SMART ARM-Based Microcontroller

AT16743: SAM V7/E7/S7 Safe and Secure Bootloader

APPLICATION NOTE

Introduction

The Atmel® | SMART ARM® Cortex®- M7 based MCUs deliver the highest performing Cortex-M7 based MCUs to the market with an exceptional memory and connectivity options for design flexibility making them ideal for the automotive, IoT, and industrial connectivity markets. The ARM Cortex-M7 architecture enhances performance and at the same time keeps the cost and power consumption in under control.

Microcontrollers are used in a variety of electronic products. The devices are becoming more flexible, due to the reprogrammable memory (Flash memory) often used to store the firmware of the product. This enables the firmware in a device to be upgraded in the field for correcting bugs or adding new functionalities.

Features

This application note discusses the boot sequence, upgrade sequence, safety, and security in bootloader. It also provides an example bootloader implementation for Atmel SAM V7/E7/S7 ARM Cortex-M7 based microcontrollers.

Table of Contents

1. Abbreviations

AES	Advanced Encryption Standard
ASF	Atmel Software Framework
CAN	Controller Area Network
CBC	Cipher Block Chaining
CDC	Communication Device Class
CFB	Cipher FeedBack
CTR	Counter
ECB	Electronic CodeBook
ICM	Integrity Check Monitor
MCU	Memory Control Unit
OFB	Output FeedBack
PC	Personal Computer
PLL	Phase Locked Loop
POR	Power On Reset
RAM	Random Access Memory
SHA	Secure Hash Algorithm
SP	Stack Pointer
USB	Universal Serial Bus
VTOR	Vector Table Offset Register

2. Bootloader

Modern microcontrollers use Flash memory to store their application code. The main advantage of Flash is that the memory can be modified by the software itself. This is the key to in-field programming: a small section of code is added to the main application to provide the ability to download updates, replacing the old firmware of the device. This code is called as bootloader, as its role is to load a new program at boot.

The bootloader implementation poses several challenges such as correctly remapping memories, effective resource utilization, ensuring firmware upgrade is successful, and including necessary safety and security precautions.

A bootloader always resides in the memory to enable the firmware of the device to be upgraded at any time. Therefore, the bootloader must be as small as possible as it does not add any direct functionality for the user.

Downloading a new firmware into the device requires a process to initiate the bootloader to prepare for the transfer. There are two types of such trigger conditions: hardware and software.

A hardware condition can be triggered using a button press during a reset. Whereas, a software condition could be lack of a valid application in the system. When the system starts, the bootloader checks the predefined conditions. If one of them is true, it will connect to a host and wait for a new firmware. This host can be any device. However, a standard PC with the appropriate software is most often used. The firmware can be transferred using any protocol supported by the target such as RS232, USB, and CAN.

2.1. Boot Sequence

The startup sequence of the Bootloader is as follows:

1. Initialization
2. Trigger condition check
3. Firmware upgrade (if trigger condition is set)
4. Firmware verification (optional)
5. Firmware loading (if verification is ok or disabled)

2.2. Upgrade Sequence

The basic upgrade flow starts with the host sending the firmware to the target, which then programs it into the memory. When the programming is complete, the new application is loaded. In theory, the “download” and “programming” steps are different, this is not the case in practice. Indeed, AT91SAM microcontrollers usually have 4x more Flash memory than RAM. The device cannot store the entire firmware in the RAM before writing it permanently to the Flash.

It is preferred that the code must be written to the memory while it is received. Since a Flash write operation consumes time, a communication protocol is required to halt the transfer when the data is being written and resume it afterwards.

The Flash memory is split up into fixed-size blocks called pages. Depending on the quantity of Flash in the microcontroller, the page size varies. A memory write operation can upgrade one page (or less) at a time; thus it is more logical to send packets containing one full page.

There are several optional post-processing features to consider. If code encryption is activated, then each page must be decrypted before being any data is written. If a digital signature or a message authentication code is available, it must be verified as soon as the download is complete.

2.3. Safety

It is important that a working firmware is embedded in the device at all times. However, the use of a bootloader can result in the conflicting situation where the new firmware has not been installed properly, compromising the behavior of the system.

This application note offers an example safety solution. Most of the techniques to circumvent those problems present a trade-off between the level of security and safety against the size and speed of the system. The safest and most secure solution is also probably the biggest and slowest in terms of performance. This also means that one must first carefully analyze safety and security requirements in a system, to implement only the required functionality.

Among the other features, a protocol stack is used in most communication standards to offer reliable transfers. This reliability is important for a bootloader application as the firmware must not get corrupted during the download.

2.4. Security

Securing a system enforces several features such as privacy, integrity, and authenticity.

2.4.1. Privacy

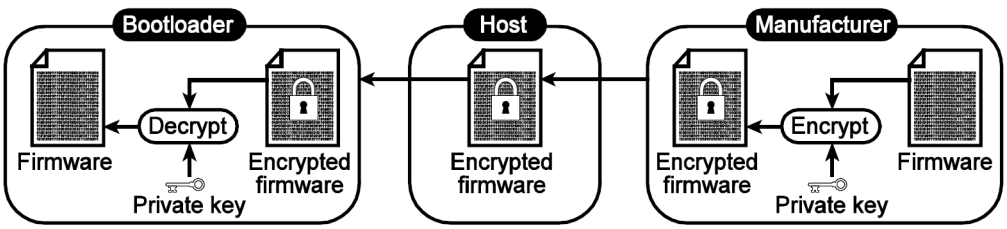
Privacy feature ensures that a piece of data cannot be accessed by unauthorized users or devices. It is a major concern for firmware developers to ensure that their application design is not accessible to the competitors. This feature ensures that the code is private and the target devices being the only authorized “users”.

Microcontrollers provide a mechanism making it difficult for malicious users to read the program code written in the device. However, for in-field firmware upgrade, the manufacturer has to provide the new code image to customers so they can patch their devices themselves. Which on the other hand enables a skilled person to potentially decompile and retrieve the original code.

Data privacy is enforced using encryption: the data is processed using a cryptographic algorithm along with an encryption key, generating a cipher text which is different from the original one. Without the specific decryption key, the data will be illegible, preventing anyone unauthorized from reading it.

A private-key algorithm is used to generate the encrypted firmware. A public-key system cannot be used, as the firmware would then be decipherable by anyone. The encryption and decryption keys are thus identical and shared only between the bootloader and the manufacturer.

Figure 2-1. Firmware Encryption



All security issues cannot be solved by encrypting the code. An attacker can probably pinpoint the location in the code of an important variable and tweak it until he gets the desired result.

The code encryption combines itself with a message authentication code. Since they both use a symmetric encryption algorithm, one can use the same algorithm to save the code size.

2.4.1.1. Selecting an Algorithm

A symmetric encryption algorithm can be defined by several characteristics:

- Key length in bits
- Block length in bits
- Security
- Size and Speed

The *Key length* used by an algorithm is an important parameter of the algorithm. The larger it is, the more difficult it is to perform a brute-force attack. As computers become faster and faster, longer keys are required. A reasonable length seems to be 128 bits at the moment, it is one of the Key lengths selected for Advanced Encryption Standard (AES) cipher.

A large *Block length* is required to avoid a “code-book attack”, i.e., someone getting enough blocks of plain text and their corresponding cipher text to build a table, enabling decipher further information. In this scenario (firmware upgrading), an attacker is very unlikely to get access to any plain text at all. Therefore, the block length can be of any reasonable size. Most block ciphers will use at least 64 bits, with modern ciphers using at least 128 bits.

2.4.1.2. Modes of Operation

Different modes of operations are possible when using a symmetric cipher:

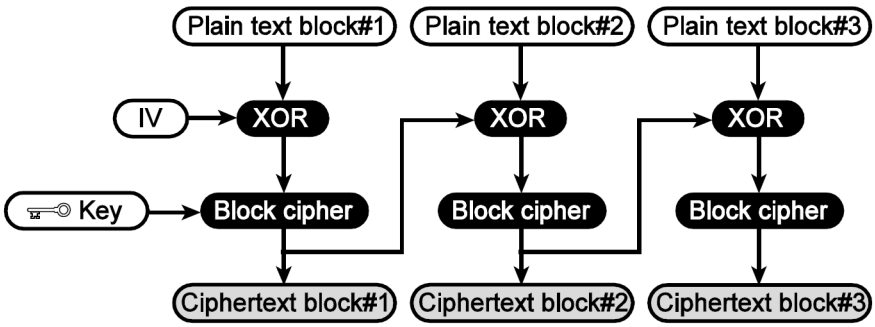
- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC, CFB, OFB, CTR)
- Authenticated Encryption (EAX, CCM, OCB)

The basic mode of operation is *ECB*. In this mode each block of plain text is encrypted using the key and the selected algorithm, resulting in a block of cipher text. However, this mode is very insecure as it does not hide the patterns. Two identical blocks of plain text will be encrypted to the same cipher text block.

To solve this issue, *Cipher Block Chaining* modes are used. Encryption is not only done with the current block of plain text, but also with the last encrypted block. It makes each block depend on the previously encrypted data, making everything interdependent.

The first block is encrypted using a random *Initialization Vector* (IV). While this vector can be transmitted in clear text, the same vector should not be reused with the same key. A manufacturer will produce more than one firmware upgrade for a product in its lifetime. So, the IV cannot be stored in the chip similar to the key. It has to be transmitted by the host.

Figure 2-2. CBC Mode of Operation



2.4.2. Integrity

Integrity feature ensures that any change in the data is detected. For example, an authorized firmware may be slightly modified. Although the firmware may appear genuine, there could be attacks which migh change the data.

To verify Integrity, check for:

- an intentional modification of the firmware
- an accidental modification of the firmware

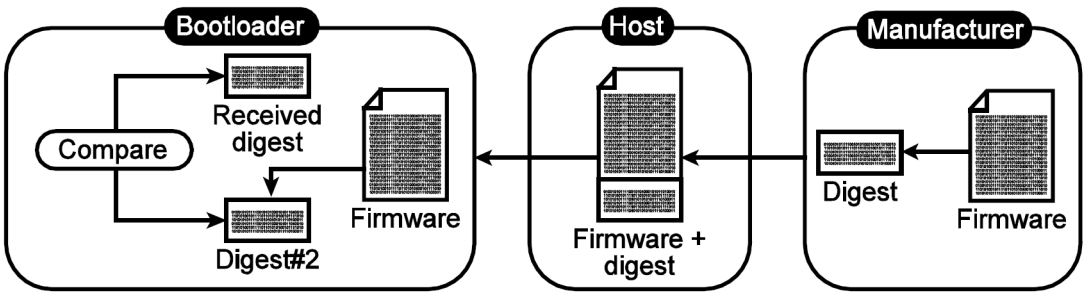
Accidental modification is a safety problem. It is typically solved by using error detection codes.

Hash Function produces a digital “fingerprint” for a piece of data. It means conversely to an error detection code, every piece of data must have its own unique fingerprint.

To verify the integrity of a firmware, its fingerprint is calculated and attached to the file. When the bootloader receives both the firmware and its fingerprint, it re-computes the fingerprint and compares it to the original fingerprint. If both are identical, then the firmware has not been altered.

In practice, a hash function takes a string of any length as an input and produces a fixed-size output called a message digest. It also has several important properties such as good diffusion (the ability to produce a completely different output even if only one bit of the input is flipped).

Figure 2-3. Firmware Hashing



Since the output length is fixed irrespective of the input, it is not possible to generate a unique digest for every piece of data. However, hash functions ensure that it is almost impossible to find two different messages with the same digest. This achieves almost the same result as uniqueness, at least in practice.

The disadvantage of hashing the firmware is its simplicity. An attacker can easily modify the file and re-compute the hash. The bootloader would not be able to detect that modification.

During run time, a hash function can be used to verify the firmware integrity to avoid executing a corrupted application.

2.4.2.1. Hash Functions

A hash function has three defining characteristics:

1. Output length
2. Security
3. Size and Speed

The output length of a hash must be large enough to make it almost impossible to find collisions (two different files having the same digest). This prevents anyone from finding a piece of data producing the same hash. Most modern hash functions has at least a 160-bit output (like SHA-1).
Note: Of late the hash algorithms with hash length of 512 bits are preferred .

The security of the hash function is much more critical than the length of its output. Indeed, MD5 (which only has a 128-bit output) would still be secure if it did not have serious design flaws in it. Similarly to block ciphers, finding a flaw in a hash does not imply that it has been cracked; most attacks are not feasible without gigantic computational power. Recent designs considered secure should be preferred rather than using any deprecated algorithms.

When deciding on a hash function, its size and speed performances should also be evaluated. However, the stronger algorithms are often the slowest ones (which is not true for block ciphers), so there will be a security/speed trade-off.

The most commonly used hash algorithms are listed in the following table.

Table 2-1. Hash Algorithms

Algorithm	Hash Length	Security	Size & Speed
MD5	128 bits	Broken	Fast
RIPEMD-160	160 bits	Secure	Slow
SHA-1	160 bits	Broken	Slow
SHA-256	256 bits	Secure	Slow
WHIRLPOOL	512 bits	Secure	Very Slow
Tiger	192 bits	Secure	Fast
HAVAL	128 to 256 bits	Broken	Moderately Fast

2.4.3. Authenticity

Authenticity feature helps to verify if the firmware is from the authentic manufacturer. While reprogramming a device, the firmware could be from a third party. This may be problematic if that firmware is intented for malicious use such as bypassing security protections and illegally using critical functions of the device.

Authentication is about verifying the identity of the sender and the receiver of a message. A bootloader should verify authenticity of the manufacturer and target.

3. Example Bootloader and User Application

3.1. Hardware/Software Requirements

- Hardware Prerequisites
  - Atmel SMART SAM V71 Xplained ULTRA Kit
  - Interfacing Cables
    - Two Micro-USB type-B cable (EDBG & Target USB)
- Software Prerequisites
  - Atmel Studio 7.0
  - ASF 3.30.1

3.2. Design Considerations

3.2.1. Safety

- Erase operations are triggered only on the user application area. This ensures that bootloader application is always available to run on the device.
- USB–CDC protocol is chosen for data transfers between host and device. This takes care of error detection, transmission loses, and packet acknowledgment.

3.2.2. Security

- AES Cipher Block Chaining (CBC) algorithm is implemented to ensure data privacy. This encryption is not only done with the current block of plain text, but also with the last encrypted block. Before sending data to the device, it is encrypted page by page with predefined Key and Initial Vector.
- SHA-1 algorithm is used to produce digital “fingerprint“ for entire user application. Before executing user application, bootloader verifies fingerprint to check integrity of the user application.

3.2.3. Software Considerations

- This design uses AES and ICM hardware modules for Decryption, Hash functions, and Software Components, Services provided in ASF.
- Applications Start address and length are aligned to Page boundaries, hardware requirements such as Vector Table locations, and alignment for SHA padding requirements.
- Other than 32 bytes used by Jump Signature, entire RAM is available for both bootloader and user applications.
- A soft reset is required to jump to User Application after completing Programming sequence.
- A soft reset is required to switch to Bootloader Application after receiving switch to bootloader command / sequence.

3.2.4. Hardware Requirements

In this example, following hardware points are considered as USB Device is activated on ATSAMV71Q21

1. An external crystal or external clock with a frequency of 12MHz or 16MHz is required to generate USB and PLL clocks correctly
2. External clock must be 2500ppm and VDDIO square wave signal
3. Flash wait states (6 wait states) are enabled as per device suggestions

3.2.5. Software Limitations

1. For the first time, it is required to program Bootloader Application with help of available programmers
   1.1. If Bootloader is erased accidentally, it must be programed again using programmers / other available options.
2. On power cycle between Jump to Bootloader instruction and board reset, control is returned to user application. In such case, it is required to issue Jump to Bootloader instruction once again.
3. Application customized for bootloader cannot run on its own because its start address is changed to different location in the flash.

3.2.6. Porting Considerations

Bootloader application can be ported to interfaces other than USB-CDC. However following points must be considered in such cases

1. Replace USB-CDC interface with preferred interface
2. Redefine functions
   2.1. To transmit and receive messages from/to Host
   2.2. To process received data and trigger actions same as in the example project
3. Revisit Firmware Packager and Updater to match with selected interface

3.3. Software Architecture

3.3.1. Features

The features of the example implementation are:

- Basic boot loading capabilities using a USB-CDC to transmit the firmware
- Code encryption using AES CBC
- Both applications are provided with Footers, which contains
  - Application Versions
  - Applications Start Address and Length
  - Option to add Authentication information
  - SHA-1 padding bytes
  - Hash Digest
- Example user application which responds to USB-CDC Messages

3.3.2. Communication Protocol

A simple communication protocol is defined over USB-CDC Class. The following tables provide information about communication protocol and the messages.

Table 3-1. Communication Protocol

Header	Command	Length (Optional)	Data n Bytes
0xFF	0xB0-0xB6	n	Data in big endian format

Table 3-2. Communication Messages

Command	Description	Sample Message
0xB0	Request to Jump to Bootloader Application	• Command - 0xFF 0xB0 • Response – 0xFF 0xB0 0x00/0x01 (Boot/User App)
0xB1	Request to raise a soft reset	• Command - 0xFF 0xB1
0xB2	Query to know device mode	• Command - 0xFF 0xB2 • Response – 0xFF 0xB2 0x00/0x01 (Boot/User App)
0xB3	Set AES Key and Initial Vector Index	• Command – 0xFF 0xB3 KEYINDEX IV_INDEX • Response – 0xFF 0xB3 0x00/0xFF (Success/Fail)
0xB4	Erase Flash	• Command - 0xFF 0xB4 • Response – 0xFF 0xB4 0x00/0xFF (Success/Fail)
0xB5	Send Upper 256 bytes of the Page data. Device receives data and wait for Lower 256 bytes.	• Command – 0xFF 0xB5 – Flash Address [3:0] – 0x01/0x00 - Encrypt ON/OFF – DATA[256:512] • Response – 0xFF 0xB5 0x00/0xFF (Success/Fail)
0xB6	Send Lower 256 bytes of the Page data. Device triggers Flash Write on receiving this command.	• Command – 0xFF 0xB6 – Flash Address [3:0] – 0x01/0x00 - Encrypt ON/OFF – DATA[0:255] • Response – 0xFF 0xB6 0x00/0xFF (Success/Fail)

3.3.3. Code Locations

The Flash is divided into 2 sections, one for Bootloader Application and another for User Application. Each section reserves 256 bytes Footer at the end of the section. Following graphic illustrates the Flash Sections and Application Footer information.

Figure 3-1. Flash Sections

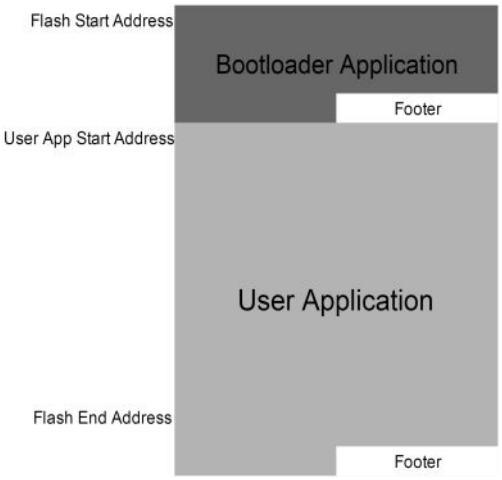


Table 3-3. Application Footer

Parameter	Data Type	Description
App Major Version	`uint32_t`	Application Software's Major Version
App Minor Version	`uint32_t`	Application Software's Minor Version
App Jump Handler	`void (*fpJumpHandler) (void)`	Function pointer to Jump to Application
Boot App Start Address	`uint32_t`	Start address of bootloader application
Boot App Length	`uint32_t`	Length of bootloader application
User App Start Address	`uint32_t`	Start Address of user application
User App Length	`uint32_t`	Length of user application
Reserved	`uint32_t[24:0]`	Reserved. Used for application authentication, if required.
SHA Padding Data	`uint32_t[15:0]`	SHA algorithms requires padding bytes. This space is reserved for them.
SHA Digest	`uint32_t[15:0]`	SHA output is stored here. Bootloader uses this value to check Integrity of user application.

3.3.4. Switching between Applications

3.3.4.1. Bootloader Application to User Application

In the bootloader application, Jump Signature plays a critical role in deciding whether control should remain in bootloader application or check for user application execution.

Jump Signature is the RAM memory reserved by both bootloader application and user application. “StayInBootLoader” is a string, which will be loaded in to Jump Signature location when control must remain in bootloader application on next soft reset.

- On POR, it contains zeros which makes bootloader application to check for user application existence.
- If user application requires control to remain in bootloader application, it updates Jump Signature with “StayInBootLoader”.
- When bootloader application detects “StayInBootLoader”, it clears Jump Signature and remains in bootloader application until next soft reset.

On soft reset, control starts executing bootloader application. The bootloader application,

1. Checks for Jump Signature, if it is not “StayInBootLoader”,
    1.1. Reads bootloader application footer to determine user application Start address and Length
    1.2. Reads user application Footer with help of Start Address and Length
    1.3. Calculates digest for the given area and compares with digest available in Footer
    1.4. On detecting valid digest,
        1.4.1. Releases resources such as peripherals
        1.4.2. Disables interrupts (if required)
        1.4.3. Updates vector table offset (VTOR) and Stack pointer (SP)
        1.4.4. Jumps to reset handler provided in vector table
2. If Jump Signature is “StayInBootLoader”,
    2.1. Initialize Flash Wait states
    2.2. Start USB-CDC Device functionality
    2.3. Wait for USB-CDC messages and start processing

Following flow charts present Boot and Upgrade sequences used in this example.

Figure 3-2. Boot Sequence Diagram

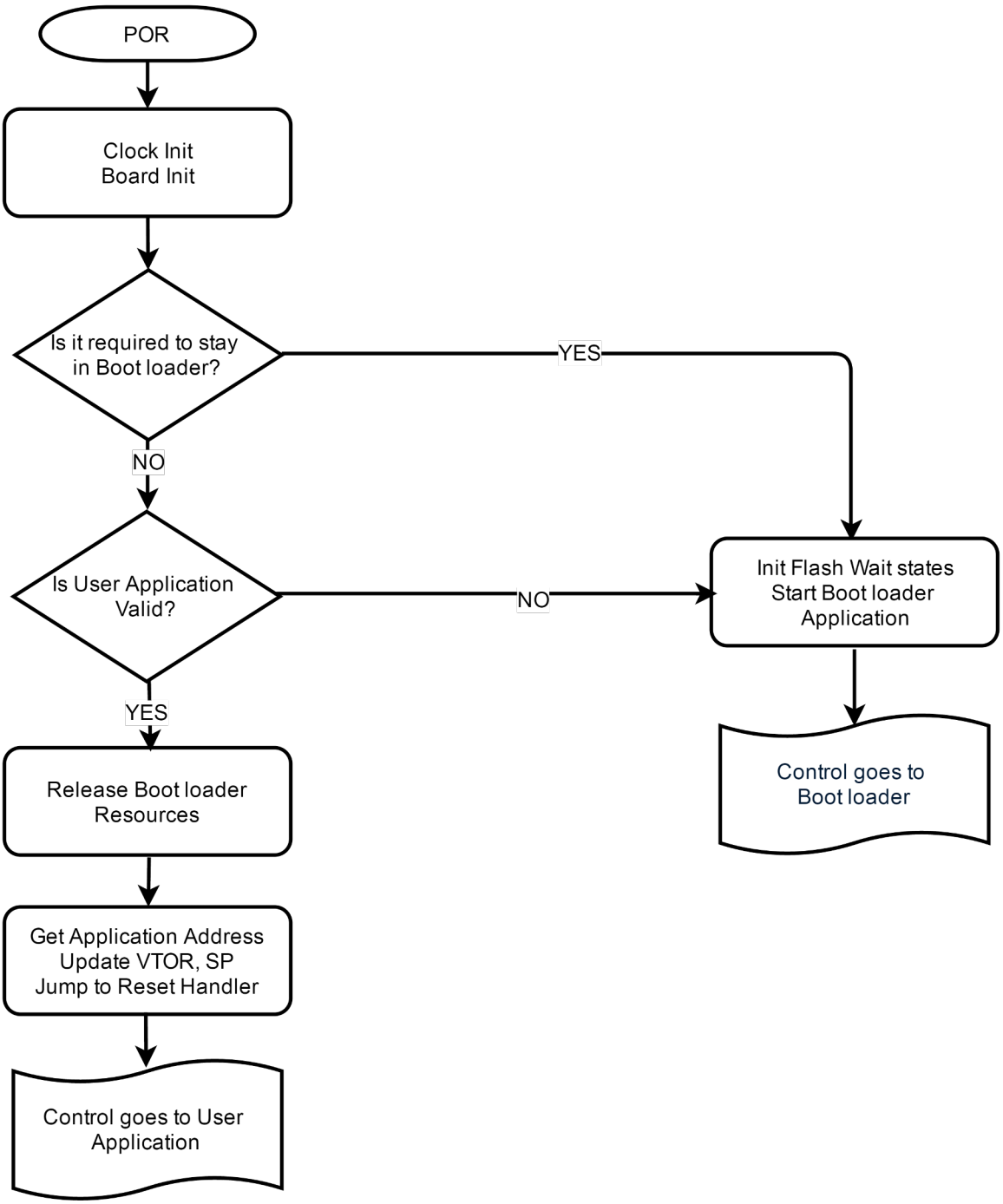
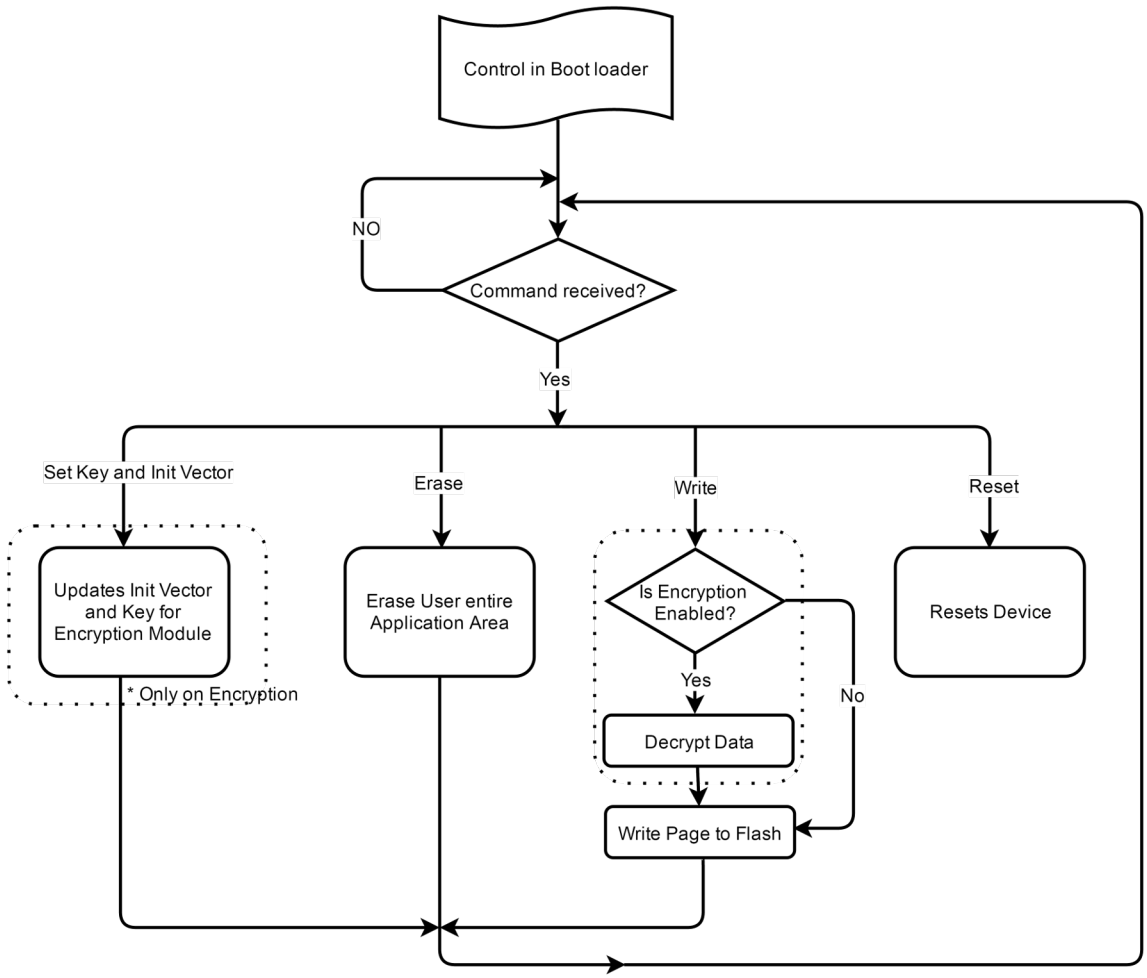


Figure 3-3. Firmware Upgrade Sequence



3.3.4.2. User Application to Bootloader Application

When the user application receives a request for firmware upgrade,

1. Reads user application Footer to know bootloader application Start Address and Length.
2. Reads bootloader application Footer for Bootloader Jump handler and executes it.
   - Bootloader Jump handler updates Jump Signature with “StayInBootLoader”.
3. At this point, a soft reset is required. On receiving soft reset, control goes to bootloader application and remain there as Jump signature indicates it.

3.3.5. Linking Options

The linker modifications are done for this example project. Linker file is updated to

- Define application’s start address and length in `rom` section
- Define footer at end of application section
- Reserve 32 bytes of RAM at the start so that both applications use it
- Fill unused locations with known data (0xFF)

In this example, bootloader application and user application are defined at following locations:

Table 3-4. Memory Allocations

Application Type	Section	Flash Start Address	Reserved Space
Bootloader Application	Application	0x00400000	0x0000BF00
	Footer	0x0040BF00	0x00000100
User Application	Application	0x0040C000	0x001F3F00
	Footer	0x005FFF00	0x00000100

32 bytes of RAM is reserved so that space is available for both bootloader and user application to share next steps to carry on.

Linker options for both bootloader and user applications are modified as follows:

- `rom (rx)` `ORIGIN` and `LENGTH` updated to match above
- Created `ApplicationFooter (rx)` section to hold footer information
- Created `JumpSignature (rwx)` section to reserve RAM
- Fill unused locations of the application with known data (0xFF)

3.3.6. User Application Customization for Bootloader

The following section provides detailed steps to convert a standard user application compatible with this bootloader application. This process requires

1. Remapping of flash locations
2. Adding application footers
3. Reserving RAM location
4. Fill unused locations with 0xFF
5. Define sequence for bootloader switching

3.3.6.1. Remapping of Flash locations

It is important to avoid storing User Application in the Bootloader Application locations. As described in previous sections, User Application should be remapped to different location. This can be achieved with help of linker file. For example,

```
rom (rx): ORIGIN = 0x0040C000, LENGTH = 0x001F3F00
```

This sets start address of the User Application to the address specified as ORIGIN defined above and allocates specified LENGTH bytes for User Application.

Ensure User Application start address and length are updated in Footers because Bootloader Application reads this information before jumping to User Application.

Note: Ensure that the Flash information stored in Bootloader Footer and Application Footer are same.

3.3.6.2. Adding Application Footer

Application Footer is an important parameter used by both Application and Bootloader. They use these parameters to read critical information before switching to other application. It is important to place this at the end of Flash memory allocated to the User Application.

- Adding Footer in Application

```
/** Application Footer information */
__attribute__ ((section(".ApplicationFooterData")))
const TS_ApplicationFooter sUserApplicationFooter =
{
    {
        U32_USB_CAN_FD_SW_RELEASE_MAJOR_VERSION,
```

```
        U32_USB_CAN_FD_SW_RELEASE_MINOR_VERSION,
        NULL,
        U32_BOOT_LOADER_APPLICATION_START_ADDRESS,
        U32_BOOT_LOADER_APPLICATION_ALLOCATED_SIZE,
        U32_USER_APPLICATION_START_ADDRESS,
        U32_USER_APPLICATION_ALLOCATED_SIZE,
    },
    {0},
    {0},
};
```

- Updates to be performed in User Application linker file
  - Add in Memory Space definitions

```
ApplicationFooter(rx) : ORIGIN = 0x005FFF00, LENGTH = 0x00000100
```

  - Add in Section definitions

```
.ApplicationFooter :
{
    KEEP(*(.ApplicationFooterData .ApplicationFooterData.*))
} > ApplicationFooter
```

3.3.6.3. Reserving RAM location

User Application and Bootloader Application uses few bytes of RAM to control applications switching between them. Both User Application and Bootloader Application must reserve the same location in both Applications to avoid conflicts. In the example project provided, 32 bytes of RAM is reserved for switching. This can be done with following modifications in linker file.

```
ram (rwx) : ORIGIN = 0x20400020, LENGTH = 0x0005FFE0
```

The value of `ram` ORIGIN is changed from default 0x20400000 to 0x20400020, which restricts linker from using first 32 bytes of RAM.

3.3.6.4. Fill Unused Locations with 0xFF

Filling unused locations with known data (0xFF) helps tool and Bootloader Application to synchronize during Hash functions. The Hash value generated by tool must match with value generated by Bootloader Application. Otherwise, Bootloader assumes User Application is invalid.

To Fill the unused locations with known data,

1. Create a dummy section in the User Application

```
/** Creating a dummy section to fill unused flash with 0xFF.. */
const U8 u8Dummy __attribute__ ((section(".fill"))) = 0xFF;
```

2. Fill this section with known data

   2.1. Allow linker to relocate initialized data

```
.relocate :
{
    . = ALIGN(4);
    _srelocate = .;
    *(.ramfunc .ramfunc.*);
    *(.data .data.*);
    . = ALIGN(4);
    _erelocate = .;
} > ram AT > rom
```

   2.2. Fill 0xFF till Application Footer

```
.Fill_FF :
{
    KEEP(*(.fill))
   FILL(0xFF);
   . = LOADADDR(.ApplicationFooter);
} AT > rom
```

3.3.6.5. Define Sequence for Bootloader Switching

This is another important step. User Application must define switching sequence to Bootloader. In the example project provided, this is achieved with help of USB-CDC command from host.

When this command is received, User Application calls `JumpToBootloaderHandler` to update JumpSignature with "`StayInBootLoader`" string to remain in Bootloader.

On soft reset, device starts executing from Bootloader and remains in Bootloader.

3.3.7. Modules Description

3.3.7.1. Bootloader Application Modules

The following table describes functions associated with the Bootloader Application.

Table 3-5. Functions and Descriptions

Function	Description
`static Bool B_IsItRequiredToStayInBootloader(void)`	This function checks if source application requested to stay in Bootloader Application • Check if Jump signature updated to remain in Bootloader Application • Clear this instruction to avoid reusing it
`static Bool B_IsUserApplicationValid(void)`	This function verifies User Application • Get application header data • Check if application footer is populated or not • Check Integrity of application • Indicate application is valid
`static void BootloaderJumpHandler(void)`	This function enables calling application to jump to Bootloader Application by updating Jump Signature • Update Jump Signature
`static void JumpToUserApplication(void)`	This function jumps to User Application after taking necessary steps in Bootloader Application. • Release resources before jumping to User Application • Disable interrupts • Get User Application header data • Update vector table • Update stack pointer • Call application reset handler

Function	Description
`U8 U8_EraseUserApplication(void)`	This function takes care of reading received message from USB buffers to application buffers and then triggers processing of those messages • Ensure User Application start address is on page boundary • Identify Pages & Sectors to Erase • Get No of Pages to Erase • Erase Pages / Erase Sector
`U8 U8_WriteToFlash(U32 u32Address, U8* pDataBuffer, U16 u16DataSize)`	This function checks whether address is on Sector boundary or not. Based on that it populates next address to Erase by adding either page(s) size or sector size • Check Address is with in User Application area • Check if Address in Page aligned or not • Check if Data size is matching with Flash Page size • Trigger Write request for entire page
`U8* P_DecryptReceivedPageData(void* pData, U16 u16DataSize)`	This function Decrypts input data of size `u16DataSize` • Enable peripheral clock • Configure the AES in CBC mode • Update Initial Vector • Loop through for all data • Feed in initial set of data • Read Decrypted data
`void CalculateICM(U32 u32StartAddress, U32 u32Size, U32* pu32Result)`	This function executes SHA-1 algorithm and updates result to output buffer. • Set region descriptor start address • Enable ICM • Check region hash is completed

3.3.7.2. User Application Modules

The following table describes functions associated with Bootloader Application in the User Application.

Table 3-6. Functions and Descriptions

Function	Description
`void JumpToBootloaderHandler(void)`	This function enables calling application to jump to Bootloader Application • Get application data from User Application Footer • Get Bootloader application Data from Bootloader Application Footer • Set to remain in Boot loader by calling application jump handler

3.4. Memory Footprint

This section provides memory utilization in Bootloader and User Application projects.

Table 3-7. Memory Details

Application Type	Program Memory Usage	Data Memory Usage
Bootloader	31596 bytes	13120 bytes
User Application	27516 bytes	14200 bytes

Note:

- Optimization level (-O1) enabled for both Bootloader Application and User Application
- ARM/GNU C Compiler version : 4.9.3
- Program Memory Usage bytes are excluding the Fill section which is filled with 0xFF for Hash functions.

4. Firmware Packager and Updater

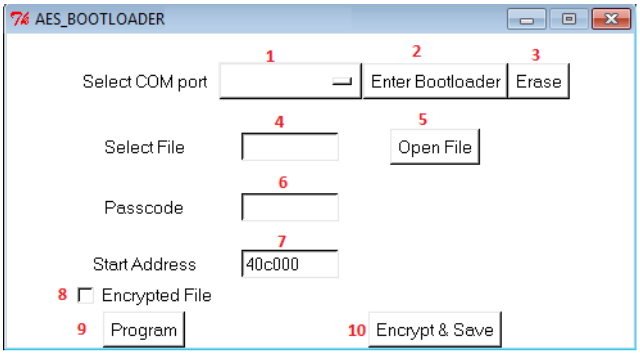
This helper program assists to Encrypt User Application and transmit the firmware using Bootloader Application. It is necessary to develop a program to interact with Applications using implemented communication protocol.

This application enables the user to:

- Trigger User Application to Bootloader Application
- Erase User Application Area
- Choose the firmware file to send to device
- Select AES Key and Initial Vector Indexes
- Launch firmware upgrade

The same application is used to prepare the firmware prior to sending it to customers. This include encrypting and generating Hash tag, etc.

Figure 4-1. PC Application



The following points explain the functionality of various options in the AES Bootloader application.

1. **Select COM port** – This drop down lists various Virtual COM ports available for the device.
2. **Enter Bootloader** – This triggers the application to switch between user application and Bootloader.
   When the button is clicked,
   1. Application queries the current status of the device
   2. Sends a command to switch to Bootloader
   3. Issues soft reset command

   Device returns to Bootloader and wait for commands from Application.
3. **Erase** – This triggers Erase Process on the device.
   Note: This command should be issued only when device is in Bootloader Application.
4. **Select File** – Displays File selected by User.
5. **Open File** – Enables user to select file.
6. **Passcode** – AES Key and Initial Vectors are predefined in device and PC Application. The passcode is combination of Key Index and Initial Vector Index (0201 – Chooses Key Index as 02 and Initial Vector Index as 01).
7. **Start Address** – Start Address of the User Application. This should be matched with Parameters in Bootloader and User Application. PC Application uses this to Start writing User Application from this Address.
8. **Encrypted File** – Indicates to application that chosen file is Encrypted. It allows PC Application to indicate the device to decrypt incoming data.

9. **Program** – Triggers Flash Programming on the device.
10. **Encrypt & Save** – Triggers Encryption process on PC Application and Prompts to Save result file.

4.1. Steps to Upgrade User Application

Upgrading User Application is divided into 2 steps based on Encryption is required or not.

4.2. Preparing Secure Application

To Encrypt the User Application,

1. Select input file using **Open File** option
2. Enter passcode in **Passcode** text box
3. Click on **Encrypt & Save**
   - This prompts **Save As** option, Please chose output file and Save

Now, encrypted file is ready for programming. Following information must be shared for programming

1. Encrypted User Application
2. Passcode
3. Start Address of the Application

4.3. Programming Application

To program the user application,

1. **Select COM port** enumerated by device.
   - This could be different for user application and bootloader application.
2. Click **Enter Bootloader**, This should make device to remain in bootloader.
3. Again, **Select COM port** enumerated by device.
4. Click **Erase** button to erase user application.
5. Select the input file using **Open File** option.
6. For encrypted application, enter **Passcode** and check **Encrypted File** option. For non-Encrypted Application, leave **Passcode** empty and uncheck **Encrypted File** option.
7. Update **Start Address**.
8. Click **Program** Button.

5. References

1. Safe and Secure Firmware Upgrade for AT91SAM Microcontrollers - http://www.atmel.com/images/doc6253.pdf
2. Federal Information Processing Standards 180-2 - http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

6. Revision History

Doc. Rev.	Date	Comments
42725A	06/2016	Initial document release.

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