Device Drivers for Space-Qualified Flash Devices

Matthew S. Colgan
Matthew.Colgan@colorado.edu
Department of Computer Sciences
University of Colorado at Boulder
July 30, 2004

Abstract
The purpose of this thesis is to provide device drivers that serve as an interface to several flash devices commonly used in spacecraft instruments and embedded systems. In particular, this library focuses on read, write, and erase functions for three NAND devices that are not CFI-compliant (i.e. do not already have device drivers), namely 1) Maxwell 29F0408 2) Maxwell 69F1608 3) Samsung KM29U128. This paper describes how and why these devices were chosen, how the interface was designed, and walks the reader through the design trades conducted in order to reach the current implementation. The paper will conclude with a discussion of how single-errors with flash are addressed by the library, as well as open issues.
Introduction: “What is a generic flash interface? Why is it useful?”

The interface is a set of device drivers that read, write, and erase a flash memory device. The purpose of the library is to give future programmers the ability to use flash devices in their software without needing to understand all of the internal architecture and quirks of the device. In particular, many flash devices used in space applications are 8-bit or 16-bit devices, and many NAND flash devices are divided into blocks and/or pages. Without an interface, these constraints prevent normal memory addressing modes (e.g. using 32-bit addresses).

In 1996 Intel, AMD, Sharp, and Fujitsu began developing the Common flash Memory Interface (CFI) as part of an industry-wide effort to increase interchangeability of current and future flash memory devices.¹ The initial purpose of CFI was to provide a standard on how to store information identifying the device, such as memory size, byte/word configuration, and necessary voltages, on the device itself. Over the next few years, the standard evolved to provide low level drivers for reading, writing, and erasing many flash devices.

However, there are several flash devices used in space applications that are not CFI-compliant. Samsung and Maxwell produce some of the few radiation-hardened flash memories available, yet these memories are not part of the CFI. Thus, the remainder of this study focuses on a) identifying devices that are used (or are planning on being used) in space applications yet are not part of the CFI b) providing device drivers for those devices c) advantages and disadvantages of the provided device drivers.
Background on Flash Devices and Requirements of Space Applications

Flash devices are commonly used in digital cameras, wireless devices, and other devices that require non-volatile storage (i.e. the memory still retains the data after being powered off). Flash memory is increasingly being used in space applications where solid-state storage is a must, and where data storage requirements exceed the limitations of EEPROM. The first type of flash memory to come on the market was NOR flash memory, which has fast random access times making it ideal for code storage and execution. However, as data storage requirements have rapidly increased over the past several years, the high density and low cost make NAND flash a formidable contender against NOR flash for data storage markets. According to Samsung and Toshiba, NAND will overcome NOR flash as the dominant architecture by 2005.2,3

Most NAND flash devices have 8-bit or 16-bit addressing, yet they have storage capacities in the megabits. How is this possible? It takes several bus cycles to send the appropriate number of address bytes, and they have internal logic that contains state machines for performing the read, write, and erase operations. This parsing of the address, along with the mechanics of erasing using a high-voltage generator, requires partitioning of the memory into blocks, pages, and areas. This partitioning is what makes addressing Flash different from SRAM.

Flash Device Candidates and Selection

Since the interface is only useful if it can be implemented for flash devices that are actually used by spacecraft electronics designers, a search was conducted to determine which devices have been used in previous missions, or are in the works for spacecraft currently being designed. The following databases were queried for Flash
memory used (or to be used) in spacecraft: NASA Technical Report Server (NTRS), AIAA Online Technical Meeting Papers, Aerospace and High Technology Database, and IEEE Electronics Library. It was found that at least six spacecraft since 1997 have used Flash memory between the sizes of 512 KB to 4 MB, and that the Mars Exploration Rovers (“Spirit” and “Opportunity”) used 256 MB of Flash memory. The earlier applications, such as on Stardust in 1999, primarily used NOR Flash to store and execute code. The Student Dust Counter (SDC), a student-built instrument flying on the New Horizons spacecraft in 2006, uses NAND Flash both for code and data storage. Future designs, such as for the Europa Orbiter, continue to incorporate NAND Flash for data and/or code storage.

A second search was then conducted to find which specific Flash models and manufacturers are being used in space applications. Searches were performed on the “JPL (Jet Propulsion Lab) Electronic Parts Engineering RadData” server and the “NASA/GSFC (Goddard Space Flight Center) Radiation Effects & Analysis” website. The search found seven flash memories that have been studied by either GSFC or JPL to assess the devices’ susceptibility to radiation in a space environment. Thus, the following devices are in either existing spacecraft or are in the design process:

1. Maxwell 29F0408: 4 MB x 8-bit NAND
2. Maxwell 69F1608: 16 MB x 8-bit NAND
3. AMD AM29LV800: 1 MB x 8-bit or 512 KB x 16-bit NAND
4. SEI/Intel E28F016SB: 1 MB x 16-bit Flash EEPROM (CMOS)
5. Sandisk Flashdisk 2 Mbit
6. Aeroflex MCM (Multi-Chip Module) NOR
7. Samsung KM29U128: 16 MB x 8-bit NAND
After looking up the datasheets on each device, it became clear some devices already have drivers implemented. The AMD and Intel devices (3 and 4 above) are both CFI-compliant, so their drivers have been implemented and released (although CFI release notes suggest further inspection may be needed to verify they are small and fast enough to be used in aerospace embedded system). From the information provided on the servers, it was unclear which models were used for 5 and 6, and datasheets could not be found for any similar devices. Thus, devices 1, 2, and 7 were selected to be included in this device driver library.

The Interface Library

With an SRAM memory device, the two modes are reading and writing. However, since NAND flash memory cells can not be re-written, the “writing” mode must be split into writing and erasing (blocks/sectors only). Thus, there are three main functions for a given flash device: readflash, writeflash, and eraseflashBlock.

Reading from flash

The prototype of the readflash function is as follows:

```
unsigned char readflash (unsigned long sourceStartAddress,
                        unsigned long *destinationStartAddress,
                        unsigned long offset)
```

This function copies X bytes (where X is “offset”) from flash to SRAM, starting at sourceStartAddress in flash and at destinationStartAddress in SRAM. The start flash address is not given as a pointer because it will be parsed into the native format of the flash device. The return value of the function is the status of the read operation. If ECC is
enabled (see description in section Error Detection and Correction), the function will return 0 if successful and will return 1 if the ECC detected two consecutive single-errors (i.e. an error occurred during the first read operation and the second back-up read operation).

This is the description of the readflash interface, which works for all three devices supported by the library. Since it is the implementation of the interface that affects the performance of the software, we will select a device as an example, the Maxwell 29F0408. The readflash function has three main sections:

1) Parse the 32-bit input address into the device’s native format
2) Send the appropriate commands to setup the read command
3) Copy the data from flash to SRAM

1) Parsing the flash Address: First, the 32-bit sourceStartAddress is parsed into the corresponding block, page, area, and byte numbers. As explained previously in section X, the 29F0408 is a 4MB device with 512 blocks, 16 pages per block, and three areas (A, B, C) per page. However, areas A and B are 256 bytes each, which are what make the device 4MB. Each Area C is 16 bytes long, so the device is actually larger than 4 MB. In order to address the bytes in area C, the area number would need to be 2 bits in length. However, since there are three areas, only three of the four values (00, 01, 10 binary) would be used. This prevents addresses from being continuous. Thus, the solution used by the library is to not let the user access area C (although the user still has access to a full 4MB worth of storage), making the area number 1 bit in length (‘0’ for Area A, ‘1’ for Area B).
Thus, the block number is 9 bits long, the page number is 4 bits long, the area number is 1 bit, and the byte number is 8 bits. Having calculated the required number of bits, these variables are simply concatenated next to each other to form the 32-bit address. The address thus takes the form:

0xGHIJKLN (hex)  (Note: bit 0 is LSB, bit 7 is MSB)

- 0xNM = byte number (8-bits)
- 0xKL = bit 0 (LSB) is area # (‘0’ = area A, ‘1’ = area B), bits 1-4 are page #, bits 5-7 are lower bits of block #
- 0xIJ = bits 0-5 are the remaining upper 6 bits of block #, bits 6-7 unused
- 0xGH = unused (0x00)

2) Setting up the Read Command: The second step of readflash is to prepare the flash device for reading from the specified address. The first part of this is to send a “command” to the flash device, indicating which area the user would like to write to. In the case of this library, only 0x00 (for Area A) or 0x01 (for Area B) will be sent. The next set of information to be sent to the device are the three address bytes (or ALE, Address Latch Enable), which include the block, page, and byte numbers. After the last address byte is sent, the flash device automatically begins copying the requested page (512 bytes) from its memory cells to its internal buffer. The microprocessor must wait for flash to complete this process before it can access the data. A “high” signal on the ready/busy line indicates the data is ready for the microprocessor.
3) **Copying the Data from flash to SRAM:** The last step of readflash for the Maxwell 29F0408 is the actually copying of data into SRAM. Once the device has finished copying the data into its buffer in Step 2, it copies the first byte in the buffer to an I/O register. The microprocessor then copies the byte from the I/O register into SRAM. Immediately after the microprocessor accesses the I/O register, the flash device automatically copies the next byte in the buffer to the I/O register. This process repeats itself until the end of the 512-byte buffer is reached (or if no more data is needed). If the caller of readflash requested data that crosses a page boundary, then Step 2 is repeated so the flash device can copy the next page’s data into the buffer. On the up side, this process of setting up a page and reading 512-byte portions can be looped as many times as necessary, which is how the library allows the user to read as many bytes from flash as necessary.

**Writing to flash**

The prototype of the writeflash function is as follows:

```c
unsigned char writeflash(unsigned long *sourceStartAddress,
                          unsigned long destinationStartAddress,
                          unsigned int offset);
```

In essence, writeflash is the opposite of readflash: it copies X bytes (where X is “offset”) from SRAM to flash, starting at sourceStartAddress in SRAM and at destinationStartAddress in flash. As with readflash, the start flash address is not given as a pointer because it will be parsed into the native format of the flash device. The return value of writeflash is 0 if successful or 1 if the internal status register indicates the
operation failed. Now the implementation of the writeflash function will be described, again using the Maxwell 29F0408 as an example.

The writeflash function for the 29F0408 has six main sections:

1) Parse the 32-bit input address into the device’s native format
2) Disable write-protect
3) Send the appropriate commands to setup the write command
4) Copy the data from SRAM to flash’s internal buffer
5) Wait for flash to copy data from its internal buffer to permanent memory
6) Enable write-protect

1) Parsing the flash Address: Same as Step 1 in readflash.
2) Disabling Write-Protect: The Maxwell 29F0408 has a built-in feature that prevents inadvertent writes/erases during power transitions. Thus in order to write to flash, the write-protect line must be set high.
3) Setting up the Write Command: This process is similar to setting up a read operation, except for a couple subtleties. The area number is again sent first, but it is sent on a different command line than the read command line. It is then followed by a special command byte (0x80) that indicates a write operation is beginning. Finally, the three ALE bytes are sent in the same format as the read operation.
4) Copying the Data from SRAM to flash buffer: This step simply copies data from SRAM into the flash buffer. Again, the write operation can not write to more than one page at a time. Even if the user wants to read less than 512 bytes, another page write operation is necessary if the addresses cross page boundaries. In order to relieve the user
of keeping track of the location of their data with respect to pages, the writeflash function determines if the user crosses page boundaries, and if so, automatically performs additional write operations. The function loops as many times as necessary to write all of the data (lifting the 512-byte limit).

5) **Waiting for flash to Copy Data:** Here the microprocessor polls the ready /busy line, waiting for the flash device to finish copying the data from its internal buffer to its permanent memory cells.

6) **Enabling Write-Protect:** The microprocessor sets the write-protect line active low, protecting the device from inadvertent writes/erases.

**Erasing flash**

The prototype of the eraseflashBlock function is as follows:

```
unsigned char eraseflashBlock(unsigned int blockNumber)
```

`eraseflashBlock` is different from reading and writing because instead of taking 32-bit addresses as parameters, it takes one 16-bit block ID. The acceptable range of ‘blockNumber’ depends on the number of blocks in the flash device, but the range begins at 0 and ends at N-1 (where N is the number of blocks). For example, in the case of the Maxwell 29F0408, a user would pass ‘0’ to erase the first block or pass ‘511’ to erase the last block. As stated previously, a block is 8 KB in the Maxwell 29F0408.

The implementation of the eraseFlashBlock function for the 29F0408 simply follows the state machine outlined by the manufacturer and has four main sections:

1) Disable write-protect

2) Send the appropriate commands to setup the erase command
3) Wait for flash to finish erasing specified area

4) Enable write-protect

1) **Disabling Write-Protect**: Same as in writeflash.

2) **Setting up the Erase Command**: First, the command byte 0x60 is sent, signaling the flash device that an erase operation is beginning. Then two address (ALE) bytes are sent, although these are in a different format than the read and write ALE bytes. This is because the page and byte numbers are not necessary, since flash erases in blocks containing many pages and bytes (e.g. 16 pages/block, 528 bytes/page for the 29F0408).

3) **Waiting for flash to Copy Data**: The microprocessor polls the ready /busy line, waiting for the flash device to finish erasing the specified block.

4) **Enabling Write-Protect**: Same as in writeflash.

**Using the Library**

There are four modifications the user must make to the library before compiling:

1) The user must select the flash device to be used. This is done by finding the section in the header file labeled “DEFINE WHICH FLASH DEVICE IS BEING USED” and un-commenting only the “#define” that matches the device. Make sure the other devices are commented out.

2) The user must select whether or not the library should use ECC. The library uses XOR sums during read operations to detect single errors. The extra Area C present in all three devices in the library is used to store the checksum, thus using the library’s ECC does not impinge on the storage area available to the user. If an
Error is detected, the read operation is automatically performed again. If an error is still detected (during the same read operation), readflash returns a ‘1’ (error). There is additional overhead when writeflash computes the checksum and writes it to flash. Un-comment “#define USE_ECC” to use this feature. (see “Problems Addressed by the Library” for more details on ECC and why XOR sums are used).

3) Assign each flash pointer an address under the comment “ASSIGN FLASH POINTER ADDRESSES HERE” These addresses will vary depending on the setup of the user’s electronics and flash chip. Some pointers are not be used by all of the flash devices; the user may wish to comment out these pointers’ declarations.

4) Add the flash function calls to the software that is using the library. The user should also familiarize themselves with the problems under “Open Issues” and implement any solutions to address invalid blocks, turning power to the device on/off, etc.

Error Detection and Correction

One of the important issues with any system that requires highly reliable memory is error detection and correction. With the NAND Flash devices included this library, there are three main types of errors that can occur. The first is due to radiation. According to a study conducted by JPL, the first component to fail due to radiation in NAND flash memory is the internal control logic. Unfortunately, once this portion fails, the device can no longer be used. The second type of failure is when a block becomes “invalid,”
which means that one or more bits within the block cannot be trusted. This occurs either during manufacturing (it is not uncommon for a device to be shipped with one or more invalid blocks) or after a cell (i.e. bit) reaches its write/erase cycle limit. Maxwell claims that the probability of a cell/bit in the 29F0408 becoming invalid, before it reaches the 1,000,000 cycle limit, is 0.1%. The third type of failure that can occur is a single event upset (SEU), where a single bit “flips” its value due to a high concentration of radiation. The JPL and Maxwell reports emphasize that these bits do not get “stuck”.

The first type of failure can obviously not be corrected, but can be detected at the system level, which is discussed below under “Open Issues and Suggestions.” The second type of failure can be detected using the return values of the erase and write operations. If either of these functions returns a 1, then it means that the block has become invalid and should not be used. One possible solution is to remap the block, but since this requires a block lookup table (which should be stored in Flash, thus consuming some of the available storage capacity) and knowledge of how many blocks the user can surrender as “spare blocks”, this is left up to the user to implement. A simple error correction code (ECC) has been implemented for the final type of error (single random errors), although it can be turned on/off at compile time. This allows the user to determine whether the benefits of the solution outweigh the disadvantages of additional execution time, code space, and reduced partial page writes.

Several error control strategies were considered before implementing the final solution. The main requirements of the code were that it a) have as a little impact as possible on the storage capacity available to the user b) have a relatively simple implementation that runs in time suitable to an embedded system c) detects and corrects
SEUs (although correcting/modifying the Flash memory cells themselves is not a requirement since “stuck” bits have not been observed in these devices). Although these requirements may seem at first too vague to be of much use, they quickly dispose of some of the more sophisticated codes. Error control strategies can be split into two categories: a) Forward Error Correction (FEC) codes, which encode extra bits into the data such that the decoder can not only detect a single error but also correct it b) Automatic Repeat Request (ARQ), which uses error detection combined with retransmission of corrupted data. Non-binary codes, such as Reed-Solomon, were thrown out early on due to their high complexity, since 8-bit and 16-bit microcontrollers, capable of only basic arithmetic, are commonly used in spacecraft embedded systems due to their low power consumption, flight heritage, and radiation tolerance. Most of the remaining FEC codes had too high overhead storage, ranging between 20%-50% of the data, compared to simpler ARQ strategies that still meet the requirements. The only Forward Error Correction (FEC) code that was considered as a final candidate was the binary Hamming code.

The binary Hamming code was initially attractive because it can provide single-error correction and double-error detection for the overhead calculated as follows. The maximum number of bytes that can be read out during a single read or write operation is 512 bytes, since a single error occurs during a read or write operation. Thus, if this is treated as the code-block size, then x check bits are needed for 4,096 bits, where $2^x = 4,096$ bits. Thus, only 12 checks bits are needed. Remembering that for the devices included in this library there is an extra Area C (16 bytes long) at the end of each page, none of the user’s storage capacity would need to be consumed to store the check bits!
However, the down side to the Hamming code is that for each check bit a separate summation has to be calculated so that if an error does occur, the location of the error bit can be identified and its value corrected (flipped). Another disadvantage of the Hamming code, in this situation, is that it is overkill since it assumed the error is an SEU. The error is only momentary and on the next read, the bit will have the correct value. In other words, it would be worth the work if the bit was flipped permanently (i.e. “stuck”) and using the Hamming code could fix this.

Since this is not the case, it became apparent that detecting the error using a simpler method, and then repeating the read operation, will still meet the code requirements at lower code size and probably faster execution time (depending on microcontroller clock speed). The simple method of detection is to perform a XOR over each byte written to the page, and store this XOR sum in the Area C section of each page. This eliminates any consumption of user storage space, and the only overhead added is the execution time necessary to calculate the checksum during read and write operations. This may not be a negligible decrease in performance, depending on the user’s timing requirements and microcontroller clock speed, thus it has been made an optional feature that the user can disable during compile time. However, compared to the aforementioned methods, the XOR-sum ARQ is an efficient way to detect and correct SEUs without impinging on the user’s storage space.
Open Issues and Suggestions

The following are problems not addressed by the library that are common to the three flash devices and to space applications. After stating the problem, a suggestion for possible resolution at the system level (or a microcontroller-specific modification to the library) is given.

- **Invalid blocks and the 1,000,000 cycle limit**: There is a 1,000,000 write/erase cycle limit on each memory cell (bit). After a cell has reached this limit, the block containing the cell is more likely to become invalid. A block can be marked as invalid when a write or erase operation accessing that block returns a ‘1’ (error). One solution to invalid blocks is to keep a block lookup table (i.e. memory map) that matches a virtual block number to a physical block number. It is up to the user to determine how many blocks are dedicated to data and how many s/he can allocate as “spare” blocks. The user may also decide to keep duplicates of each data block such that when a block becomes invalid, its identical twin can be copied to a fresh block. If it is not anticipated that a cell will reach the write/erase cycle limit yet it is crucial that a block not become invalid, an alternative to remapping is to keep triplicate copies, one copy in each block, and majority vote when reading the data.

- **Limited Number (10) of Partial Page Writes**: A notorious “feature” of some flash devices (including all three in this library!) is that there is a limit to how many write operations may be performed on a page before it is erased. For example, if the user writes 50 consecutive bytes to a page, but only 5 bytes in 10 separate write operations, then the remaining 528-50 = 478 bytes cannot be written to
(before the block containing the page is erased). Depending on which manufacturer’s data sheet you look at, the maximum number is 1, 2, and 10, and sometimes 3 for area C (although the generally accepted value is 10)! One common solution is to always write in “large” chunks (that is, ~256 bytes or more) such that the partial page write limit is not approached. Another strategy would be to keep track of the number of times the page has been written to in Area C. This would involve turning ECC off and writing a small routine to write to Area C.

- **Polling Statements are Potential Infinite Loops**: Inside each of the readFlash, writeFlash, and eraseFlashBlock functions are while-loops that poll the ready/busy line (active low) of the flash device. If for whatever reason the ready/busy line never goes high, then the software could be caught in an infinite loop. A rather straightforward solution to this is to start a timer inside readFlash, writeFlash, and eraseFlashBlock, and then poll the timer’s overflow flag in the same while-loop that is polling the ready-busy line. This was not implemented in the library because the user could be using the timer(s) for other functions, and because timers are microcontroller-specific.

- **Radiation Susceptibility**: Flash devices have been shown to be susceptible to radiation in two forms. The first form is an SEU, which was discussed in detail “Error Detection and Correction.” The second form occurs when the device reaches a certain cumulative amount of radiation exposure (a.k.a. Total Integrated Dose (TID)) that causes the internal command/control logic to stop functioning, rendering the device useless. One option of detecting this is to routinely erase and
write to a designated “test” block that will not approach its write/erase cycle limit. If both the write and erase operations fail consecutively, it is highly probable that the internal logic of the Flash device is not working, and that the device should not be used. Although this is only a form of detection, it can give the software a chance to go into a back-up mode that can continue without flash.

Conclusion

The purpose of this thesis was to provide device drivers for Flash devices commonly used in space applications. These drivers will hopefully help future programmers by allowing them to address these flash devices in a fashion similar to SRAM (i.e. using 32-bit addresses, not needing to know how many bytes can be read or written in a single operation, etc). The study began by summarizing the results of the search for current and future spacecraft that incorporate flash memory into their design. It then described how the supported Flash devices were down-selected from the flash memories that have been radiation-tested by JPL or GSFC. This selection resulted in devices that are not part of the Common Flash Interface (CFI), and thus devices that do not already have drivers implemented (as a brief side note, it should be mentioned that some CFI implementations are not optimized or intended for aerospace applications, and a separate study identifying which drivers could be rewritten for space applications might be useful to future aerospace programmers). These devices included in the library are: 1) Maxwell 29F0408 2) Maxwell 69F1608 3) Samsung KM29U128. Each of the functions in the drivers was then discussed in detail, using the Maxwell 29F0408 to illustrate how the interface was implemented. Error detection and correction methods were discussed, again narrowing down the possible choices according to the requirements of single-
random errors. Although a binary Hamming code was considered, it was found that a XOR sum, used to detect a single error, paired with calling the read operation again, is a simpler solution and meets the requirements. Finally, open issues and possible solutions that the user should be aware of were outlined, including limits on the number of write/erase cycles per memory cell, the number of write operations to a page before it needs to be erased, the potential infinite loop of polling ready/busy lines, and radiation susceptibility.

Due to the limited number of radiation-hardened NAND flash devices available, it is hoped that providing these device drivers will aid future designers in integrating flash devices into their embedded systems. It is the sincere hope of the author to reduce the pain and agony that is occasionally associated with writing software for flash devices.
References

2) “Selecting the Right Flash Partner to Turn Technology Advantages into Profits.” Samsung, 2003.
3) “NAND Flash Memory”, eeProductCenter.com
   http://www.eeproductcenter.com/showArticle.jhtml?printable=true&articleID=22103055
4) “Spirit Condition Upgraded As Twin Rover Nears Mars”. Jet Propulsion Laboratory (JPL).
6) “JPL Electronic Parts Engineering RadData.”
   http://radnet.jpl.nasa.gov
7) “NASA/GSFC Radiation Effects & Analysis.”
   http://radhome.gsfc.nasa.gov/top.htm