Lab 2: MemTest

Taylor Okel

9/21/2018

Table of Contents

Table of Figures

Problem Statement

As Stated

To determine the size of working memory on the Maple board. The documentation says 128k of flash, 20 kbytes of sram. Perhaps they are lying to us. Maybe some of the memory failed due to radiation exposure. We want to know what we have to work with.

Derived

The task described above models a memory test (henceforth “memtest”). This program should loop through the given memory maps and perform a “read\write\read\verify” exercise to confirm that the memory is accessible, can be written to, and can write correctly.

Assumptions

1. The provided memory begins at the addresses listed in the memory map provided by the manufacturer
2. The provided memory is *no larger* than the size indicated by the datasheet
3. The flash memory has enough remaining erase/write cycles to complete the test

The rationale for these assumptions is as follows, in order:

1. If the device is at a different address, users would have no reasonable way of determining its address. Further, the manufacturer has no distinguishable reason to intentionally mislead users
2. Testing above/below the device is tricky, as this can affect other registers (namely, control registers) and cause unpredictable effects. The manufacturer also has no distinguishable reason to provide more than advertised.
3. If the flash memory is about to die, there is no way of determining this (at the writer’s current experience level). Further, the number of cycles provided by the device is sufficiently high to assume this with a high degree of certainty.

Equipment

- Olimexino-STM32 Development Board (henceforth “board”)
  - STM32F103RBT6 Arm Processor (Olimexino-STM32 development board User's manual 2016)
- USB programming cable
- Programming Workstation (henceforth “computer”)
  - Windows 10 operating system
  - Arduino IDE

Architecture

The board is connected to the computer via the USB programming computer. The code for the application will be written and programmed using the Arduino IDE compilers.

Design

The board has two memory devices with unique access mechanisms, so the design was split into two sub-processes - one for each device. The main program will call the tests and display test results.

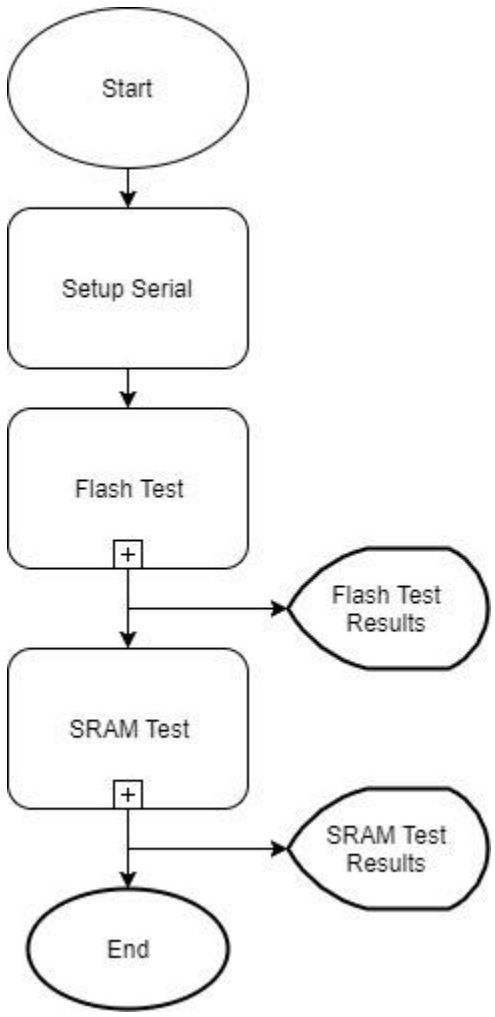


Figure 1: Lab2 Main Flowchart

The basic idea of each test will involve writing a pattern and reading the pattern back from each address in each device. However, both devices have unique challenges to consider, which be discussed in detail later in this section.

For both designs, the memory map of the STM32 processor (STM32F103xB Reference Manual 2015) was referenced for memory locations.

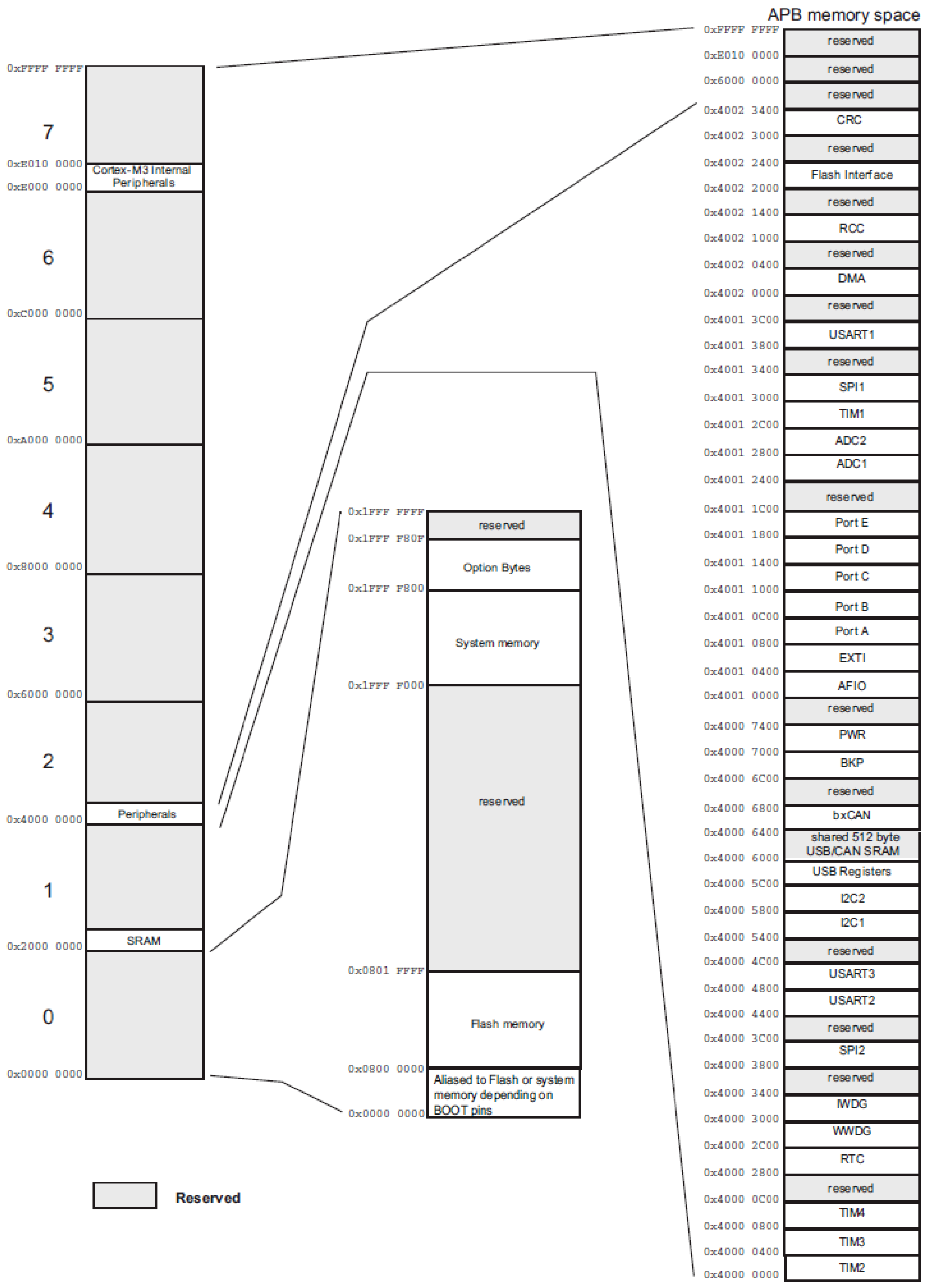


Figure 2: STM32 Memory Map

SRAM

From Figure 2, the starting address of the SRAM is 0x20000000. Starting at this address, the memory is exercised by writing walking ones, DEADBEEF, and zeros. This provides decent coverage to ensure that the reads and writes are happening correctly, and that the addresses are, in fact accessible. Figure 3 demonstrates this algorithm in a flowchart.

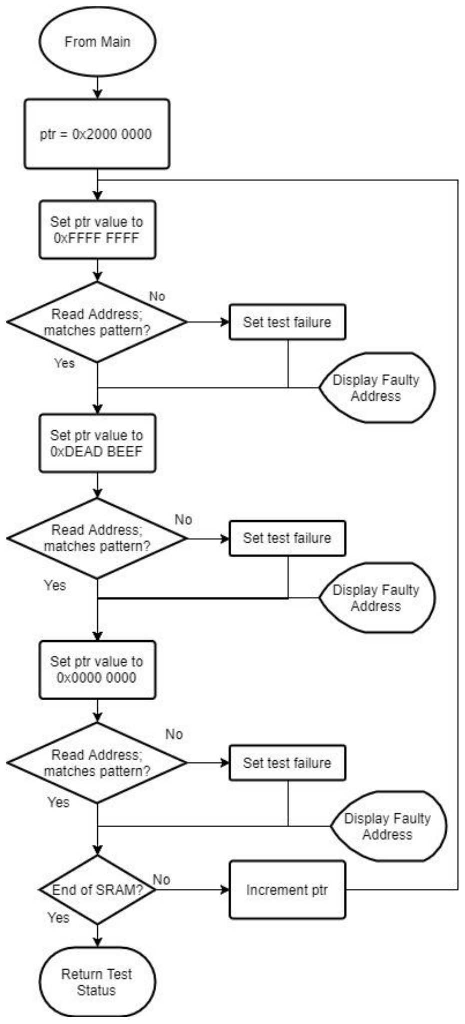


Figure 3: SRAM Test Flow Diagram

The issue with this algorithm is it essentially erases the entire SRAM, which is a major issue considering this is where the program code resides. A simple remediation of backing up the memory and restoring after the test will solve this error. This leaves just one memory address that would cause program crash: the address of the backup data itself. When the algorithm reaches that address, the backup would store its own data, and become erased, leading to a program crash. So, two variables store the backup data. The “walking zeros” test comes last, as the data can be easily restored by or'ing the two backups back into the address – when one is cleared, the data will still be preserved by the other.

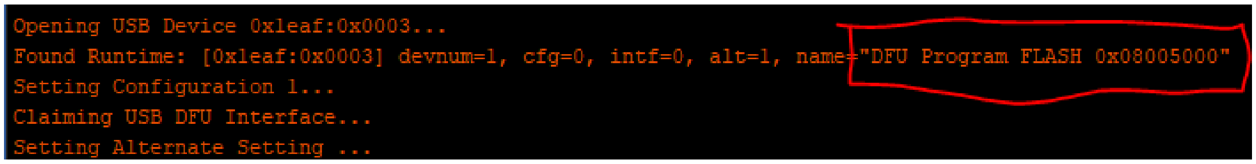
Flash

The Flash device is not as simple. To begin, Flash has an inherent limit to read/write cycles (PM0075 STM32 Flash Programming Manual 2012), so the number of times this test was run in the debug stage was limited to preserve the Flash life. Further, Flash can only be written once the address is erased; in other words, the address must be 0xFFFF before any data can be written (PM0075 STM32 Flash Programming Manual 2012). Finally, the Flash can only erase one page (1KB) at a time via a specific erase procedure (PM0075 STM32 Flash Programming Manual 2012).

Therefore, due to the complexity of Flash control, I wrote a controller module in case this code will need to be reused in later labs. This controller is explained in detail further in this section.

The controller is implemented into the top-level Flash Test. It is assumed at this point that all sub-process (denoted by '+' in a process box) work as advertised at this point. From this level of abstraction, the Flash Test begins to resemble the SRAM test.

The Flash address begins at 0x08000000 (PM0075 STM32 Flash Programming Manual 2012). However, overwriting this carelessly is extremely dangerous; the bootloader resides in the beginning section of the flash, erasing the bootloader would render the board unusable. While it is possible to restore the bootloader, circumventing the bootloader is safer. From the output of the Arduino Upload process (Figure 4), the bootloader code ends at address 0x08005000.



```
Opening USB Device 0x1eaf:0x0003...
Found Runtime: [0x1eaf:0x0003] devnum=1, cfg=0, intf=0, alt=1, name="DFU Program FLASH 0x08005000"
Setting Configuration 1...
Claiming USB DFU Interface...
Setting Alternate Setting ...
```

Figure 4: Snip of Arduino Upload with Bootloader End Address Callout

Following this code lies the program code. While this is less dangerous and can be found by looking at the compiled metadata, time constraints forced me to roughly start the test at the (supposed) halfway mark of the Flash, 0x08010000. It is assumed that, since the program code and bootloader are operating correctly – or else the test would not run – that this beginning section is operational.

The test patterns chosen here resemble a test performed while searching for shorts; one pattern is the inverse of the other, with every other bit active. This reduces the number of patterns to be written (saving the Flash block), and demonstrates another important memtest algorithm. Figure 5 demonstrates the algorithm.

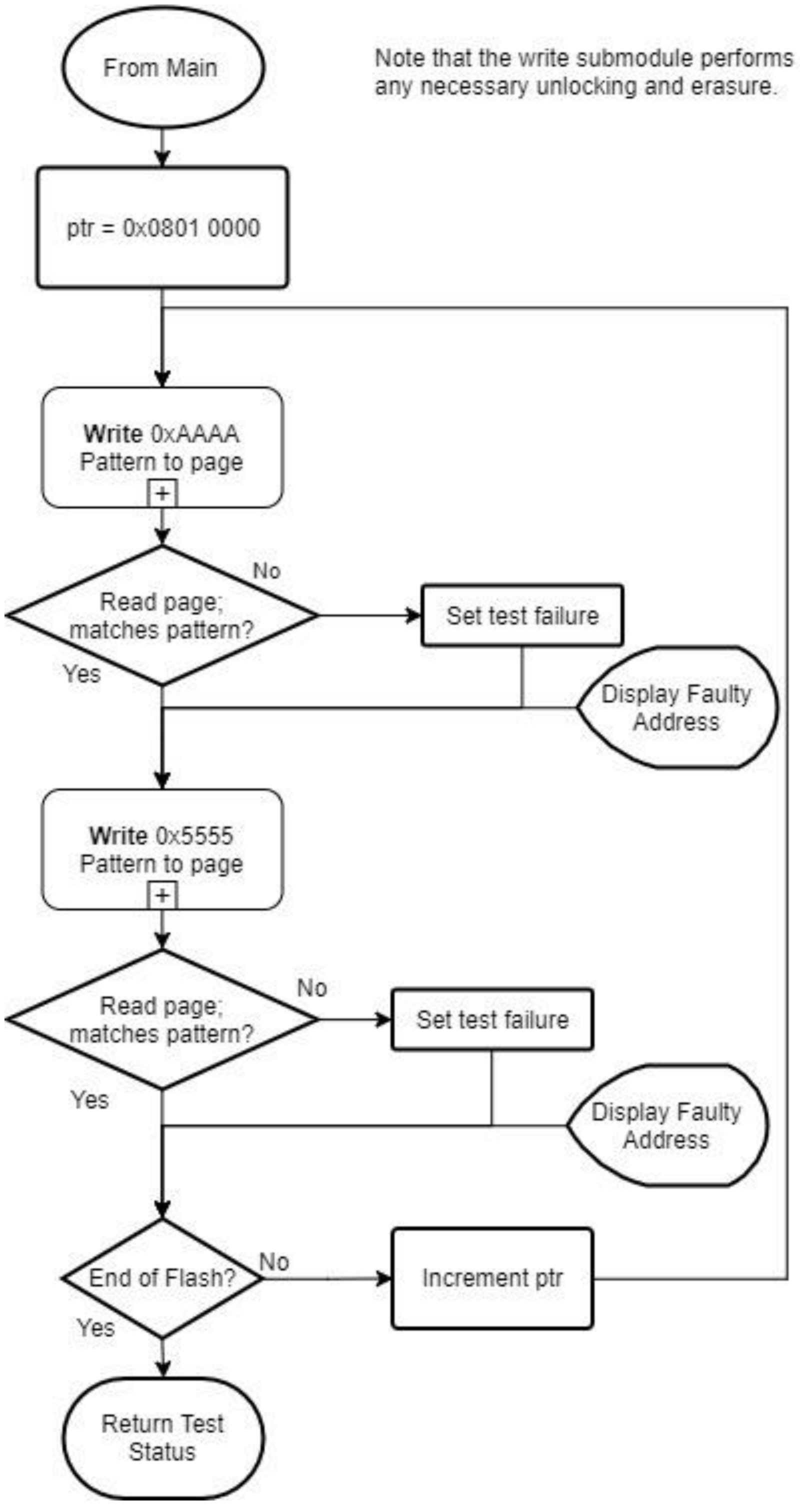

From Main
Note that the write submodule performs any necessary unlocking and erasure.
ptr = 0x0801 0000
Write 0xAAAA Pattern to page
+
Read page; matches pattern?
No
Yes
Set test failure
Display Faulty Address
Write 0x5555 Pattern to page
+
Read page; matches pattern?
No
Yes
Set test failure
Display Faulty Address
End of Flash?
No
Yes
Increment ptr
Return Test Status

Figure 5: Flash Test Top

Flash Controller

Due to the complexity of Flash operations, a Flash Controller was written to abstract the interface. This is how the algorithm in Figure 5 seems as simple as it does. The following sections details the operation underneath this controller.

Note that the following flowcharts do not show waiting on the busy bit to clear, for simplicity of the diagrams. This wait happens any time a memory address is read/written/erased. See the code (

Write

The description begins with the write, as it is the first called from the FlashTest. Writing Flash has two dependencies: first, the flash must be unlocked, and second, the data entry must be erased, 0xFFFF.[1] These dependencies are checked (and, if necessary, corrected), by the corresponding sub-processes.

The programming sequence, as described by the Flash Programming Manual (PM0075 STM32 Flash Programming Manual 2012) on page 14, section 2.3.3, is as follows:

- Check that no main Flash memory operation is ongoing by checking the BSY bit in the FLASH_SR register.
- Set the PG bit in the FLASH_CR register.
- Perform the data write (half-word) at the desired address.
- Wait for the BSY bit to be reset.
- Read the programmed value and verify.

The controller uses this sequence to implement the algorithm depicted in Figure 6. Compare this with Figure 1 on page 13 of the Flash Programming Manual (PM0075 STM32 Flash Programming Manual 2012). The manual fails to note that the address must be written to the FLASH_AR register. The FPEC needs this the address value to ensure that the address has been previously erased.

[1] There are other dependencies, such as write protected pages, but these considerations were non-issues in this implementation.

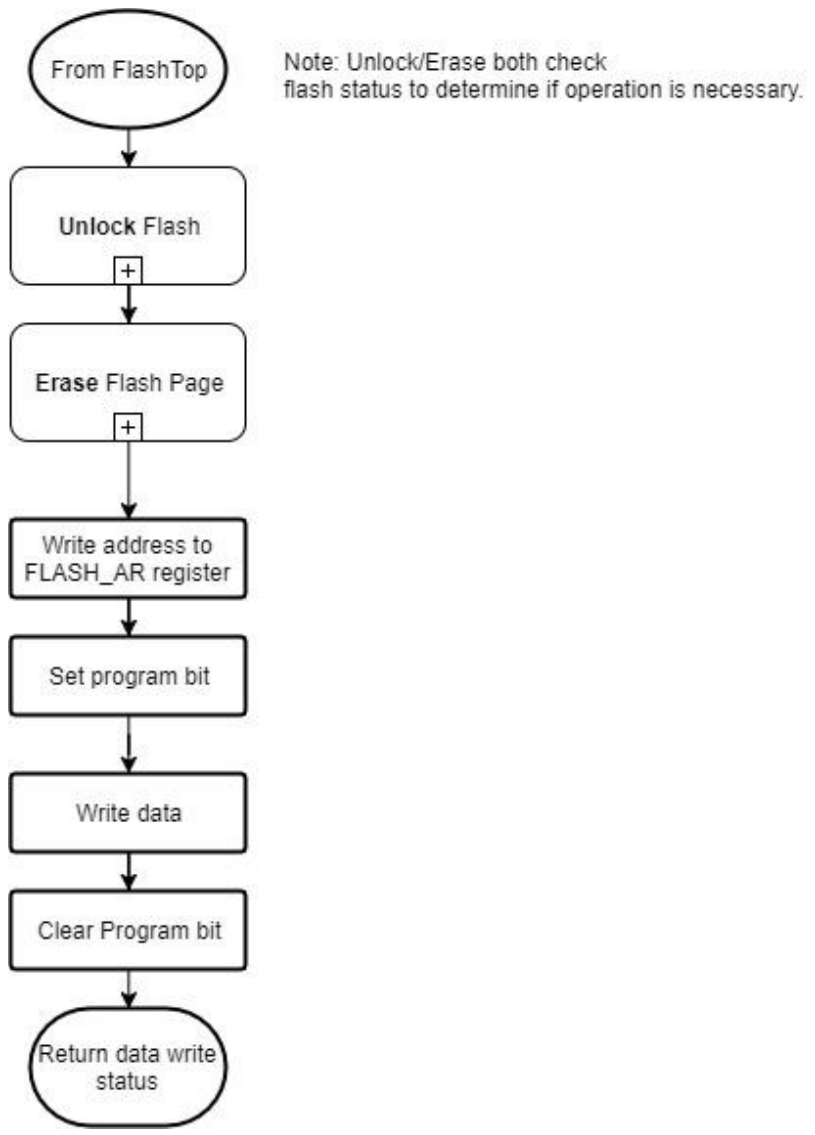

From FlashTop
Note: Unlock/Erase both check flash status to determine if operation is necessary.
Unlock Flash
Erase Flash Page
Write address to FLASH_AR register
Set program bit
Write data
Clear Program bit
Return data write status

Figure 6: Flash Controller Write

Unlock

The unlock sequence is described in the Flash Programming Manual (PM0075 STM32 Flash Programming Manual 2012), page 13 section 2.3.2. A bit in the control register, FLASH_CR_LOCK, denotes whether the flash is locked. If it is set, a key sequence must be written to the FLASH_KEYR register. A failed unlock sequence will lock the flash until the next reset. The appropriate keys are provided in section 2.3.1 of the Flash Programming Manual (PM0075 STM32 Flash Programming Manual 2012).

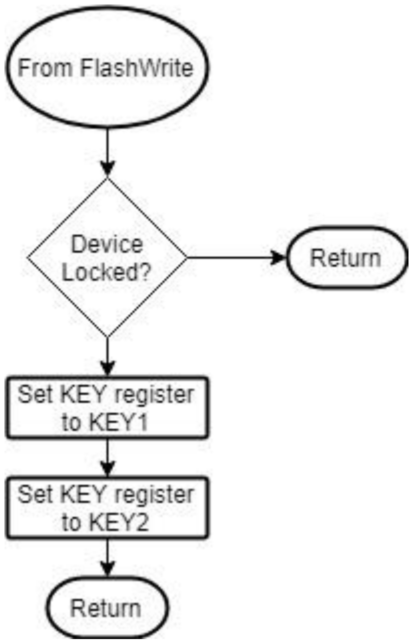


Figure 7: Flash Unlock Flow Diagram

Erase

The erase procedure is more complex, as the Erase Pattern (all ones, 0xFFFF) is not written directly to the Flash. Instead, a erase sequence using the control and address register must be performed. This sequence is described in section 2.3.4 on page 14 of the Flash Programming Manual (PM0075 STM32 Flash Programming Manual 2012).

This procedure is implemented by the Flash Controller Erase function, depicted in Figure 8. This time, there are no major discrepancies between the two processes, save for an unlock check and (redundant) clearing of the control bits in this implementation.

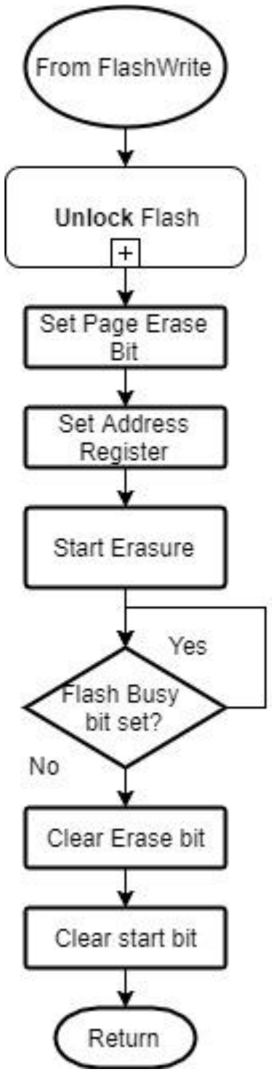


Figure 8: Flash Erase Flow Diagram

Read

The read is by far the simplest algorithm. Section 2.2 of the Flash Programming Manual (PM0075 STM32 Flash Programming Manual 2012) describes the read process. A read is performed simply by accessing the appropriate memory slot. Figure 9 demonstrates the read algorithm of this implementation.

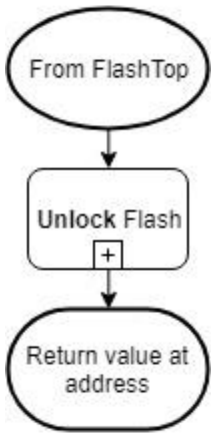


Figure 9: Flash Read Diagram

Implementation

The uploaded package consists of 3 files: lab2.ino, FlashController.h, and TestSpeedup.h.

The first file (lab.ino) contains entry point of the payload, and runs the SRAM tests, manipulates the Flash Controller to perform the Flash Test, and prints test results.

The second (FlashController.h) contains the code implementing the design discussed in the Flash Controller section. This code implements each of the four functions, as well as providing global information to Flash addresses and register bit masks.

The final file (TestSpeedup.h) is a set of macros designed to quickly control the rate of the SRAM test. If debugging info is printing in this test, the test may take a long time to complete. As such, the INC() macro-defined function allows the developer to easily choose a faster increment rate by changing the defined macro on line 20 of lab.ino.

This package is compiled by the Arduino IDE and uploaded to the board via USB.

Test Plan

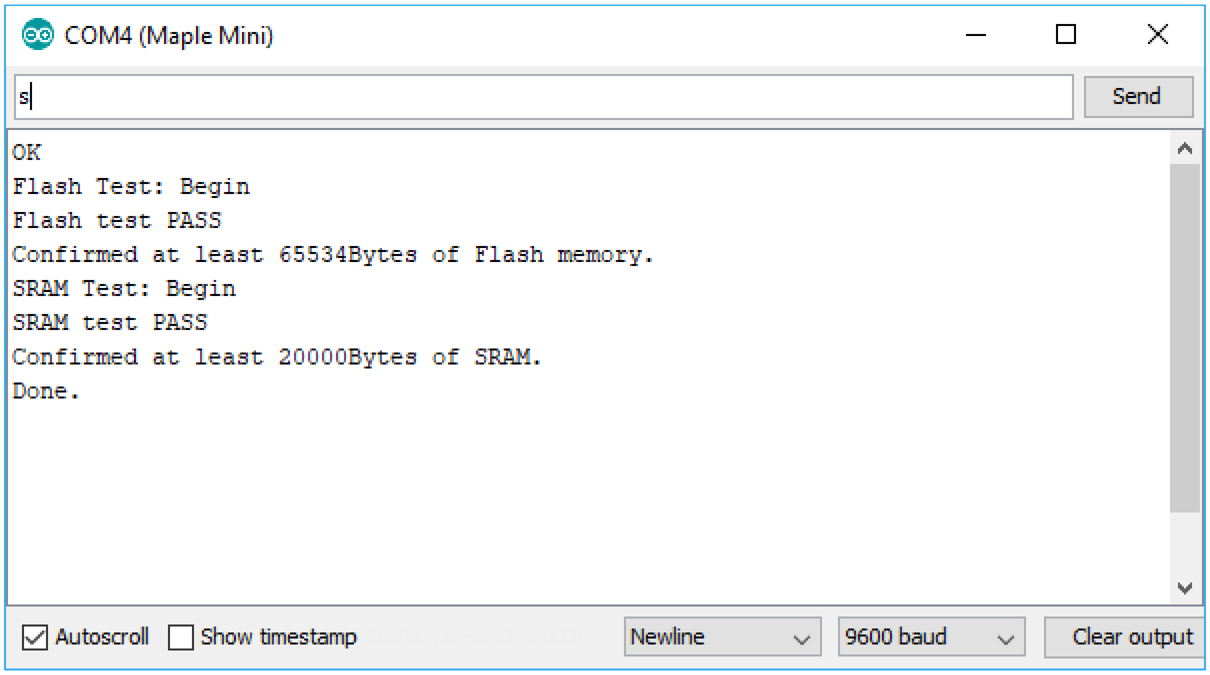
- Compile the test Payload.
- Plug the USB into the computer and board.
- Upload the test payload
- Observe the output
- If needed, make adjustments or add more debugging printouts and run again
- Repeat this process until both tests are (a) fully implemented and (b) passing on the test board.

Test Results

This test took several tries to get right. There were several complications along the way; to name a few:

- Overwriting program memory (SRAM and Flash) corrupts program and crashes with little info
- Timing issues with the Flash interfaces
- Unclear/Incomplete Flash Programming Manual

However, the tests finally passed:



```
OK
Flash Test: Begin
Flash test PASS
Confirmed at least 65534Bytes of Flash memory.
SRAM Test: Begin
SRAM test PASS
Confirmed at least 20000Bytes of SRAM.
Done.
```

As the output states, 65534 Bytes (64KB) of Flash and 200000 Bytes (20KB) of SRAM were verified. As discussed in the Flash section of the design, only half of the flash was verified. As such, these values are as expected, meaning the devices meet the advertised specifications and functionality.

Suggestions

Future Improvements to this Implementation

- Serial prints are needed to get the unlock to function correctly.
    - This indicates a timing issue, the time needed to verify the KEYs.
    - Find a more graceful method than empty prints.
- Flash test improvements – all Flash memory besides bootloader
    - Possible if pages are backed up, but reliant on SRAM containing all program code.
    - Discussions on blackboard of a way to load program straight into flash

Future Improvements to Lab Description

- Provide the tips and tricks earlier
- While the open-endedness of this lab was very interesting, it is hard to determine if enough was done. Since this results in a grade, it is preferred to have a better handle on what requirements need met.

Code

The code can be viewed on the author’s GitHub page, https://github.com/okeltw/EmbeddedMicrosystems/tree/master/Labs/Module3/lab2.

This release of the code was tagged as ‘Lab2-Rev2’, to preserve this code in case of future updates as noted in the Suggestions section.

References

2016. "Olimexino-STM32 development board User's manual." Olimex Ltd, September.

2012. "PM0075 STM32 Flash Programming Manual." STMicroelectronics, August.

2015. "STM32F103xB Reference Manual." STMicroelectronics, August. 34.