AVR1316: XMEGA Self-programming

8-bit AVR® Microcontrollers

Application Note

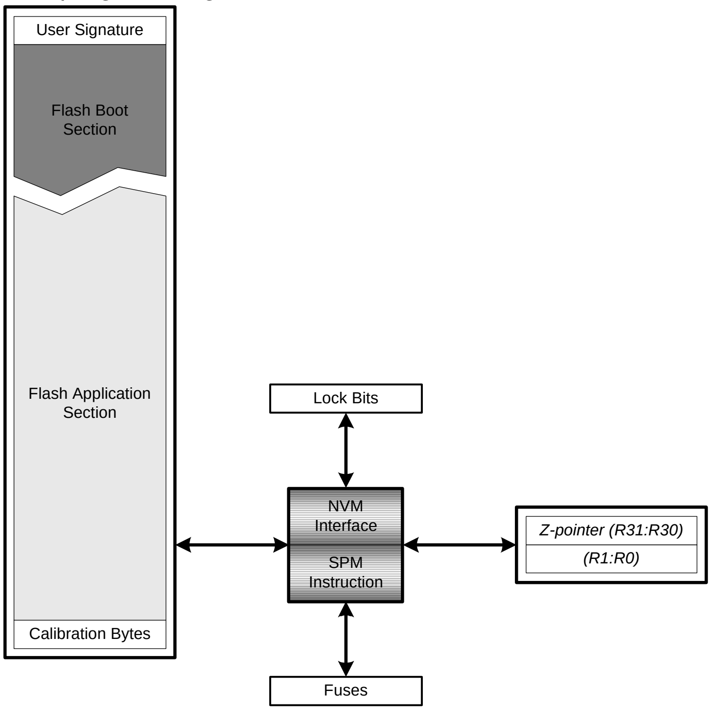
Features

- Software Access to Non-volatile Memories
- Read Calibration Byte
- Read Fuse Bytes
- Read and Write Lock Bits
- Erase, Read, and Write User Signature Row
- Erase, Read, and Write Application Section
- Erase, Read, and Write Boot Section
- CRC Generation for Application and Boot Section
- Optimized Assembly Implementation
- Driver Source Code Included

1 Introduction

This application note contains descriptions of the basic functionality of the XMEGA™ Self-programming feature and code examples to get up and running quickly. A driver interface written in assembly with a C interface is included as well.

Figure 1-1 Self-programming Overview



Rev. 8070B-AVR-11/08

2 Module Overview

This section provides an overview of the functionality and basic configuration options of the XMEGA Self-programming features.

2.1 The Non-volatile Memory Module

The XMEGA Flash memory, signature and calibration bytes, fuses and lock bits can all be accessed using the Non-volatile Memory (NVM) module. The NVM module is also used for access to EEPROM. Please refer to application note "AVR1315: Accessing the XMEGA EEPROM" or the device datasheet for more details.

As the NVM module serves several purposes, it is important to check that the NVM module is not busy with other operations (such as EEPROM update) before using it to access Flash memory etc. The *NVM Busy* bit (`NVMBUSY`) in the *NVM Status* register (`STATUS`) is set when the NVM module is busy.

Using the NVM module to perform Self-programming involves using *NVM Commands*. There are commands for erasing, writing, reading etc. Several commands require addresses or indexes to be set before issuing the command. Some commands also return values.

There are three different command types, depending on memory and access type. The command types are described in the next three sections. An overview of commands and their use can be found in Section 2.5 below.

2.1.1 NVM Action-based Commands

NVM Action-based commands are instructions where the *Command Execute* bit (`CMDEX`) in *NVM Control Register A* (`CTRLA`) has to be set in order to start an NVM Action. The *Command Execute* bit has to be set after the required control registers are set up. The NVM Action-based commands are used to access fuse bits, lock bits, EEPROM and to calculate CRC for the Flash memory sections.

The exact procedure is as follows:

1. Load address or index into the *Address* registers (`ADDRn`) if required.
2. Load data into the *Data* registers (`DATAn`) if required.
3. Load the command code into the *Command* register (`CMD`).
4. Load the *Protect IO Register* signature (byte value `0xD8`) into the *Configuration Change Protection* register (`CCP`). This will automatically disable all interrupts for the next four CPU instruction cycles.
5. Within the next four CPU instruction cycles, set the *Command Execute* bit (`CMDEX`) in *NVM Control Register A* (`CTRLA`).
6. The operation is finished when the *NVM Busy* bit (`NVMBUSY`) is cleared.
7. Results from the operation will be available in the *NVM Data* registers (`DATAn`).

The following operations use NVM Action-based commands:

- Reading fuse bytes
- Writing lock bits
- Generating CRC for Application Section
- Generating CRC for Boot Section

2.1.2 LPM-based Commands

LPM-based commands are commands where the LPM instruction is used. The LPM instruction is executed after the required control registers are set up. LPM-based commands are used for reading Flash memories. In XMEGA devices, the application code, calibration bytes, and signature bytes are located in Flash memory.

The exact procedure is as follows:

1. Load address or index into the Z pointer register (R31 and R30) if required.
2. Load the command code into the *Command* register (`CMD`).
3. Execute the LPM instruction.
4. The operation finishes immediately.
5. Results from the operation will be available in registers R0.

The following operations use LPM-based commands:

- Reading calibration bytes
- Reading user-accessible signature bytes
- Read Flash memory

2.1.3 SPM-based Commands

SPM-based commands are commands where the CPU executes the SPM instruction. The SPM instruction is executed after the required control registers are set up. SPM-based commands are used for erasing and writing Flash memories. In XMEGA devices, the application code, calibration bytes, and signature bytes are located in Flash memory.

The exact procedure is as follows:

1. Load address or index into the Z pointer register (R31 and R30) if required.
2. Load data into registers R1 and R0 if required.
3. Load the command code into the *Command* register (`CMD`).
4. Load the *Protect SPM Register* signature (byte value `0x9D`) into the *Configuration Change Protection* register (`CCP`). This will automatically disable all interrupts for the next four CPU instruction cycles.
5. Within the next four CPU instruction cycles, execute the SPM instruction.
6. The operation is finished when the *NVM Busy* bit (`NVMBUSY`) is cleared.

The following operations use SPM-based commands:

- Erasing user signature row
- Writing user signature row
- Erasing Flash memory
- Loading Flash page buffer
- Writing Flash memory
- Flushing Flash page buffer

2.2 Erasing and Writing Flash memory

There are two ways to update the Flash memory: *atomic write* and *split operation*. With atomic write, Flash locations are erased and written in one operation. With split operation, erasing and writing are separate operations.

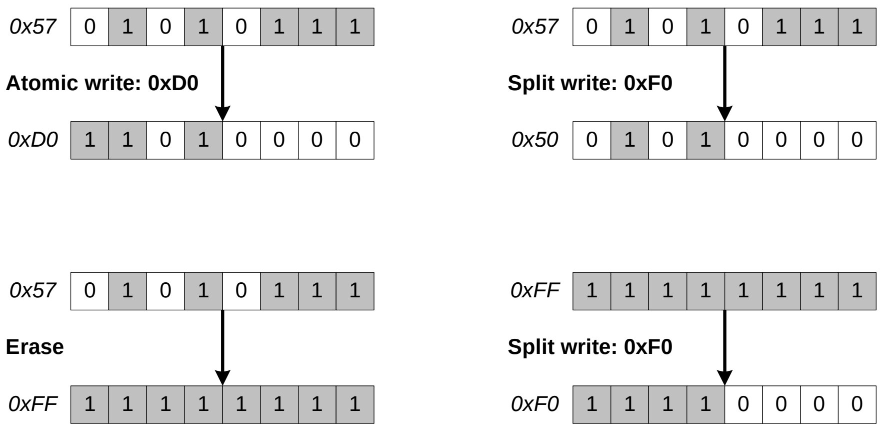
When erasing Flash locations, all bits are set to logic one. A split write can then program selected bits to logic zero. Split write operations cannot change a bit from zero to one, as opposed to an atomic write that first erase to logic one and then writes logic zeros to selected bits. This means that multiple writes to one location with different values, without erases in between, eventually results in all locations being logic zeros. This is similar to a logic AND operation between existing value and written value.

An atomic write takes approximately twice the time of one erase or one write. Therefore, split operation can be used to save time by erasing Flash locations in advance, e.g. during initialization. For instance, if the application needs to store vital data when a power drop is detected, a split write will take less time than an atomic write.

Another useful application for the split operation is to increase Flash memory endurance, especially for applications that update Flash locations frequently. By implementing a scheme where Flash locations are not erased unless they have to, the split operation feature will increase Flash endurance. This is similar to split and atomic write for EEPROM. For more details, refer to the application note “AVR101: High Endurance EEPROM Storage”.

Different erase and write operations, together with bit values before and after the operation, are illustrated in Figure 2-1 below. For simplicity, the figure shows byte addresses although the Flash sections are word-addressable.

Figure 2-1. Atomic Write, Split Write and Erase Operations



2.3 The Temporary Page Buffer

The Flash memory is organized in pages, which in turn are word-addressable. Both erase and write operations operate on pages. The page size depends on memory size and is given in the device datasheet. Erase and write operations use a temporary page buffer to store the individual words until the entire page is ready to be written to Flash.

The Flash page buffer is very similar to the EEPROM page buffer, which is described in application note "AVR1315: Accessing the XMEGA EEPROM". The difference is that even if only a few words are loaded in the Flash page buffer, the entire page will be updated when writing data, or the entire page will be erased when performing a page erase. For the EEPROM page buffer, only the loaded buffer locations will be affected when writing or erasing EEPROM pages. Please study the example code for usage details.

A Flash page write actually consists of two operations: (1) loading the page buffer with data and (2) writing data to a Flash page. When loading the page buffer, the lower part of the Flash address is used to select a word in the page buffer, while the upper part is ignored. When writing or erasing a page, the upper part of the address selects the page while the lower part is ignored. As the Flash is word-addressable, the LSB of the address is always ignored.

Once a word has been loaded into the buffer for the first time, the *Flash Page Buffer Active Loading* bit (FLOAD) in the *NVM Status* register (STATUS) will be set. The bit remains set until either the buffer is flushed or a page write is performed (atomic or split write).

Note that each word location in the page buffer can only be written once before flushing the buffer or writing a page.

2.4 Application and Boot Sections

The XMEGA Flash memory is divided into two sections, the Application Section and the Boot Section. The Application Section stores ordinary application firmware, while the Boot Section is most often used to store a bootloader application.

The Boot Section can also be used for ordinary application firmware. Since code that executes SPM-based commands can only run from the Boot Section, the section is most often used for bootloaders and/or parts of the application firmware that needs to update Flash memory or run other SPM-based commands.

There are separate lock bits for the Application Section and the Boot Section, so that read and/or write access to the two sections can be individually limited. Also, the upper part of the Application Section has its own set of lock bits, independent from the rest of the section. This upper section is called the Application Table Section, since it is ideal for storing and maintaining larger tables in Flash memory.

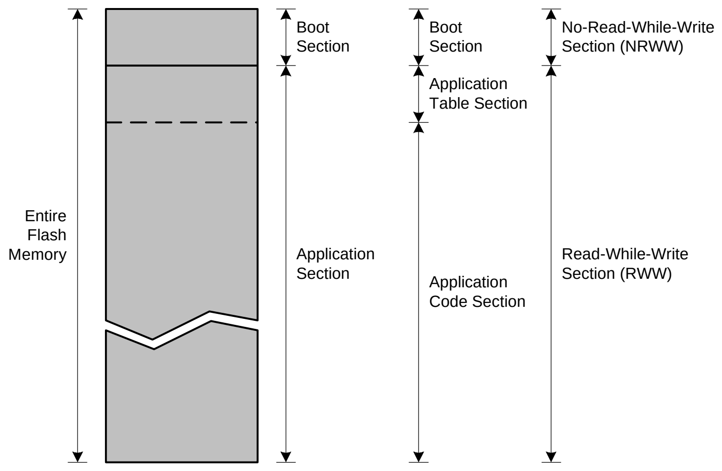
A common scenario for using the Application Table Section is as follows:

- Store application firmware in the Application Section and lock it using corresponding lock bits.
- Store and maintain larger tables in the Application Table Section, and leave it unlocks in case it needs updates later.
- Store the parts of the application firmware that maintains the Flash tables (using SPM-based commands) in the Boot Section and lock it using corresponding lock bits.

Note that the size of the Application Table Section will always be equal to the size of the Boot Section.

Figure 2-2 below shows an overview of the different Flash memory sections related to each other.

Figure 2-2. Flash Memory Organization


Entire Flash Memory
Boot Section
Application Section
Boot Section
Application Table Section
Application Code Section
No-Read-While-Write Section (NRWW)
Read-While-Write Section (RWW)

2.4.1 No-Read-While-Write Section

The entire Boot Section is referred to as a No-Read-While-Write Section (NRWW). It is not possible to read from any location in Flash memory while the NRWW section is being written to. Reading includes executing code and reading Flash data using LPM-based commands.

SPM-based commands can only be executed from the Boot Section. If a bootloader application wants to update its own code, instruction execution is halted while the Boot Section is being written.

2.4.2 Read-While-Write Section

The entire Application Section is referred to as a Read-While-Write Section (RWW). This does *not* mean that it is possible to read from *this* section while it is being written to. It means that it is possible to read and execute code from the *Boot Section* while the Application Section is being updated.

This feature makes it possible to keep critical functions running while updating the application firmware or Flash tables in the Application Table Section.

2.4.3 SPM Lock

Some applications update the application section after startup, and then leave it untouched until the next reset or power cycle. As a safety precaution for this usage, the NVM module can lock all further access to any SPM-based command.

The SPM Lock command requires a variant of the NVM Action-based command. The exact procedure is as follows:

1. Load the *Protect IO Register* signature (byte value `0xD8`) into the *Configuration Change Protection* register (`CCP`). This will automatically disable all interrupts for the next four CPU instruction cycles.
2. Within the next four CPU instruction cycles, set the *SPM Lock* bit (`SPMLOCK`) in *NVM Control Register B* (`CTRLB`).

The example software contains an implementation of this functionality.

2.4.4 Boot Reset Vector

Applications that use bootloader firmware to update the Application Section needs to configure the XMEGA to start executing code in the Boot Section after power-up or reset instead of the Application Section. A dedicated fuse bit is used to configure the startup address to the first location (RESET interrupt vector) in the Boot Section instead of the Application Section. Please refer to the device datasheet for fuse settings and programming information.

2.4.5 Interrupt Vector Table Location

Some applications need to keep selected vital functions running even when the Application Section is being updated. These functions are often interrupt-controlled and need to be enabled and executable at all times during the update. When updating the Application Section, no code can be running from it. The interrupt handler code needs to be duplicated in the Boot Section and the interrupt vector table must be relocated.

For details on how to move the interrupt table and information on interrupts in general, please refer to the application note "AVR1305: XMEGA Interrupts and the Programmable Multilevel Interrupt Controller".

2.4.6 Reducing Power Consumption

It is possible to configure the NVM module to disable the Flash section that is unused at any given time (Application Section or Boot Section). This feature minimizes power consumption in the application since there is no current consumption in the disabled Flash section. By default, both sections are always enabled, but by setting the *Flash Power Reduction Enable* bit (`FPRM`) in *Control Register B* (`CTRLB`), only the section where code executes is enabled. If the execution flow moves from one section to another, e.g. a function call, the CPU is halted for 6 clock cycles while one section is enabled and the other is disabled.

The 6-cycle penalty also applies when an LPM operation accesses a disabled Flash section.

Note that the Application Table Section is a part of the Application Section and will be enabled when the Application Section is.

2.5 Command Summary

Table 2-1 below gives an overview of all NVM commands available for self-programming operations along with the symbolic name used in the source code. The byte value of the command, its type, and a short description are also included. Please study the example source code for more details.

Table 2-1. Relevant NVM Commands

Command	Code	Type	Description
`NO_OPERATION`	`0x00`	(LPM)	Read Flash byte from address Z into R0.
`READ_USER_SIG_ROW`	`0x01`	LPM	Read user signature byte at index Z into R0.
`READ_CALIB_ROW`	`0x02`	LPM	Read calibration byte at index Z into R0.
`READ_FUSES`	`0x07`	NVM	Read fuse byte at index `ADDR0` into `DATA0`.
`WRITE_LOCK_BITS`	`0x08`	NVM	Write `DATA0` to lock bits.
`ERASE_USER_SIG_ROW`	`0x18`	SPM	Erase user signature byte at index Z.

Command	Code	Type	Description
`WRITE_USER_SIG_ROW`	`0x1A`	SPM	Write R0 to user signature byte at index Z.
`ERASE_APP`	`0x20`	SPM	Erase entire Application Section.
`ERASE_APP_PAGE`	`0x22`	SPM	Erase application page at address Z, using only the upper address bits.
`LOAD_FLASH_BUFFER`	`0x23`	SPM	Load word R1:R0 into page buffer at address Z, using only the lower address bits.
`WRITE_APP_PAGE`	`0x24`	SPM	Write page buffer to application page at address Z, using only the upper address bits.
`ERASE_WRITE_APP_PAGE`	`0x25`	SPM	Erase application page at address Z and write page buffer to application page at address Z, using only the upper address bits.
`ERASE_FLASH_BUFFER`	`0x26`	SPM	Flush the page buffer.
`ERASE_BOOT_PAGE`	`0x2A`	SPM	Erase boot page at address Z, using only the upper address bits.
`WRITE_BOOT_PAGE`	`0x2C`	SPM	Write page buffer to boot page at address Z, using only the upper address bits.
`ERASE_WRITE_BOOT_PAGE`	`0x2D`	SPM	Erase boot page at address Z and write page buffer to boot page at address Z, using only the upper address bits.
`APP_CRC`	`0x38`	NVM	Generate CRC from Application Section and store in `DATA2`:`DATA1`:`DATA0`.
`BOOT_CRC`	`0x39`	NVM	Generate CRC from Boot Section and store in `DATA2`:`DATA1`:`DATA0`.

The algorithm details for the CRC generation can be found in the device datasheet.

2.5.1 Special Case: Reading Lock Bits

The lock bits are I/O-mapped into the *NVM Lock Bits* register (`LOCKBITS`), and can be read directly without any NVM commands.

2.6 Interrupt Considerations

When using interrupts and self-programming operations in the same application, the following should be considered:

- Take care not to corrupt the Flash page buffer contents. When loading the page buffer, make sure that no interrupt handlers access the page buffer. If an interrupt handler is used to access the page buffer, make sure that other parts of the application do not access the buffer at the same time.
- Instead of continuously polling the *NVM Busy* flag (`NVMBUSY`) to detect when an NVM operation is finished, it is possible to enable the SPM Interrupt by setting an appropriate interrupt level with the *SPM Interrupt Level* bitfield (`SPMLVL`) in the *Interrupt Control* register (`INTCTRL`). The corresponding interrupt handler will be called whenever the *NVM Busy* flag (`NVMBUSY`) is not set. This can be used to implement an interrupt-controlled Flash memory update. More details on interrupts

can be found in application note "AVR1305: XMEGA Interrupts and the Programmable Multi-level Interrupt Controller".

3 Getting Started

This section walks you through the basic steps for getting up and running with the XMEGA Self-programming. A few common scenarios are described in the sections below. For further examples and details, please study the example software.

3.1 Full Application Section Update

A common scenario is for a Bootloader to get updated Flash data from serial communication through a PC application. In the case of a full Application Section update, the recommended procedure is as follows:

1. Erase entire application section with the SPM command `ERASE_APP`.
2. Wait for *NVM Busy* flag to be cleared.
3. Get one page worth of data over the communication channel.
4. Load the page buffer with the data using the SPM command `LOAD_PAGE_BUFFER`.
5. Use the SPM command `WRITE_APP_PAGE` to write the new data to the Flash page.
6. Wait for *NVM Busy* flag to be cleared.
7. Repeat from step 2 until all pages are updated.

3.2 Update Selected Locations

Another common scenario is to update selected locations on Flash, for instance constant tables or parameters stored in Flash memory, preferably in the Application Table Section. The recommended procedure for such read-modify-write operations is as follows:

1. Read the Flash page that contains the locations to be updated into an SRAM buffer using plain LPM read operations (`NO_OPERATION`).
2. Updated selected locations in the SRAM buffer.
3. Load the page buffer with the data from the updated SRAM buffer using the SPM command `LOAD_PAGE_BUFFER`.
4. Use the SPM command `ERASE_WRITE_APP_PAGE` to erase previous contents and write the new data to the Flash page.
5. Wait for *NVM Busy* flag to be cleared.

4 Driver Implementation

This application note includes a source code package with a basic Self-programming driver implemented in assembly with a C interface. Please refer to the driver source code and device datasheet for more details.

4.1 Files

The source code package consists of three files:

- *sp_driver.s/sp_driver.s90* – Self-programming driver source file
- *sp_driver.h* – Self-programming driver header file

- *sp_example.c* – Example code using the driver

For a complete overview of the available driver interface functions and their use, please refer to the source code documentation.

4.2 Doxygen Documentation

All source code is prepared for automatic documentation generation using Doxygen. Doxygen is a tool for generating documentation from source code by analyzing the source code and using special keywords. For more details about Doxygen please visit http://www.doxygen.org. Precompiled Doxygen documentation is also supplied with the source code accompanying this application note, available from the *readme.html* file in the source code folder.

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Unit 1-5 & 16, 19/F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
Hong Kong
Tel: (852) 2245-6100
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
avr@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Request
www.atmel.com/literature