AVR 8-bit Microcontrollers

AVR106: C Functions for Reading and Writing to Flash Memory

APPLICATION NOTE

Introduction

The Atmel® AVR® devices have a feature called Self programming Program memory. This feature enables an AVR device to reprogram the Flash memory while executing the program. Such a feature is helpful for applications that must self-update firmware or store parameters in Flash.

This application note provides the details about the C functions for accessing the Flash memory.

Features

- C functions for accessing Flash memory
  - Byte read
  - Page read
  - Byte write
  - Page write
- Optional recovery on power failure
- Functions can be used with any device having Self programming Program memory
- Example project for accessing Application Flash section for parameter storage

Table of Contents

1. Theory of Operation

This section contains some basic theory around using the Self programming Program memory feature in AVR. For a better understanding of all features concerning Self programming, refer to the device datasheet or application note “AVR109: Self Programming”.

1.1. Using SPM Instructions

The Flash memory may be programmed using the Store Program Memory (SPM) instruction. On devices containing the Self Programming feature the program memory is divided into two main sections: (1) Application Flash Section and (2) Boot Flash Section.

On devices with boot block, the SPM instruction has the ability to write to the entire Flash memory, but can only be executed from the Boot section. Executing SPM from the Application section will have no effect. On the smaller devices that do not have a boot block, the SPM instruction can be executed from the entire flash memory.

During Flash write to the Boot section the CPU is always halted. However, most devices may execute code (read) from the Boot section while writing to the Application section. It is important that the code executed while writing to the Application section do not attempt to read from the Application section. If this happens the entire program execution may be corrupted.

The size and location of these two memory sections are depending upon device and fuse settings. Some devices have the ability to execute the SPM instruction from the entire Flash memory space.

1.2. Write Procedure

The Flash memory is written in a page-by-page fashion. The write function is performed by storing data for an entire page into a temporary page buffer prior to writing the Flash. Which Flash address to write to is decided by the content of the Z-register and RAMPZ-register. A Flash page must be erased before it can be programmed with the data stored in the temporary buffer. The functions contained in this application note use the following procedure when writing a Flash page:

- Fill temporary page buffer
- Erase Flash page
- Write Flash page

As it is evident in this sequence there is a possibility for loss of data, if a reset or power failure should occur immediately after a page erase. Loss of data can be avoided by taking necessary precautions in software, involving buffering in non-volatile memory. The write functions contained in this application note provide optional buffering when writing. These functions are further described in the firmware section.

Devices that support the read-while-write feature allows the boot loader code to be executed while writing. In such devices, the write functions will not return until the write has completed.

1.3. Addressing

The Flash memory in AVR is divided into 16-bit words. This means that each Flash address location can store two bytes of data. For an ATmega128, it is possible to address up to 65k words or 128k bytes of Flash data. In some cases the Flash memory is referred to by using word addressing and in other cases by using byte addressing, which can be confusing. All functions contained in this application note use byte addressing. The relation between byte address and word address is as follows:

Byte address = word address • 2

A Flash page is addressed by using the byte address for the first byte in the page. The relation between page number (ranging 0, 1, 2…) and byte address for the page is as follows:

Byte address = page number • page size (in bytes)

Example on byte addressing:

A Flash page in an ATmega128 is 256 bytes long.

Byte address 0x200 (512) will point to:

- Flash byte 0x200 (512), equal to byte 0 on page 2
- Flash page 2

When addressing a page in ATmega128, the lower byte of the address is always zero. When addressing a word, the LSB of the address is always zero.

1.4. Page Size

The constant `PAGESIZE` must be defined to be equal to the Flash page size (in bytes) of the device being used.

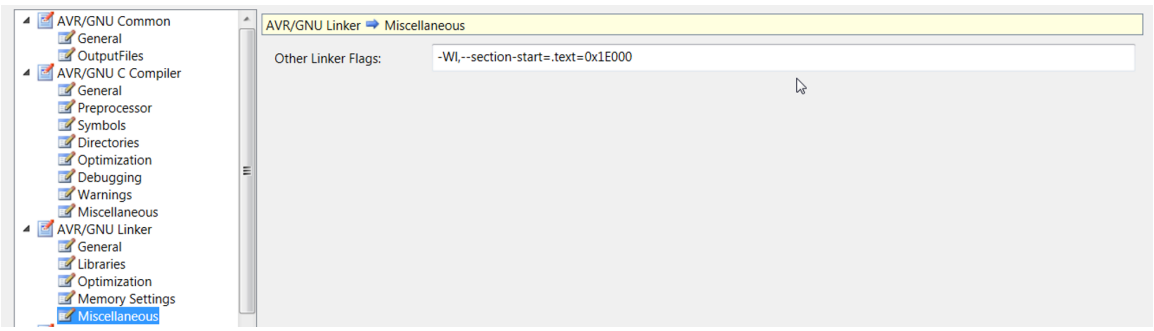
1.5. Defining Flash Memory For Writing

The memory range in which the functions are allowed to write is defined by the constants `ADR_LIMIT_LOW` and `ADR_LIMIT_HIGH`. The write functions can write to addresses higher or equal to `ADR_LIMIT_LOW` and lower than `ADR_LIMIT_HIGH`.

1.6. Placing the Entire Code Inside the Boot Section

It is necessary to include the linker options as shown in the following figure to place the entire application code in the Boot section of Flash. The location and size of the Boot section varies with the device being used and fuse settings. Programming the `BOOTRST` fuse will move the reset vector to the beginning of the Boot section. It is also possible to move all the interrupt vectors to the Boot section. For more information, refer to the interrupt section in the device datasheet.

Figure 1-1. Adding the Linker Script



The linker miscellaneous option should be defined with the corresponding start address of the bootloader. The Bootloader starting word address of ATmega128 4K bootloader is 0xF000. The linker option should be updated with the byte address Hence, equivalent byte address = 0x1E000. By including the linker script as shown in the preceeding figure, the entire application code will be placed inside the boot section.

1.7. Placing Selected Functions Inside the Boot Section

Alternatively, it is possible to place only selected functions into defined segments of the Flash memory. In fact, it is only the functions for writing that must be located inside the Boot section. This can be done by defining a new Flash segment corresponding to the Boot memory space and use the attribute `BOOTLOADER SECTION` to place the desired functions into this segment as shown in the following example.

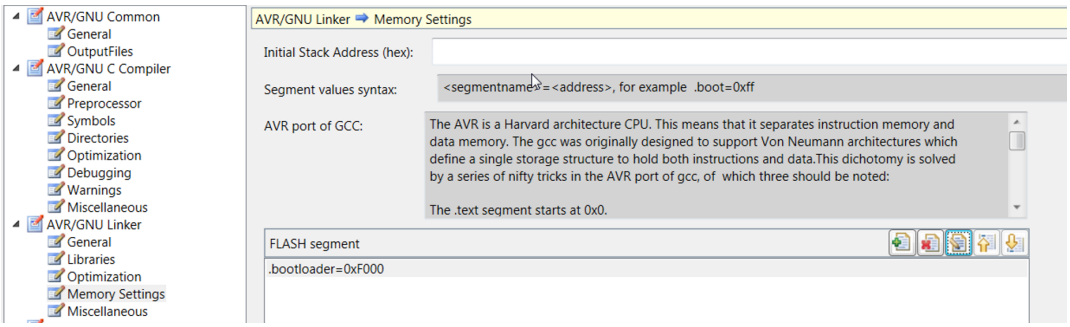
Definition of Boot segment:

The bootloader segment definition should be done at the memory settings tab in the AVR/GNU Linker option of the project properties window as shown in the following figure. The syntax is as follows.

`".bootloader=0xF000"`

Where 0xF000 is the start address of the 4K bootloader segment of ATmega128.

Figure 1-2. Definition of Boot Segment



Placing a C function into the defined segment:

```
#define BOOTLOADER_SECTION __attribute__ ((section(".bootloader")))
void Example_Function ()BOOTLOADER_SECTION;
void Example_Function (){
-------
}
```

On defining the bootloader segment in the project properties window as shown in the above figure and building the above C-code, it will place the `example function ()` into the defined memory segment `".bootloader"`.

2. Software Implementation and Usage

The firmware is made for the AVR-GCC compiler with Atmel Studio 7.0.582. The functions may be ported to other compilers, but this may require some work since several functions from the avr-gcc toolchain are used. When using Self-programming it is essential that the functions for writing are located inside the Boot section of the Flash memory. The procedure for placing the flash write function inside the boot section is explained under the section Placing Selected Functions Inside the Boot Section. The remaining functions can be placed in the application section of the flash. All other necessary configurations concerning the firmware are done inside the file `Self_programming.h`.

The zip file available with this application note consists of an example project created using Atmel Studio 7.0 for the device ATmega128. In this example project, the flash write function is located in the Boot section of Flash and remaining code placed in the application section of flash.

2.1. Flash Recovery

Defining the constant `__FLASH_RECOVER` enables the Flash recovery option for avoiding data loss in case of power failure. When Flash recovery is enabled, one Flash page will serve as a recovery buffer. The value of `__FLASH_RECOVER` will determine the address to the Flash page used for this purpose. This address must be a byte address pointing to an address in the application section of a Flash page and the write functions will not be able to write to this page. Flash recovery is carried out by calling the function `RecoverFLASH()` at program startup.

When the Flash recovery option is enabled a page write will involve pre-storing of data into a dedicated recovery page in Flash, before the actual write to a given Flash page takes place. The address for the page to be written to is stored in EEPROM together with a status byte indicating that the Flash recovery page contains data. This status byte will be cleared when the actual write to a given Flash page is completed successfully. The variables in EEPROM and the Flash recovery buffer are used by the Flash recovery function `RecoverFlash()` to recover data when necessary. The writing of one byte to EEPROM takes about the same time as writing an entire page to Flash. Thus, when enabling the Flash recovery option the total write time will increase considerably. EEPROM is used instead of Flash because reserving a few bytes in Flash will exclude flexible usage of the entire Flash page containing these bytes.

2.2. Descriptions of C Functions

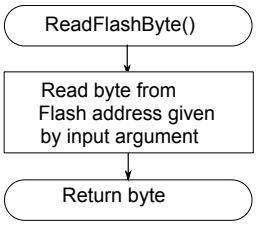
Function	Arguments	Return
`ReadFlashByte( )`	MyAddressType flashAdr	unsigned char
`ReadFlashPage()`	MyAddressType flashStartAdr, unsigned char *dataPage	unsigned char
`WriteFlashByte( )`	MyAddressType flashAddr, unsigned char data	unsigned char
`WriteFlashPage()`	MyAddressType flashStartAdr, unsigned char *dataPage	unsigned char
`RecoverFlash()`	Void	unsigned char

The datatype `MyAddressType` is defined in `Self_programming.h`. The size of this datatype depends on the device that is being used. It can be defined as an `long int` when using devices with more than 64KB of Flash memory, and as an `int` (16 bit) using devices with 64KB or less of Flash memory. The datatypes are actually used as `__flash` or `__farflash` pointers (consequently 16 and 24 bit). The

reason why a new datatype is defined is that integer types allow a much more flexible usage than pointer types.

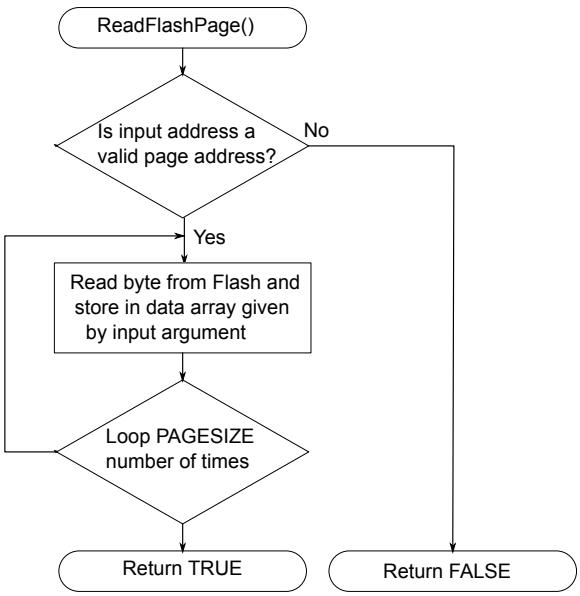
The `ReadFlashByte()` returns one byte located on Flash address given by the input argument `FlashAdr`.

Figure 2-1. Flowchart for the ReadFlashByte() Function



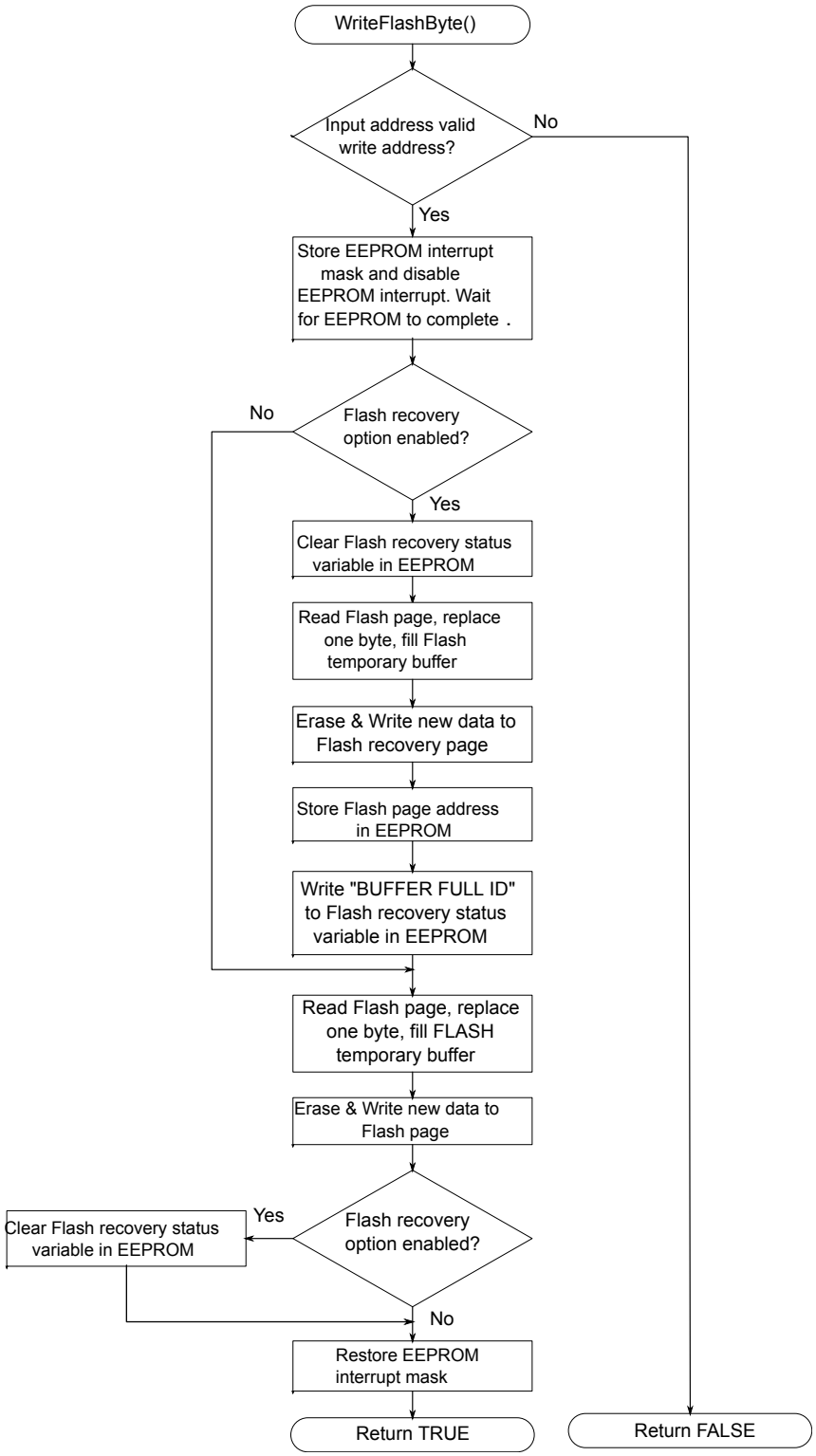
The `ReadFlashPage()` reads one Flash page from address given by the input argument `FlashStartAdr` and stores data in array given by the input argument `DataPage[]`. The number of bytes stored is depending upon the Flash page size. The function returns FALSE if the input address is not a Flash page address, else TRUE.

Figure 2-2. Flowchart for the ReadFlashPage() Function



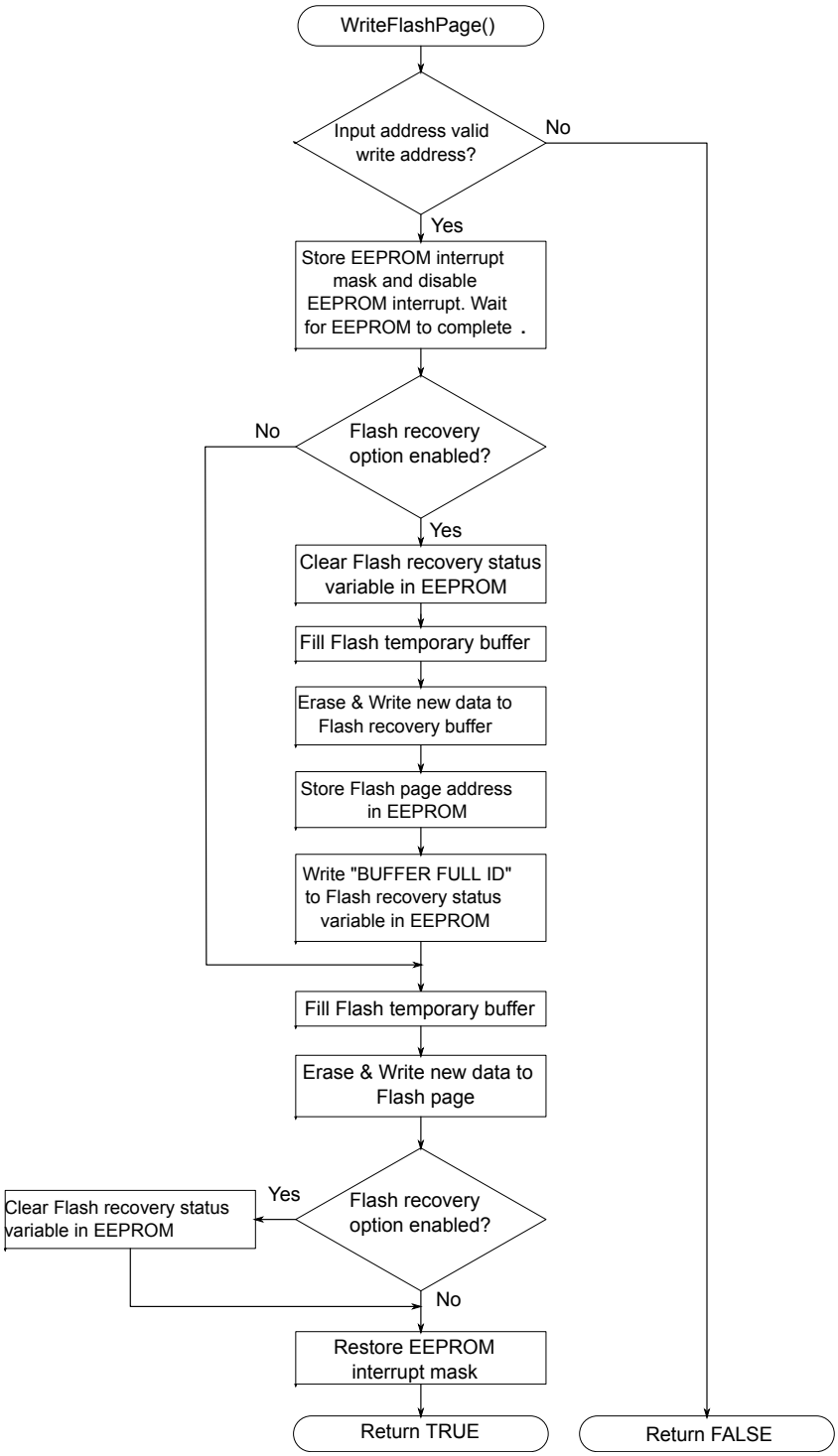
The `WriteFlashByte()` writes a byte given by the input argument `Data` to Flash address given by the input argument `FlashAddr`. The function returns FALSE if the input address is not a valid Flash byte address for writing, else TRUE.

Figure 2-3. Flowchart for the WriteFlashByte() Function



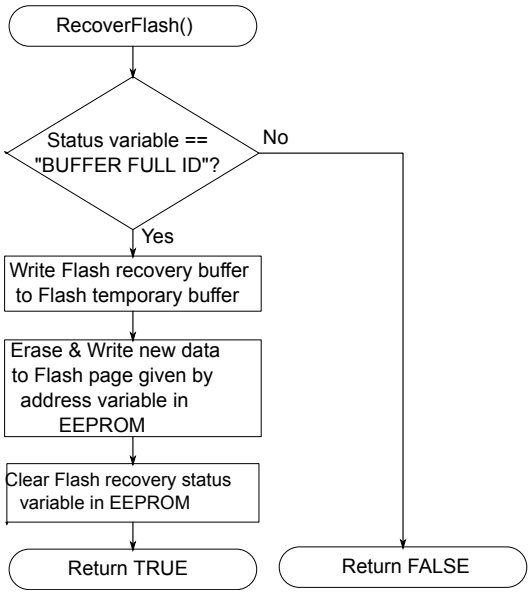
The `WriteFlashPage()` writes data from array given by the input argument `DataPage[]` to Flash page address given by the input argument `FlashStartAdr`. The number of bytes written is depending upon the Flash page size. The function returns FALSE if the input address is not a valid Flash page address for writing, else TRUE.

Figure 2-4. Flowchart for the WriteFlashPage() Function



The `RecoverFlash()` reads the status variable in EEPROM and restores Flash page if necessary. The function must be called at program start-up if the Flash recovery option is enabled. The function Returns TRUE if Flash recovery has taken place, else FALSE.

Figure 2-5. Flowchart for the RecoverFlash() Function



2.3. Steps for Implementing in Other Devices

This firmware can be reused with other AVR devices having specific bootloader segment.

The steps for reusing the code are as follows:

1. Change the following macro available in the `self_programming.h` file with the available boot segment size of the devie in use.
   ```
   #define BOOTSECTORSIZE 0x2000// 4096 words
   ```
2. Change the following macro available in the `self_programming.h` file with the location for flash recovery buffer based on the available flash size.
   ```
   #define ADR_FLASH_BUFFER 0xEF00
   ```
3. Change the flash segment definition available under the project properties window corresponding to the boot memory space and use the attribute BOOTLOADER SECTION to place the desired functions into this segment as shown in the Figure 1-2 Definition of Boot Segment. This figure shows the memory segment definition for ATmega128. The Bootloader starting word address of ATmega128 4K bootloader is 0xF000.
   Similarly update the memory segment definition for the device in use.
4. Ensure that the required BOOTSZ fuse setting is programmed. For ATmega128, it should be 4096W_F000.

Implementing in Devices without dedicated Boot Section

While implementing the code for the devices which do not have dedicated boot section, ensure that the following conditions are satisfied,

1. For the devices which do not have a dedicated boot section, use the last few pages of the flash as boot section to place the flash write function. The flash write routine consumes approximately 550 bytes of memory. Hence, make sure that the emulated boot section is not less that 550 bytes.
2. Provide the address of the flash write function and address of the recovery flash buffer in such a way that the actual application code is not overwritten.

3. Summary

This application note provides four different C functions for accessing the flash memory.

The following functionalities are covered in this application note.

1. Flash page write.
2. Flash page read.
3. Flash byte write.
4. Flash byte read.

The firmware places the flash write routines inside the boot section and the remaining code in the application section of the flash memory. The firmware available with this application note performs the above mentioned operations on ATmega128. It also explains the procedure to re-use the application code for other devices.

4. Further Readings

- ATmega128 Datasheet
- AVR109: Using Self Programming on tinyAVR and megaAVR devices
- AVR105: Power efficient high endurance parameter storage in tinyAVR and megaAVR devices Flash memory
- AVR108: Setup and use of the LPM Instructions on tinyAVR and megaAVR devices
- Atmel AVR116: Wear Leveling on DataFlash
- Atmel AVR947: Single-Wire Bootloader for any MCU with Self Programming Capability

5. Revision History

Doc Rev.	Date	Comments
2575C	3/2016	The firmware is ported to Atmel Studio 7.0 and steps to implement in other devices is added. In addition to that we have modified the code to place the selected function inside the boot section.
2575B	08/2006	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**