EEPROM Emulation For Flash Microcontrollers

1. Introduction

Non-volatile data storage is an important feature of many embedded systems. Dedicated, byte-writeable EEPROM devices are commonly used in such systems to store calibration constants and other parameters that may need to be updated periodically. These devices are typically accessed by an MCU in the system using a serial bus. This solution requires PCB real estate as well as I/O pins and serial bus resources on the MCU. Some cost efficiencies can be realized by using a small amount of the MCU's flash memory for the EEPROM storage. This note describes firmware designed to emulate EEPROM storage on Silicon Laboratories' flash-based C8051Fxxx MCUs. Figure 1 shows a map of the example firmware. The highlighted functions are the interface for the main application code.

2. Key Features

- Compile-Time Configurable Size: Between 4 and 255 bytes
- Portable: Works across C8051Fxxx device families and popular 8051 compilers
- Fault Tolerant: Resistant to corruption from power supply events and errant code
- Small Code Footprint: Less than 1 kB for interface functions + minimum two pages of Flash for data storage

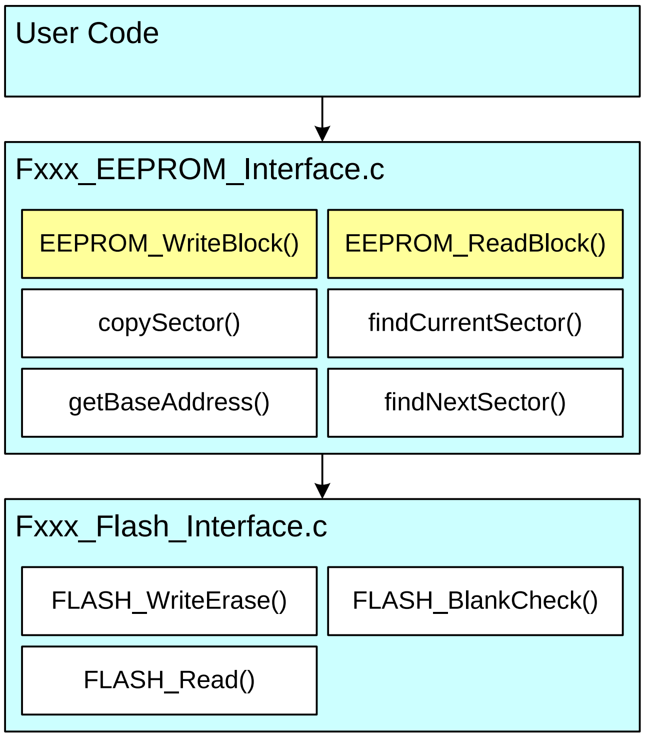


Figure 1. EEPROM Emulation Firmware

3. Basic Operation

A very simple example project and wrapper code is included with the source firmware. The example demonstrates how to set up a project with the appropriate files within the Silicon Labs IDE and how to call the EEPROM access functions from user code.

The EEPROM emulation firmware consists of two layers: an emulated EEPROM interface and a hardware interface layer. Application code calls into the EEPROM interface, which calls the hardware layer to access flash as needed. The file *Fxxx_EEPROM_Interface.c* defines the generic EEPROM interface, while *Fxxx_Flash_Interface.c* is the hardware flash access layer. Both files should be included in the project build.

The header file, *Fxxx_EEPROM_Interface.h*, exposes the EEPROM access functions to the application layer. The file, *Fxxx_EEPROM_Configuration.h*, can also be included at the application layer if any custom parameters need to be accessed by application code.

3.1. EEPROM Access

Two functions provide read/write access to the EEPROM area. The EEPROM_WriteBlock() function is used to write to the EEPROM, and the EEPROM_ReadBlock() function is used to read from the EEPROM area. These functions are described in detail in the following two sections.

3.1.1. EEPROM_WriteBlock()

Description:	This function writes a block of data to the EEPROM. Writes of any length between 1 and the specified EEPROM size are supported.
Prototype:	U8 EEPROM_WriteBlock (U8 address, U8 *dataBuf, U8 length)
Parameters:	address - The logical address in EEPROM space where the first byte will be written.
	*dataBuf - A generic pointer to the first byte of data to write into EEPROM space.
	length - The number of bytes to write into the EEPROM.
Return Value:	EE_NO_ERROR = Write was successful
	EE_WRITE_ERROR = This error indicates that either the specified address and length is outside of the EEPROM area or a Flash erase problem occurred.
	EE_SECTOR_ERROR = The current EEPROM sector could not be determined. This could indicate corruption in the emulated EEPROM space.

3.1.2. EEPROM_ReadBlock()

Description:	This function reads a block of data from the EEPROM. Reads of any length between 1 and the specified EEPROM size are supported.
Prototype:	U8 EEPROM_ReadBlock (U8 address, U8 *dataBuf, U8 length)
Parameters:	address - The logical address in EEPROM of the first byte to be read.
	*dataBuf - A generic pointer to a RAM location where data will be returned.
	length - The number of bytes to read from the EEPROM.
Return Value:	EE_NO_ERROR = Read was successful
	EE_READ_ERROR = This error indicates that the specified address and length is outside of the EEPROM area.
	EE_SECTOR_ERROR = The current EEPROM sector could not be determined. This could indicate corruption in the emulated EEPROM space.

4. Configuring the EEPROM and Device Parameters

The file, *Fxxx_Flash_Parameters.h*, defines several macros used by the hardware layer. These macros are dependent on which device is being used. The first element that needs to be customized is the device name, located in *Fxxx_Flash_Parameters.h*. This is not a named parameter, but a stand-alone definition. The device to be used should be specified on a line of code that looks like the following:

```
#define C8051F310
```

The target device name should be used without any suffix. For example, C8051F311 or C8051F122 are valid here, while C8051F300-GM is not. Allowable values are specified in the comments field near the definition. An error will be generated by the compiler if the device name does not match a known value.

The file *Fxxx_EEPROM_Configuration.h* defines a handful of parameters that can be configured by the user. These constants are used at compile time to determine the characteristics of the emulated EEPROM. the following is a list of these parameters and what they configure:

- EE_BASE_ADDR: This parameter specifies the EEPROM base address in flash memory. This is a 16-bit constant that should be defined as the first (lowest) address byte of the first page in flash memory where the EEPROM data will be stored. For devices which have more than 64 kB of flash memory, the firmware assumes that this address specifies a location in the highest code bank available on that device. Please refer to the specific device datasheet for details on banked code storage.
- FL_PAGES: This parameter specifies the number of flash pages to use for EEPROM data storage. Typically, this value will be set to its minimum value of 2.
- EE_SIZE: This parameter specifies the number of data bytes available in the emulated EEPROM. The minimum allowable size of the EEPROM depends directly on the amount of flash reserved for EEPROM data. A compiler error will be generated if the specified size is too small. The maximum value for this parameter is 255.
- FL_ERASE_LIMIT: This parameter specifies how many times the EEPROM code will retry a flash page erase before it generates an error. This parameter should typically be left at 1.
- RSTSRC_VAL: This parameter allows the user to specify the value that is written to RSTSRC in the low-level flash routines to enable the desired reset sources for the application. It will be ORed with 0x02 in all cases to enable the on-chip VDD monitor as a reset source.

5. Detailed Functionality

This section contains auxiliary information for those who wish to modify or extend the example firmware or those who simply want a deeper understanding of the internal workings. The method of EEPROM emulation is discussed, as are data integrity and portability considerations.

5.1. EEPROM Emulation

On the surface, the main difference between Flash memory and EEPROM memory is the size of an individually-modifiable area. On most EEPROM devices, memory can be modified one byte at a time. However, on flash devices, memory is typically organized into “pages” consisting of a larger number of bytes. These pages must be erased (writing 1s to all bits) one full page at a time, while individual bits can be written to zero. To emulate byte-writeable EEPROM memory in a Flash device, it is necessary to perform a read-modify-write operation. To ensure data integrity, at least two pages of Flash memory must be defined as “EEPROM space” so that a valid EEPROM image is always present in the device.

The approach to EEPROM emulation presented here uses two or more pages of Flash, divided into sub-sectors, each consisting of data and an informative “tag”. The number of flash pages used and the number of sectors per flash page are determined by the configuration information defined in *Fxxx_EEPROM_Configuration.h*. A memory map of the resulting EEPROM storage area is shown in Figure 2.

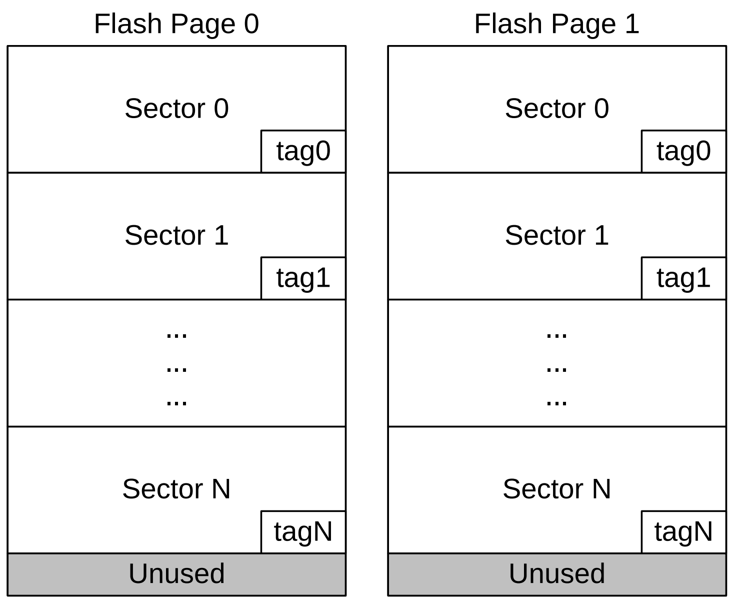


Figure 2. EEPROM Data Storage Area

When a write of EEPROM memory is requested, the firmware first determines what the current and next available sectors are and whether the next sector is on a page that needs to be erased. An erase is initiated if necessary, and then the information is copied from the current sector into the new sector, replacing the written addresses with the new data bytes. During this process, the sector tags are maintained so that the EEPROM routines will always know which is the most current valid sector, and read the appropriate data on request.

5.2. Sector Tags and Data Integrity

Sector tags are an additional piece of information written to each EEPROM sector. They provide a means of ensuring data integrity and identifying the most recent valid sector. After a page of flash is erased, the sector tags on that page will all read 0xFF. The value, 0xFF, indicates that the sector is unwritten. When data is written to a new sector, a specific sequence of writes to the sector tag and the data area provide a level of data integrity. The primary goal of the sequence is to prevent an interrupted EEPROM write from later being considered valid by the EEPROM access routines. If a write is interrupted before it completes, no data will be lost as a result. The sequence that ensures this is shown in Figure 3 and described in the list below.

1. Before any data is written to the new sector, the MSB of the sector tag is cleared to 0. The sector tag at this point becomes 0x7F, which is considered a "work in progress" and not valid.
2. Data is then read from the current sector, modified where indicated, and written to the new sector.
3. Once all data has been written to the sector, the seven LSBs of the sector tag are updated with the previous sector tag value, plus one. These bits will wrap after 0x7E and begin again at 0x01.
4. Upon a page erase, the sector tag returns to 0xFF.

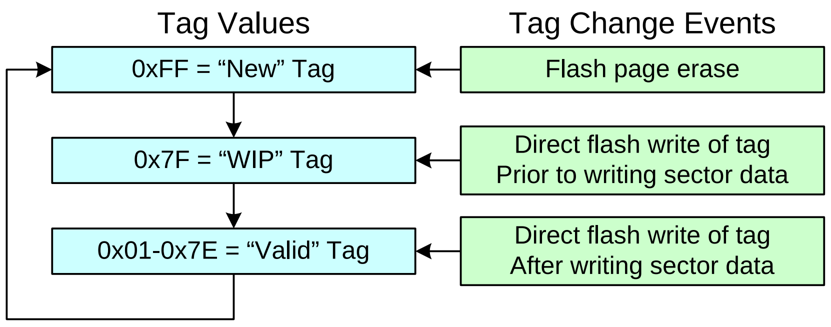

Tag Values
Tag Change Events
0xFF = "New" Tag
Flash page erase
0x7F = "WIP" Tag
Direct flash write of tag Prior to writing sector data
0x01-0x7E = "Valid" Tag
Direct flash write of tag After writing sector data

Figure 3. Sector Tag Progression

Two functions are responsible for interpreting these sector tags and providing the EEPROM read and write routines with the correct page and sector pointers.

The function, findCurrentSector(), is used to determine which sector holds the most recent valid copy of the EEPROM data. Sector tag values between 0x01 and 0x7E are considered to be valid, and if more than one valid tag is found, the routine will look for a sequence of values. The most recent sequential value denotes the latest valid EEPROM data sector.

The function, findNextSector(), is used to determine where the next empty sector is. It does so by searching for the next sector with an empty (0xFF) tag. If no empty tag is found, the "next" sector is considered to be the first sector of the next flash page, and the findNextSector() function will indicate that the page needs to be erased before it can be used.

5.3. Flash Interface

The Flash interface is the only piece of the example firmware that directly drives the hardware. There are a variety of different capabilities and flash interface mechanisms across the C8051Fxxx device families. The file, *Fxxx_Flash_Parameters.h*, defines the differences as a set of macros for each device. These macros are used in the low-level flash access routines defined in *Fxxx_Flash_Interface.c* to provide hardware-appropriate code. If the example code is to be modified, it may be helpful to unroll these macros for the specific target device to aid in debugging. Not all macros are used on all devices. When a macro is not needed, it is defined as white space. The defined macros are listed below:

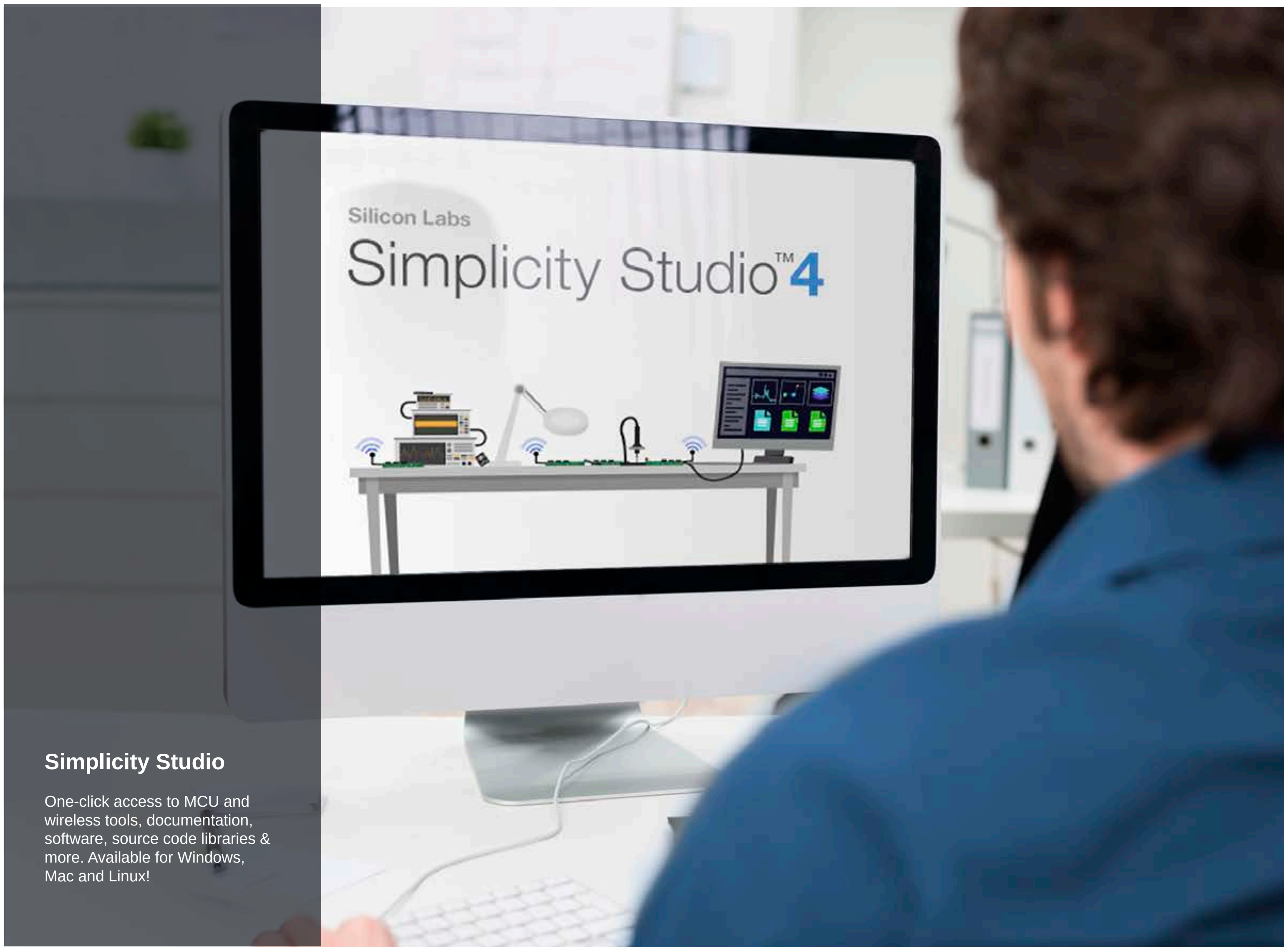
- ENABLE_VDDMON(): Enables the supply monitor for the device.
- DISABLE_WDT(): Disables the watchdog timer.
- SFRPAGE_SWITCH(): Saves the current SFRPAGE and changes to the SFR page containing the flash access registers.
- SFRPAGE_RESTORE(): Restores the saved SFRPAGE setting.
- PSBANK_STORE(): Saves the current PSBANK setting.
- PSBANK_SWITCH(): Switches PSBANK to the highest bank available on the device.
- PSBANK_RESTORE(): Restores the saved PSBANK setting.
- ENABLE_FL_MOD(): Writes the sequence to enable flash modification. On devices without a hardware lock-and-key mechanism, a software key mechanism is implemented by this macro.
- DISABLE_FL_MOD(): Writes the sequence to disable flash modification.
- FL_PROTECT(): Enables hardware flash protection when available, or resets device when not.

In addition to this list of macros, three other constants are defined for each device:

- FL_PAGE_SIZE: Defines the flash page size for the device family - either 512 or 1024 bytes.
- LOCK_PAGE: Defines the first address in the lock byte page. The example firmware will not allow any address above this to be used as EEPROM data storage.
- FLASH_SAFE_ADDR: Defines a “safe” address that will not have an effect when written or erased.

The flash interface firmware implements the low-level routines to read, write, and erase flash. From a fault-tolerance standpoint, the most important of these is the *FLASHWriteErase()* routine, which is responsible for modifying flash contents. The routine takes the following precautions to help prevent errant code execution from modifying the flash.

1. The VDD monitor is enabled within the routine. The VDD monitor should always be enabled at all times in any system that writes or erases flash memory from firmware. Doing so in this routine is an additional precaution in case the VDD monitor happens to be disabled when a flash modification is about to occur.
2. The flash hardware unlock sequence and the target address are stored in dedicated RAM locations. These RAM locations are reset to benign values after each write. The two flash keys are reset to 0x00, and the address is reset to the defined FLASH_SAFE_ADDR. Code must go through specific areas of the routine in sequence before the keys and address are set to the correct values.
3. In the case where an error is detected, the routine will attempt to lock out flash writes or reset the device. If the function was reached as the result of errant code, any further attempts will be minimized.



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com