ACUBESAT

FLASH Memory Operations Report
November 17, 2018
Dimitrios Stoupis

Contents

1 Summary **2**

2 FLASH Memory Overview **2**

2.1 Memory Structure 2

2.2 Programming Specifics 3

3 Complete Tasks **4**

3.1 Overview of Flash Writing 4

3.2 Software 4

3.2.1 Functionality Description 4

3.2.2 Resetting Implementation 5

4 Bugs Encountered **6**

4.1 The Hidden Treasure 6

4.2 No Fault, But No Write Too 6

4.3 Wrote Successfully, But Not All Data 6

5 Incomplete Tasks **7**

6 Conclusion **7**

Acronyms **7**

1 Summary

Memory operations, reading and writing, are a crucial part of a remote embedded system and especially if that system resides in space, where some interesting phenomena can occur, like bit flip from high amounts of surrounding radiation. One very important type of memory is the FLASH memory, which contains the copy of the executable code, and operations on that memory are important for on the fly code fixes and validations. In this report the structure of the FLASH memory for *STM32F103* is presented, along with the detailed explanation of the completed code tasks, regarding our CubeSat mock-up. Finally, the bugs encountered during the development time are listed and discussed.

2 FLASH Memory Overview

Recomended file(s): PM0075-(STM32F10xxx_Flash_Manual).pdf
RM0008-(STM32F103xx_Reference_Manual).pdf

The FLASH memory holds a copy of the compiled code and any mistakes on writing operations on the memory could result in a permanent fault of the application, which is only recovered by re-uploading the compiled code. The memory's structure differs between MicroController Unit (MCU), even those in the same family, and the reason is mostly the density of the device, with the higher density devices having wider FLASH memory pages. A detailed overview of the memory's structure is provided on the PM0075 PDF file given above.

2.1 Memory Structure

Generally regardless the MCU, the FLASH memory is divided into pages and some MCUs also have banks, which is something not considered in this report. In our case, medium density MCU, FLASH memory is divided into **128 pages**, each **1KByte** in length. The so called **base address** of the FLASH memory is the **0x0800 0000** in HEX and it is the first memory address used by FLASH. An overview of the FLASH pages is provided in the image below.

Table 5. Flash module organization (medium-density devices)

Block	Name	Base addresses	Size (bytes)
Main memory	Page 0	0x0800 0000 - 0x0800 03FF	1 Kbyte
	Page 1	0x0800 0400 - 0x0800 07FF	1 Kbyte
	Page 2	0x0800 0800 - 0x0800 0BFF	1 Kbyte
	Page 3	0x0800 0C00 - 0x0800 0FFF	1 Kbyte
	Page 4	0x0800 1000 - 0x0800 13FF	1 Kbyte
	. . .	. . .	. . .
	Page 127	0x0801 FC00 - 0x0801 FFFF	1 Kbyte

Figure 1: Flash memory page organization *(image taken from Table 5 of the RM0008 manual)*

2.2 Programming Specifics

Read access in the FLASH memory, as per the *RM0008* manual is done through normal pointers, by pointing to the desired address and then dereferencing to read the stored data, with an example code being like `*(uint16_t *)ulAddress`, `ulAddress` is the desired FLASH memory address. The reason that the previous pointer is cast to 16-bit int, comes from the fact that FLASH values are stored in half-word fashion, which is equal to 16-bits.

The previous statement brings us to the next section of the programming discussion and that is that the FLASH memory is programmed in a half-word fashion, meaning 16-bits at a time. A **very important** remark here is that while reading is allowed in any address, both aligned and not, writing is **ONLY** allowed in aligned addresses, which means address ending from **0x0** and incrementing by 2, so **0x00**, **0x02**, **0x04** ... **0x0E**. Failure to comply with the forementioned rule, will result in hardware fault error, causing the application to crash.

3 Complete Tasks

Operations on FLASH memory, both reading and writing, are accomplished using the code from files `flashOps.c` and `flashOps.h`, which are the core for operating on FLASH. Below, along with the completed task details, there is an explanation on how the FLASH writing is accomplished.

3.1 Overview of Flash Writing

The logic of the FLASH programming is **"write once"**, which simply means, first erase and then write. This implies that a page containing the desired write address, must first be deleted, in order to be writable. The erase and programming procedures are very well outlined in the **PM0075** manual, on pages **13** and **15**.
So the FLASH memory writing operation is as follows:

1. Read the contents of the page containing the desired writing address
2. Save the read contents in an array
3. Unlock the FLASH, before any writing or erasing operations is initiated
4. Erase the page containing the desired writing address
5. Reprogram the original values in other addresses and program the new value in the desired address
6. Check if everything was programmed correctly by comparing the values before and after
7. Lock the FLASH again

In the case of this report's code, the crucial unlocking part is accomplished through calling the `HAL_FLASH_Unlock();` function, which takes care of the procedure described in the **PM0075** manual.

An important note is that all the operations performed on the FLASH memory in our code, are performed under a **critical section** of the FreeRTOS, to avoid any task switching, resulting in application failure or permanent crash.

3.2 Software

Memory read and write operations were implemented as described above and code implementations are found in the files `flashOps.c` and `flashOps.h`.

3.2.1 Functionality Description

Currently the program takes the **value change** command through **NRF24** communication with the ground station, sending the desired value to be programmed in the FLASH. For the moment though, there is are hard-coded values and pre-specified FLASH memory areas to be changed upon the command reception and that is done for demonstration purposes. The idea remains the same, even when sending custom payload to be written to memory. A sample screenshot depicting the successful value

change is shown below.

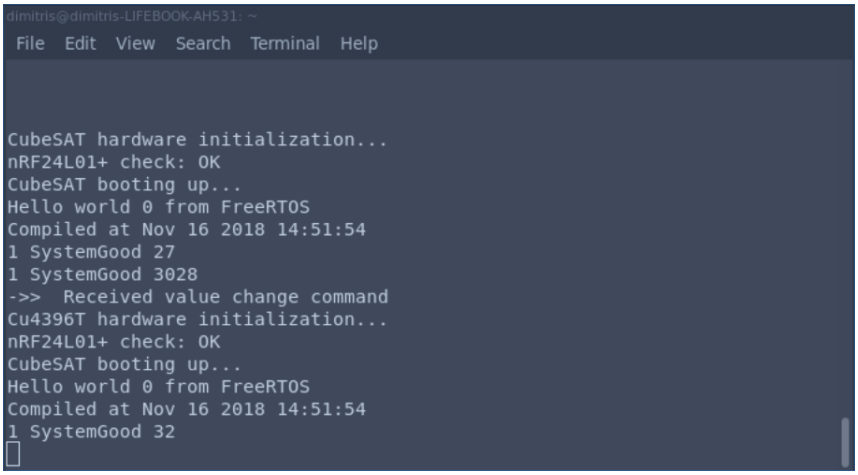

Figure 2: Value change success screenshot

For the above result, the memory address of the string *"CubeSAT hardware initialization…"*, was taken from the list debug file and two consecutive address were written, so the values were changed accordingly.

One important remark that should not go unmentioned is the order in which the 16-bits are written in the flash memory. As we know American Standard Code for Information Interchange (ASCII) characters are 8-bit characters, so in a single memory address *(16-bits length)*, two ASCII characters are written, so the value of one ASCII characters is shifted left and ORED with the value of the other ASCII value. In the case of our MCU, the value that appears second is the one that has to be shifted left. In other words as we can see in the example above **"CubeSAT"**, changed to **"Cu43..."**, because the value **0x3334** was programmed in the corresponding address, which corresponds to ASCII **0x33** shifted left *(or number 3)* and **0x34 (or number 4)** ORED with the first shifted character. So to sum up, we can see that the storage of the values in the FLASH memory is inverted.

3.2.2 Resetting Implementation

After every FLASH memory write operation, even on successful operations, the FreeRTOS scheduler seems to stop, or all tasks are suspended and only interrupts are working. This is probably caused by not updating the watchdog for long time, thus causing the program to hang. The currently implemented code overcomes this issue by initiating a software reset after FLASH writing operation. This is definitely not the optimal solution and a better one needs to be found in future versions, but for the moment the resetting implementation causes no problem to the general code implementation of the mock-up.

4 Bugs Encountered

4.1 The Hidden Treasure

The greatest bug of this code implementation was hidden under the huge amounts of transistors in the FLASH memory! More specifically, there was a hardware fault going off, with PRECISERR all the time, whenever there was an effort to write to the FLASH memory, seemingly in any address. The reason that caused this hardware fault was a misunderstanding of FLASH memory programming, about aligned addresses. When aligned addresses were used, the hardware fault magically disappeared and the full glory of the bit treasure was uncovered.

4.2 No Fault, But No Write Too

Having solved the hardware fault issue, there was still a problem in writing to the FLASH memory, this time with no fault or other issues. The root cause for that was that the FLASH memory page was not erased first. After implementing a page erase *the page at which the desired address was in*, and then writing everything seemed to be completely fine. Although there was still an issue and that is the writing was not done in full and that leads us to the next and final bug encountered in this code implementation.

4.3 Wrote Successfully, But Not All Data

With the previously described solutions implemented there was still an issue with data write. Data was written in the FLASH, but not in full. Only the first part was written and the cause for that was the use of the macro `FLASH_TYPEPROGRAM_WORD`, instead of `FLASH_TYPEPROGRAM_HALFWORD`, which caused the programming to skip a whole memory block, because as it was described in this report, FLASH memory is programmed in half-words at a time.

5 Incomplete Tasks

There are definitely some important incomplete tasks and those tasks along with others are listed below:

- Code merge with the master branch is pending
- Resetting implementation needs to be replaced by a better process
- Rewrite big blocks of data to FLASH to test code patching, by re-purposing some functions on the fly

6 Conclusion

The FLASH operations code implementation is fully functional and all tests passed. An important note to take into consideration is that before any writing event in the FLASH, erase operation must precede the writing. Also, writing is done half-word at a time, with each half word being *"inverted"*. As described earlier in this report, the resetting implementation has to be reconsidered and replaced by a better service, in order to avoid resetting. Finally, I would like to mention the decisive role that community help played in the resolution of the most important bug and the related issue on GitLab can be found here.

Acronyms

ASCII American Standard Code for Information Interchange. 5

MCU MicroController Unit. 2