SMART ARM-based Microcontrollers

AT03247: SAM D/R/L/C Non-Volatile Memory (NVM) Driver

APPLICATION NOTE

Introduction

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of non-volatile memories within the device, for partitioning, erasing, reading, and writing of data.

The following peripheral is used by this module:

- NVM (Non-Volatile Memory)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

Table of Contents

1. Software License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The name of Atmel may not be used to endorse or promote products derived from this software without specific prior written permission.

4. This software may only be redistributed and used in connection with an Atmel microcontroller product.

THIS SOFTWARE IS PROVIDED BY ATMEL "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE EXPRESSLY AND SPECIFICALLY DISCLAIMED. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

2. Prerequisites

There are no prerequisites for this module.

3. Module Overview

The Non-Volatile Memory (NVM) module provides an interface to the device's Non-Volatile Memory controller, so that memory pages can be written, read, erased, and reconfigured in a standardized manner.

3.1. Driver Feature Macro Definition

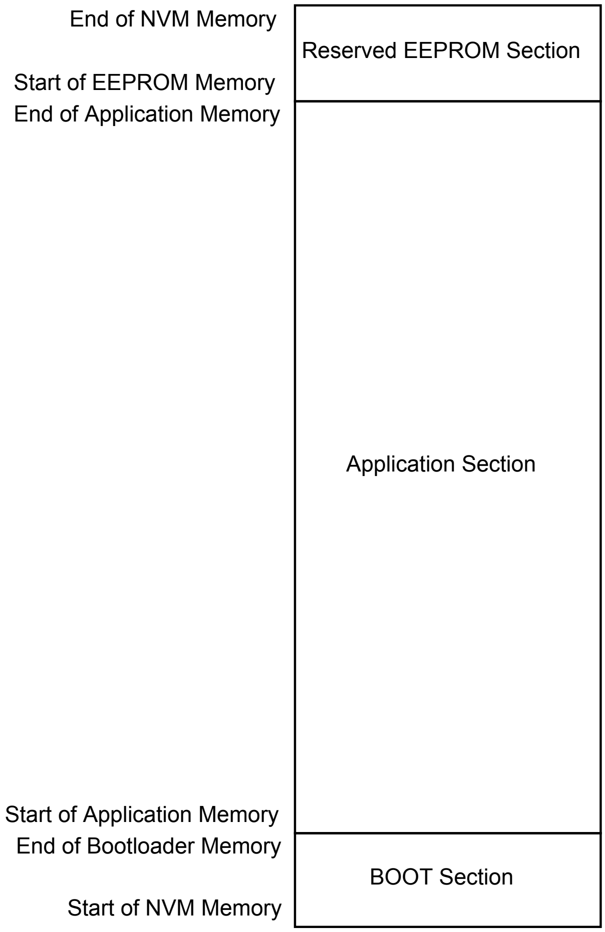
Driver feature macro	Supported devices
FEATURE_NVM_RWWEE	SAM L21/L22, SAM D21-64K, SAM DA1, SAM C20/C21
FEATURE_BOD12	SAM L21

Note: The specific features are only available in the driver when the selected device supports those features.

3.2. Memory Regions

The NVM memory space of the SAM devices is divided into two sections: a Main Array section, and an Auxiliary space section. The Main Array space can be configured to have an (emulated) EEPROM and/or boot loader section. The memory layout with the EEPROM and bootloader partitions is shown in Figure 3-1 Memory Regions on page 7.

Figure 3-1 Memory Regions



The Main Array is divided into rows and pages, where each row contains four pages. The size of each page may vary from 8-1024 bytes dependent of the device. Device specific parameters such as the page size and total number of pages in the NVM memory space are available via the nvm_get_parameters() function.

An NVM page number and address can be computed via the following equations:

$$PageNum = (RowNum \times 4) + PagePosInRow$$
$$PageAddr = PageNum \times PageSize$$

Figure 3-2 Memory Regions on page 7 shows an example of the memory page and address values associated with logical row 7 of the NVM memory space.

Figure 3-2 Memory Regions

Row 0x07	Page 0x1F	Page 0x1E	Page 0x1D	Page 0x1C
Address	0x7C0	0x780	0x740	0x700

3.3. Region Lock Bits

As mentioned in Memory Regions, the main block of the NVM memory is divided into a number of individually addressable pages. These pages are grouped into 16 equal sized regions, where each region can be locked separately issuing an NVM_COMMAND_LOCK_REGION command or by writing the LOCK bits in the User Row. Rows reserved for the EEPROM section are not affected by the lock bits or commands.

Note: By using the NVM_COMMAND_LOCK_REGION or NVM_COMMAND_UNLOCK_REGION commands the settings will remain in effect until the next device reset. By changing the default lock setting for the regions, the auxiliary space must to be written, however the adjusted configuration will not take effect until the next device reset.

Note: If the Security Bit is set, the auxiliary space cannot be written to. Clearing of the security bit can only be performed by a full chip erase.

3.4. Read/Write

Reading from the NVM memory can be performed using direct addressing into the NVM memory space, or by calling the nvm_read_buffer() function.

Writing to the NVM memory must be performed by the nvm_write_buffer() function - additionally, a manual page program command must be issued if the NVM controller is configured in manual page writing mode.

Before a page can be updated, the associated NVM memory row must be erased first via the nvm_erase_row() function. Writing to a non-erased page will result in corrupt data being stored in the NVM memory space.

4. Special Considerations

4.1. Page Erasure

The granularity of an erase is per row, while the granularity of a write is per page. Thus, if the user application is modifying only one page of a row, the remaining pages in the row must be buffered and the row erased, as an erase is mandatory before writing to a page.

4.2. Clocks

The user must ensure that the driver is configured with a proper number of wait states when the CPU is running at high frequencies.

4.3. Security Bit

The User Row in the Auxiliary Space cannot be read or written when the Security Bit is set. The Security Bit can be set by using passing NVM_COMMAND_SET_SECURITY_BIT to the nvm_execute_command() function, or it will be set if one tries to access a locked region. See Region Lock Bits.

The Security Bit can only be cleared by performing a chip erase.

5. Extra Information

For extra information, see Extra Information for NVM Driver. This includes:

- Acronyms
- Dependencies
- Errata
- Module History

6. Examples

For a list of examples related to this driver, see Examples for NVM Driver.

7. API Overview

7.1. Structure Definitions

7.1.1. Struct nvm_config

Configuration structure for the NVM controller within the device.

Table 7-1 Members

Type	Name	Description
enum nvm_cache_readmode	cache_readmode	Select the mode for how the cache will pre-fetch data from the flash
bool	disable_cache	Setting this to true will disable the pre-fetch cache in front of the NVM controller
bool	manual_page_write	Manual write mode; if enabled, pages loaded into the NVM buffer will not be written until a separate write command is issued. If disabled, writing to the last byte in the NVM page buffer will trigger an automatic write. **Note:** If a partial page is to be written, a manual write command must be executed in either mode.
enum nvm_sleep_power_mode	sleep_power_mode	Power reduction mode during device sleep
uint8_t	wait_states	Number of wait states to insert when reading from flash, to prevent invalid data from being read at high clock frequencies

7.1.2. Struct nvm_fusebits

This structure contain the layout of the first 64 bits of the user row which contain the fuse settings.

Table 7-2 Members

Type	Name	Description
enum nvm_bod12_action	bod12_action	BOD12 Action at power on
bool	bod12_enable	BOD12 Enable at power on
bool	bod12_hysteresis	
uint8_t	bod12_level	BOD12 Threshold level at power on

Type	Name	Description
enum nvm_bod33_action	bod33_action	BOD33 Action at power on
bool	bod33_enable	BOD33 Enable at power on
bool	bod33_hysteresis	
uint8_t	bod33_level	BOD33 Threshold level at power on
enum nvm_bootloader_size	bootloader_size	Bootloader size
enum nvm_eeprom_emulator_size	eeprom_size	EEPROM emulation area size
uint16_t	lockbits	NVM Lock bits
bool	wdt_always_on	WDT Always-on at power on
enum nvm_wdt_early_warning_offset	wdt_early_warning_offset	WDT Early warning interrupt time offset at power on
bool	wdt_enable	WDT Enable at power on
uint8_t	wdt_timeout_period	WDT Period at power on
bool	wdt_window_mode_enable_at_poweron	WDT Window mode enabled at power on
enum nvm_wdt_window_timeout	wdt_window_timeout	WDT Window mode time-out at power on

7.1.3. Struct nvm_parameters

Structure containing the memory layout parameters of the NVM module.

Table 7-3 Members

Type	Name	Description
uint32_t	bootloader_number_of_pages	Size of the Bootloader memory section configured in the NVM auxiliary memory space
uint32_t	eeprom_number_of_pages	Size of the emulated EEPROM memory section configured in the NVM auxiliary memory space
uint16_t	nvm_number_of_pages	Number of pages in the main array

Type	Name	Description
uint8_t	page_size	Number of bytes per page
uint16_t	rww_eeprom_number_of_pages	Number of pages in read while write EEPROM (RWWEE) emulation area

7.2. Macro Definitions

7.2.1. Driver Feature Definition

Define NVM features set according to the different device families.

7.2.1.1. Macro FEATURE_NVM_RWWEE

```
#define FEATURE_NVM_RWWEE
```

Read while write EEPROM emulation feature.

7.2.1.2. Macro FEATURE_BOD12

```
#define FEATURE_BOD12
```

Brown-out detector internal to the voltage regulator for VDDCORE.

7.3. Function Definitions

7.3.1. Configuration and Initialization

7.3.1.1. Function nvm_get_config_defaults()

Initializes an NVM controller configuration structure to defaults.

```
void nvm_get_config_defaults(
        struct nvm_config *const config)
```

Initializes a given NVM controller configuration structure to a set of known default values. This function should be called on all new instances of these configuration structures before being modified by the user application.

The default configuration is as follows:

- Power reduction mode enabled after sleep mode until first NVM access
- Automatic page write mode disabled
- Number of FLASH wait states left unchanged

Table 7-4 Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

7.3.1.2. Function nvm_set_config()

Sets the up the NVM hardware module based on the configuration.

```
enum status_code nvm_set_config(
        const struct nvm_config *const config)
```

Writes a given configuration of an NVM controller configuration to the hardware module, and initializes the internal device struct.

Table 7-5 Parameters

Data direction	Parameter name	Description
[in]	config	Configuration settings for the NVM controller

Note: The security bit must be cleared in order successfully use this function. This can only be done by a chip erase.

Returns
Status of the configuration procedure.

Table 7-6 Return Values

Return value	Description
STATUS_OK	If the initialization was a success
STATUS_BUSY	If the module was busy when the operation was attempted
STATUS_ERR_IO	If the security bit has been set, preventing the EEPROM and/or auxiliary space configuration from being altered

7.3.1.3. Function nvm_is_ready()

Checks if the NVM controller is ready to accept a new command.

```
bool nvm_is_ready( void )
```

Checks the NVM controller to determine if it is currently busy execution an operation, or ready for a new command.

Returns
Busy state of the NVM controller.

Table 7-7 Return Values

Return value	Description
true	If the hardware module is ready for a new command
false	If the hardware module is busy executing a command

7.3.2. NVM Access Management

7.3.2.1. Function nvm_get_parameters()

Reads the parameters of the NVM controller.

```
void nvm_get_parameters(
        struct nvm_parameters *const parameters)
```

Retrieves the page size, number of pages, and other configuration settings of the NVM region.

Table 7-8 Parameters

Data direction	Parameter name	Description
[out]	parameters	Parameter structure, which holds page size and number of pages in the NVM memory

7.3.2.2. Function nvm_write_buffer()

Writes a number of bytes to a page in the NVM memory region.

```
enum status_code nvm_write_buffer(
        const uint32_t destination_address,
        const uint8_t * buffer,
        uint16_t length)
```

Writes from a buffer to a given page address in the NVM memory.

Table 7-9 Parameters

Data direction	Parameter name	Description
[in]	destination_address	Destination page address to write to
[in]	buffer	Pointer to buffer where the data to write is stored
[in]	length	Number of bytes in the page to write

Note: If writing to a page that has previously been written to, the page's row should be erased (via nvm_erase_row()) before attempting to write new data to the page.

Note: For SAM D21 RWW devices, see `SAMD21_64K`, command `NVM_COMMAND_RWWEE_WRITE_PAGE` must be executed before any other commands after writing a page, refer to errata 13588.

Note: If manual write mode is enabled, the write command must be executed after this function, otherwise the data will not write to NVM from page buffer.

Returns
Status of the attempt to write a page.

Table 7-10 Return Values

Return value	Description
STATUS_OK	Requested NVM memory page was successfully read
STATUS_BUSY	NVM controller was busy when the operation was attempted

Return value	Description
STATUS_ERR_BAD_ADDRESS	The requested address was outside the acceptable range of the NVM memory region or not aligned to the start of a page
STATUS_ERR_INVALID_ARG	The supplied write length was invalid

7.3.2.3. Function nvm_read_buffer()

Reads a number of bytes from a page in the NVM memory region.

```
enum status_code nvm_read_buffer(
        const uint32_t source_address,
        uint8_t *const buffer,
        uint16_t length)
```

Reads a given number of bytes from a given page address in the NVM memory space into a buffer.

Table 7-11 Parameters

Data direction	Parameter name	Description
[in]	source_address	Source page address to read from
[out]	buffer	Pointer to a buffer where the content of the read page will be stored
[in]	length	Number of bytes in the page to read

Returns
Status of the page read attempt.

Table 7-12 Return Values

Return value	Description
STATUS_OK	Requested NVM memory page was successfully read
STATUS_BUSY	NVM controller was busy when the operation was attempted
STATUS_ERR_BAD_ADDRESS	The requested address was outside the acceptable range of the NVM memory region or not aligned to the start of a page
STATUS_ERR_INVALID_ARG	The supplied read length was invalid

7.3.2.4. Function nvm_update_buffer()

Updates an arbitrary section of a page with new data.

```
enum status_code nvm_update_buffer(
        const uint32_t destination_address,
        uint8_t *const buffer,
        uint16_t offset,
        uint16_t length)
```

Writes from a buffer to a given page in the NVM memory, retaining any unmodified data already stored in the page.

Note: If manual write mode is enable, the write command must be executed after this function, otherwise the data will not write to NVM from page buffer.

Warning This routine is unsafe if data integrity is critical; a system reset during the update process will result in up to one row of data being lost. If corruption must be avoided in all circumstances (including power loss or system reset) this function should not be used.

Table 7-13 Parameters

Data direction	Parameter name	Description
[in]	destination_address	Destination page address to write to
[in]	buffer	Pointer to buffer where the data to write is stored
[in]	offset	Number of bytes to offset the data write in the page
[in]	length	Number of bytes in the page to update

Returns
Status of the attempt to update a page.

Table 7-14 Return Values

Return value	Description
STATUS_OK	Requested NVM memory page was successfully read
STATUS_BUSY	NVM controller was busy when the operation was attempted
STATUS_ERR_BAD_ADDRESS	The requested address was outside the acceptable range of the NVM memory region
STATUS_ERR_INVALID_ARG	The supplied length and offset was invalid

7.3.2.5. Function nvm_erase_row()

Erases a row in the NVM memory space.

```
enum status_code nvm_erase_row(
        const uint32_t row_address)
```

Erases a given row in the NVM memory region.

Table 7-15 Parameters

Data direction	Parameter name	Description
[in]	row_address	Address of the row to erase

Returns
Status of the NVM row erase attempt.

Table 7-16 Return Values

Return value	Description
STATUS_OK	Requested NVM memory row was successfully erased
STATUS_BUSY	NVM controller was busy when the operation was attempted
STATUS_ERR_BAD_ADDRESS	The requested row address was outside the acceptable range of the NVM memory region or not aligned to the start of a row

7.3.2.6. Function nvm_execute_command()

Executes a command on the NVM controller.

```
enum status_code nvm_execute_command(
        const enum nvm_command command,
        const uint32_t address,
        const uint32_t parameter)
```

Executes an asynchronous command on the NVM controller, to perform a requested action such as an NVM page read or write operation.

Note: The function will return before the execution of the given command is completed.

Table 7-17 Parameters

Data direction	Parameter name	Description
[in]	command	Command to issue to the NVM controller
[in]	address	Address to pass to the NVM controller in NVM memory space
[in]	parameter	Parameter to pass to the NVM controller, not used for this driver

Returns
Status of the attempt to execute a command.

Table 7-18 Return Values

Return value	Description
STATUS_OK	If the command was accepted and execution is now in progress
STATUS_BUSY	If the NVM controller was already busy executing a command when the new command was issued
STATUS_ERR_IO	If the command was invalid due to memory or security locking
STATUS_ERR_INVALID_ARG	If the given command was invalid or unsupported
STATUS_ERR_BAD_ADDRESS	If the given address was invalid

7.3.2.7. Function nvm_get_fuses()

Get fuses from user row.

```
enum status_code nvm_get_fuses(
        struct nvm_fusebits * fusebits)
```

Read out the fuse settings from the user row.

Table 7-19 Parameters

Data direction	Parameter name	Description
[in]	fusebits	Pointer to a 64-bit wide memory buffer of type struct nvm_fusebits

Returns
Status of read fuses attempt.

Table 7-20 Return Values

Return value	Description
STATUS_OK	This function will always return STATUS_OK

7.3.2.8. Function nvm_set_fuses()

Set fuses from user row.

```
enum status_code nvm_set_fuses(
        struct nvm_fusebits * fb)
```

Set fuse settings from the user row.

Note: When writing to the user row, the values do not get loaded by the other modules on the device until a device reset occurs.

Table 7-21 Parameters

Data direction	Parameter name	Description
[in]	fusebits	Pointer to a 64-bit wide memory buffer of type struct nvm_fusebits

Returns
Status of read fuses attempt.

Table 7-22 Return Values

Return value	Description
STATUS_OK	This function will always return STATUS_OK
STATUS_BUSY	If the NVM controller was already busy executing a command when the new command was issued
STATUS_ERR_IO	If the command was invalid due to memory or security locking

Return value	Description
STATUS_ERR_INVALID_ARG	If the given command was invalid or unsupported
STATUS_ERR_BAD_ADDRESS	If the given address was invalid

7.3.2.9. Function nvm_is_page_locked()

Checks whether the page region is locked.

```
bool nvm_is_page_locked(
        uint16_t page_number)
```

Extracts the region to which the given page belongs and checks whether that region is locked.

Table 7-23 Parameters

Data direction	Parameter name	Description
[in]	page_number	Page number to be checked

Returns
Page lock status.

Table 7-24 Return Values

Return value	Description
true	Page is locked
false	Page is not locked

7.3.2.10. Function nvm_get_error()

Retrieves the error code of the last issued NVM operation.

```
enum nvm_error nvm_get_error( void )
```

Retrieves the error code from the last executed NVM operation. Once retrieved, any error state flags in the controller are cleared.

Note: The nvm_is_ready() function is an exception. Thus, errors retrieved after running this function should be valid for the function executed before nvm_is_ready().

Returns
Error caused by the last NVM operation.

Table 7-25 Return Values

Return value	Description
NVM_ERROR_NONE	No error occurred in the last NVM operation
NVM_ERROR_LOCK	The last NVM operation attempted to access a locked region
NVM_ERROR_PROG	An invalid NVM command was issued

7.4. Enumeration Definitions

7.4.1. Enum nvm_bod12_action

What action should be triggered when BOD12 is detected.

Table 7-26 Members

Enum value	Description
NVM_BOD12_ACTION_NONE	No action
NVM_BOD12_ACTION_RESET	The BOD12 generates a reset
NVM_BOD12_ACTION_INTERRUPT	The BOD12 generates an interrupt

7.4.2. Enum nvm_bod33_action

What action should be triggered when BOD33 is detected.

Table 7-27 Members

Enum value	Description
NVM_BOD33_ACTION_NONE	No action
NVM_BOD33_ACTION_RESET	The BOD33 generates a reset
NVM_BOD33_ACTION_INTERRUPT	The BOD33 generates an interrupt

7.4.3. Enum nvm_bootloader_size

Available bootloader protection sizes in kilobytes.

Table 7-28 Members

Enum value	Description
NVM_BOOTLOADER_SIZE_128	Boot Loader Size is 32768 bytes
NVM_BOOTLOADER_SIZE_64	Boot Loader Size is 16384 bytes
NVM_BOOTLOADER_SIZE_32	Boot Loader Size is 8192 bytes
NVM_BOOTLOADER_SIZE_16	Boot Loader Size is 4096 bytes
NVM_BOOTLOADER_SIZE_8	Boot Loader Size is 2048 bytes
NVM_BOOTLOADER_SIZE_4	Boot Loader Size is 1024 bytes
NVM_BOOTLOADER_SIZE_2	Boot Loader Size is 512 bytes
NVM_BOOTLOADER_SIZE_0	Boot Loader Size is 0 bytes

7.4.4. Enum nvm_cache_readmode

Control how the NVM cache prefetch data from flash.

Table 7-29 Members

Enum value	Description
NVM_CACHE_READMODE_NO_MISS_PENALTY	The NVM Controller (cache system) does not insert wait states on a cache miss. Gives the best system performance.
NVM_CACHE_READMODE_LOW_POWER	Reduces power consumption of the cache system, but inserts a wait state each time there is a cache miss
NVM_CACHE_READMODE_DETERMINISTIC	The cache system ensures that a cache hit or miss takes the same amount of time, determined by the number of programmed flash wait states

7.4.5. Enum nvm_command

Table 7-30 Members

Enum value	Description
NVM_COMMAND_ERASE_ROW	Erases the addressed memory row
NVM_COMMAND_WRITE_PAGE	Write the contents of the page buffer to the addressed memory page
NVM_COMMAND_ERASE_AUX_ROW	Erases the addressed auxiliary memory row. **Note:** This command can only be given when the security bit is not set.
NVM_COMMAND_WRITE_AUX_ROW	Write the contents of the page buffer to the addressed auxiliary memory row. **Note:** This command can only be given when the security bit is not set.
NVM_COMMAND_LOCK_REGION	Locks the addressed memory region, preventing further modifications until the region is unlocked or the device is erased
NVM_COMMAND_UNLOCK_REGION	Unlocks the addressed memory region, allowing the region contents to be modified
NVM_COMMAND_PAGE_BUFFER_CLEAR	Clears the page buffer of the NVM controller, resetting the contents to all zero values
NVM_COMMAND_SET_SECURITY_BIT	Sets the device security bit, disallowing the changing of lock bits and auxiliary row data until a chip erase has been performed
NVM_COMMAND_ENTER_LOW_POWER_MODE	Enter power reduction mode in the NVM controller to reduce the power consumption of the system

Enum value	Description
NVM_COMMAND_EXIT_LOW_POWER_MODE	Exit power reduction mode in the NVM controller to allow other NVM commands to be issued
NVM_COMMAND_RWWEE_ERASE_ROW	Read while write (RWW) EEPROM area erase row
NVM_COMMAND_RWWEE_WRITE_PAGE	RWW EEPROM write page

7.4.6. Enum nvm_eeprom_emulator_size

Available space in flash dedicated for EEPROM emulator in bytes.

Table 7-31 Members

Enum value	Description
NVM_EEPROM_EMULATOR_SIZE_16384	EEPROM Size for EEPROM emulation is 16384 bytes
NVM_EEPROM_EMULATOR_SIZE_8192	EEPROM Size for EEPROM emulation is 8192 bytes
NVM_EEPROM_EMULATOR_SIZE_4096	EEPROM Size for EEPROM emulation is 4096 bytes
NVM_EEPROM_EMULATOR_SIZE_2048	EEPROM Size for EEPROM emulation is 2048 bytes
NVM_EEPROM_EMULATOR_SIZE_1024	EEPROM Size for EEPROM emulation is 1024 bytes
NVM_EEPROM_EMULATOR_SIZE_512	EEPROM Size for EEPROM emulation is 512 bytes
NVM_EEPROM_EMULATOR_SIZE_256	EEPROM Size for EEPROM emulation is 256 bytes
NVM_EEPROM_EMULATOR_SIZE_0	EEPROM Size for EEPROM emulation is 0 bytes

7.4.7. Enum nvm_error

Possible NVM controller error codes, which can be returned by the NVM controller after a command is issued.

Table 7-32 Members

Enum value	Description
NVM_ERROR_NONE	No errors
NVM_ERROR_LOCK	Lock error, a locked region was attempted accessed
NVM_ERROR_PROG	Program error, invalid command was executed

7.4.8. Enum nvm_sleep_power_mode

Power reduction modes of the NVM controller, to conserve power while the device is in sleep.

Table 7-33 Members

Enum value	Description
NVM_SLEEP_POWER_MODE_WAKEONACCESS	NVM controller exits low-power mode on first access after sleep
NVM_SLEEP_POWER_MODE_WAKEUPINSTANT	NVM controller exits low-power mode when the device exits sleep mode
NVM_SLEEP_POWER_MODE_ALWAYS_AWAKE	Power reduction mode in the NVM controller disabled

7.4.9. Enum nvm_wdt_early_warning_offset

This setting determine how many GCLK_WDT cycles before a watchdog time-out period an early warning interrupt should be triggered.

Table 7-34 Members

Enum value	Description
NVM_WDT_EARLY_WARNING_OFFSET_8	8 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_16	16 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_32	32 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_64	64 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_128	128 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_256	256 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_512	512 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_1024	1024 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_2048	2048 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_4096	4096 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_8192	8192 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_16384	16384 clock cycles

7.4.10. Enum nvm_wdt_window_timeout

Window mode time-out period in clock cycles.

Table 7-35 Members

Enum value	Description
NVM_WDT_WINDOW_TIMEOUT_PERIOD_8	8 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_16	16 clock cycles

Enum value	Description
NVM_WDT_WINDOW_TIMEOUT_PERIOD_32	32 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_64	64 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_128	128 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_256	256 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_512	512 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_1024	1024 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_2048	2048 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_4096	4096 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_8192	8192 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_16384	16384 clock cycles

8. Extra Information for NVM Driver

8.1. Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
NVM	Non-Volatile Memory
EEPROM	Electrically Erasable Programmable Read-Only Memory

8.2. Dependencies

This driver has the following dependencies:

- None

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
Removed BOD12 reference, removed nvm_set_fuses() API
Added functions to read/write fuse settings
Added support for NVM cache configuration
Updated initialization function to also enable the digital interface clock to the module if it is disabled
Initial Release

9. Examples for NVM Driver

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

- Quick Start Guide for NVM - Basic

9.1. Quick Start Guide for NVM - Basic

In this use case, the NVM module is configured for:

- Power reduction mode enabled after sleep mode until first NVM access
- Automatic page write commands issued to commit data as pages are written to the internal buffer
- Zero wait states when reading FLASH memory
- No memory space for the EEPROM
- No protected bootloader section

This use case sets up the NVM controller to write a page of data to flash, and then read it back into the same buffer.

9.1.1. Setup

9.1.1.1. Prerequisites

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

9.1.1.2. Code

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

```
void configure_nvm(void)
{
    struct nvm_config config_nvm;

    nvm_get_config_defaults(&config_nvm);

    config_nvm.manual_page_write = false;

    nvm_set_config(&config_nvm);
}
```

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

```
configure_nvm();
```

9.1.1.3. Workflow

1. Create an NVM module configuration struct, which can be filled out to adjust the configuration of the NVM controller.

   ```
   struct nvm_config config_nvm;
   ```

2. Initialize the NVM configuration struct with the module's default values.

   ```
   nvm_get_config_defaults(&config_nvm);
   ```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Enable automatic page write mode. The new data will be written to NVM automaticly.

```
config_nvm.manual_page_write = false;
```

Note: If automatic page write mode is disabled, the data will not write to NVM until the NVM write command has been invoked. For safe use of the NVM module, disable automatic page write mode and use write command to commit data is recommended.

4. Configure NVM controller with the created configuration struct settings.

```
nvm_set_config(&config_nvm);
```

9.1.2. Use Case

9.1.2.1. Code

Copy-paste the following code to your user application:

```
uint8_t page_buffer[NVMCTRL_PAGE_SIZE];

for (uint32_t i = 0; i < NVMCTRL_PAGE_SIZE; i++) {
    page_buffer[i] = i;
}

enum status_code error_code;

do
{
    error_code = nvm_erase_row(
            100 * NVMCTRL_ROW_PAGES * NVMCTRL_PAGE_SIZE);
} while (error_code == STATUS_BUSY);

do
{
    error_code = nvm_write_buffer(
            100 * NVMCTRL_ROW_PAGES * NVMCTRL_PAGE_SIZE,
            page_buffer, NVMCTRL_PAGE_SIZE);
} while (error_code == STATUS_BUSY);

do
{
    error_code = nvm_read_buffer(
            100 * NVMCTRL_ROW_PAGES * NVMCTRL_PAGE_SIZE,
            page_buffer, NVMCTRL_PAGE_SIZE);
} while (error_code == STATUS_BUSY);
```

9.1.2.2. Workflow

1. Set up a buffer, one NVM page in size, to hold data to read or write into NVM memory.

```
uint8_t page_buffer[NVMCTRL_PAGE_SIZE];
```

2. Fill the buffer with a pattern of data.

```
for (uint32_t i = 0; i < NVMCTRL_PAGE_SIZE; i++) {
    page_buffer[i] = i;
}
```

3. Create a variable to hold the error status from the called NVM functions.

```
enum status_code error_code;
```

4. Erase a page of NVM data. As the NVM could be busy initializing or completing a previous operation, a loop is used to retry the command while the NVM controller is busy.

```
do
{
    error_code = nvm_erase_row(
            100 * NVMCTRL_ROW_PAGES * NVMCTRL_PAGE_SIZE);
} while (error_code == STATUS_BUSY);
```

Note: This must be performed before writing new data into an NVM page.

5. Write the data buffer to the previously erased page of the NVM.

```
do
{
    error_code = nvm_write_buffer(
            100 * NVMCTRL_ROW_PAGES * NVMCTRL_PAGE_SIZE,
            page_buffer, NVMCTRL_PAGE_SIZE);
} while (error_code == STATUS_BUSY);
```

Note: The new data will be written to NVM memory automatically, as the NVM controller is configured in automatic page write mode.

6. Read back the written page of page from the NVM into the buffer.

```
do
{
    error_code = nvm_read_buffer(
            100 * NVMCTRL_ROW_PAGES * NVMCTRL_PAGE_SIZE,
            page_buffer, NVMCTRL_PAGE_SIZE);
} while (error_code == STATUS_BUSY);
```

10. Document Revision History

Doc. Rev.	Date	Comments
42114E	12/2015	Added support for SAM L21/L22, SAM C21, SAM D09, and SAM DA1
42114D	12/2014	Added support for SAM R21 and SAM D10/D11
42114C	01/2014	Added support for SAM D21
42114B	06/2013	Corrected documentation typos
42114A	06/2013	Initial document 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**

DISCLAIMER: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. EXCEPT AS SET FORTH IN THE ATMEL TERMS AND CONDITIONS OF SALES LOCATED ON THE ATMEL WEBSITE, ATMEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS AND PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and products descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

SAFETY-CRITICAL, MILITARY, AND AUTOMOTIVE APPLICATIONS DISCLAIMER: Atmel products are not designed for and will not be used in connection with any applications where the failure of such products would reasonably be expected to result in significant personal injury or death (“Safety-Critical Applications”) without an Atmel officer's specific written consent. Safety-Critical Applications include, without limitation, life support devices and systems, equipment or systems for the operation of nuclear facilities and weapons systems. Atmel products are not designed nor intended for use in military or aerospace applications or environments unless specifically designated by Atmel as military-grade. Atmel products are not designed nor intended for use in automotive applications unless specifically designated by Atmel as automotive-grade.