APPLICATION NOTE

Atmel AT4066: EEPROM Emulation Using Internal Flash (SAM4)

Atmel Microcontroller

Description

This application note aims to provide a driver to emulate an EEPROM using the internal flash of a microcontroller. This application note is based on SAM4S device but is fully compatible with SAM4E and SAM4N as well because these devices share the same embedded flash controller. With minor modifications, this application note can be applied to other SAM devices as well.

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Architecture
- Limitations and Future Scope

Features

This design presents a flexible solution with:

- Easy to use APIs
- Automatic wear leveling
- Configurable EEPROM size
- Both IAR™ and Atmel® Studio compatibility
- Easy portability to other devices

Table of Contents

1. Prerequisites

This application note comes with an example code which includes the emulated EEPROM driver. To compile the driver and use the code, following are the pre-requisites:

1. IAR Workbench 6.50 (or higher) or Atmel Studio 6.1.
2. SAM4S Xplained Board with SAM-ICE™ programmer or SAM4S Xplained PRO board (with on-board debugger).

2. Module Overview

Many embedded systems rely on nonvolatile parameters that are preserved across reset or power-loss events. In some systems this static information is used to initialize the system to a correct state at start-up. In other systems it is used to log system history or accumulated data. Traditionally these tasks have been implemented using EEPROM; first with off-chip EEPROM and later in on-chip EEPROM as levels of system integration have increased. This application note describes how to emulate the behavior of an on-chip EEPROM.

2.1 Flash Memory Basics

Flash memory consists of independent cells each representing a single data bit. The flash cells are based on floating gate transistor technology: an electrical charge “trapped” on the floating gate determines the logic value of the cell. “Erasing” a cell charges the floating gate, allowing the cell to read as logic one. “Programming” a cell discharges the floating gate, bringing the logic value to zero. Therefore it is only possible to program (discharge) a cell that was previously erased (charged). Bit cells are grouped into data bytes, but bits within the byte can be programmed individually. Since only the cells being programmed are discharged, the remaining unprogrammed cells remain charged. Any unprogrammed cell can be programmed at a later stage. Therefore programming a byte that is already programmed, without erasing it in between, will result in a bit-wise AND between the old value and the new value. If the byte is not erased in advance, it may not be possible to program it to the intended value. For example, assuming that a byte was FEh and was then programmed to 01h; the result would be 00h since the LSB cannot be changed from zero to one by a program operation.

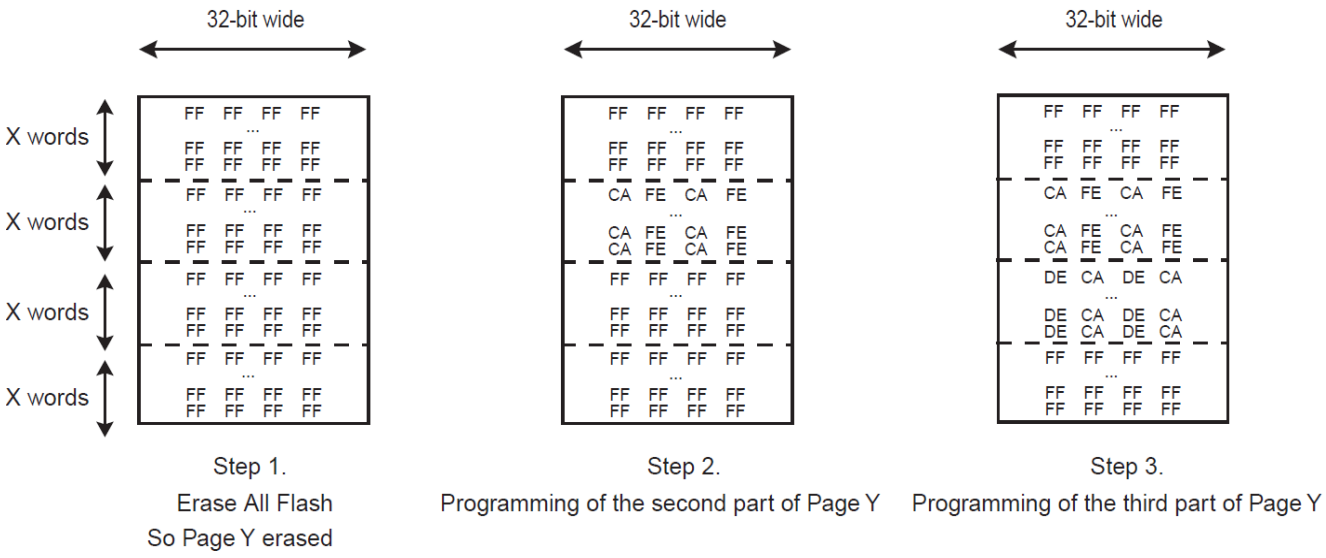
Flash memory is arranged in pages of multiple bytes. An erase operation acts on an entire page; that is, all the bits of all the bytes in the page are charged at one time. A program operation can be performed on the entire page; that is, one or more bytes, up to the maximum page size, can have some or all of their bits discharged at one time. If a single bit in the page must change from zero to one, the entire page must be erased and all bytes reprogrammed.

Traditional EEPROM memory is similar to Flash memory except that the “Erase” and “Program” operations are merged into a single atomic “Write” operation that acts on a single byte. The “Write” operation first erases (charges) all bits in a byte and then programs (discharges) those bits that must be zero. Therefore an EEPROM can update a single byte without regard to its previous value or the value of its neighbors. However, most EEPROMs cannot update multiple bytes simultaneously.

2.2 Partial Programming

In this design, partial programming capability of SAM4S is exploited. Partial programming is a mode in which a page can be programmed in several steps if it has been erased before (see Figure 2-1). After any power-on sequence, the flash memory internal latch buffer is not initialized. Thus the latch buffer must be initialized by writing the part-select to be programmed with user data and the remaining of the buffer must be written with logical 1.This action is not required for the next partial programming sequence because the latch buffer is automatically cleared after programming the page.

Figure 2-1. Partial Programming Example



2.3 Design Considerations

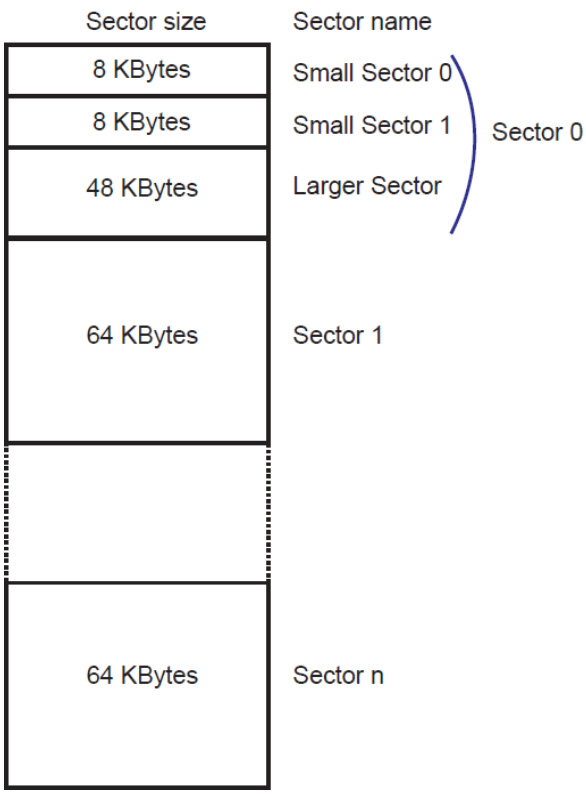
The biggest limitation of flash compared to EEPROM is endurance. Generally EEPROM have at least 100,000 erase cycles compared to 10,000 for flash. To emulate EEPROM in flash, some kind of wear leveling and translation is necessary. In this design, the algorithm uses ten times the EEPROM size in flash and moves the data around in such a way that it is invisible to the end user. By having 10 times the memory at disposal, 100,000 erase cycles can be achieved with same flash.

As explained in Section 2.1, flash is only erasable in blocks. If one variable is to be updated in flash with 512 bytes page size, the entire block would have to be erased first. This causes unnecessary wear. For algorithms, without wear leveling, smaller blocks have advantages over bigger blocks as less memory needs to be erased for a data update. However, when a circular buffer is implemented in firmware (for wear leveling), the smaller block presents no real advantage over big blocks. It is because only after filling the entire buffer, an erase is required. This block erase is generally faster and more power efficient than page by page erase.

2.3.1 Flash Overview

The memory is organized in sectors. Each sector has a size of 64KB. The first sector of 64KB is divided into three smaller sectors. The three smaller sectors are organized to consist of two sectors of 8KB and one sector of 48KB. Refer to Figure 2-2 Global Flash Organization.

Figure 2-2. Global Flash Organization



Each Sector is organized in pages of 512 bytes.

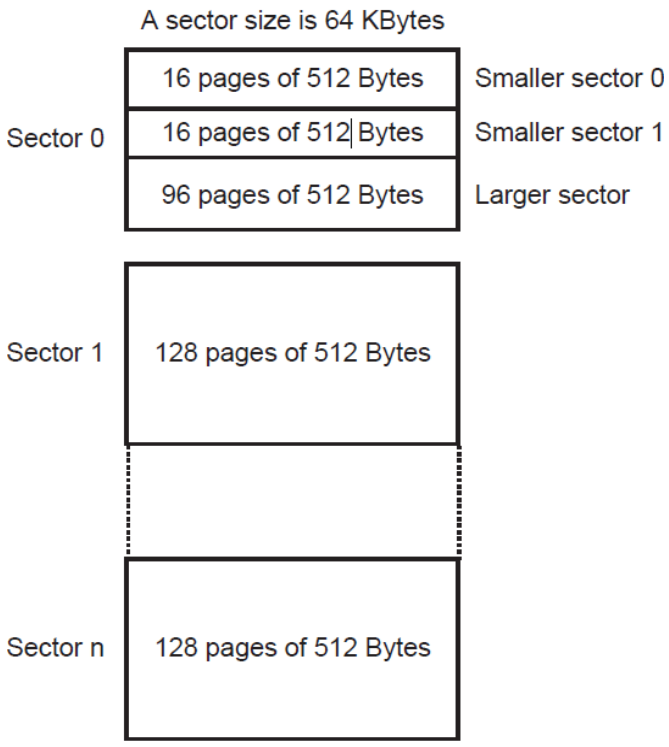
For sector 0:

- The smaller sector 0 has 16 pages of 512 bytes
- The smaller sector 1 has 16 pages of 512 bytes
- The larger sector has 96 pages of 512 bytes

From sector 1 to n:

The rest of the array is composed of 64-KB sectors of 128 pages, each page of 512 bytes. Refer to Figure 2-3 Flash Sector Organization.

Figure 2-3. Flash Sector Organization



Flash size varies by product:

- SAM4S8/S16: the Flash size is 512/1024KB
- Internal Flash address is 0x0040_0000
- SAM4SD16/SA16: the Flash size is 2 x 512KB
- Internal Flash0 address is 0x0040_0000
- Internal Flash1 address is 0x0048_0000
- SAM4SD32: the Flash size is 2 x 1024KB
- Internal Flash0 address is 0x0040_0000
- Internal Flash1 address is 0x0050_0000

Erasing the memory can be performed as follows:

- On a 512-byte page inside a sector, of 8KB
- On a 4-KB Block inside a sector of 8KB/48KB/64KB
- On a sector of 8KB/48KB/64KB
- Complete Chip erase

Note: The Write commands of the Flash cannot be used under 330kHz.

2.3.2 Firmware Design

This application note has been designed using the SAM4S Xplained board which has a SAM4S16C device. In this design, we reserve the last 0x2000 bytes of flash (memory 0x4FE000 to 0x4FFFF) for EEPROM emulation. Sector 0 has not been used for this design even though it is page erasable. It is because:

1. Page erasability of sector 0 it presents no real advantage for this design as this design implements wear leveling using a circular buffer. The erase is done only after the entire buffer (8KB) is filled up. This block erase is faster and more energy efficient than erasing 16 pages one by one.
2. For both IAR and Atmel Studio, it is much easier to reserve flash at the end than in the middle. The linker file changes and the code changes are minimalistic and portable to other microcontrollers.
3. The intended use of sector 0 is mainly for a custom bootloader.

The number of pages to be reserved for EEPROM emulation is user configurable and can be changed as per requirements.

In this design, 16 pages are used to implement wear leveling by means of a circular buffer. Each EEPROM block makes one element of this buffer. Every time there is a data update, this data is written to the next element of the buffer. After the buffer is full, the entire block is erased and the data is wrapped around to the beginning of the buffer. Because of the circular buffer, the effective endurance for the simulated EEPROM is 16 times the endurance of flash. In other words,

Effective endurance = Number of elements in circular buffer X flash endurance.

Choosing 16 pages in this design, gives us effective endurance of 16 x 10,000 = 160,000 erase cycles which satisfies the endurance requirements of a typical EEPROM.

2.3.3 Linker File Changes

As mentioned above, in this design, 16 pages (512 bytes each) are reserved for EEPROM emulation. Hence, the linker file needs to be modified to tell the linker that the last 0x2000 bytes of flash (= 8KB) are not available for program memory. For SAM4S, the flash memory is mapped from 0x00400000 and ends at 0x004FFFFF.

In IAR, this is done by modifying the linker file as follows:

define symbol __ICFEDIT_region_ROM_end__ = 0x004FDFFF;

In Atmel Studio, this can be done by modifying the linker file ‘flash.ld’ as follows:

rom (rx): ORIGIN = 0x00400000, LENGTH = 0xFE000 /* flash, 1024K */

Note: The memory used for EEPROM emulation can no longer be used for program memory. By making these changes, the last 0x2000 bytes are made invisible to the linker so that no code is placed at the address.

3. Architecture

In SAM4 devices, embedded flash controller (EEFC) ensures the interface of the flash block with 32-bit internal bus. It also manages the programming, erasing, locking and unlocking sequences of flash using a full set of commands. The functions to execute these commands are located in ROM (In Application Programming or IAP) so these command functions are accessed via function pointers. All the source code is included in the provided source files.

Note: Detailed information on the embedded flash controller and its commands can be found in SAM4S device datasheet under section “Enhanced embedded flash controller”. In this application, embedded flash controller drivers (efc.h, efc.c), are used for accessing flash.

Using IAP routines, any flash location can be programmed. If there is no wear leveling requirement, any available flash location can be used to store data. Since wear leveling in an intrinsic requirement of this application, a translation layer is needed that manages the location to store data. In this application, the last 16 pages of flash are used as circular buffer. The translation layer keeps tracks of the current page being used and also when the entire sector is to be erased.

Note: During the flash update, the MCU cannot access flash. This is the reason why IAP routines are placed in ROM. After the flash update, the control returns to MCU and it can start executing out of flash again.

3.1 Translation Layer Algorithm

The translation layer algorithm implements a circular data buffer in flash. Every time there is a new data to be written, the data pointer is incremented by one EEPROM block. If there are no valid/empty blocks, entire sector is erased and then index resets to zero.

First byte of each block is the status byte. This byte, if 1, signifies that the block is ready to be written. Every time, at startup, the status byte of each block is checked for validity and stored in RAM. This index becomes the next pointer for incoming data. Also at startup, the previous block data is read and stored in RAM. This becomes the EEPROM current data. Every time there is a read, this RAM buffer is used to send the data. This saves the controller time and resources to read flash every time there is an EEPROM read.

In SAM4S device, the flash page size is 512 bytes. Hence, emulated EEPROM size cannot be greater than 511 bytes (one byte reserved for status). For 511 byte emulated EEPROM, using 16 pages circular buffer, the endurance is 16x flash endurance which exceeds the 100,000 erase count requirement.

Erase count and hence endurance can be increased further by using smaller EEPROM size. The code with this application note gives user the flexibility to choose between 511, 255, or 127 bytes emulated EEPROM. This is done by breaking each page into further sub pages (using partial programming), hence increasing the size of circular buffer. For 255 byte EEPROM, the effective endurance is 320,000 cycles and for 127 byte EEPROM endurance is 640,000 cycles. The EEPROM size is set by setting the “#define EEPROM_SIZE” (conf_nvm_example.h) to appropriate value.

The translation layer algorithm is implemented in translation.c and translation.h files and invisible to the application layer.

3.2 Code Overview

The driver consists of following important files in addition to peripheral driver files imported using ASF.

1. nvm_example.c.
2. translation.c.
3. translation.h.
4. conf_nvm_example.h.

3.2.2 nvm_example.c

This file is the main solution file and contains the main function. Main function implements a typical use case and also provides the reference code that implements the correct initialization and use of this driver. This file also contains RTT peripheral implementation and driver to generate 1 sec and 1 hour ticks. This may come in handy when EEPROM is to be updated in a timely fashion.

3.2.3 translation.c

This file contains all the wear leveling and translation algorithm required by this application. It provides three important APIs that implement all the functionality of this application.

- uint8_t eep_init(void)

  This function should be called first upon reset. This function traverses through the flash pages used for EEPROM emulation, finds the index of the next valid block and read the most recent data into the EEPROM buffer located in RAM. The function returns 1 of successful, else 0.

- uint8_t eep_write(uint8_t* data, uint16_t add, uint16_t size)

  This function writes ‘size’ number of bytes of data pointed by ‘data’ at address ‘add’. This address is the logical address. For example, for 255 byte EEPROM, address range is 0 to 254. The function returns 1 of successful, else 0.

- uint8_t eep_read(uint8_t* data, uint16_t add, uint16_t size)

  This function reads ‘size’ number of bytes from emulated EEPROM into pointer ‘data’ starting at address ‘add’. This address is the logical address within the EEPROM. For example, for 255 byte EEPROM, address range is 0 to 254. The function returns 1 of successful, else 0.

This file use only two functions to access flash and are tightly bound to the embedded flash controller. These functions are:

- flash_erase()
- flash_write()

These functions link to ASF drivers for embedded flash controller access. To port this code to other devices, these functions will have to be replaced with appropriate counterparts for the target device.

3.2.4 translation.h

This file contains the prototype declarations for translation.c.

3.2.5 conf_nvm_example.h

This file is the configurations file for this project. It contains all the user configurable parameters as described below:

- EEPROM_SIZE: Selects the emulated EEPROM size. Can be 511, 255, or 127
- PAGE_SIZE: Page size for the selected device. For SAM4S, it is 512
- NUMBER_PAGES: Sets the number of pages to be used for circular buffer. This value is set at 16

The only parameter that can be safely changed without any code modification is EEPROM_SIZE. Other parameters may require some minor code modifications.

3.3 Steps to Build the Project

For IAR:

- Extract the zip file into local directory
- Go to NVM_EXAMPLE_IAR directory
- Open EEPROM_example.eww and compile the source

For Atmel Studio:

- Extract the zip file into local directory
- Go to NVM_EXAMPLE_AS directory
- Open NVM_EXAMPLE1.atsln and compile the source

4. Limitations and Future Scope

This application intends to match EEPROM usability but due to inherent physical differences between EEPROM and flash, it will never be possible to perfectly to do. One of the limitations that this application presents is that it puts a cap on the maximum write cycles in flash. Even updating one location in this emulated EEPROM would count towards one write cycle. In real EEPROMs, when one location is updated, it is counted as one erase cycle of that particular address while all other locations are untouched. That means in worst case, using this emulated driver, if user wants to update only one location out of available 511 bytes, after 160,000 writes, the flash will be worn out even though rest of the bytes were not touched.

There are three ways to overcome this limitation:

- This application can be extended to use any part of flash for a bigger circular buffer. It will require some code modifications but even 4K blocks of flash can be used as EEPROM
- The user must be careful in selecting the required EEPROM size. Even though this application provides user with flexibility to choose between 511, 255, and 127 bytes of EEPROM, same idea can be extended to implement 63 bytes of EEPROM. Every time the size of EEPROM is cut by 2, the endurance increases by a factor of 2
- The user must be frugal about the number writes done to EEPROM. By minimizing unnecessary writes, the endurance effectively can be increased. For example, if the system has a bulk capacitor to provide last gasp power, the system can be set up to write to EEPROM only during a power fail sequence. As long the system is running on power, a simple RAM buffer can be used for storing data

5. Revision History

Doc. Rev.	Date	Comments
42218A	12/2013	Initial document release

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation
1600 Technology Drive
San Jose, CA 95110
USA
Tel: (+1)(408) 441-0311
Fax: (+1)(408) 487-2600
www.atmel.com

Atmel Asia Limited
Unit 01-5 & 16, 19F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
Tel: (+852) 2245-6100
Fax: (+852) 2722-1369

Atmel Munich GmbH
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
Tel: (+49) 89-31970-0
Fax: (+49) 89-3194621

Atmel Japan G.K.
16F Shin-Osaki Kangyo Building
1-6-4 Osaki, Shinagawa-ku
Tokyo 141-0032
JAPAN
Tel: (+81)(3) 6417-0300
Fax: (+81)(3) 6417-0370