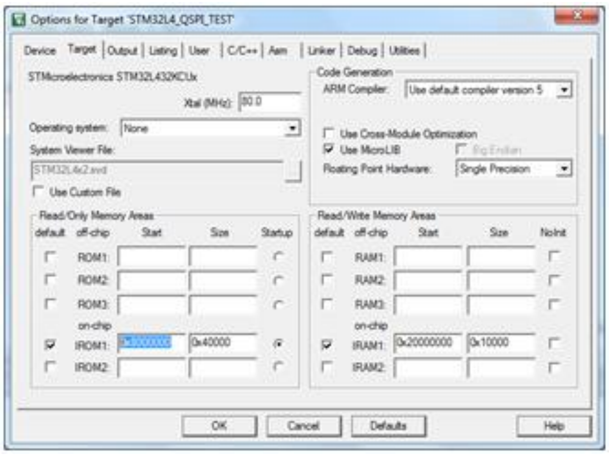
Suppose, Developer has to flash three different codes to internal flash at different sectors. The prerequisite to do so:

1) Size of the application code and sector start address.
Developer has to verify the code size, by checking the code raw binary file size. Based on the sizes of the code, the developer has to decide which sectors of the flash to be used to store the application code. suppose, code sizes are 10Kbytes each, then developer can choose:

- 0x08000000(sector 0 base address) for custom boot-loader code.
- 0x08004000(sector 1 base address) for application 1 code.
- 0x08008000(sector 2 base address) for application 2 code.

2) KeiluVision settings
There are three major settings to be notices in order to flash the code to different locations.

- Sector start address where the code to be flashed.


Options for Target 'STM32L4_QSPI_TEST'

STMicroelectronics STM32L432KCUx
Xtal (MHz): 80.0
Operating system: None
System Viewer File: STM32L4x2.svd
Use Custom File
Code Generation
ARM Compiler: Use default compiler version 5
Use Cross-Module Optimization
✓ Use MicroLIB
Big Endian
Floating Point Hardware: Single Precision

Read/Only Memory Areas

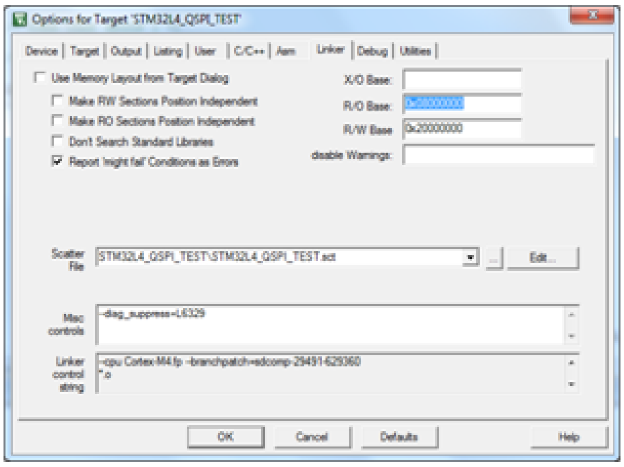
default	off-chip	Start	Size	Startup
	ROM1:			
	ROM2:			
	ROM3:			
	on-chip			
✓	IROM1:	[illegible]	0x40000	●
	IROM2:			

Read/Write Memory Areas

default	off-chip	Start	Size	NoInit
	RAM1:			
	RAM2:			
	RAM3:			
	on-chip			
✓	IRAM1:	0x20000000	0x10000	
	IRAM2:			

Figure 3: Writing sector start address

- Sector address change in linker setting


Options for Target 'STM32L4_QSPI_TEST'

Use Memory Layout from Target Dialog
Make RW Sections Position Independent
Make RO Sections Position Independent
Don't Search Standard Libraries
✓ Report 'might fail' Conditions as Errors
X/O Base:
R/O Base: [illegible]
R/W Base: 0x20000000
disable Warnings:
Scatter File: STM32L4_QSPI_TEST\STM32L4_QSPI_TEST.sct
Misc controls: --diag_suppress=L6329
Linker control string: --cpu Cortex-M4.fp --branchpatch=sdcomp-29491-629360
*.o

Figure 4: Linker setting

3) Vector table offset setting
The reset is the highest priority interrupt for the controller. The reset handler ISR is either written in assembly or C language. Whenever the controller gets reset, the controller calls the reset handler ISR. The reset handler ISR performs some system initializationslike clock settings, clear all reset and setting vector offset register.

The vector table offset register has to be updated as per the start address of the sector, where the code is going to be placed. we can find this in controller startup code. As per the cortex M processor design, the processor has some exceptional handlers like Initial MSP value which is the first handler in the vector table and reset handler vector is the second in vector table. Based on this, the controller initialize the stack area for very first time. Then controller reads the reset vector value. Based on the reset vector value controller jumps to that particular location and from there after developer application instructions start execute.

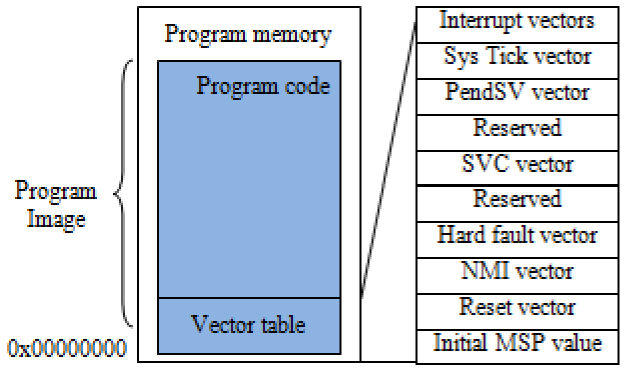


Figure 5: Vector table in a program image

The reset sequence of the ARM cortex M processor is shown below.

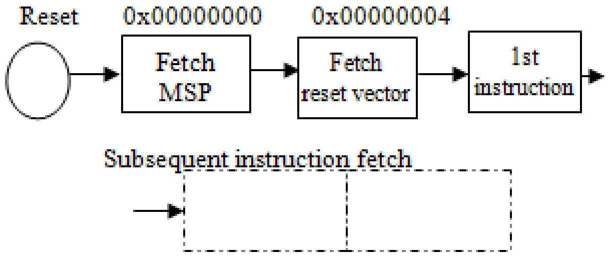


Figure 6: Reset Sequence

Custom boot-loader flow chart and algorithm

- **Flow chart**

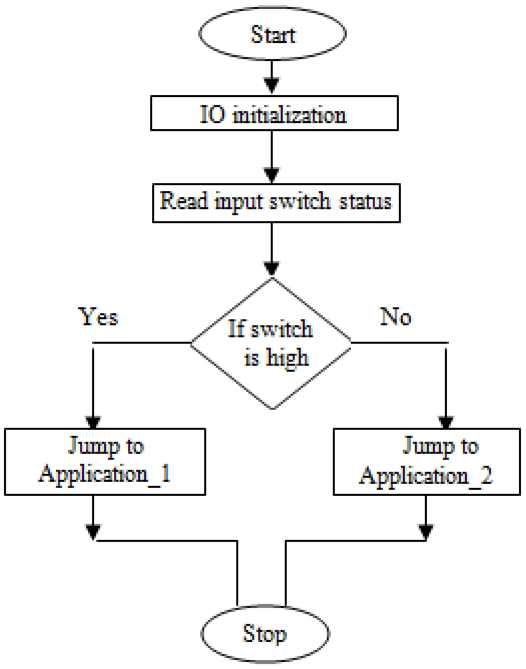


Figure7: Flow Chart

- **Algorithm for custom boot-loader**

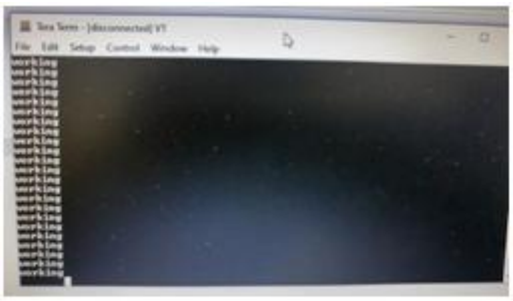
The below algorithm is explained with reference to the flash sector address which are mentioned in the section IV.

Step 1: START
Step 2: System and I/O initialization
Step 3: Read switch input
Step 4: If switch input is LOW call application_1 function
Step 5: In application_1 function set application 1 MSP value, which is located at the address 0x08004000.
Step 6: Call the reset handler address of the application 1, which is stored at the address 0x08004004 based on which controller jumps to application 1code section.
Step 7: If switch is HIGH call application_2 function
Step 8: In application_2 function set application 2 MSP value, which is located at the address 0x08008000.
Step 9: Call the reset handler address of the application 2, which is stored at the address 0x08008004 based on which controller jumps to application 2 code section.
Step 10: STOP

4. Conclusion

Storing two different code to the controller internal flash and booting the code from the custom boot-loader, reduces the complexity of merging two different codes to a single code.

I have considered the application 1 to print some message over serial port. The below shown figure is an output of the application 1, which is booted when the user switch is not pressed under controller reset.


Tera Term - [disconnected] VT
File Edit Setup Control Window Help

```
working
working
working
working
working
working
working
working
working
working
working
working
working
working
working
working
working
working
working
working
working
working
working
working
```

Figure 8: Application 1 output

controller resets. The below shown figure is the output of the application 2. In application 2 I have I am just glowing the user LED, where this application 2 is booted when the user switch is pressed under controller reset.

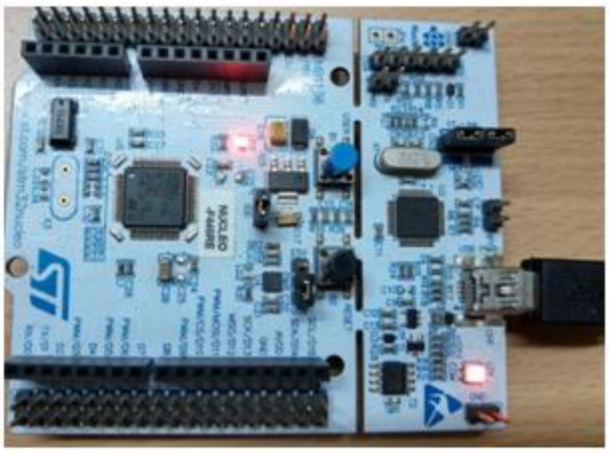


Figure 9: Application 2 output

References

[1] Joseph Yiu, "The definitive guide to the ARM cortex-M0(book style with paper title and editor), " 1st edition, 2011.
[2] Byte Craft, “First Steps with Embedded Systems”, ByteCraft Limited, first edition.

[3] Andrew Sloss, Dominic Symes and Chris Wright “ARM System Developers Guide – Designing and Optimizing System Software” , published by ELSEVIER, 2009.
[4] Michael Pont, “Embedded C”, Pearson Publication, 2002.
[5] Ted Van Sickle, “Programming Microcontrollers in C”, LLH Technology Publishing, 2001.
[6] Michael Barr, “Programming Embedded Systems in C and C++”, Oreilly Publication, edition January 1999.
[7] Intel. Intel hexadecimal object file format specification, Revision A, 1/6/88, 1988.
[8] Raj Kamal, “Microcontrollers: Architecture, Programming, Interfacing and System Design”, Fourth Edition, Pearson Education.