AT03265: SAM D10/D11/D20/D21/R/L/C EEPROM Emulator (EEPROM) Service

APPLICATION NOTE

Introduction

This driver for Atmel® | SMART ARM®-based microcontrollers provides an emulated EEPROM memory space in the device's FLASH memory, for the storage and retrieval of user-application configuration data into and out of non-volatile memory.

The following peripherals are used by this module:

- NVM (Non-Volatile Memory Controller)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM C20/C21
- Atmel | SMART SAM DA1

The outline of this documentation is as follows:

Table of Contents

1. Software License

2. Prerequisites

The SAM device fuses must be configured via an external programmer or debugger, so that an EEPROM section is allocated in the main NVM flash memory contents. If a NVM section is not allocated for the EEPROM emulator, or if insufficient space for the emulator is reserved, the module will fail to initialize.

3. Module Overview

As the SAM devices do not contain any physical EEPROM memory, the storage of non-volatile user data is instead emulated using a special section of the device's main FLASH memory. The use of FLASH memory technology over EEPROM presents several difficulties over true EEPROM memory; data must be written as a number of physical memory pages (of several bytes each) rather than being individually byte addressable, and entire rows of FLASH must be erased before new data may be stored. To help abstract these characteristics away from the user application an emulation scheme is implemented to present a more user-friendly API for data storage and retrieval.

This module provides an EEPROM emulation layer on top of the device's internal NVM controller, to provide a standard interface for the reading and writing of non-volatile configuration data. This data is placed into the EEPROM emulated section of the device's main FLASH memory storage section, the size of which is configured using the device's fuses. Emulated EEPROM is exempt from the usual device NVM region lock bits, so that it may be read from or written to at any point in the user application.

There are many different algorithms that may be employed for EEPROM emulation using FLASH memory, to tune the write and read latencies, RAM usage, wear levelling and other characteristics. As a result, multiple different emulator schemes may be implemented, so that the most appropriate scheme for a specific application's requirements may be used.

3.1. Implementation Details

The following information is relevant for **EEPROM Emulator scheme 1, version 1.0.0**, as implemented by this module. Other revisions or emulation schemes may vary in their implementation details and may have different wear-leveling, latency, and other characteristics.

3.1.1. Emulator Characteristics

This emulator is designed for **best reliability, with a good balance of available storage and write-cycle limits**. It is designed to ensure that page data is automatically updated so that in the event of a failed update the previous data is not lost (when used correctly). With the exception of a system reset with data cached to the internal write-cache buffer, at most only the latest write to physical non-volatile memory will be lost in the event of a failed write.

This emulator scheme is tuned to give best write-cycle longevity when writes are confined to the same logical EEPROM page (where possible) and when writes across multiple logical EEPROM pages are made in a linear fashion through the entire emulated EEPROM space.

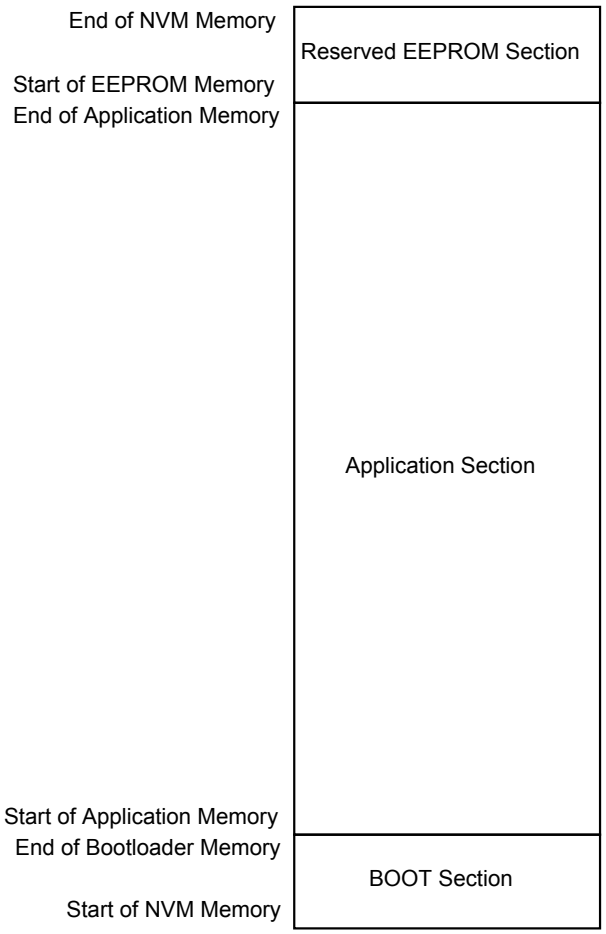
3.1.2. Physical Memory

The SAM non-volatile FLASH is divided into a number of physical rows, each containing four identically sized flash pages. Pages may be read or written to individually, however pages must be erased before being reprogrammed and the smallest granularity available for erasure is one single row.

This discrepancy results in the need for an emulator scheme that is able to handle the versioning and moving of page data to different physical rows as needed, erasing old rows ready for re-use by future page write operations.

Physically, the emulated EEPROM segment is located at the end of the physical FLASH memory space, as shown in Figure 3-1 Physical Memory on page 7.

Figure 3-1 Physical Memory



3.1.3. Master Row

One physical FLASH row at the end of the emulated EEPROM memory space is reserved for use by the emulator to store configuration data. The master row is not user-accessible, and is reserved solely for internal use by the emulator.

3.1.4. Spare Row

As data needs to be preserved between row erasures, a single FLASH row is kept unused to act as destination for copied data when a write request is made to an already full row. When the write request is made, any logical pages of data in the full row that need to be preserved are written to the spare row along with the new (updated) logical page data, before the old row is erased and marked as the new spare.

3.1.5. Row Contents

Each physical FLASH row initially stores the contents of two logical EEPROM memory pages. This halves the available storage space for the emulated EEPROM but reduces the overall number of row erases that are required, by reserving two pages within each row for updated versions of the logical page contents. See Figure 3-3 Initial Physical Layout of The Emulated EEPROM Memory on page 8 for a visual layout of the EEPROM Emulator physical memory.

As logical pages within a physical row are updated, the new data is filled into the remaining unused pages in the row. Once the entire row is full, a new write request will copy the logical page not being written to in the current row to the spare row with the new (updated) logical page data, before the old row is erased.

This system allows for the same logical page to be updated up to three times into physical memory before a row erasure procedure is needed. In the case of multiple versions of the same logical EEPROM page being stored in the same physical row, the right-most (highest physical FLASH memory page address) version is considered to be the most current.

3.1.6. Write Cache

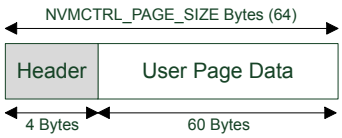
As a typical EEPROM use case is to write to multiple sections of the same EEPROM page sequentially, the emulator is optimized with a single logical EEPROM page write cache to buffer writes before they are written to the physical backing memory store. The cache is automatically committed when a new write request to a different logical EEPROM memory page is requested, or when the user manually commits the write cache.

Without the write cache, each write request to an EEPROM memory page would require a full page write, reducing the system performance and significantly reducing the lifespan of the non-volatile memory.

3.2. Memory Layout

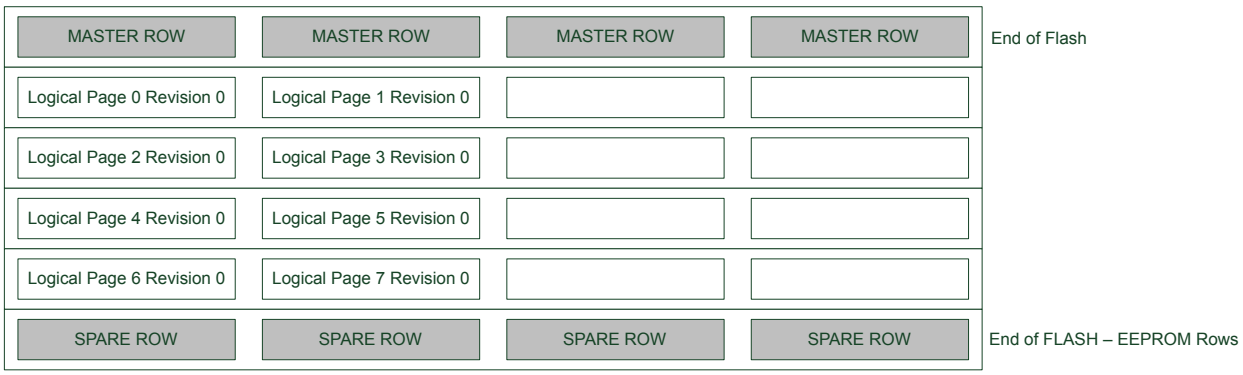
A single logical EEPROM page is physically stored as the page contents and a header inside a single physical FLASH page, as shown in Figure 3-2 Internal Layout of An Emulated EEPROM Page on page 8.

Figure 3-2 Internal Layout of An Emulated EEPROM Page



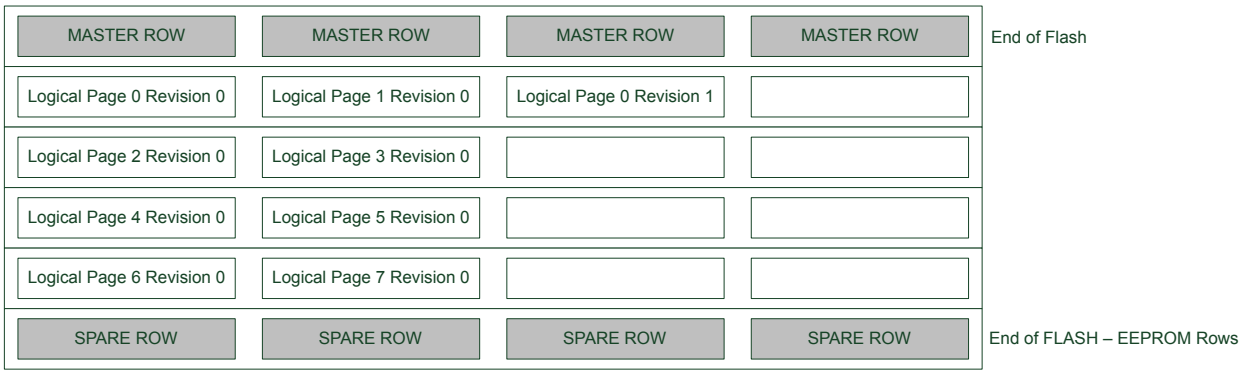
Within the EEPROM memory reservation section at the top of the NVM memory space, this emulator will produce the layout as shown in Figure 3-3 Initial Physical Layout of The Emulated EEPROM Memory on page 8 when initialized for the first time.

Figure 3-3 Initial Physical Layout of The Emulated EEPROM Memory



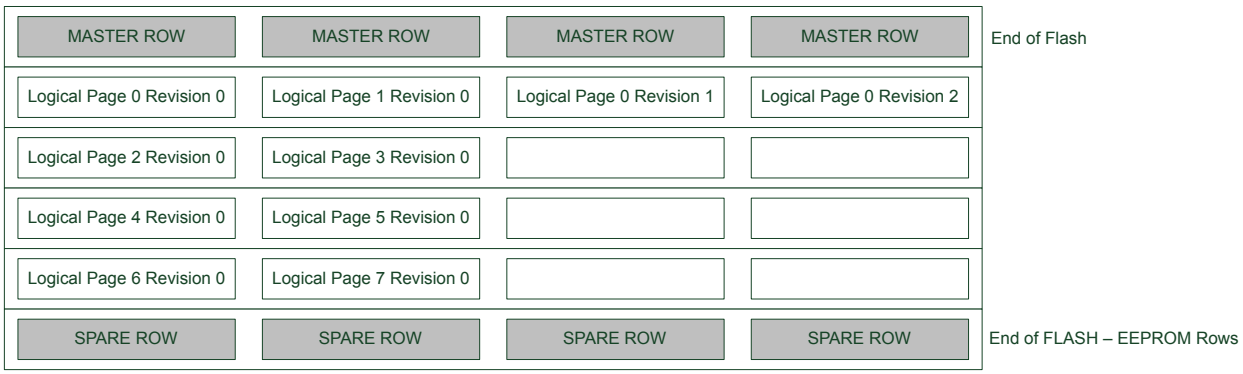
When an EEPROM page needs to be committed to physical memory, the next free FLASH page in the same row will be chosen - this makes recovery simple, as the right-most version of a logical page in a row is considered the most current. With four pages to a physical NVM row, this allows for up to three updates to the same logical page to be made before an erase is needed. Figure 3-4 First Write to Logical EEPROM Page N-1 on page 9 shows the result of the user writing an updated version of logical EEPROM page `N-1` to the physical memory.

Figure 3-4 First Write to Logical EEPROM Page N-1



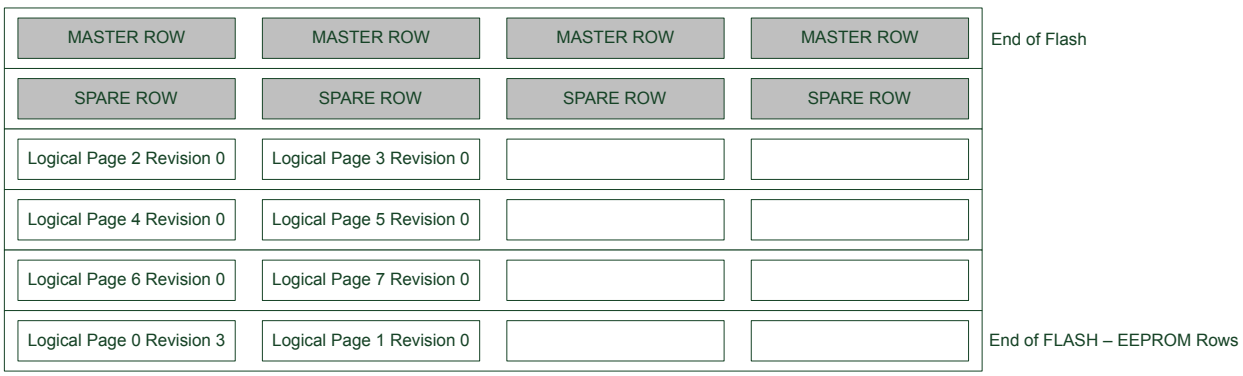
A second write of the same logical EEPROM page results in the layout shown in Figure 3-5 Second Write to Logical EEPROM Page N-1 on page 9.

Figure 3-5 Second Write to Logical EEPROM Page N-1



A third write of the same logical page requires that the EEPROM emulator erase the row, as it has become full. Prior to this, the contents of the unmodified page in the same row as the page being updated will be copied into the spare row, along with the new version of the page being updated. The old (full) row is then erased, resulting in the layout shown in Figure 3-6 Third Write to Logical EEPROM Page N-1 on page 9.

Figure 3-6 Third Write to Logical EEPROM Page N-1



4. Special Considerations

4.1. NVM Controller Configuration

The EEPROM Emulator service will initialize the NVM controller as part of its own initialization routine; the NVM controller will be placed in Manual Write mode, so that explicit write commands must be sent to the controller to commit a buffered page to physical memory. The manual write command must thus be issued to the NVM controller whenever the user application wishes to write to a NVM page for its own purposes.

4.2. Logical EEPROM Page Size

As a small amount of information needs to be stored in a header before the contents of a logical EEPROM page in memory (for use by the emulation service), the available data in each EEPROM page is less than the total size of a single NVM memory page by several bytes.

4.3. Committing of the Write Cache

A single-page write cache is used internally to buffer data written to pages in order to reduce the number of physical writes required to store the user data, and to preserve the physical memory lifespan. As a result, it is important that the write cache is committed to physical memory **as soon as possible after a BOD low power condition**, to ensure that enough power is available to guarantee a completed write so that no data is lost.

The write cache must also be manually committed to physical memory if the user application is to perform any NVM operations using the NVM controller directly.

5. Extra Information

For extra information, see Extra Information. This includes:

- Acronyms
- Dependencies
- Errata
- Module History

6. Examples

For a list of examples related to this driver, see Examples for Emulated EEPROM Service.

7. API Overview

7.1. Structure Definitions

7.1.1. Struct eeprom_emulator_parameters

Structure containing the memory layout parameters of the EEPROM emulator module.

Table 7-1 Members

Type	Name	Description
uint16_t	eeprom_number_of_pages	Number of emulated pages of EEPROM.
uint8_t	page_size	Number of bytes per emulated EEPROM page.

7.2. Macro Definitions

7.2.1. EEPROM Emulator Information

7.2.1.1. Macro EEPROM_EMULATOR_ID

```
#define EEPROM_EMULATOR_ID
```

Emulator scheme ID, identifying the scheme used to emulated EEPROM storage.

7.2.1.2. Macro EEPROM_MAJOR_VERSION

```
#define EEPROM_MAJOR_VERSION
```

Emulator major version number, identifying the emulator major version.

7.2.1.3. Macro EEPROM_MINOR_VERSION

```
#define EEPROM_MINOR_VERSION
```

Emulator minor version number, identifying the emulator minor version.

7.2.1.4. Macro EEPROM_REVISION

```
#define EEPROM_REVISION
```

Emulator revision version number, identifying the emulator revision.

7.2.1.5. Macro EEPROM_PAGE_SIZE

```
#define EEPROM_PAGE_SIZE
```

Size of the user data portion of each logical EEPROM page, in bytes.

7.3. Function Definitions

7.3.1. Configuration and Initialization

7.3.1.1. Function eeprom_emulator_init()

Initializes the EEPROM Emulator service.

```
enum status_code eeprom_emulator_init( void )
```

Initializes the emulated EEPROM memory space; if the emulated EEPROM memory has not been previously initialized, it will need to be explicitly formatted via eeprom_emulator_erase_memory(). The EEPROM memory space will **not** be automatically erased by the initialization function, so that partial data may be recovered by the user application manually if the service is unable to initialize successfully.

Returns
Status code indicating the status of the operation.

Table 7-2 Return Values

Return value	Description
STATUS_OK	EEPROM emulation service was successfully initialized
STATUS_ERR_NO_MEMORY	No EEPROM section has been allocated in the device
STATUS_ERR_BAD_FORMAT	Emulated EEPROM memory is corrupt or not formatted
STATUS_ERR_IO	EEPROM data is incompatible with this version or scheme of the EEPROM emulator

7.3.1.2. Function eeprom_emulator_erase_memory()

Erases the entire emulated EEPROM memory space.

```
void eeprom_emulator_erase_memory( void )
```

Erases and re-initializes the emulated EEPROM memory space, destroying any existing data.

7.3.1.3. Function eeprom_emulator_get_parameters()

Retrieves the parameters of the EEPROM Emulator memory layout.

```
enum status_code eeprom_emulator_get_parameters(
        struct eeprom_emulator_parameters *const parameters)
```

Retrieves the configuration parameters of the EEPROM Emulator, after it has been initialized.

Table 7-3 Parameters

Data direction	Parameter name	Description
[out]	parameters	EEPROM Emulator parameter struct to fill

Returns
Status of the operation.

Table 7-4 Return Values

Return value	Description
STATUS_OK	If the emulator parameters were retrieved successfully
STATUS_ERR_NOT_INITIALIZED	If the EEPROM Emulator is not initialized

7.3.2. Logical EEPROM Page Reading/Writing

7.3.2.1. Function eeprom_emulator_commit_page_buffer()

Commits any cached data to physical non-volatile memory.

```
enum status_code eeprom_emulator_commit_page_buffer( void )
```

Commits the internal SRAM caches to physical non-volatile memory, to ensure that any outstanding cached data is preserved. This function should be called prior to a system reset or shutdown to prevent data loss.

Note: This should be the first function executed in a BOD33 Early Warning callback to ensure that any outstanding cache data is fully written to prevent data loss.

Note: This function should also be called before using the NVM controller directly in the user-application for any other purposes to prevent data loss.

Returns
Status code indicating the status of the operation.

7.3.2.2. Function eeprom_emulator_write_page()

Writes a page of data to an emulated EEPROM memory page.

```
enum status_code eeprom_emulator_write_page(
        const uint8_t logical_page,
        const uint8_t *const data)
```

Writes an emulated EEPROM page of data to the emulated EEPROM memory space.

Note: Data stored in pages may be cached in volatile RAM memory; to commit any cached data to physical non-volatile memory, the eeprom_emulator_commit_page_buffer() function should be called.

Table 7-5 Parameters

Data direction	Parameter name	Description
[in]	logical_page	Logical EEPROM page number to write to
[in]	data	Pointer to the data buffer containing source data to write

Returns
Status code indicating the status of the operation.

Table 7-6 Return Values

Return value	Description
STATUS_OK	If the page was successfully read
STATUS_ERR_NOT_INITIALIZED	If the EEPROM emulator is not initialized
STATUS_ERR_BAD_ADDRESS	If an address outside the valid emulated EEPROM memory space was supplied

7.3.2.3. Function eeprom_emulator_read_page()

Reads a page of data from an emulated EEPROM memory page.

```
enum status_code eeprom_emulator_read_page(
        const uint8_t logical_page,
        uint8_t *const data)
```

Reads an emulated EEPROM page of data from the emulated EEPROM memory space.

Table 7-7 Parameters

Data direction	Parameter name	Description
[in]	logical_page	Logical EEPROM page number to read from
[out]	data	Pointer to the destination data buffer to fill

Returns
Status code indicating the status of the operation.

Table 7-8 Return Values

Return value	Description
STATUS_OK	If the page was successfully read
STATUS_ERR_NOT_INITIALIZED	If the EEPROM emulator is not initialized
STATUS_ERR_BAD_ADDRESS	If an address outside the valid emulated EEPROM memory space was supplied

7.3.3. Buffer EEPROM Reading/Writing

7.3.3.1. Function eeprom_emulator_write_buffer()

Writes a buffer of data to the emulated EEPROM memory space.

```
enum status_code eeprom_emulator_write_buffer(
        const uint16_t offset,
        const uint8_t *const data,
        const uint16_t length)
```

Writes a buffer of data to a section of emulated EEPROM memory space. The source buffer may be of any size, and the destination may lie outside of an emulated EEPROM page boundary.

Note: Data stored in pages may be cached in volatile RAM memory; to commit any cached data to physical non-volatile memory, the eeprom_emulator_commit_page_buffer() function should be called.

Table 7-9 Parameters

Data direction	Parameter name	Description
[in]	offset	Starting byte offset to write to, in emulated EEPROM memory space
[in]	data	Pointer to the data buffer containing source data to write
[in]	length	Length of the data to write, in bytes

Returns
Status code indicating the status of the operation.

Table 7-10 Return Values

Return value	Description
STATUS_OK	If the page was successfully read
STATUS_ERR_NOT_INITIALIZED	If the EEPROM emulator is not initialized
STATUS_ERR_BAD_ADDRESS	If an address outside the valid emulated EEPROM memory space was supplied

7.3.3.2. Function eeprom_emulator_read_buffer()

Reads a buffer of data from the emulated EEPROM memory space.

```
enum status_code eeprom_emulator_read_buffer(
        const uint16_t offset,
        uint8_t *const data,
        const uint16_t length)
```

Reads a buffer of data from a section of emulated EEPROM memory space. The destination buffer may be of any size, and the source may lie outside of an emulated EEPROM page boundary.

Table 7-11 Parameters

Data direction	Parameter name	Description
[in]	offset	Starting byte offset to read from, in emulated EEPROM memory space
[out]	data	Pointer to the data buffer containing source data to read
[in]	length	Length of the data to read, in bytes

Returns
Status code indicating the status of the operation.

Table 7-12 Return Values

Return value	Description
STATUS_OK	If the page was successfully read
STATUS_ERR_NOT_INITIALIZED	If the EEPROM emulator is not initialized
STATUS_ERR_BAD_ADDRESS	If an address outside the valid emulated EEPROM memory space was supplied

8. Extra Information

8.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
EEPROM	Electronically Erasable Read-Only Memory
NVM	Non-Volatile Memory

8.2. Dependencies

This driver has the following dependencies:

- Non-Volatile Memory Controller Driver

8.3. Errata

There are no errata related to this driver.

8.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Fix warnings
Initial Release

9. Examples for Emulated EEPROM Service

This is a list of the available Quick Start guides (QSGs) and example applications for SAM EEPROM Emulator (EEPROM) Service. QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that QSGs can be compiled as a standalone application or be added to the user application.

- Quick Start Guide for the Emulated EEPROM Module - Basic Use Case

9.1. Quick Start Guide for the Emulated EEPROM Module - Basic Use Case

In this use case, the EEPROM emulator module is configured and a sample page of data read and written. The first byte of the first EEPROM page is toggled, and a LED is turned on or off to reflect the new state. Each time the device is reset, the LED should toggle to a different state to indicate correct non-volatile storage and retrieval.

9.1.1. Prerequisites

The device's fuses must be configured to reserve a sufficient number of FLASH memory rows for use by the EEPROM emulator service, before the service can be used. That is:
`NVMCTRL_FUSES_EEPROM_SIZE` has to be set to less than 0x5 in the fuse setting, then there will be more than 8 pages size for EEPROM. Atmel Studio can be used to set this fuse(Tools->Device Programming).

9.1.2. Setup

9.1.2.1. Prerequisites

There are no special setup requirements for this use-case.

9.1.2.2. Code

Copy-paste the following setup code to your user application:

```
void configure_eeprom(void)
{
    /* Setup EEPROM emulator service */
    enum status_code error_code = eeprom_emulator_init();

    if (error_code == STATUS_ERR_NO_MEMORY) {
        while (true) {
            /* No EEPROM section has been set in the device's fuses */
        }
    }
    else if (error_code != STATUS_OK) {
        /* Erase the emulated EEPROM memory (assume it is unformatted or
         * irrecoverably corrupt) */
        eeprom_emulator_erase_memory();
        eeprom_emulator_init();
    }
}

#if (SAMD || SAMR21)
void SYSCTRL_Handler(void)
{
    if (SYSCTRL->INTFLAG.reg & SYSCTRL_INTFLAG_BOD33DET) {
        SYSCTRL->INTFLAG.reg |= SYSCTRL_INTFLAG_BOD33DET;
```

```
        eeprom_emulator_commit_page_buffer();
    }
}
#endif
static void configure_bod(void)
{
#if (SAMD || SAMR21)
    struct bod_config config_bod33;
    bod_get_config_defaults(&config_bod33);
    config_bod33.action = BOD_ACTION_INTERRUPT;
    /* BOD33 threshold level is about 3.2V */
    config_bod33.level = 48;
    bod_set_config(BOD_BOD33, &config_bod33);
    bod_enable(BOD_BOD33);

    SYSCTRL->INTENSET.reg |= SYSCTRL_INTENCLR_BOD33DET;
    system_interrupt_enable(SYSTEM_INTERRUPT_MODULE_SYSCTRL);
#endif

}
```

Add to user application initialization (typically the start of `main()`):

```
configure_eeprom();
```

9.1.2.3. Workflow

1. Attempt to initialize the EEPROM emulator service, storing the error code from the initialization function into a temporary variable.

   ```
   enum status_code error_code = eeprom_emulator_init();
   ```

2. Check if the emulator failed to initialize due to the device fuses not being configured to reserve enough of the main FLASH memory rows for emulated EEPROM usage - abort if the fuses are mis-configured.

   ```
   if (error_code == STATUS_ERR_NO_MEMORY) {
       while (true) {
           /* No EEPROM section has been set in the device's fuses */
       }
   }
   ```

3. Check if the emulator service failed to initialize for any other reason; if so assume the emulator physical memory is unformatted or corrupt and erase/re-try initialization.

   ```
   else if (error_code != STATUS_OK) {
       /* Erase the emulated EEPROM memory (assume it is unformatted or
        * irrecoverably corrupt) */
       eeprom_emulator_erase_memory();
       eeprom_emulator_init();
   }
   ```

   Config BOD to give an early warning, so that we could prevent data loss.

   ```
   configure_bod();
   ```

9.1.3. Use Case

9.1.3.1. Code

Copy-paste the following code to your user application:

```
uint8_t page_data[EEPROM_PAGE_SIZE];
eeprom_emulator_read_page(0, page_data);

page_data[0] = !page_data[0];
port_pin_set_output_level(LED_0_PIN, page_data[0]);

eeprom_emulator_write_page(0, page_data);
eeprom_emulator_commit_page_buffer();

page_data[1]=0x1;
eeprom_emulator_write_page(0, page_data);

while (true) {

}
```

9.1.3.2. Workflow

1. Create a buffer to hold a single emulated EEPROM page of memory, and read out logical EEPROM page zero into it.

   ```
   uint8_t page_data[EEPROM_PAGE_SIZE];
   eeprom_emulator_read_page(0, page_data);
   ```

2. Toggle the first byte of the read page.

   ```
   page_data[0] = !page_data[0];
   ```

3. Output the toggled LED state onto the board LED.

   ```
   port_pin_set_output_level(LED_0_PIN, page_data[0]);
   ```

4. Write the modified page back to logical EEPROM page zero, flushing the internal emulator write cache afterwards to ensure it is immediately written to physical non-volatile memory.

   ```
   eeprom_emulator_write_page(0, page_data);
   eeprom_emulator_commit_page_buffer();
   ```

5. Modify data and write back to logical EEPROM page zero. The data is not committed and should call `eeprom_emulator_commit_page_buffer` to ensure that any outstanding cache data is fully written to prevent data loss when detecting a BOD early warning.

   ```
   page_data[1]=0x1;
   eeprom_emulator_write_page(0, page_data);
   ```

10. Document Revision History

Doc. Rev.	Date	Comments
42125F	12/2015	Added support for SAM L22, SAM DA1, and SAM C20/C21
42125E	11/2014	Added support for SAM L21
42125D	09/2014	Added support for SAM R21, and SAM D10/D11
42125C	07/2014	Add support for SAM D21
42125B	11/2013	• ASF 3.13: Fixed bugs related to eeprom_emulator_write_buffer() and eeprom_emulator_read_buffer(). The functions now handle offsets that are multiples of 60. The length can now be smaller than one page without risking corruption. Addresses that are multiples of 60 will be written correctly • Updated module figures and re-worded the module overview. Corrected documentation typos
42125A	06/2013	Initial release

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**