Atmel®

APPLICATION NOTE

# Atmel AT02333: Safe and Secure Bootloader Implementation for SAM3/4

**Atmel 32-bit Microcontroller**

## Features

- Getting familiar with the conception of in-field upgrading and bootloader
- Discussing design considerations in the development of safe and secure bootloader
- Understanding how the bootloader for SAM3/4 works

## Description

This document introduces in-field firmware upgrading and describes various aspects of the implementation of a safe & secure bootloader. It discusses several design considerations in developing this kind of software, which readers can refer to when consulting this document.

Correct bootloader implementation poses several challenges, such as correctly remapping the chip memory with the new-loaded program. Those issues will be described and solved in the following sections, with focus on Atmel® ARM® Cortex®-M based microcontroller. Simple bootloader software is also provided along with this application note as example.

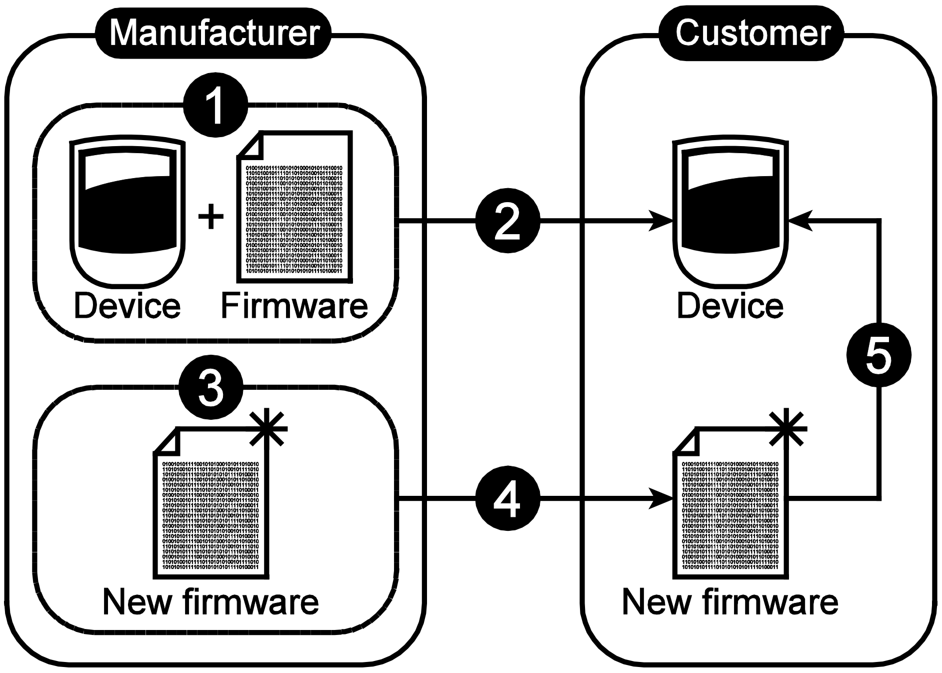
## Table of Contents

## 1. Introduction

Microcontrollers are used increasingly in a variety of electronic products. The devices are becoming more flexible, thanks to the reprogrammable memory (typically flash) often used to store the firmware of the product. This means that a device which has been sold can still be upgraded in-field, e.g., to correct bugs or add new functionalities.

Figure 1-1 illustrates this concept.

**Figure 1-1. In-field Upgrading**



1. Manufacturer designs a device and an initial firmware
2. Devices are sold to customers
3. Manufacturer develops a new version of the firmware
4. New firmware is distributed to customers
5. Customers patch their devices with the new firmware

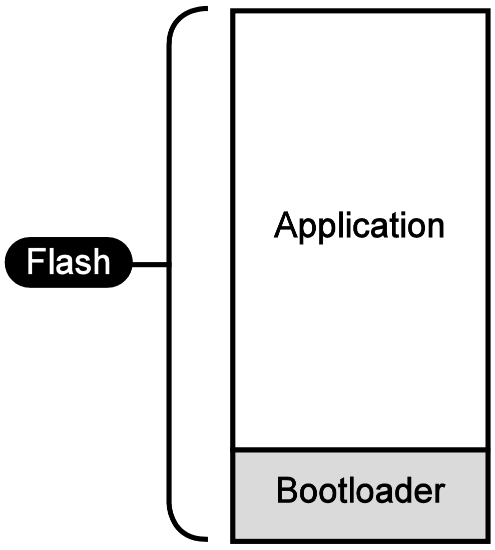
# 2. In-field Upgrading

## 2.1 The Bootloader

Many 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 piece 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 often called a bootloader, as its role is to load a new program at boot.

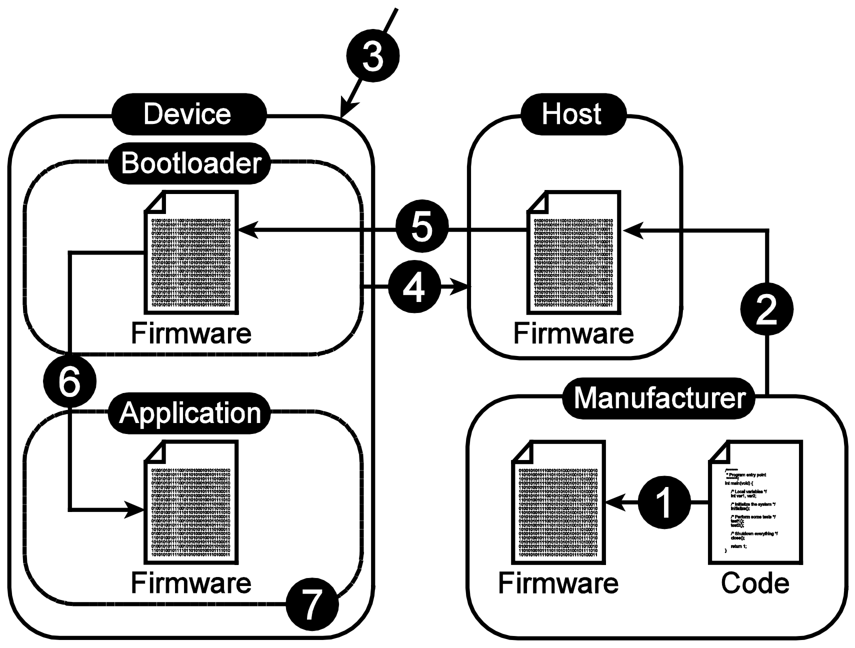
A bootloader always resides in memory to make it possible for the device to be upgraded at anytime. Therefore, it must be as small as possible. Since one does not want to waste a large amount of memory on a piece of code which does not add any direct functionality for the user.

**Figure 2-1. Memory Organization with a Bootloader**



To download a new firmware onto the device, there must be a way to tell the bootloader to prepare for the transfer. There are two types of trigger conditions: hardware and software. A hardware condition might be a pressed button during a reset, whereas a software condition could be the 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 try to connect to a host and wait for a new firmware. This host can theoretically be any device; however, a standard PC with the appropriate software is most often used. The transmission of the firmware can be done via any media supported by the target, i.e., RS232, USB, CAN and so forth.

**Figure 2-2. Firmware Upgrade Using a Bootloader**



1. Manufacturer releases a new firmware version
2. New firmware is distributed to users
3. Boot condition is triggered by customer
4. Bootloader connects to host
5. Host sends the new firmware
6. Existing application is replaced
7. New application is run

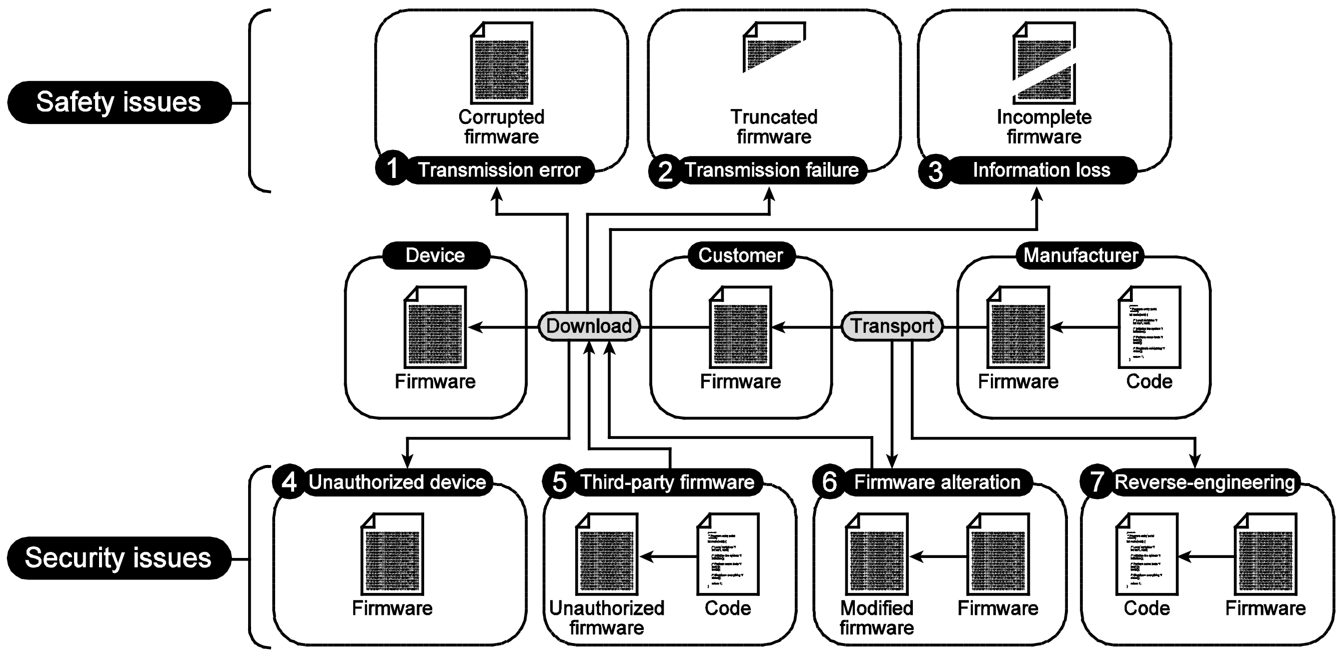
Once the transfer is finished, the bootloader replaces the old firmware with the new version. This new application is then loaded.

There are several other ways of carrying out in-field programming of a product. For example, the main application might do the upgrade itself: for a device using external memory storage, the new firmware could be written on it as a file. The main advantage of using a bootloader is that you do not have to design your application in a different way. Therefore, while modifying your application to include an upgrade mechanism can be tedious, a bootloader can always be used without additional programming (providing the bootloader is readily available, of course).

## 2.2 Issues

There are several issues associated with using such a simple bootloader. The issues might happen at two points of the upgrading flow: either during the transport of the firmware from the manufacturer to the customer or during the download on the target device. Figure 2-3 is a diagram showing several issues which will be discussed in the following sections.

**Figure 2-3. Upgrading Flow Issues**



### 2.2.1 Safety

It is critical for most devices to have a working firmware embedded in them at all times, since they probably cannot function properly or even cannot function at all without it. However, the use of a bootloader can result in the problematic situation, where the new firmware has not been installed properly, compromising the behavior of the system. The following problems (in Figure 2-3) might happen:

- Issue #1, Transmission error: As a result, part of the code is corrupted.
- Issue #2, Transmission failure: As a result, the application area would then be corrupted and unusable. This issue arises:
  - If the device suddenly loses power during the update process.
  - If the connection to the host is lost during the transmission.
- Issue #3, Information loss: some data might be lost while transmitting the firmware, which would completely corrupt the code after the missing part.

### 2.2.2 Security

Securing a system means enforcing several features: **privacy**, **integrity** and **authenticity**.

**Privacy** means that a piece of data cannot be read by unauthorized users or devices. A major concern of firmware developers is to ensure that the application they have designed cannot be leaked by competitors. They thus want their code to be private, the target devices being the only authorized "users".

Microcontrollers typically provide a mechanism making it impossible for malicious users to read the program code written in the device. However, for in-field firmware upgrading, the manufacturer has to give the new code image to customers so they can patch their devices themselves.

This means that a skilled person could potentially decompile it to retrieve the original code (issue #7 in Figure 2-3).

**Authenticity** makes it possible to verify that the firmware is from the manufacturer itself, not anybody else. Indeed, another problem of reprogrammability is that a device could be given a firmware which is not designed by the original manufacturer, but by a third-party (issue #5 in Figure 2-3). This may be especially problematic if that firmware is developed for malicious use, i.e., to bypass security protections, to illegally use critical functions of the device, and so forth.

A genuine firmware could also be used on a different device other than the one that it is intended for (issue #4 in Figure 2-3). This could be an unauthorized hardware copy of the product, or a device designed for hacking purpose. This is again an authenticity concern, this time regarding the target device.

Finally, **integrity** is required to detect a modification of the data. For example, an authorized firmware may be slightly modified (issue #6 in Figure 2-3). It would appear as genuine, but attacks that are similar to those described in the previous paragraph could be achieved in this way.

Possible security issues list:

- Issue #4, Use of a firmware on an unauthorized device.
- Issue #5, Use of an unauthorized firmware.
- Issue #6, Firmware modification.
- Issue #7, Firmware reverse-engineering.

Without any kind of security feature, a firmware will be subject to all attacks regarding privacy, integrity and authenticity. Therefore, some techniques are needed to enforce those three aspects.

## 2.3 Possible Solutions

The following two sections offer a practical solution to the issues identified in previous sections. However, most of the techniques to circumvent those problems present a trade-off between the level of security and safety and the size and speed of the system. As such, the safest and most secure solution is also probably the biggest and slowest one. This means that one must first carefully analyze what is needed in terms of security and safety in a system, to implement only the required functionalities.

Several techniques for enforcing safety and security are presented in the following sections. Please note that no software solution can give perfect security. Indeed, there are many hardware-based attacks (like micro-probing, power analysis, timing analysis, etc.) which enable a malicious user to break software protections. These attacks are best solved by using a dedicated secure chip. However, using soft protection is not inappropriate, as they make it more costly (both in time and money) to attack your system.

## 2.4 Safety Solutions

The following techniques are ways to prevent safety-related errors from happening. However, it is interesting to note that, since the bootloader should never be compromised (as it cannot be updated), the user can simply try upgrading his device again if there is a failure. Naturally, this may not always be a desirable alternative, which is why the following solutions are presented.

### 2.4.1 Communication Protocol Stack

A protocol stack is used in most communication standards to offer, among other features, reliable transfers. This reliability is important for a bootloader, as the firmware must not be corrupted during the download (see issue #2 and #3 in Figure 2-3).

In the OSI standard model, a communication system is divided into seven layers, which form the protocol stack. Each layer is responsible for providing a set of features. For example, the physical layer is responsible for the physical interconnection of the devices. Reliability is typically implemented at the transport layer.

Transport reliability is usually obtained by using several techniques: error detection/correction code, block numbering and packet acknowledgement. They are presented below. Existing protocols for the media available on Atmel® ARM® Thumb®-2 based AT91SAM microcontrollers are then described in Section 2.4.1.4.

### 2.4.1.1 Error Detection/Correction

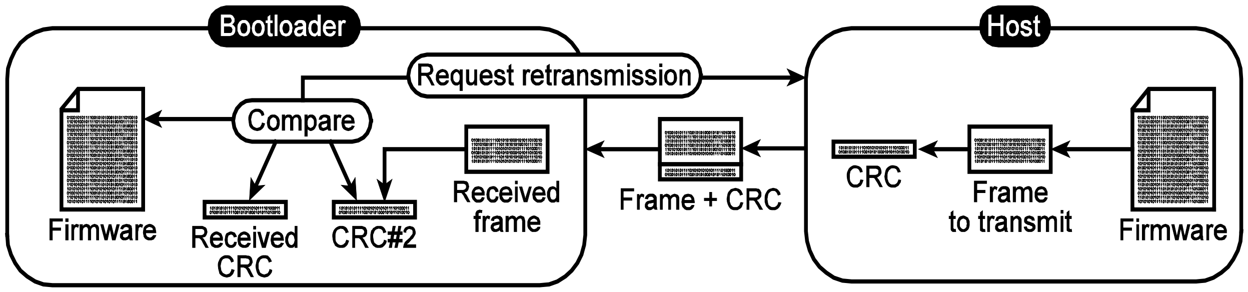
It is common to use **error detection** and **error correction** codes for transmitting data. Indeed, many operations cannot handle receiving a corrupted piece of data: loading a new firmware, sending a file across a network, and so on. Therefore, several codes have been designed to make it possible to detect and even correct transmission errors.

**Detection codes** use simple mathematical properties to compute a value over the data which is to be sent. That value is then transmitted along with the original data. When the target receives the data, it recomputes the value and compares it to the one it has been given. If both are equal, the transfer was successful; otherwise, there are one or more invalid bits.

**Correction codes** work in the same way, except that they are able to detect errors, as well as to recover some of them. This is useful to avoid requesting the sender to retransmit the erroneous data.

To be useful, error detection/correction cannot be carried out on the whole firmware. Since it is written into the memory as it is transmitted, it would be pointless to detect an error only when the file has been completely received. Instead, the firmware is transmitted in small pieces called frames. A code is calculated and checked for each frame; if an error is detected, it is either corrected via the code (if possible), or the frame is retransmitted.

**Figure 2-4. Error Detection during Firmware Transmission**



However, there are limitations to error detection/correction codes. Depending on how they are mathematically constructed, they will have a maximum number of detectable/correctable errors. As such, a thorough analysis of the system must be carried out prior to selecting the method to use, to avoid choosing an inappropriate code.

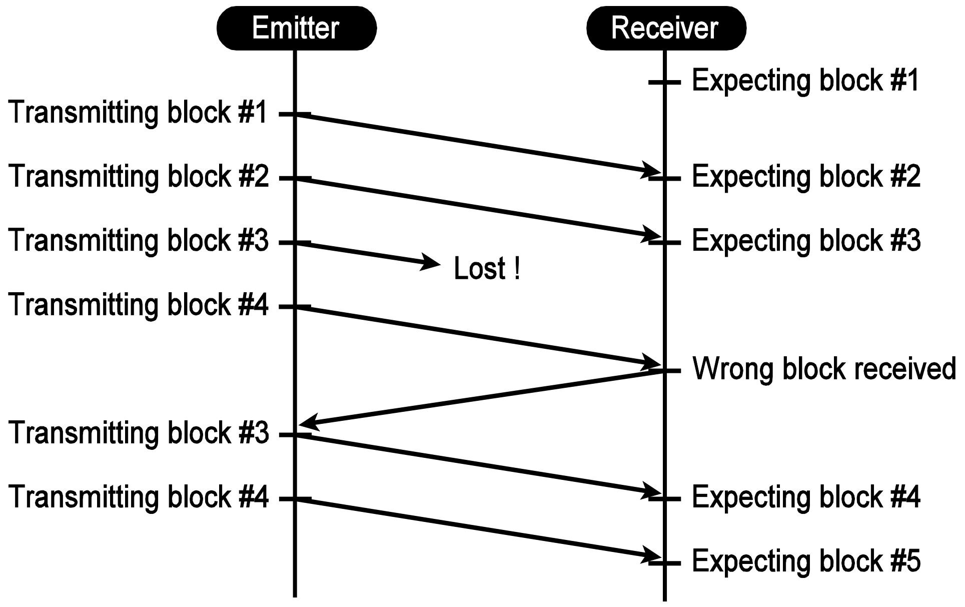
Finally, error correction is not really necessary in this particular case. It is most suited when it is unpractical to resend the erroneous data, which is not a concern here. Since error correction codes typically incur a bigger overhead than error detection ones, they should be implemented with good reasons.

### 2.4.1.2 Block Numbering

The purpose of block numbering is to avoid losing a block of data or having two blocks arrive in the wrong order. This is critical in a file-oriented transfer such as firmware downloading: those errors would render the received code unusable.

As its name suggests, block numbering is simply about adding a sequence number to each transmitted block. This number increases by one for each block. Therefore, the receiver can easily detect that two blocks have been swapped if it gets block #3 before block #2. Likewise, if the sequence goes straight from #3 to #5, then block #4 was lost.

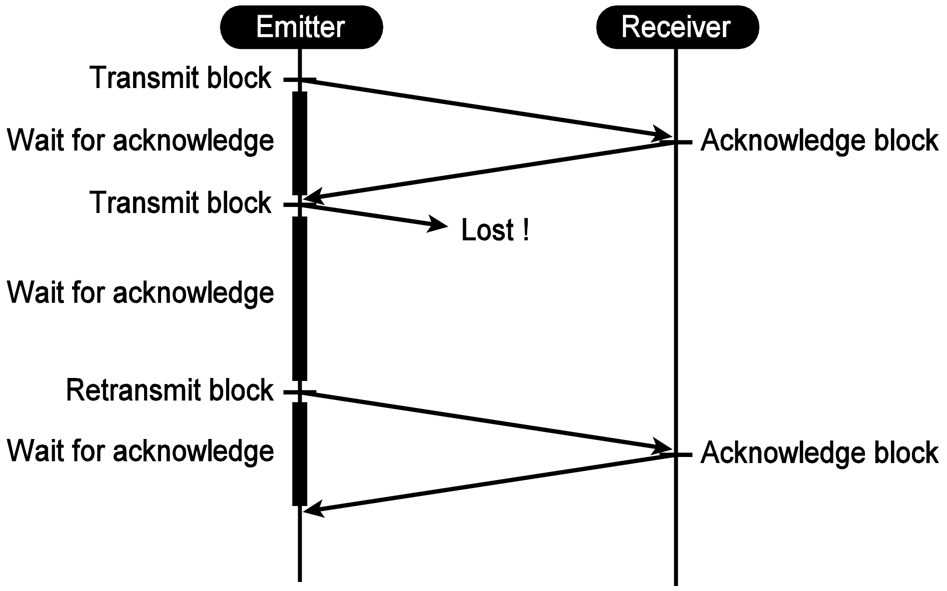
**Figure 2-5. Block Numbering**



### 2.4.1.3 Packet Acknowledgement

Packet acknowledgement works in the following way. Each time the sender transmits a block of data, it waits for the receiver to acknowledge it (i.e., reply that it has been correctly received). If nothing comes back after a fixed amount of time, then the sender assumes that the packet has been lost, and retransmits it.

**Figure 2-6. Packet Acknowledgements**



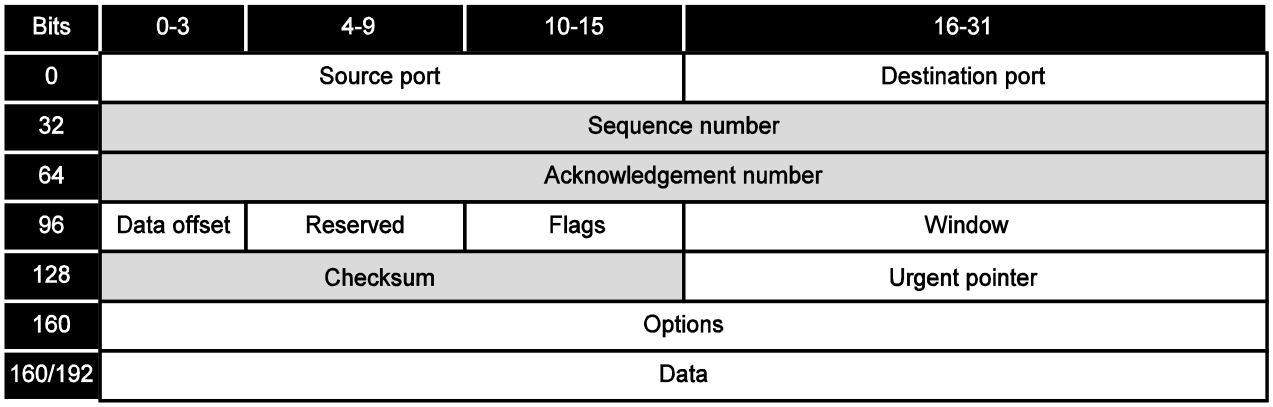
No other data is sent by the emitter while it is waiting for an acknowledgement. Therefore, packets cannot be received out of order since only one is sent at a time.

### 2.4.1.4 Existing Protocols

A communication medium (such as RS-232 or Ethernet) is rarely used as it is. A protocol stack is most often required to make full use of the communication medium.

TCP/IP is the most widely used protocol stack on top of Ethernet. The Transport Control Protocol (TCP) implements reliability by using a packet sequence number as well as a checksum (a simple error detection code). It also uses a variant of packet acknowledgment. But since several packets can be sent at once, they can arrive out of order (thus block numbering is still needed).

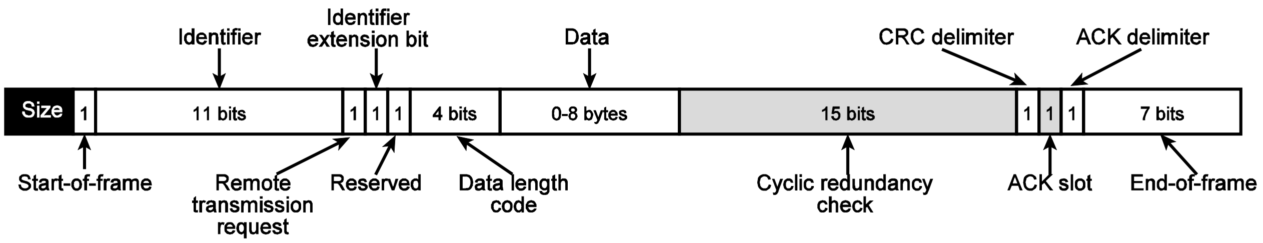
**Figure 2-7. TCP Frame Structure**



| Bits | 0-3 | 4-9 | 10-15 | 16-31 |
|---|---|---|---|---|
| 0 | Source port | | | Destination port |
| 32 | Sequence number | | | |
| 64 | Acknowledgement number | | | |
| 96 | Data offset | Reserved | Flags | Window |
| 128 | Checksum | | | Urgent pointer |
| 160 | Options | | | |
| 160/192 | Data | | | |

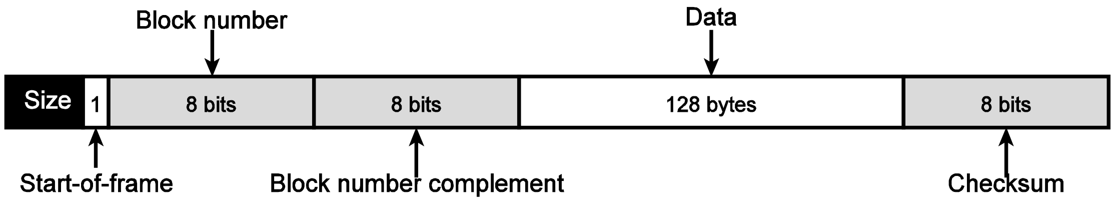
The USB protocol uses a Cyclic Redundancy Check (CRC) for error detection. There is no sequence number on the packets, but a receiver acknowledges each block of data. This is the same for the CAN bus.

**Figure 2-8. CAN Frame Structure**



Lastly, there are several file-oriented communication protocols for the RS-232 interface. One of them is X-MODEM, which was developed in the 1970's. It features a simple single byte checksum, block numbering and packet acknowledgment.

**Figure 2-9. X-Modem Frame Structure**



## 2.4.2 Memory Partitioning

The main idea of memory partitioning is to have, at all times, a copy of a working firmware somewhere in memory. Achieving this means that even if anything goes wrong during an update, it is still possible to revert back to that firmware.

### 2.4.2.1 Memory & Memory Banking

The memory embedded in microcontroller is usually organized in one or several banks (or planes), which are hardware dependent. In practice one (single) bank or two (dual) banks are often used. The dual bank function enables programming one bank while the other one is read (typically while the application code is running). It also enables boot code selection by mapping different physical banks to boot program area.

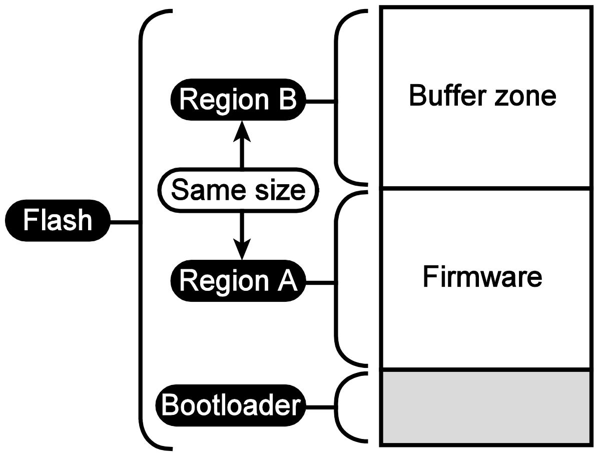
The following sections will describe solutions based on single banked or dual banked memory.

#### 2.4.2.2 Single Banked Partitioning

The solution presented here takes a little twist on that technique. The memory is partitioned at all times in two distinct regions (excluding the bootloader region):

- The application code (region A)
- A buffer for the new firmware (region B)

**Figure 2-10. Single Banked Memory Organization with Partitioning**



Region B is used as a buffer. The new firmware is downloaded to this region and verified. If firmware downloaded on this region is verified “OK”, it is programmed to region A, so that downloading errors will not affect the working firmware on region A. This method ensures that there is always a working firmware on a device after an upgrade, whether it was successful or not.

For more details on single banked partitioning consideration and implementation, please refer to Section 3.1.3.1.

#### 2.4.2.3 Dual Banked Partitioning

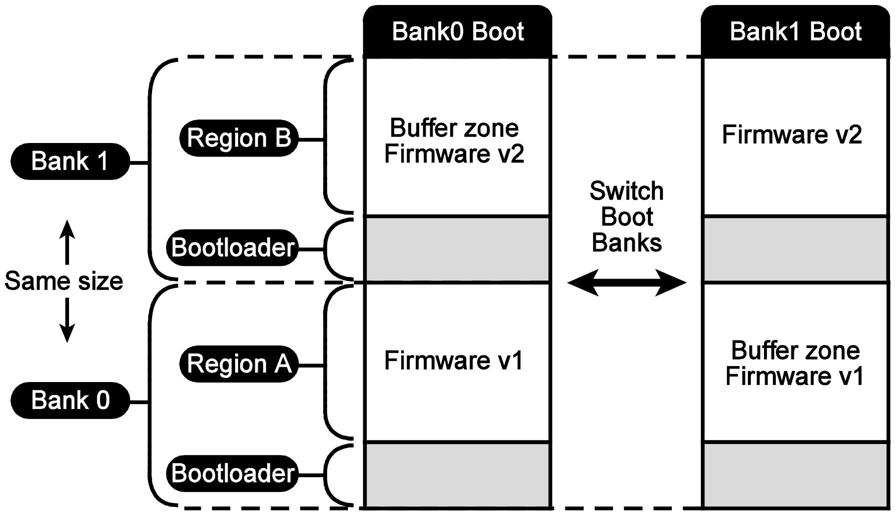
Since the memory is already distinguished into banks, and they can be mapped to boot area to boot in this case, the partitioning just takes the advantage of the organization: each bank has the same copy of bootloader at the beginning and then the firmware follows. When booting from one bank, another bank is used as upgrade buffer, to accept new firmware. After the new firmware is received and verified, the boot banks are switched. Thus there can be two workable firmware versions in memory. To load different firmware version, just change the startup memory mapping to map the bank which contains that firmware version as the startup bank, as follows:

- At first, memory bank 0 is boot bank and firmware (v1) on it runs.
- Firmware region in bank 1 is used as upgrade buffer zone, which accepts the newly downloaded firmware (v2).
- After downloading firmware (v2) to bank 1 and verifying it “OK”, the boot bank is changed to bank 1, that is, bank 1 is mapped to boot area for the next system boot.
- Now, memory bank 1 is boot bank and firmware (v2) on it will run on start-up.
- When system restarts, it runs firmware (v2) on bank1. This time firmware region on bank 0 is used as upgrade buffer zone, to receive new firmware (the old firmware is not deleted, and the new downloaded firmware will then overwrite it in this zone).

The advantage of this method is that there are maximum two workable firmware versions in memory, and there is no need to perform additional programming (simply changing the boot mapping instead of copying firmware from region to region), but in each bank there must be a bootloader so that the available application memory is a bit smaller than single banked solution.

For more details on dual banked partitioning consideration and implementation, please refer to Section 3.1.3.2.

**Figure 2-11. Dual Banked Memory Organization with Partitioning**



### 2.4.3 Summary

Error detection and correction (Section 2.4.1.1)

- Pros
  - Detects transmission errors
- Cons
  - Code must be chosen wisely
  - Slightly increased code size
  - Slightly reduced speed (during upgrade only)

Memory partitioning (Section 2.4.2)

- Pros
  - Solves all safety-related issues
- Cons
  - The required memory size is doubled
  - Slightly increased code size
  - Reduced execution speed (during upgrade only, for dual banked memory, it is not a problem since code need not be copied)

## 2.5 Security Solutions

Several security-related techniques to solve the afore-mentioned issues (see Section 2.2.2) are presented in this section. See Section 2.6.2 for in-depth information about security considerations.

### 2.5.1 Integrity

Verifying integrity means checking the following:

- Purposeful modification of the firmware
- Accidental modification of the firmware

Accidental modification is a safety problem. It is typically solved by using error detection codes (see Section 2.4.1.1).

There are several ways to verify that a firmware has not been voluntarily modified by a mischievous user. They are presented in the following paragraphs, and make it possible to solve issue #6 described in Figure 2-3.

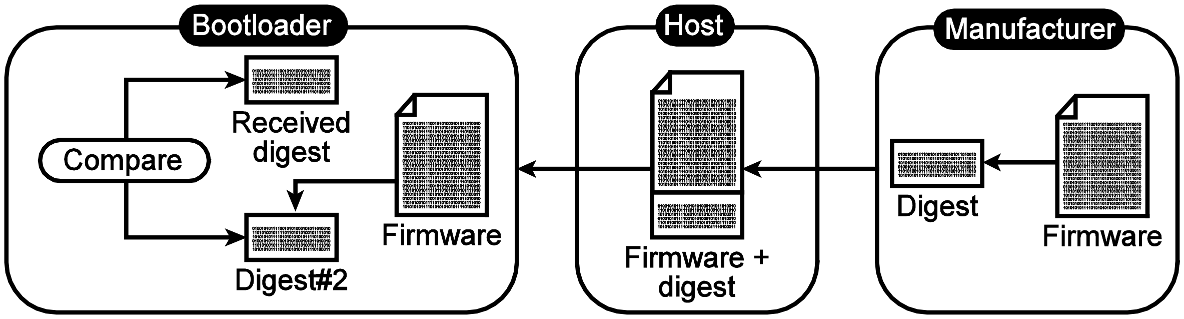
#### 2.5.1.1 Hash Function

Conceptually, the goal of a hash function is to produce a digital “fingerprint” of a piece of data. This means that, 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 ones. If they 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, e.g., a good diffusion (the ability to produce a completely different output even if only one bit of the input is flipped).

**Figure 2-12. Firmware Hashing**



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

The downside of simply hashing the firmware is that anybody can do it. This means that an attacker could modify the file and re-compute the hash. The bootloader would thus not be able to tell that an alteration was made.

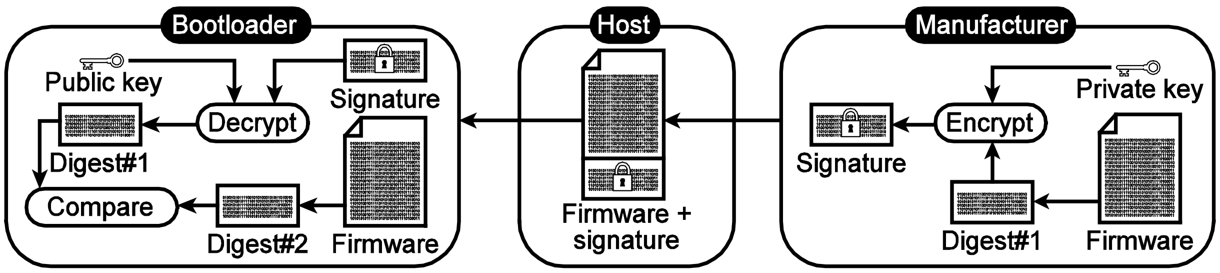
However, a hash function alone can still be used to verify the firmware integrity at runtime, to avoid running a damaged application.

#### 2.5.1.2 Digital Signature

Since a hash can be easily recomputed, the solution is to encrypt it. This is the basis of a digital signature: the digest of a firmware is computed (using a hash function) and then encrypted using public-key cryptography. This produces a digital signature, akin to the signatures used in everyday life.

Public-key (or asymmetric) encryption relies on the use of two keys. The manufacturer uses his private key (secret) to encrypt the signature, while the device uses the corresponding public key to decrypt it.

**Figure 2-13. Digital Signature Creation and Verification**

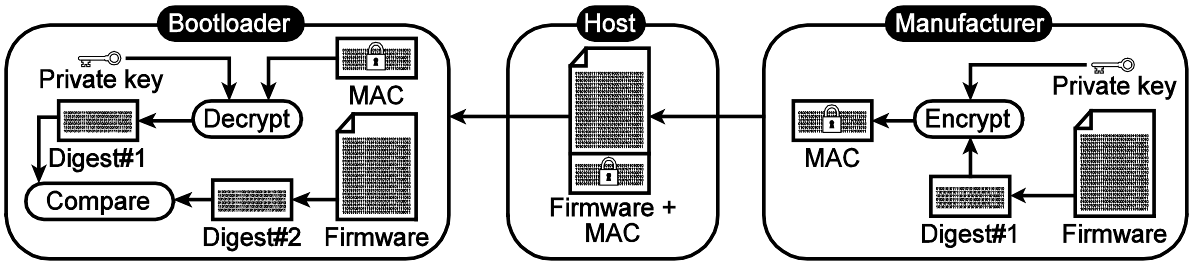


Since only the private key can encrypt data, nobody except the manufacturer can produce the signature. Thus, a malicious user would not be able to perform the attack described in the previous section. But anybody can verify the signature using the public key of the manufacturer.

#### 2.5.1.3 Message Authentication Codes

Message Authentication Codes (MACs) provide the same functionality as digital signatures, except that they use **private key cryptography**. Modern private key encryption algorithms (also called ciphers) are mostly **block ciphers** (i.e., they work on a block of data of a fixed size), as opposed to **stream ciphers** (which work on a stream of data).

**Figure 2-14. Message Authentication Code Verification**



Private Key encryption relies on only one secret key, which is shared between the manufacturer and the devices. This has several implications, compared to a digital signature:

- Anybody who can verify a MAC can also produce it.
- If the private key inside the device gets exposed, then the security of the system is completely compromised.

The first point is not a concern in practice; as a device will not use its private key to produce MACs, only to verify them. The second implication means that if an attacker manages to retrieve the key from the bootloader (which is supposed to be locked using security bits), then he will be able to modify a firmware and still have it accepted as unmodified by a target. Depending on your requirements, this may or may not be an issue.

It should be noted that since private-key cryptography is much faster than public-key, a MAC will be computed and verified faster than a digital signature. But since only one MAC/signature is required for the firmware, it would probably not make a big difference in practice.

### 2.5.2 Authentication

Authentication is about verifying the identity of the sender and the receiver of a message. In the case of the bootloader, this means verifying that the firmware has been issued by the manufacturer, and that the target is a genuine one. It solves issue #4 and #5 described in Figure 2-3.

It happens that the methods which provide authentication also provide integrity: **digital signatures** and **MACs**. Since they are described from the integrity point of view (see Section 2.5.1), this section only discusses their authentication properties.

This section only discusses firmware authentication; authentication of the target device will be treated further.

#### 2.5.2.1 Digital Signature

Only the manufacturer is supposed to possess the private key used to produce the signature attached to a firmware. This means that any valid signature (once decrypted using the corresponding public key) will certify that the signed data comes from the manufacturer and not from anyone else.

However, since the signature is freely decipherable by anyone possessing the public key (which is not supposed to be secret), the computed hash of the firmware can be obtained by anyone. This means that an attacker could find a collision in the hash function which is used, e.g., two different texts giving the same hash. The signature would also authenticate this data as produced by the original sender.

This may not be a problem in practice however, as a collision is extremely hard to find and it is unlikely that it would result in a valid program. It would only enable a malicious user to create a fake firmware which would render the device unusable.

#### 2.5.2.2 Message Authentication Code

Converse to a signature, a MAC cannot be used to certify that it is the sender who created the message. Indeed, since the receiver also has the private key used to compute the MAC, he may have generated it. The advantage is that only the two parties can decrypt the MAC, preventing anyone else from verifying that the message is indeed valid.

In practice, this does not create an issue as only the firmware would be MAC'ed. The bootloader would not use its private key to generate any other MAC, thus achieving the same authenticity verification as a digital signature.

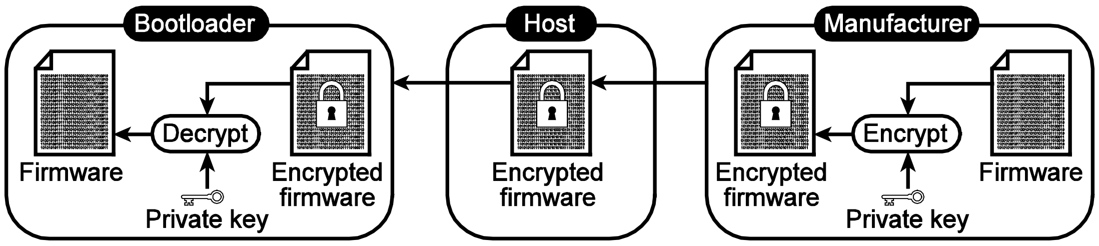
The additional concern of a MAC compared to a digital signature is that an attacker should never be able to retrieve the private key inside the bootloader. If he manages to do that, he would be able to create or modify a firmware, issuing the associated MAC needed to authenticate it as a genuine one to any target.

### 2.5.3 Privacy

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 (plain) original one. Without the required decryption key, the data will look like complete nonsense, preventing anyone unauthorized from reading it. This takes care of issue #7, as described in Figure 2-3.

In practice, a private-key algorithm is used to generate the encrypted firmware. It is obvious that 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 between the bootloader and the manufacturer.

**Figure 2-15. Firmware Encryption**



Note that code encryption does not solve every security issue all by itself. For example, the firmware might still be modified, even if it is quite difficult. An attacker could manage to pinpoint the location in the code of an important variable and tweak it until he gets the desired result.

Code encryption also combines itself well with a message authentication code. Since they both use a symmetric encryption algorithm, they can use the same one to save code size. There are also secure modes to combine both a block cipher and a MAC while using the same key (see Section 2.6.2.1).

### 2.5.4 Target Device Authentication

There are two ways of verifying that a device is genuine. The first one is passive, i.e., no special functionality is added to perform the verification. Instead, the authenticity of the device is implicitly checked by other security mechanisms.

In this particular case, encrypting the firmware will also authenticate the device. Indeed, it will need the private key to decrypt the firmware. As only genuine device shave them embedded in their bootloader, an unauthorized target will not be able to recover the original code and run the application.

This is especially applicable as target authentication cannot be achieved without encrypting the code anyway; otherwise, it could be simply downloaded to the device.
An active authentication method would involve adding an authentication technique for the target. Since the device identity would be verified during the upgrade process by the host, a message authentication code cannot be used. Indeed, since it would require the host to have the private key, an attacker could easily retrieve it.

Adding such a mechanism would also incur a significant overhead, both in terms of bootloader size (for storing the additional key and the digital signature encryption algorithm) and upgrade speed (because of the transactions needed to identify the device). In addition, the host upgrading program could be modified to get rid of that additional mechanism anyway.

### 2.5.5 Summary

Hash function (Section 2.5.1.1)

- Pros
  - Detects accidental and voluntary changes
  - Can be used to check firmware integrity at runtime
- Cons
  - Can be recomputed by a malicious user
  - Slightly increased code size
  - Slightly reduced execution speed

Digital signature (Section 2.5.2.1)

- Pros
  - Detects third-party and modified firmware
  - If the key inside the bootloader is compromised, the system remains safe
- Cons
  - Slower than a MAC
  - Requires a large key length
  - Increased code size
  - Reduced execution speed (during upgrade only)

Message authentication code (Section 2.5.1.3 and Section 2.5.2.2)

- Pros
  - Detects third-party and modified firmware
  - Faster than a digital signature
- Cons
  - If the key inside the bootloader is compromised, the system is broken
  - Increased code size
  - Slightly reduced execution speed (during upgrade only)

Code encryption (Section 2.5.3)

- Pros
  - Prevents reverse-engineering
  - Authenticates the target
- Cons
  - If the key inside the bootloader is compromised, the system is broken
  - Increased code size
  - Reduced execution speed (during upgrade only)

## 2.6 Design Considerations

There are several choices and problems which arise when designing a bootloader such as the one described in this application note. This section gives an overview of the major topics.

### 2.6.1 Transmission Media

SAM microcontrollers provide a wide variety of peripherals to communicate with an external host, such as:

- USB
- CAN
- RS232
- Ethernet
- External memory (e.g., DataFlash®)

Choices should be made on the implementation priority (and relevance) of each method. Given the simplest one to implement is probably RS232; it could be used to get the system ready. Other interfaces could then be added in an easier way.

It should be noted that there is a USB device class geared toward firmware upgrading. This class, referred to as Device Firmware Upgrade (DFU), may be used to implement the bootloader functionality. However, please note that since it is not supported by Microsoft® Windows®, it may not be easy to do so.

Finally, media such as CAN or Ethernet have the potential to allow for batch programming, i.e., programming several devices at once. The host could broadcast all the messages it sends, enabling every connected device to receive them and upgrade their firmware.

### 2.6.2 Cryptographic Algorithms

The secure part of the bootloader relies on different types of cryptographic primitives (hash functions, MACs, digital signature algorithms, block ciphers, etc.). But for every type of primitive, there are many different algorithms to choose from.

This section tries to give a brief overview of the choices available for the following: symmetric block ciphers, hash functions, message authentication codes, digital signature algorithms and pseudorandom number generators (PRNG). While the latter has not been introduced before because it is not a security method itself, it is critical to the design of a secure system.

For further recommendations, you may also look at those made by committees such as CRYP-TREC or NESSIE, which carefully analyze existing and new algorithms.

### 2.6.2.1 Symmetric Ciphers

Symmetric (or private key) ciphers are used both for computing MACs and encrypting the program code. Thus, they are an important part of the bootloader security and must be chosen wisely.

A symmetric encryption algorithm can be defined and chosen by several characteristics:

- **Key length in bits**: The larger it is, the more difficult it is to perform a brute force attack. A reasonable length seems to be128 bits at the moment, as it is the one key length selected for the Advanced Encryption Standard (AES) cipher.
- **Block length in bits**: is needed to avoid a "codebook attack". Most block ciphers will use at least 64 bits, with modern ciphers using at least 128 bits.
- **Security**: If the algorithm has flaws, then it may be easily breakable.
- Size & speed: depend on the techniques they use. Since embedded system has limits in resources, more suitable encryption should be chosen.

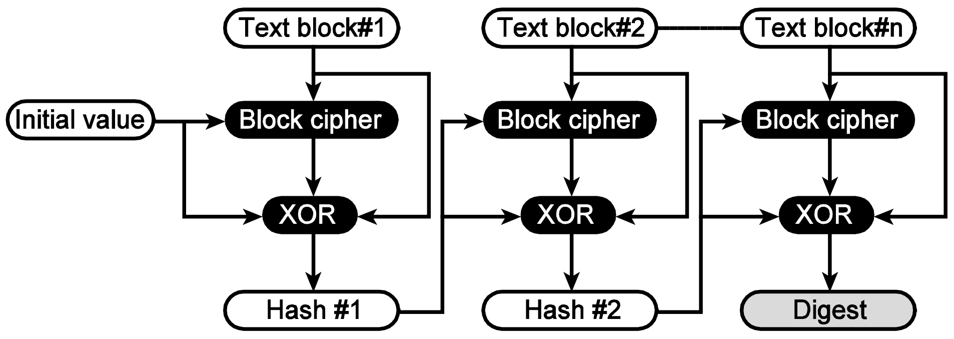
Table 2-1 gives an overview of several popular ciphers. Note that if the target platform has hardware acceleration available, the resulting code size will be much smaller and the system will be much faster.

**Table 2-1. Symmetric Encryption Algorithms**

| Algorithm | Key Length | Block Length | Security | Speed & Size |
|---|---|---|---|---|
| AES | 128 to 256 bits | 128 bits | Secure | Fast, small code, small RAM footprint |
| Blowfish | 32 to 448 bits | 64 bits | Secure | Fast, large RAM footprint |
| DES | 56 bits | 64 bits | Broken | Slow |
| Triple-DES | 168 bits | 64 bits | Secure | Very Slow |
| RC6 | 128 to 256 bits | 128 bits | Secure | Small code ,large RAM footprint |
| Serpent | 128 to 256 bits | 128 bits | Secure | Slow, small code, small RAM footprint |
| Twofish | 128 to 256 bits | 128 bits | Secure | Small RAM function |

Note that block ciphers can also be used (with modifications) as hash functions and message authentication codes. This can be useful to save code size when several primitives are needed (by reusing the same algorithm more than once).

**Figure 2-16. Block Cipher as Hash Function**



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

- **Electronic codebook** (ECB): The basic mode of operation, each block of plain text is encrypted using the key and the chosen algorithm, resulting in a block of cipher text. However, this mode is very insecure, as it does not hide patterns.
- **Cipher block chaining** (CBC, CFB, OFB, CTR ...): Encryption is not only done with the current block of plain text, but also with the last encrypted block. Make everything interdependent.

- **Authenticated encryptions** (EAX, CCM, OCB ...): are used to provide privacy, integrity and authentication at once. They are basically the combination of a MAC algorithm and a symmetric block cipher. They are useful when the three components are needed, as using a mode such as EAX will be faster and has less overhead than applying a MAC and a symmetric cipher separately.

The first block is encrypted using a random **Initialization Vector** (IV). While this vector can be transmitted in clear text, the same vector shall never be used more than once with the same key. It is likely that a manufacturer will produce more than one firmware upgrade for a product in its lifetime. This means that the IV cannot be stored in the chip in the same way the key is. Therefore, it will have to be transmitted by the host.

**Figure 2-17. CBC Mode of Operation**

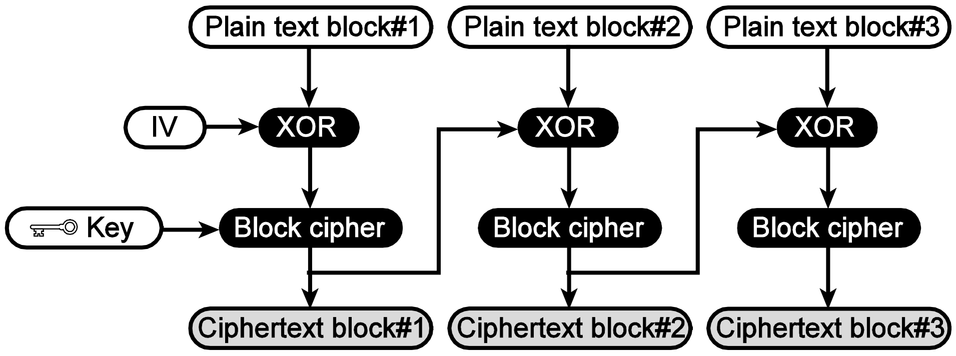
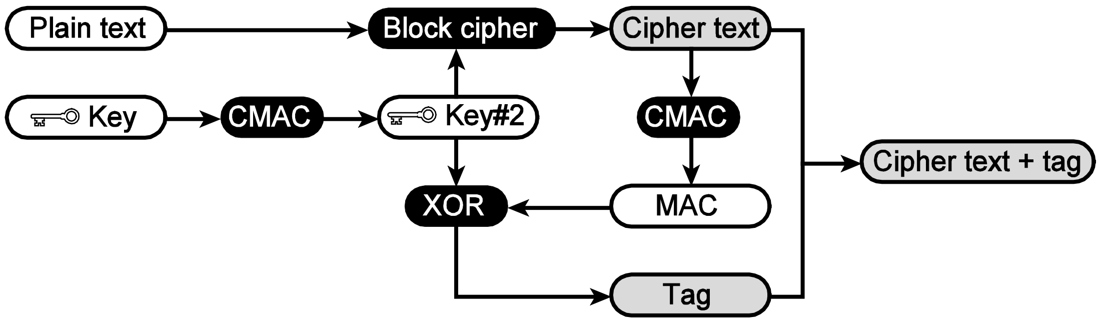


**Figure 2-18. EAX Mode of Operation**



### 2.6.2.2 Hash Functions

A hash function has three defining characteristics:

- **Output length**: needs to be large enough to make it almost impossible to find collisions.
- **Security**: 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.
- **Size & speed**: the stronger algorithms are often the slowest ones (which are not true for block ciphers), so there will be a security/speed trade-off

**Table 2-2. Hash Algorithms**

| Algorithm | Block Length | Security | Speed & Size |
|---|---|---|---|
| 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 |

### 2.6.2.3 Message Authentication Codes

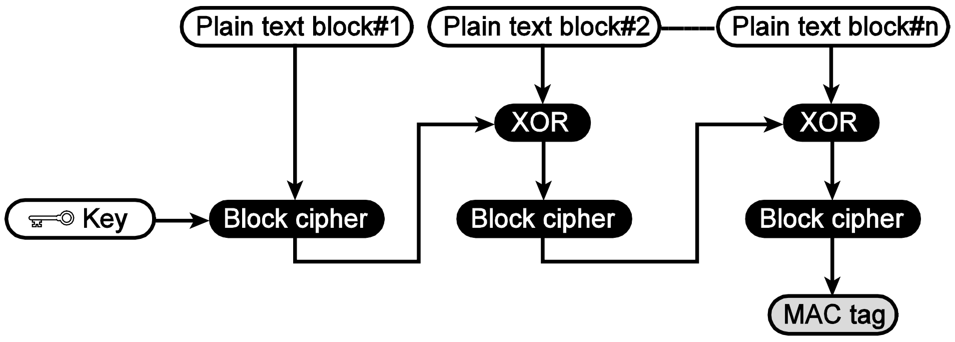
MACs are constructed by using other cryptographic primitives. Therefore, the choice of which type of MAC to use is mostly dictated by which algorithms are used by other functionalities. Of course, some MAC algorithms have been found faulty; care should be taken to avoid them.

Here are the different types of (secure) MACs available:

- HMAC: a hash function along with a private key
- UMAC: many hash functions and a block cipher
- OMAC/CMAC: block cipher in CBC mode
- PMAC: block cipher in CBC mode

Note that UMAC might not be usable in practice, as it requires many different hash algorithms. The incurred size overhead would thus be far too important for a bootloader

**Figure 2-19. CMAC Message Authentication Code**



### 2.6.2.4 Digital Signature Algorithms

There are basically two main systems for generating and verifying digital signatures:

- The Digital Signature Standard (DSS): specifically designed for digital signatures. It is based on a public-key algorithm known as the ElGamal scheme. The key length required to have a strong enough security is at least 1024 bits.
- A system based on the Rivest-Shamir-Adleman (RSA) public-key algorithm: the most popular method, used with a padding scheme (to append data to the message to encrypt). There are three commonly used schemes:
  - Full-domain hashing: involves using a hash function that has an output size equal to the RSA key length.

  - Optimal Asymmetric Encryption Padding (OEAP): adding a quantity of random data to convert RSA into a probabilistic encryption scheme.
  - Probabilistic Signature Scheme (PSS): adding a quantity of random data to convert RSA into a probabilistic encryption scheme.

#### 2.6.2.5 Pseudorandom Number Generators (PRNG)

There are many things in a secure system which are "random" or need some kind of randomized value. Secret keys are the most basic example. Thus there must be a method to generate those random values in a secure way, to avoid weakening the whole system. A chain is as strong as its weakest link, so even indirect security issues should not be overlooked.

However, note that a PRNG is not needed by the target. It is only used on the manufacturer side, for the following operations:

- Generating private keys
- Generating initialization vectors
- Padding data when using RSA/OEAP or RSA/PSS

This means that in practice, there is neither real speed nor size constraint on the PRNG algorithm. Only its security matters.

PRNGs work by using a starting seed to generate successive random values. Initializing that seed is a core problem, which is referred to as gathering entropy. Consider the case where the current date & time are used to seed the PRNG. An attacker could obtain that information and thus reconstruct every random number generated using that seed: private keys, nonce, etc.

Operating systems usually provide a mechanism to provide entropy, e.g., /dev/random on UNIX systems. They use, for example, the response time of devices such as hard disks to gather the required entropy.

Most PRNGs then rely on another cryptographic primitive (such as a block cipher or a hash function) to generate pseudo-random outputs. Here is a list of several secure PRNGs:

- Any block cipher in CTR mode
- Yarrow
- Fortuna
- Blum-Blum-Shub random number generator

#### 2.6.2.6 Available Libraries

This section lists web sites providing libraries and/or reference implementations of several cryptographic primitives described in this document. Those are all open-source or freely available implementations.

- libTomCrypt (http://libtom.org)
- OpenSSL crypto (http://www.openssl.org)
- Crypto++ (http://www.cryptopp.com)
- Brian Gladman (http://www.gladman.me.uk)
- Libmcrypt (http://mcrypt.hellug.gr/#_libmcrypt)

#### 2.6.2.7 Performances on an SAM3X8 Chip

Several aforementioned ciphers and modes have been tested using different implementations on a SAM3X8. This section presents the results obtained when compiled by EWARM 5.50.

The AES cipher has been tested using different implementations. The first one uses the libTomCrypt library, freely available from http://libtom.org. The second one is based on a standard implementation provided by Paulo Baretto and Vincent Rijmen.

**Table 2-3. Performance of AES (128-bit key and 128-bit blocks)**

| Encryption Mode | Implementation Source | Size Overhead (bytes) | Decryption Time for a 128KB File (ms) |
|---|---|---|---|
| ECB | libTomCrypt | 8576 | 680.0 |
| | Reference | implement | 2632 |
| CBC | libTomCrypt | 8660 | 734.7 |
| | Reference | implement | 2708 |
| CTR | libTomCrypt | 8820 | 752.7 |
| | Reference | implement | 2096 |

**Table 2-4. Performance of Triple-DES (168-bit key and 64-bit blocks)**

| Encryption Mode | Implementation Source | Size Overhead (bytes) | Decryption Time for a 128KB File (ms) |
|---|---|---|---|
| ECB | libTomCrypt | 4742 | 2006.9 |
| CBC | libTomCrypt | 4862 | 2065.3 |
| CTR | libTomCrypt | 5022 | 2085.3 |

### 2.6.3 Error Detection Codes

Since SAM microcontrollers are based on a 32-bit architecture, it seems logical to implement codes which are at least as long (as 32-bit). They will not cost more in terms of speed and size than an 8-bit or 16-bit variant.

Originally, simple checksums were used to detect errors. They operate by simply adding all the bytes in a piece of data to get a final value. However, they are very limited: they cannot, for example, detect that null bytes (0x00) have been appended or deleted. More reliable techniques are now available, so very simple checksums should be avoided.

There are two algorithms which are worth mentioning here. The first one is the well-known Cyclic Redundancy Check (CRC), which has strong mathematical properties and is quite fast. The 32-bit version, called CRC-32, is used in the IEEE® 802.3 specification.

Adler-32 is an algorithm that is slightly less reliable but significantly faster than CRC-32. It has a weakness for very short messages (< 100 bytes), but this is not a concern if a whole page of data (≥256 bytes) is transmitted at one time.

### 2.6.4 Firmware File Format

Compilers support a wide variety of output file formats. The most basic of them is the binary format (.bin): it is simply a binary image of the firmware. Since no information apart from the application code is required, the firmware can simply be transmitted in binary format and directly written to memory by the bootloader. Since using other formats would mean adding the necessary code on the bootloader side to handle them, this may not be worth it.

### 2.6.5 Target Chips

Some of the chips in the SAM family have different IPs, such as the Flash controller. This means that they are programmed differently; therefore, several versions of the code must be written to accommodate all the microcontrollers.

Thus, the bootloader will be developed for a particular chip first, but in a modular way. This means that functions which are chip-dependent are wrapped in an abstraction layer. Porting the software to another chip is easy: only the necessary low-level functions have to be coded, without touching the bootloader core.

The following implementation example will use SAM3X8 as candidate, since it has no cryptographic accelerators, a software solution with libTomCrypt or Reference design is used to encrypt.

# 3. Bootloader Implementation

This section details several issues that one may encounter while implementing a safe & secure bootloader, along with some insight on how to approach them. Example code from a working implementation is given in the following sections to illustrate the solutions.

## 3.1 Bootloader Flow

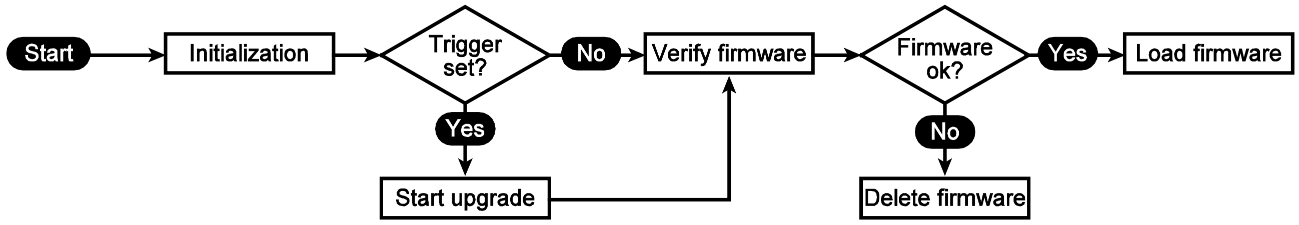
### 3.1.1 Boot Sequence

The startup sequence of the bootloader is as follows:

- Initialization
- Trigger condition check
  - Firmware upgrade (if trigger condition is offered)
  - Firmware selection (if trigger condition is offered, for dual banked device)
- Firmware verification (optional)
- Firmware loading (if verification is ok or disabled)

The following is a simple bootloader example that supports only Firmware upgrade:

**Figure 3-1. Basic Boot Sequence Diagram Example**



Here is the corresponding code in C:

```
// Bootloader initialization
trigger_init();
memory_init();
media_init();
communication_init();
encryption_init();
// Check trigger condition
if (trigger_poll()) {
      // Upgrade firmware
      bootloader_load(APP_START_ADDRESS);
}
// Verify firmware
if (integrity_check() != OK) {
//
}
// Execute the firmware
binary_exec(APP_START_ADDRESS);
```

The function *binary_exec()* uses the application start address to start the firmware.

### 3.1.2 Upgrade Sequence

The basic upgrade flow starts with the host sending the firmware to the target, which then programs it in memory. Once the programming is done, the new application is loaded.

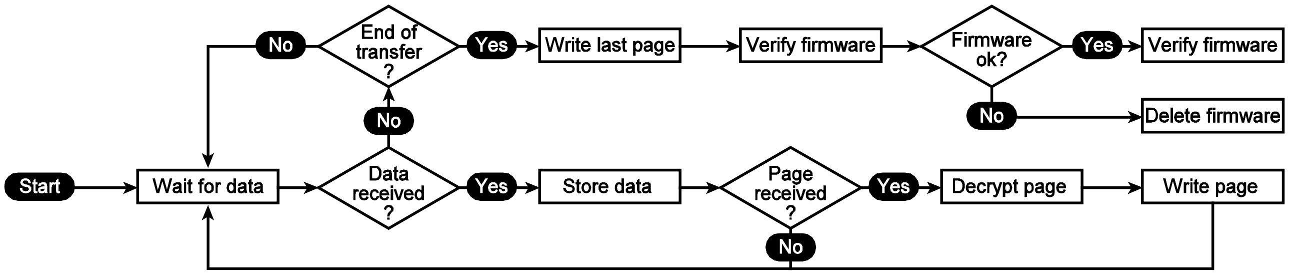
While, in theory, the "downloading" and "programming" steps are different, this is not the case in practice. Indeed, SAM microcontrollers usually have much more Flash memory than RAM. Thus, they cannot store the whole firmware in RAM before writing it permanently to the Flash.

This means that the code must be written to the memory while it is received, but not after. Since a Flash write operation takes approximately 6 ms (with a page erase), a communication protocol is needed to halt the transfer when the memory is being written and resume it afterwards.

Note also that the Flash memory is split up into fixed-size blocks called pages. A memory write operation can upgrade one page (or less) at a time; thus it seems logical to send packets containing one full page.

Finally, there are several optional post-processing features to take into account. If code encryption is activated, then each page must be decrypted before being written. If a digital signature or a message authentication code is present, it must be verified once the download is completed.

**Figure 3-2. Basic Firmware Upgrading Process Example**



Here are some sample C codes implementing the upgrade flow:

```
// Receive and write firmware
pCurrent = APPLICATION_STARTING_ADDRESS;
do {
        bytes = communication_receive(page);
        // If at least one byte is received, write the page
        if (bytes > 0) {
                // Decrypt firmware
                encryption_decrypt(page, page, bytes);
                // Pad page data
                while (bytes < MEMORY_PAGE_SIZE) {
                        page[bytes] = 0xFF;
                        bytes++;
                }
                // Write page
                memory_write(pCurrent, page);
                pCurrent += MEMORY_PAGE_SIZE;
        }
} while (bytes > 0);
```

In this example, the *communication_receive()* function waits for a whole page of data while detecting the transfer end. A returned value "0" means that the transfer is finished.

### 3.1.3 Memory Partitioning

As described in Section 2.4.2, using memory partitioning makes it possible to have at least one working version of the firmware in the device at anytime. This is useful to avoid firmware corruption if a problem occurs during an upgrade, such as a power loss or a connection loss.

The following sections will describe the implements for single and dual banked memory.

#### 3.1.3.1 Partitioning on Single Banked Memory

As described in Section 2.4.2.2 and Figure 2-10, the whole Flash memory is divided into two distinct regions, A and B (excluding the bootloader region). The first one, A, is located right above the bootloader and contains the code which is to be loaded by the bootloader. B acts as a buffer during an upgrade: the code is downloaded to that region, verified, and finally copied to the first region if valid.

The boot and upgrade sequences are thus slightly modified. During the upgrade, the code is written to region B (the buffer zone). Several steps are then added to the boot sequence, regardless of whether or not a firmware upgrade has been requested. The bootloader first checks if the two codes present in regions A and B are identical (via code compare, hash code compare or other ways). If they are, then the code in region A is loaded.

If two codes are not the same, this means that either an upgrade has just taken place, or there has been an error during a previous upgrade. The validity of the code in region B is verified. If it is indeed valid, then it is copied over region A. If not, it is deleted.

Memory partitioning also makes it possible to use a slightly different bootloading strategy. Since there is always a working firmware embedded in the device, some functionalities of the bootloader can be moved to the user application. By doing this they can be upgraded as well.

For example, the firmware transfer can be delegated to the user firmware. It simply downloads the new code in the buffer region and resets the chip. The bootloader then performs the remaining operations, i.e., verify region B and copy it over region A.

Figure 3-3. Boot Sequence with Single Banked Memory Partitioning

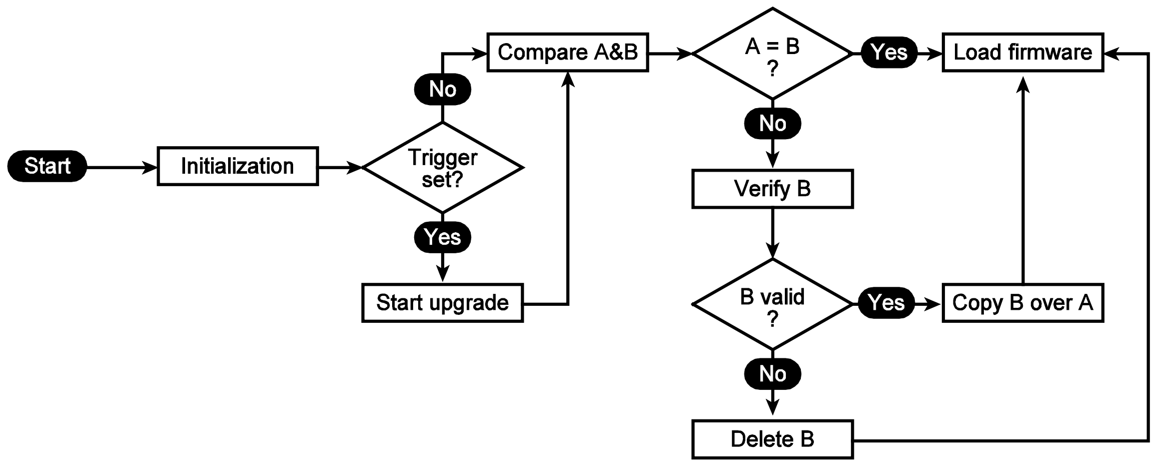
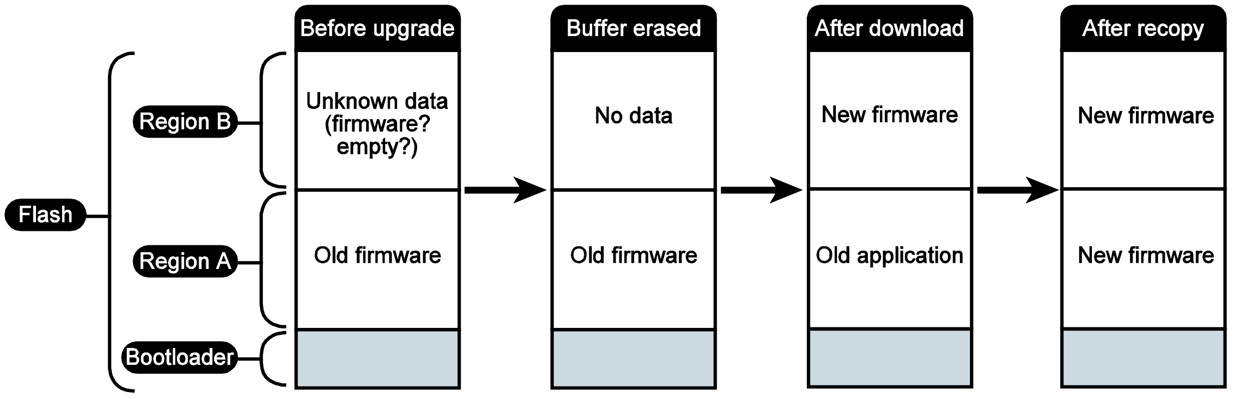


Figure 3-4. Single Banked Memory Content during Upgrade Process



### 3.1.3.2 Partitioning on Dual Banked Memory

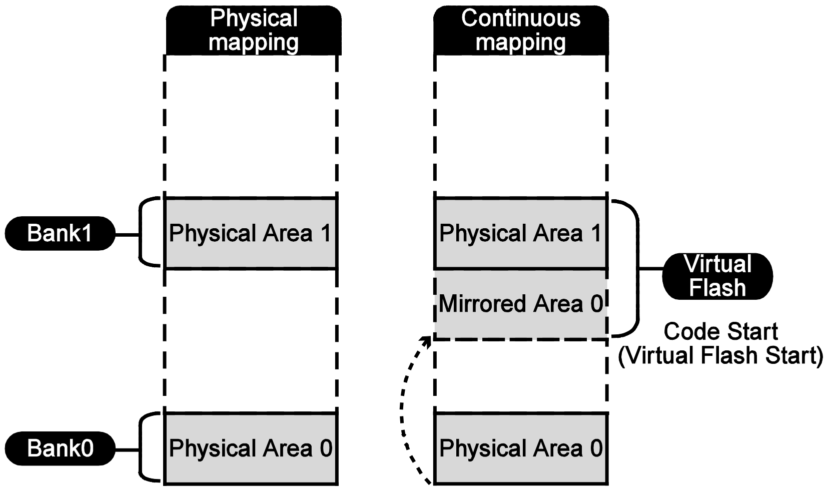
As there are two memory banks, more choices are available on memory partitioning:

- Choice #1, manage the whole memory as a single space, and accesses to different memory banks are translated to continuous memory address. In this way, the whole memory is looked as a single banked memory to partition. For more details, please refer to Section 3.1.3.1 and Section 3.2.1.
- Choice #2, as described in Section 2.4.2.3, manages the Flash by banks, and each bank has its bootloader and firmware region. Here the two banks have different physical addresses, but they are mapped to the same boot program address on boot time, so binary should use a link file that links to boot program address space. When the boot memory layout is changed, different firmware version is chosen.

The first choice can apply if the two memory banks should be in a continuous address (such as SAM3X, SAM3SD8, SAM4SD16 and SAM4SD32, the Flash address of two banks are continuous), or the banks are able to be mapped to a continuous address (such as SAM3U, the Flash address of two banks are not continuous, but the bank in Flash area is mapped to reserved space to access, see Figure 3-5).

The link file should use this continuous address to generate the application firmware. For more information about Flash banks, please refer to datasheet for SAM3/4 chips.

**Figure 3-5. Map Two Memory Banks to a Continuous Address**



Then the second choice: since Flash memory has two banks, the two firmware regions A & B are placed at the same offset of these banks; they are remarked as boot region and buffer region. Both of them are located right above the bootloader, on its location bank and contain the firmware code. The boot region A is loaded by bootloader on that bank when system starts up. The buffer region, B, is a buffer on upgrading: the code is downloaded to that region and verified. Finally, the roles of these regions can switch, by selecting different boot bank. That is, when booting from bank 0, the firmware region on it acts as boot region (A) and the firmware region on bank 1 acts as buffer region (B); when booting from bank 1, the firmware region on bank 0 acts as buffer region (B) and the firmware region on bank 1 acts as boot region (A).

In this case, when performing upgrade, the firmware is always loaded to buffer region (B) and after downloaded data in buffer region (B) is verified “OK”, the boot bank is switched and new firmware will be loaded when system restarts. As a result, the old firmware is still kept in buffer region but not used, so the two regions both contain workable firmware and the latest downloaded one is to be loaded.

**Figure 3-6. Boot Sequence with Dual Banked Memory**

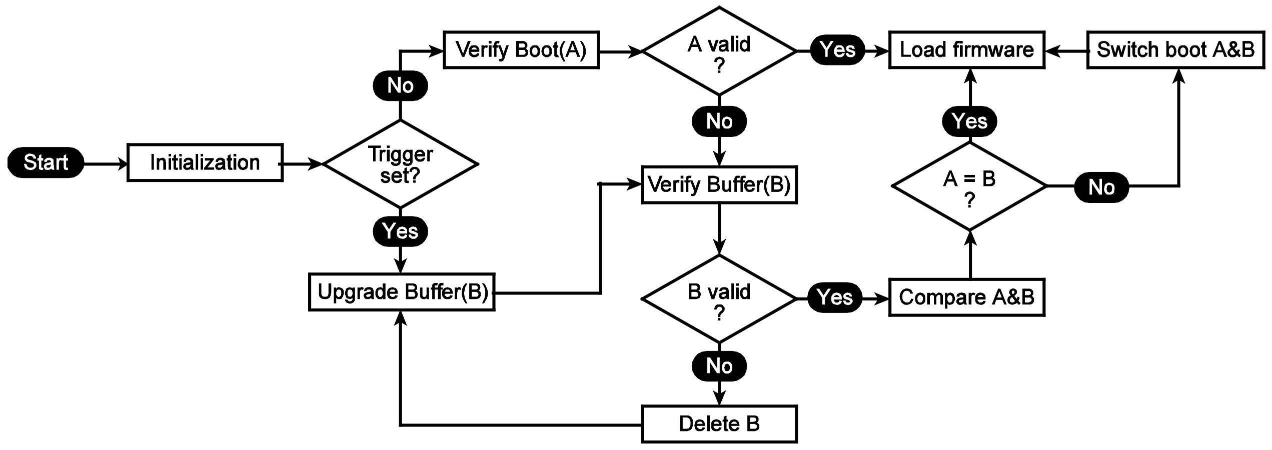
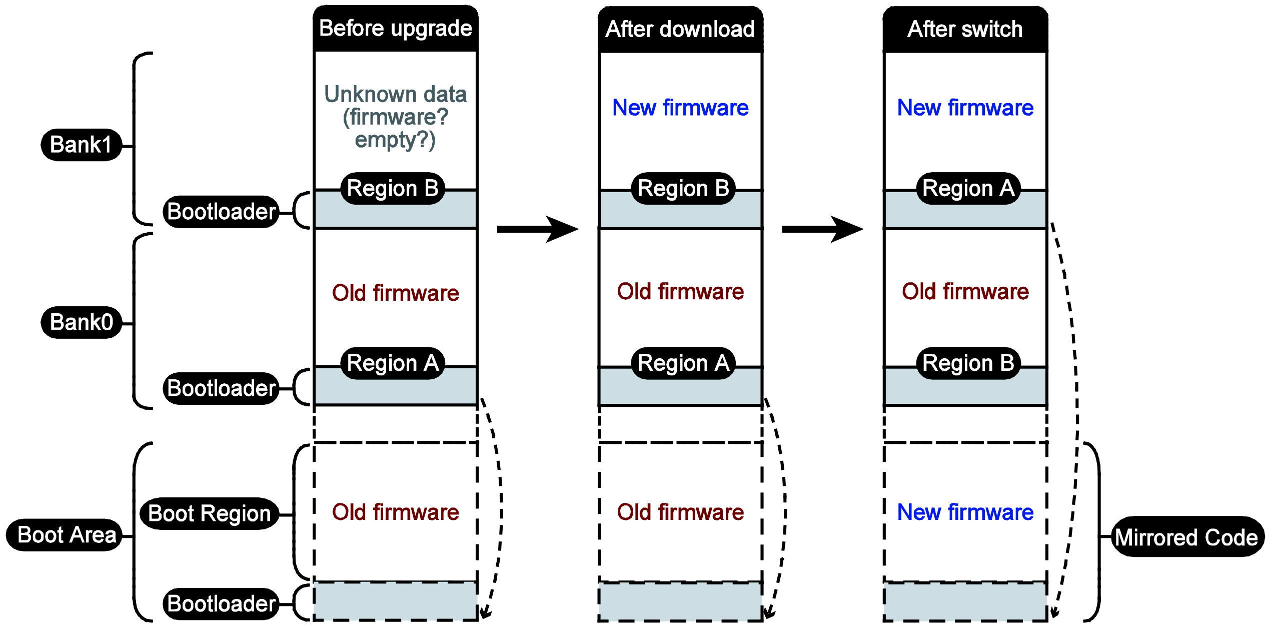
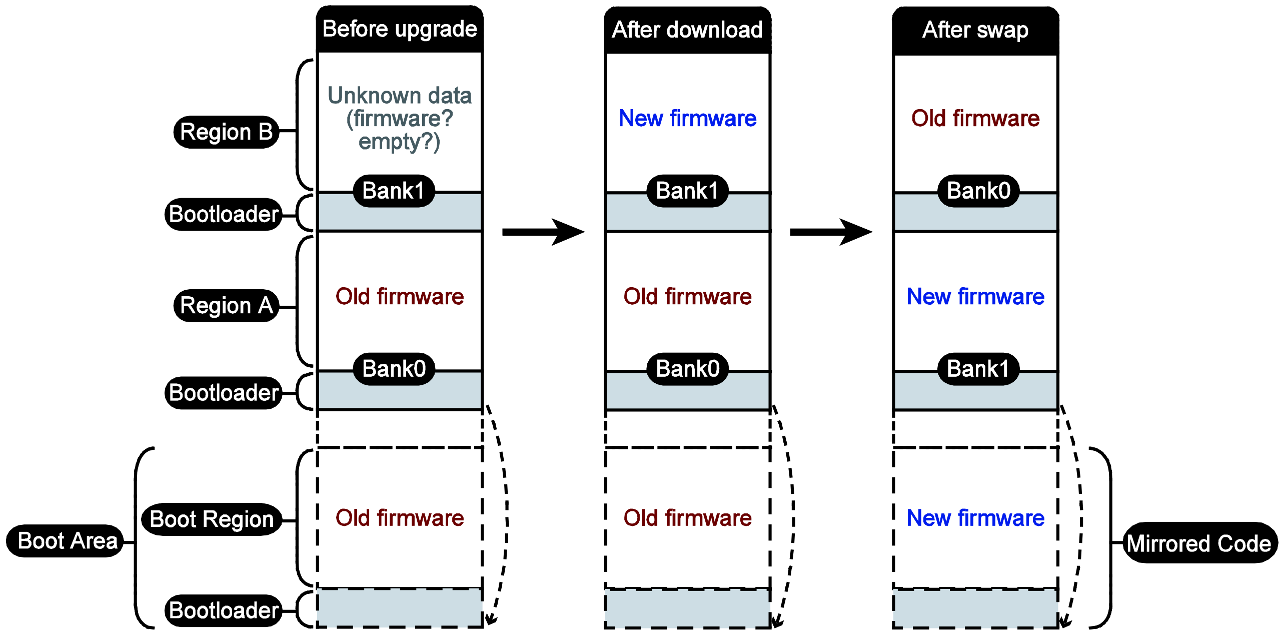


**Figure 3-7. Dual Banked Memory Content during Upgrade Process**



Note that in SAM3SD8, SAM4SD16 and SAM4SD32, there is a slight difference when changing boot banks. It swaps the physical banks! E.g., before swap, the old firmware is in address area 0 and the new firmware is in address area 1, while after swap, the new firmware will be in address area 0 and the old firmware in address area 1. This feature keeps the boot firmware in the same physical address, whether Flash banks have been swapped or not.

**Figure 3-8. Dual Banked Memory Content during Upgrade Process for SAM3SD8, SAM4SD16, SAM4SD32**



## 3.2 Bootloader Programming on SAM3/4 Chips

In practice, for SAM3 and SAM4 chips, there are two types of flash controller:

- The Enhanced Embedded Flash Controller (EEFC): Used by most of SAM3 and SAM4 chips
- The FLASHCALW: Used by SAM4L

Here we take EEFC as example to show how internal flash of SAM3/4 chips are programmed. For operations that using FLASHCALW, please refer to SAM4L documents (can be found here).

### 3.2.1 Flash Programming

Usually the Flash of the SAM3 chip is organized in one or two planes (banks) that are made of pages of 256 bytes. SAM4 chip has the same organization while the page size is 512 bytes. The EEFC is used to manage the Flash access. The Fast Flash Programming Interface (FFPI) is used to program the device. Depending on the device, there might be one or two Flash Controllers, and the physical address of two Flashes (banks) can be continuous or not. Table 3-1 gathers the rough information. For more detailed information, please refer to chip's data sheet.

Note:

- The Flash memory of SAM microcontrollers cannot be read and written at the same time for one single bank case.
- For SAM3 the programming command can perform erase and program automatically.
- For SAM4, before writing the Flash memory, an erase operation must be performed,
- If flash erase block size are bigger than write block size (this is a usual situation for flash devices, e.g., in SAM4 writing bases on one page but minimum erasing size is eight pages), the partition must be aligned with erase block size so that the unexpected area will not be erased.

**Table 3-1. Flash Organizations for SAM3/4 Chips**

| Device | Flash Organization | Number of Flash Controllers | Flash (bank) Address is continuous | Flash Controller |
|---|---|---|---|---|
| SAM3U2/1 | Single Bank | 1 | - | EEFC |
| SAM3U4 | Dual Bank | 2 | No | |
| SAM3S4/2/1 | Single Bank | 1 | - | |
| SAM3N Series | Single Bank | 1 | - | |
| SAM3X(A) Series | Dual Bank | 2 | Yes | |
| SAM3S8 | Single Bank | 1 | - | |
| SAM3SD8 | Dual Bank | 1 | Yes | |
| SAM4S16/8 | Single Bank | 1 | - | |
| SAM4SA16 | Single Bank | 1 | - | |
| SAM4SD32/16 | Dual Bank | 2 | Yes | |
| SAM4E Series | Single Bank | 1 | - | |
| SAM4L4/2 | Single Bank | 1 | - | FLASHCALW |

#### 3.2.1.1 Single Banked Flash Programming

Single banked Flash cannot be read and written at the same time since it is single plane. Therefore, a program executing from the Flash cannot perform a memory write. Since the bootloader is located inside the Flash, it must be copied to the RAM prior to execution.

A way is available for certain toolchains such as IAR Embedded Workbench®. Using the *__ramfunc* attribute for a function will tell the compiler that it is supposed to run from the RAM, not the Flash. The C-initialization routine copies the function at runtime. Note that you must disable interrupts during a Flash write if this solution is used, since exception vectors will still be read from the Flash by the ARM core.

Example function performs a Flash command using the *__ramfunc* attribute:

```
__ramfunc void flash_cmd(Efc* pEfc, uint32_t dwCmd, uint32_t dwArg) {
      // Start Flash operation and wait for completion
      // ITs are masked during this.
      while (!(efc->EEFC_FSR & EEFC_FSR_FRDY));
      __disable__irq();
      efc->EEFC_FCR = EEFC_FCR_FKEY(0x5A)
            | EEFC_FCR_FARG(dwArg)
            | EEFC_FCR_FCMD(dwCmd)
      while (!(efc->EEFC_FSR & EEFC_FSR_FRDY));
      __enable__irq();
}
```

#### 3.2.1.2 Dual Banked Flash Programming

Dual banked Flash enables Flash writing to another bank while code is running on one bank since it is in different Flash bank. So there are several different ways to write.

- Put function codes in bank 0 to perform writing in bank 1 and vice versa. When writing to a Flash bank, judge which function to use by the writing address.
- Use RAM functions, the same as single banked Flash programming in Section 3.2.1.1.

### 3.2.1.3 Flash Programming via IAP Function

SAM3/4S/4E chips offer IAP function in ROM code to send the Flash command to EEFC and wait for the Flash to be ready. Since the code is executed from ROM, it allows Flash programming to be done while the code is still running in Flash, to any of the banks.

Example function performs a Flash command using the IAP function:

```
#define CHIP_FLASH_IAP_ADDRESS (IROM_ADDR + 8)
void flash_cmd(uint8_t ucFlashID, uint32_t dwCmd, uint32_t dwArg) {
      // Obtain Pointer on IAP function in ROM
      static uint32_t (*IAP_PerformCommand)( uint32_t, uint32_t ) ;
      IAP_PerformCommand = (uint32_t (*)( uint32_t, uint32_t ))
      *((uint32_t*)CHIP_FLASH_IAP_ADDRESS ) ;
      // Start Flash operation and wait for completion
      // ITs are masked during this.
      __disable__irq();
      IAP_PerformCommand( ucFlashID, EEFC_FCR_FKEY(0x5A)
            | EEFC_FCR_FARG(dwArg)
            | EEFC_FCR_FCMD(dwCmd) ) ;
      __enable__irq();
}
```

### 3.2.2 Vector Table Relocate & Application Execute

Atmel SAM3/4 chips are based on an ARM® Cortex® core. When the core boots, it always starts from address 0x0, where a vector table exists. The first word of vector table is used to initialize stack (SP) and the second word is used as a start entry (PC). The other words in this vector table offer exception vectors to the ARM® Cortex® core.

The vector table location can be changed by modifying the VTOR register to a different memory location, in the range from 0x00000080 to 0x3FFFFF80.

Before executing the firmware, the vector table must be set to the address of firmware vector table.

Then SP is loaded from the first word of vector table. After that, PC is loaded from the second word of vector table, to jump to the application.

Example code for vector table relocating and application executing:

```
// -- Disable interrupts
// Disable IRQ
__disable_irq();
// Disable IRQs
for (i = 0; i < 8; i ++) NVIC->ICER[i] = 0xFFFFFFFF;
// Clear pending IRQs
for (i = 0; i < 8; i ++) NVIC->ICPR[i] = 0xFFFFFFFF;
// -- Modify vector table location
// Barriars
__DSB();
__ISB();
// Change the vector table
SCB->VTOR = ((uint32_t)vStart & SCB_VTOR_TBLOFF_Msk);
// Barriars
__DSB();
__ISB();
// -- Enable interrupts
__enable_irq();
// -- Execute application
__asm ("mov r1, r0 \n"
      "ldr r0, [r1, #4] \n"
      "ldr sp, [r1] \n"
      "blx r0"
```

```
    );
```

### 3.2.3 Boot Code Remapping

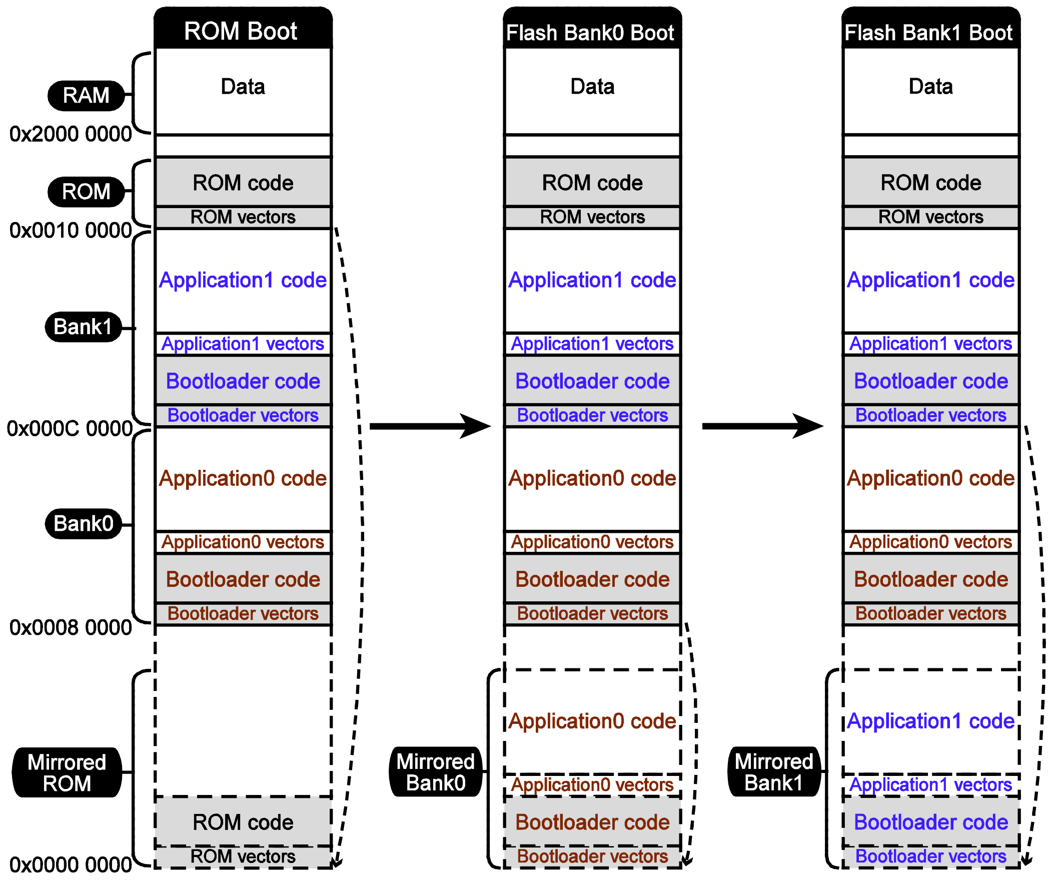
SAM3/SAM4S/SAM4E chips can boot from ROM or Flash. If there are two banks, each of the banks can be the boot bank. The boot selection is changed by modifying boot code mapping.

E.g., for SAM3X8, there are two 256-Kbyte Flash banks; the boot mapping is changed by configuring GPNVM bits 1 & 2. Table 3-2 and Figure 3-9 are table and diagram for reference:

**Table 3-2. Boot Code Mapping for Dual Banked Device**

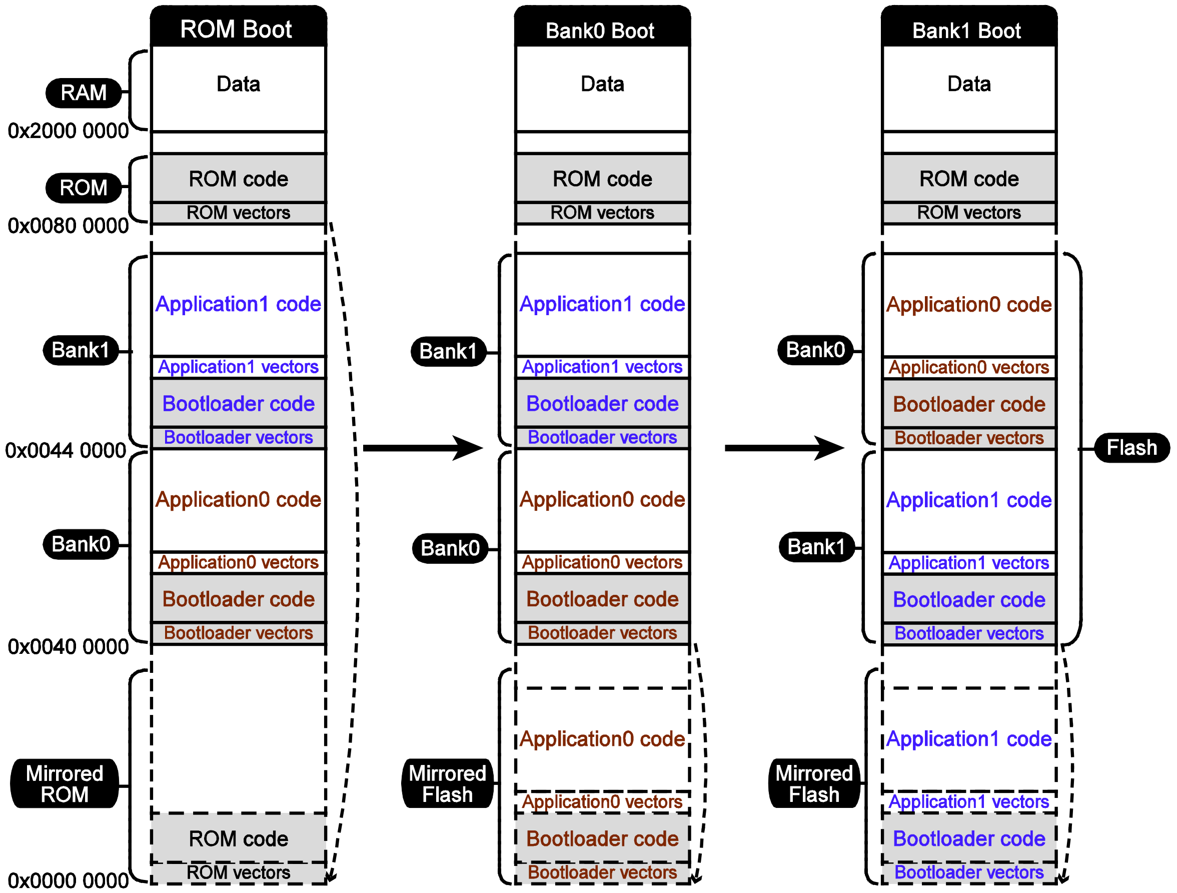
| GPNVM[1] | GPNVM[2] | Boot From |
|---|---|---|
| Clear | Clear/Set | ROM |
| Set | Clear | Flash Bank0 |
| Set | Set | Flash Bank1 |

**Figure 3-9. Memory Organization Before and After Boot Code Selection**



Note that for SAM3SD8 and SAM4SD32/16, setting GPNVM bit 2 will swap the two Flash banks, so the memory mapping change is as follows:

**Figure 3-10. Memory Organization Before and After Boot Code Selection SAM3SD8, SAM4SD32/16**



## 3.2.4 Application Code Linking

The next sections focus on application code linking in different cases.

The example uses a linker script that is based on address 0x0, which satisfies both single banked and dual banked situation on SAM3X chips.

### 3.2.4.1 Single Banked Code Linking

Ordinarily, the code segment of a user application is linked at the beginning of the Flash. This is because the ARM core of SAM microcontrollers starts fetching program code at address 0x0, where the beginning of Flash is mapped. However, this cannot be the case when using a bootloader, since it will be located at the beginning of the Flash.

The solution is to modify the address at which the application is linked, depending on the reserved bootloader size. To do that, you have to modify the linker script of your project. Simply define a “bootloader_size” constant which is equal to the size of the compiled bootloader, and the first available ROM address to “flash_start+bootloader_size”. Here is an example for SAM3X8 chip using IAR Embedded Workbench 5.50:

```
/*--- Bootloader size (32K) ---*/
define symbol __BOOTLOADER_size__ = 0x8000;
...
/*--- Code region 3X8 ---*/
define symbol __region_ROM_start__ = 0x00080000+__BOOTLOADER_size__;
define symbol __region_ROM_end__ = 0x00040000-1;
```

```
...
```

Code segments are then defined between *__region_ROM_start__* and *__region_ROM_end__*.

Note that when single banked Flash is selected to boot, the whole Flash is mapped to address 0x0, so it's also possible to arrange bootloader and application based on this mirrored address. It also works for a dual banked Flash with continuous bank addresses. But when a dual banked Flash without continuous bank addresses is used (as choice #1 described in Section 3.1.3.2), the linking map must be based on the virtual continuous Flash start address (see Figure 3-5) but not on this boot mirror, since the boot mirror might only include the single bank that is used for boot.

#### 3.2.4.2 Dual Banked Code Linking

In the choice #2 case in Section 3.1.3.2, the code segment should be placed on different physical bank, but arranged in the same mapping. Since the code will never exceed one bank size, and all banks can be mapped to address 0x0, which is used as a base to place bootloader and application, so the application linker script can be like the following:

```
/*--- Bootloader size (32K) ---*/
define symbol __BOOTLOADER_size__ = 0x8000;
...
/*--- Code region 3X8 ---*/
define symbol __region_ROM_start__ = 0x00000000+__BOOTLOADER_size__;
define symbol __region_ROM_end__ = 0x00040000-1;
...
```

The code segments start address is modified to place code based on 0x0 but not on physical Flash address.

Note that for some dual-bank chips such as SAM3SD8, SAM4SD32/16, thanks to its Flash banks swapping feature (Figure 3-8 and Figure 3-10, two Flash banks are swapped, but Flash start address is always mapped to address 0x0, no matter whether it's banks are swapped or not), it's also possible to put code segments at Flash start address. In this way, the code can be debugged directly in IAR Embedded Workbench.

### 3.2.5 Boot Region Locking

To avoid having the bootloader region accidentally erased by the user application, it is safe to lock it with the Flash controller. Any write operation to a locked Flash region will be aborted automatically.

A command is available in the Enhanced Embedded Flash Controller (EEFC) for locking a specified memory region. The problem is that the region is quite large: between 4K and 16K bytes. Since only the bootloader region must be locked, this means that the application will have to start at the beginning of the next region.

In practice, the users have to set the bootloader segment size to a multiple of a region size. This effectively means that you waste the region space which is not used by the bootloader.

Note that for dual bank, the bootloader region for both banks should be locked.

The security bit, used to protect the internal memory (Flash + SRAM) from external access, can also be set in the EEFC.

# 4. Example Implementation

This section describes a sample implementation of the safe & secure bootloader presented in this document. It is made up of three programs: the bootloader itself and two program tools. One program tool is used to transfer the firmware between a PC and the bootloader. The other one is necessary to encrypt the firmware before sending it. The three pieces of software are detailed below.

## 4.1 Bootloader

It's general implementation for device with single banked Flash or dual banked Flash.

### 4.1.1 Features

The following features have been implemented:

- Basic bootloading capabilities using a USART to transmit the firmware
- XON/XOFF protocol for flow control during downloading/Flash programming
- Boot region locking
- Code encryption using AES or Triple DES
- Memory partitioning

The software has been developed for IAR Embedded Workbench 5.50.

### 4.1.2 Configurability

The bootloader has been designed in a way which makes it easy to add new components to it, like a new media or a new communication protocol.

The /inc/config.h file controls the configuration of both mandatory components (such as which media to use) and optional ones (security, safety). A particular component can be selected by defining the following constant:

```
#define USE_COMPONENTNAME
```

The COMPONENTNAME is the name of the components (e.g., USART0, UART, and BUTTONS for SAM3/4). Note that only one component can be selected for each mandatory category, which are:

- Communication protocol
  - XON/XOFF (*USE_XON_XOFF*)
- Media layer
  - USART (*USE_USART0*)
- Debug
  - DBGU (*USE_DBGU*)
  - UART (*USE_UART*)
- Memory type
  - Flash (*USE_Flash*)
- Trigger condition
  - Dummy (*USE_DUMMY*)
  - Switches (*USE_SWITCHES*)
  - Button (*USE_BUTTONS*)

- Timing measurement
  - Timer0 (*USE_TIMER0*)

Both the debug and timing categories are not truly mandatory. If no debug driver is defined, for example, then the bootloader does not output debug messages. The timing module can be used to perform benchmarks on several features.

The following optional parameters are available:

- Boot region locking (*USE_BOOT_REGION_LOCKING*)
- Code encryption (*USE_ENCRYPTION*)
- Memory partitioning (*USE_MEMORY_PARTITIONING*)

### 4.1.3 Code Location

#### 4.1.3.1 Main Bootloader Algorithm

The /src/main.c file contains the core bootloader algorithm. This includes the boot sequence (Section 3.1.1), the upgrade sequence (Section 3.1.2) as well as the modified boot & upgrade sequence for memory partitioning (Section 3.1.3). Memory locking of the bootloader region is also done during initialization of this bootloader.

#### 4.1.3.2 Vector Table Relocating & Application Execute

The code is located in /src/main.c, implemented in function *binary_exec()*.

#### 4.1.3.3 Flash Programming

Algorithms for programming the Flash as well as locking/unlocking memory regions are located in the /src/media/flash.c file.

#### 4.1.3.4 Linking Code

All the files used for both bootloader and user application linking and startup are located in the /resources directory. It includes:

- bootloader_sam3x8_flash_0.icf: startup and linker file for the bootloader without boot code remap (based on Flash beginning address)
- bootloader_sam3x8_flash_remap.icf: startup and linker file for the bootloader with boot code remap (based on address 0x0, the mapped address of Flash)
- firmware_sam3x8_flash_0.icf: startup and linker file for a firmware without remapped boot code (based on Flash beginning address)
- firmware_sam3x8_flash_remap.icf: startup and linker file for a firmware with remapped boot code (based on address 0x0, the mapped address of Flash)

## 4.2 Dual Banked Bootloader

As described in previous sections, dual bank gives the chance of modifying data on another bank while code is running on one bank, and also allows booting from different banks.

- It allows customers to implement the FW upgrade in application without bootloader: let application run in one bank and buffer FW upgrade data in another bank.
- It adds the advantage to let the same code work on both banks without modifying the binary mapping, since both banks can be mapped to boot program area.
  - It allows 2 firmware versions in 2 banks and selection of boot firmware version.

This simple example adds implementation of dual banked bootloader as described in Section 3.1.3.2. It's added as an alternative partitioning method. The following constant should be defined to enable it:

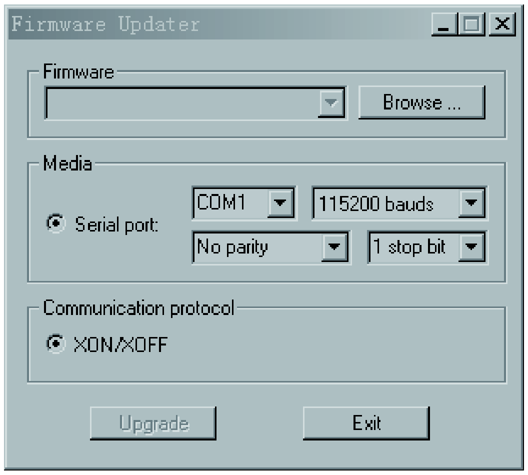
```
#define PARTITION_DUAL_BANK
```

When using this option, the two firmware region size (for region A & B, defined as *REGION_SIZE*) should also be decreased, because the bootloader code is duplicated.

## 4.3 Firmware Updater

This program is used to transmit the firmware from the host PC to the bootloader. Since an RS232 (through a PC COM port) link is implemented at the moment, a standard terminal application (like HyperTerminal) could be used to perform the same operation. However, it will be necessary to develop a program if another media or communication protocol is implemented and not supported by standard programs.

**Figure 4-1. Firmware Updater Main Window**



Serial port: COM1, 115200 bauds, No parity, 1 stop bit

Communication protocol: XON/XOFF

The main window of the application enables the user to perform the following operations:

- Choose the firmware file to send to the bootloader
- Select the media and associated parameters
- Select the communication protocol and associated parameters
- Launch an upgrade

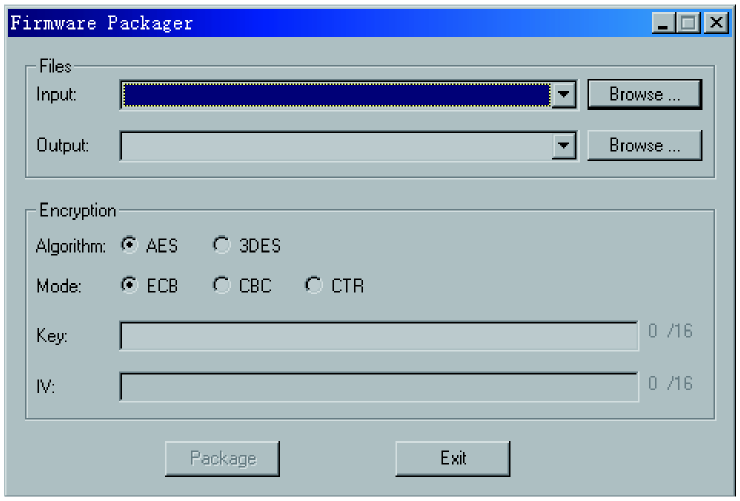
Note that you must select the same parameters in the Firmware Updater as the ones which have been selected for the bootloader. For example, if the bootloader is configured to connect using a USART configured at 115200 bps, no parity and 1 stop bit, select those parameters in the Firmware Updater.

## 4.4 Firmware Packager

The Firmware Packager is used to prepare the firmware prior to sending it to customers, including encrypting it, generating a signature or MAC tag, etc.

Note that when the bootloader enables code encryption (*USE_ENCRYPTION*), the Firmware Packager must be used, with the same encryption settings, to generate encrypted firmware for Firmware Updater to download.

**Figure 4-2. Firmware Packager Main Window**



Currently, the following functionalities are implemented:

- Select the firmware file to package
- Select the output file
- Select an encryption method (AES or Triple DES) with associated parameters
  - Encryption mode, key and Initialization Vector (IV)
- Launch the firmware packaging

Note that you must select the same parameters in the Firmware Packager as the ones selected for the bootloader. For example, if the bootloader is configured to accept a firmware encryption in AES-CBC with a particular key and IV, enter the same parameters in the Firmware packager.

## 5. Related Documents

[1] Atmel Corp., 2006, Safe and Secure Firmware Upgrade for AT91SAM Microcontrollers, literature no. 6253.

[2] Atmel Corp., 2006, Safe and Secure Bootloader Implementation, literature no. 6282.

## 6. Revision History

| Doc. Rev. | Date | Comments |
|---|---|---|
| 42141A | 06/2013 | Initial revision |

Atmel | Enabling Unlimited Possibilities®

**Atmel Corporation**
1600 Technology Drive
San Jose, CA 95110
USA
**Tel:** (+1)(408) 441-0311
**Fax:** (+1)(408) 487-2600
www.atmel.com

**Atmel Asia Limited**
Unit 01-5 & 16, 19F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
**Tel:** (+852) 2245-6100
**Fax:** (+852) 2722-1369

**Atmel Munich GmbH**
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
**Tel:** (+49) 89-31970-0
**Fax:** (+49) 89-3194621

**Atmel Japan G.K.**
16F Shin-Osaki Kangyo Building
1-6-4 Osaki
Shinagawa-ku, Tokyo 141-0032
JAPAN
**Tel:** (+81)(3) 6417-0300
**Fax:** (+81)(3) 6417-0370