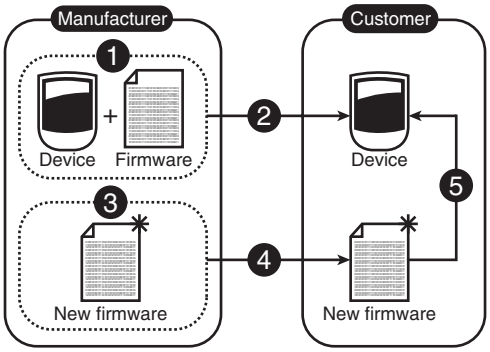
# Safe and Secure Firmware Upgrade for AT91SAM Microcontrollers

## 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 Principle




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. Customer patches his device with the new firmware

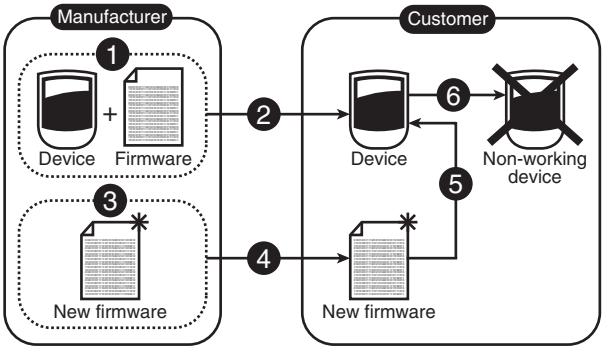
However, two major issues arise from this. First, the device must not be rendered useless because of an error during the update process. Common problems include power loss and connection loss during the transmission of the new firmware. We will refer to this concern as the **safety** of the device.

AT91 ARM Thumb Microcontrollers

Application Note

6253A–ATARM–07-Sep-06

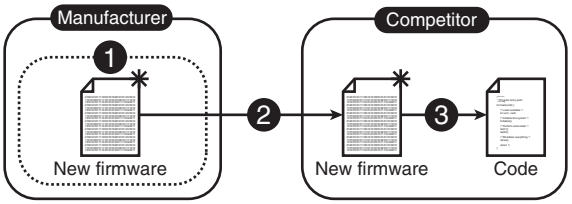
**Figure 1-2.** Safety Problems of In-field Upgrading




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. Customer patches his device with the new firmware
6. An error occurs during the transmission (e.g., the device batteries run out), rendering the device unable to operate

**Security** is the other problem. One of the main concerns is that firmware developers do not want their work to be disclosed and used by competitors; therefore, the new application must be protected from unauthorized access on its way to the target product.

**Figure 1-3.** Security Problems of In-field Upgrading




1. Manufacturer releases a new firmware for a device
2. Competitor obtains new firmware
3. Firmware is reverse-engineered to retrieve original code

"In-field Upgrading" on page 3 explains how a traditional update mechanism works; the different problems associated with it will then be described, as well as our solutions to address them. Finally, we discuss several important design points.

# 2. In-field Upgrading

Programming a microcontroller is easy with a testbench. Developers have access to a variety of tools to get their code into the device. This includes debugging ports such as JTAG, hardware programmers, and so on.

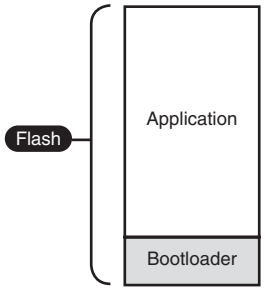
However, to be able to write a new firmware on a final product, the previous solutions cannot be used: debugging support is disabled, ports are not available, the end-user may be inexperienced, etc. Thus, in-field upgrading typically uses another technique to get the new code onto the device.

## 2.1 The Bootloader

Many modern microcontrollers use NOR 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 any time. Therefore, it must be as small as possible; 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.

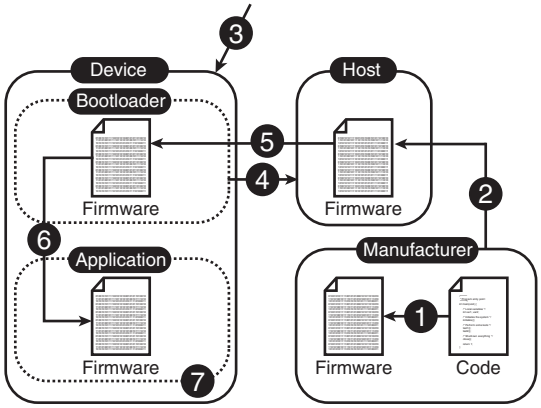
**Figure 2-1.** Memory Organization with a Bootloader




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 tries 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.

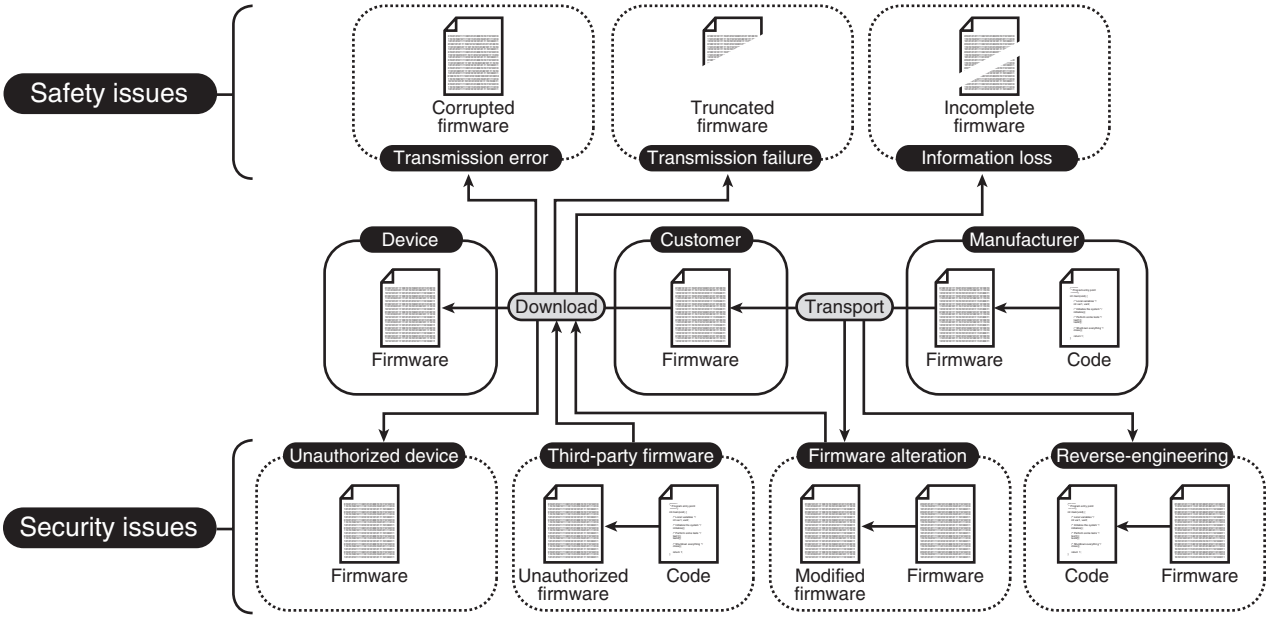
There are several other ways of carrying out in-field programming of a product. For example, the main application may do the upgrade itself: for a device using an external memory storage, the new firmware could be written on it as a file. The main advantage of using a bootloader is that you do not have 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, as described in the previous section. They are presented according to which area they are related, i.e., to the safety of the device or to the security of the firmware.

Those issues may 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. Below is a diagram showing several issues which are discussed in following sections.

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

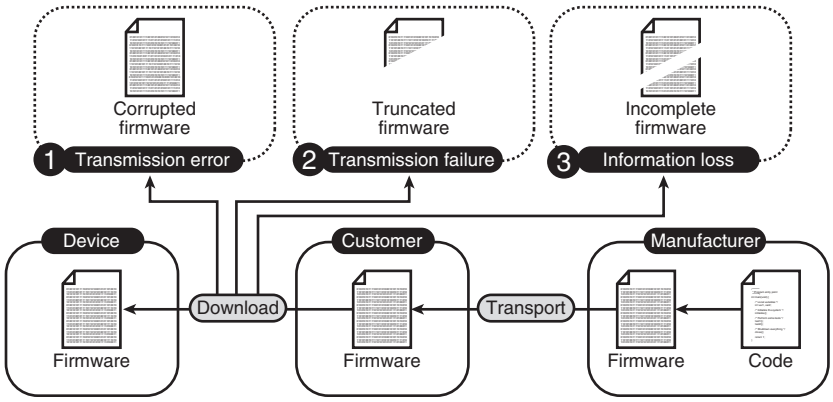



Section 2.2.1 ”Safety” on page 5 and Section 2.2.2 ”Security” on page 6 describe the following concerns:

- Safety
  - Transmission error
  - Transmission failure
  - Information loss
- Security
  - Firmware reverse-engineering
  - Use of unauthorized firmware
  - Firmware modification
  - Use of a firmware on an unauthorized device

### 2.2.1 Safety

**Figure 2-4.** Safety Problems with a Basic Bootloader



It is critical for most devices to have a working firmware embedded in them at all times, since they probably cannot function properly or 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.

An obvious problem arises if the device suddenly loses power during the update process. The application area would then be corrupted and unusable. This may be considered as a transfer failure (**issue #2**).

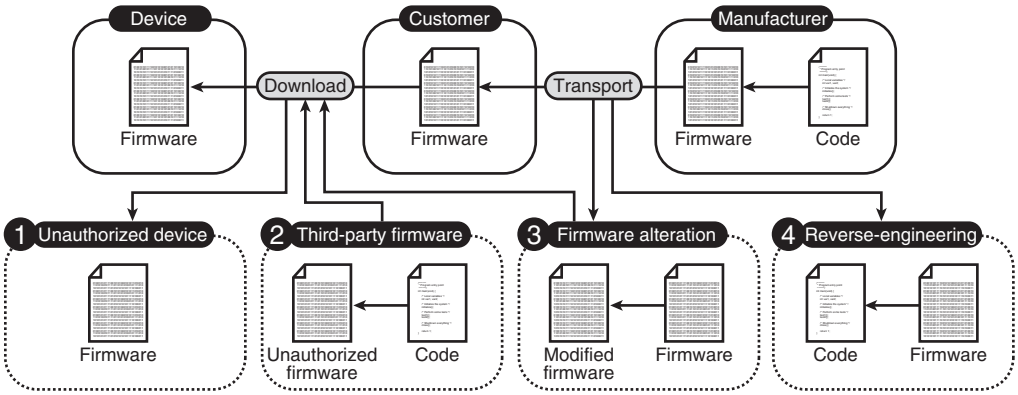
Same result (or almost) if the connection to the host is lost during the transmission. Since the firmware is downloaded and upgraded at the same time, the product would end up with part of both the new and old firmwares. Although there is a small chance that the system remains functional (i.e., in the case that only unmodified parts between the old and new firmwares have been written), it is unlikely to happen.

Alternatively, if a transmission error occurs during the transfer (e.g., a bit is flipped in a block of data, **issue #1**), part of the code is corrupted. The result may vary from a small computation error to a crash when that code is executed (depending on the instruction which is modified and how it is changed).

Lastly, some data may be lost while transmitting the firmware (**issue #3**). This would completely corrupt the code after the missing part. Likewise, some media might reorder blocks of data, making it necessary to check which block is received or not and where it belongs.

### 2.2.2 Security

**Figure 2-5.** Security Problems with a Basic Bootloader




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 de-compile it to retrieve the original code (**issue #4**).

**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 has not been designed by the original manufacturer, but by a third-party (**issue** #**2**). This may be especially problematic if that firmware has been developed for malicious use, i.e., to bypass security protections, illegally use critical functions of the device, and so forth.

A genuine firmware could also be used on a different device than the one it is intended for (**issue** #**1**). 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** #**3**). It would appear as genuine, but attacks similar to those described in the previous paragraph could be achieved in this way.

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.

# 3. Proposed Solution

This document offers a practical solution to the issues we have identified. 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.

## 3.1 Safety

The following techniques are ways to prevent a safety-related errors to happen. 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 we present the following solutions.

### 3.1.1 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 issues #2 and #3 on figure 2-4).

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 codes**, **block numbering** and **packet acknowledgement**. They are presented below. Existing protocols for the media available on Atmel® ARM® Thumb® based AT91SAM microcontrollers are then described in Section 3.1.1.4 ”Existing Protocols” on page 10.

#### *3.1.1.1 Error Detection/Correction*

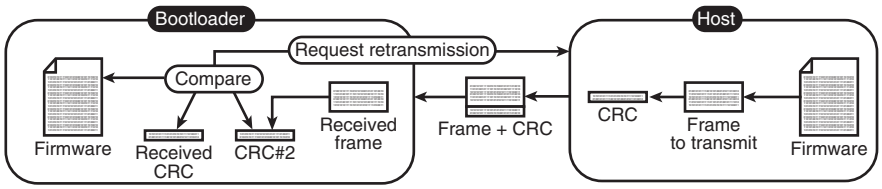
It is common to use **error detection** and **error correction** codes for transmitting data. Indeed, many operations cannot handle receiving a corrupted piece of data: loading a new firmware, sending a file across a network, and so on. Therefore, several codes have been designed to make it possible 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 3-1.** 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.

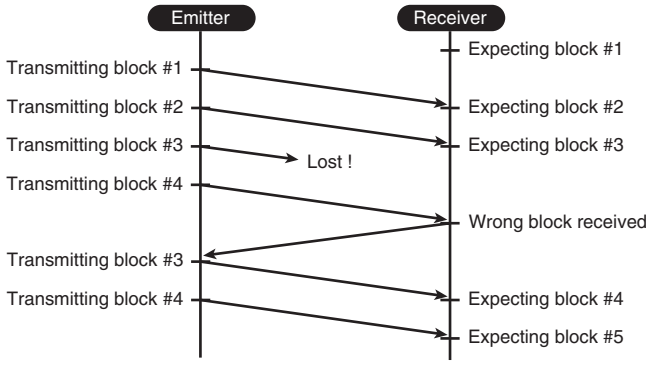
Finally, error **correction** is not really necessary in this particular case. It is most suited when it is unpractical to resend the erroneous data, which is not a concern here. Since error correction codes typically incur a bigger overhead than error detection ones, they should be implemented with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

### *3.1.1.2 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 3-2.** Block Numbering

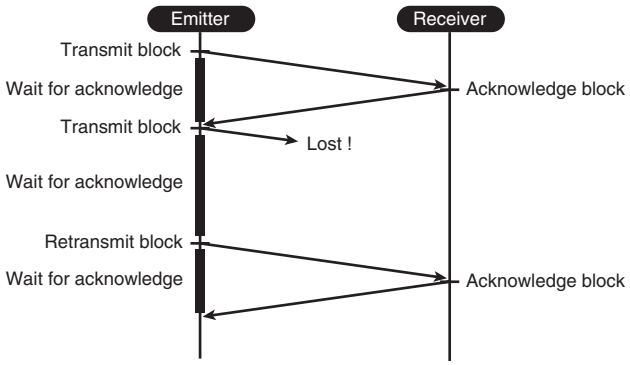



### *3.1.1.3 Packet Acknowledgement*

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 3-3.** Packet Acknowledgement




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.

*3.1.1.4 Existing Protocols*

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

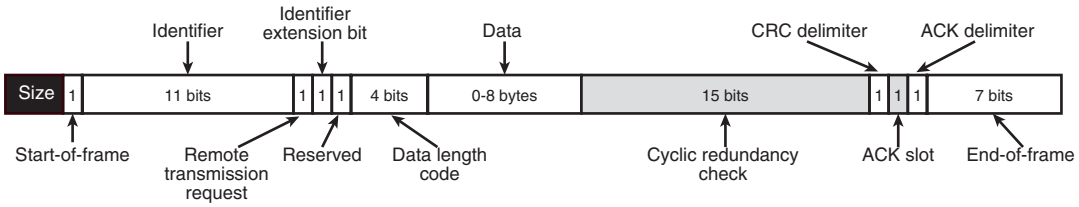
**TCP/IP** is the most widely used protocol stack on top of Ethernet. The Transport Control Protocol (TCP) implements reliability by using a **packet sequence number** as well as a **checksum** (a simple error detection code). It also uses a variant of packet acknowledgment, but since several packets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

**Figure 3-4.** 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 | | | |

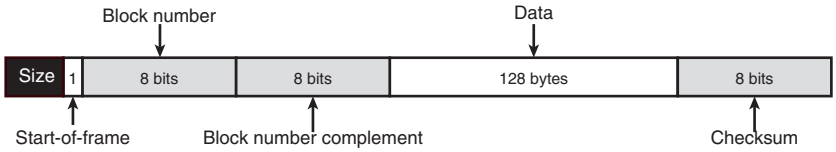
The USB protocol uses a Cyclic Redundancy Check (CRC) for error detection. There is no sequence number on the packets, but a receiver acknowledges each block of data. This is the same for the CAN bus.

**Figure 3-5.** 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 3-6.** XMODEM Frame Structure



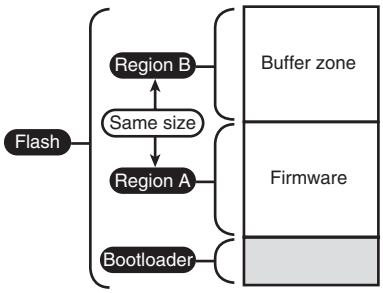

### 3.1.2 Memory Partitioning

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.

The solution we present 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 3-7.** Memory Organization with Memory Partitioning




Region B is used as a buffer for the new firmware, to download it entirely before programming it in region A. This method ensures that there is always a working firmware on a device after an upgrade, whether it was successful or not.

### 3.1.3 Summary

**Error detection and correction**

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

**Memory partitioning**

- Pros
  - Solves all safety-related issues
- Cons
  - The required memory size is doubled
  - Slightly increased code size
  - Reduced execution speed (during upgrade only)

## 3.2 Security

Several security-related techniques to solve the aforementioned issues (see Section 2.2.2 on page 6) are presented in this section. See Section 5.2 on page 20 for in-depth information about security considerations.

### 3.2.1 Integrity

Verifying integrity means checking the following:

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

Accidental modification is a safety problem, and has been already discussed. It is typically solved by using error detection codes (see Section 3.1.1.1 on page 8).

There are several ways to check 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 problem #3** described in Figure 2-5 on page 6.

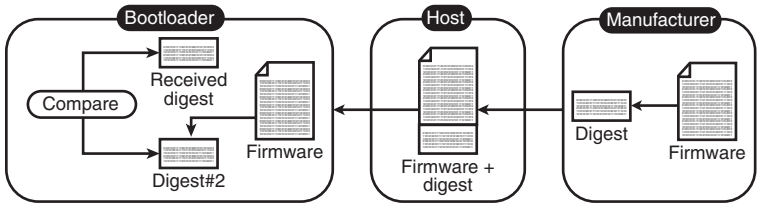
#### *3.2.1.1 Hash function*

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 recomputes the fingerprint and compares it to the original ones. If both are identical, then the firmware has not been altered.

In practice, a hash function takes a string of any length has 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 3-8.** 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 recompute the hash. The bootloader would thus not be able to tell that an alteration was made.

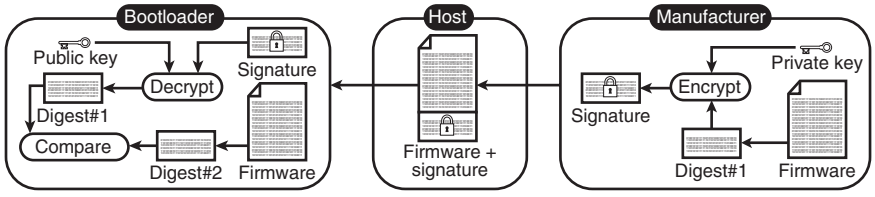
However, a hash function alone can still be used to verify the firmware integrity at runtime, to avoid running a damaged application.

### *3.2.1.2 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 the 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 3-9.** Digital Signature Creation and Verification

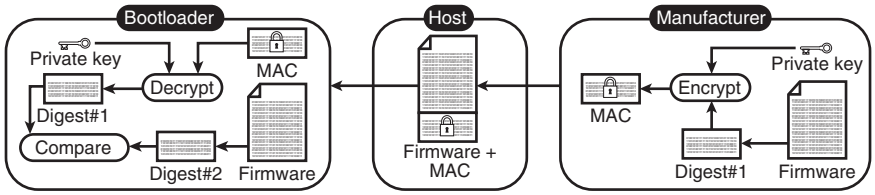



Since only the private key can encrypt data, nobody except the manufacturer can produce the signature. Thus, a malicious user would not be able to perform the attack described in the previous section. But anybody can verify the signature using the public key of the manufacturer.

### *3.2.1.3 Message Authentication Codes*

**Message Authentication Codes** (MACs) provide the same functionality as digital signatures, except 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 3-10.** 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 up a big difference in practice.

### 3.2.2 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 problems #1 and #2** described in Figure 2-4 on page 5.

It happens that the methods which provide authentication also provide integrity: **digital signatures** and **MACs**. Since they have already been described from the integrity point-of-view (see Section 3.2.1 on page 12), this section will only discuss their authentication properties.

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

#### *3.2.2.1 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 used, i.e. two different texts giving the same hash. The signature would also authentify 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.

#### *3.2.2.2 Message Authentication Code*

Conversely 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.

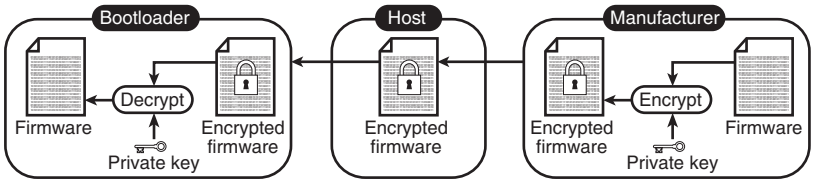
The additional concern of a MAC compared to a digital signature is that an attacker should never be able to retrieve the private key inside the bootloader. If he manages to do that, he would be able to create or modify a firmware, issuing the associated MAC needed to authentify it as a genuine one to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

### 3.2.3 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 problem 4**, as described in Figure 2-4 on page 5.

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 3-11.** Firmware Encryption




Note that code encryption does not solve every security issue all by itself. For example, the firmware may 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 5.2.1.2 on page 22).

### 3.2.4 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 authentify the device. Indeed, it will need the private key to decrypt the firmware. As only genuine devices have 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.

### 3.2.5 Summary

**Hash function**

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

**Digital signature**

- Pros
  - Detects third-party and modified firmwares
  - 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**

- Pros
  - Detects third-party and modified firmwares
  - 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**

- 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)

# 4. Examples of Use

In this section, several possible uses and requirements for a bootloader are presented. The possible solutions to each scenario are discussed in term of safety & security advantages versus code size & speed performance.

## 4.1 Security

### 4.1.1 Scenario 1

#### *4.1.1.1 Requirements*

The only requirement of this first basic scenario is to prevent competitors/malicious users to obtain access to the firmware code.

#### *4.1.1.2 Core Solution*

The first step is to use code encryption to protect the firmware. This requires the implementation of a symmetric cipher, as well as the generation and inclusion in the bootloader of a private key.

On the manufacturer side, the same private key needs to be secured. It will be used to encrypt new firmware versions as they come along.

#### *4.1.1.3 Options*

Since you have to include a symmetric cipher for encryption, it can be reused to implement a message authentication code (such as OMAC/CMAC). The MAC will make it possible for the bootloader to check that the firmware has not been modified (either accidentally or voluntarily).

### 4.1.2 Scenario 2

#### *4.1.2.1 Requirements*

A customer must never have his device compromised by a modified or third-party malicious firmware.

#### *4.1.2.2 Core Solutions*

Authenticity and integrity are needed in this case. But while theoretically both digital signatures and message authentication codes can be used to provide them, the former is much more secure.

Consider the case when the bootloader code (containing the private or public key needed for authentication) is dumped (using an hardware attack, for example). Now, in the case of a digital signature, the attacker has no more power: he cannot create valid signatures with only the public key. Whereas with a MAC, he will be able to do so.

However, a digital signature algorithm will probably be more size & speed consuming than a MAC. If the requirement is very strict, then opt for digital signatures. If you do not expect people to have access to advanced hardware hacking tools, than a MAC might be acceptable.

#### *4.1.2.3 Options*

If a MAC based on a symmetric cipher is used (such as OMAC/CMAC), than you might consider encrypting the firmware. Since the same algorithm (and even the same key) can be reused, the size overhead will be small.

### 4.1.3 Scenario 3

#### *4.1.3.1 Requirements*

Maximum security is required:

- Firmware code cannot be read by users
- A customer must not be able to download a non-genuine firmware
- Non-authorized targets must not be able to use the firmware

#### *4.1.3.2 Core Solutions*

This scenario requires privacy, authenticity and integrity. Only code encryption can provide privacy; a symmetric cipher will thus have to be implemented. Code encryption will also prevent unauthorized device from loading the code (see Section 3.2.4 on page 16).

Authenticity and integrity can be provided both by either a MAC or a digital signature. Using a MAC in this case will keep the code size down (by reusing the block cipher added for code encryption). A digital signature will provide a better security margin.

Note that combining a MAC and a digital signature has no added security benefit over simply using a signature.

# 5. 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.

## 5.1 Transmission Media

AT91 microcontrollers provide a wide variety of peripherals to communicate with an external host, such as:

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

Choice should be made on the implementation priority (and relevance) of each method. Given the simplest 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. Please note however 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 firmwares.

## 5.2 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 pseudo-random number generators (PRNG). While the latter has not been introduced before because it is not a security method by 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 CRYPTREC or NESSIE, which carefully analyze existing and new algorithms.

Note: While developing a custom cryptographic algorithm has the advantage of hiding how it is implemented, it is preferable to apply standards. Indeed, the standards have been thoroughly tested and attacked, making the knowledge of which of them is used irrelevant. Similarly, avoid (when possible) implementing a standard algorithm yourself and try to use standard implementations to avoid bugs which will weaken the encryption.

### 5.2.1 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.

#### 5.2.1.1 Choosing an Algorithm

A symmetric encryption algorithm can be defined by several characteristics:

- Key length in bits
- Block length in bits
- Security
- Size & speed

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

A large **block length** is needed to avoid a “codebook attack”, i.e., someone getting enough blocks of plain text and their corresponding cipher text to build a table, enabling him to 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 (but still reasonable) size. Most block ciphers will use at least 64 bits, with modern ciphers using at least 128 bits.

The **security** of the cipher is, of course, critical. If the algorithm has flaws, then it may be easily breakable. Old ciphers like DES have been found flawed over time and thus abandoned. As cryptanalysis evolves, new methods may be found to break ciphers which are currently considered secure.

Breaking a cipher does not mean that it becomes instantly decryptable; it may mean that instead of searching through $2^{128}$ keys, only $2^{100}$ searches are necessary. Many attacks on and breaches of ciphers are thus quite unusable, even for old algorithms like DES.

Finally, different algorithms have different **speeds and sizes**, depending on the techniques they use. Some ciphers are especially suited for embedded systems, as they require less memory (both in code and data). Similarly, some algorithms may be more suited to fast encryption, fast decryption, efficient hardware implementation and so on.

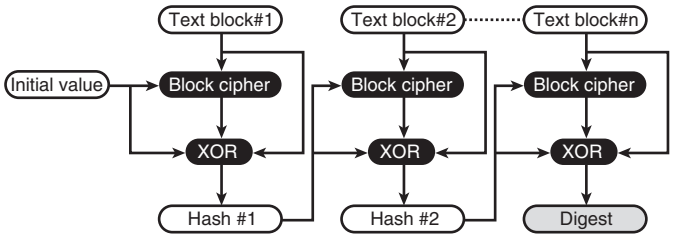
The table below gives an overview of several popular ciphers. Note that if the target platform has hardware acceleration available, the resulting code size will be much smaller, and the system much faster.

**Table 5-1.** 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 footprint |

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 5-1.** Block Cipher as Hash Function




*5.2.1.2 Modes of Operation*

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

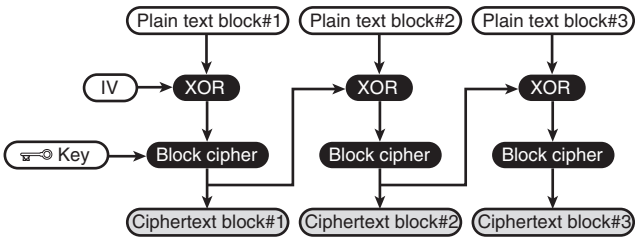
- Electronic codebook (ECB)
- Cipher block chaining (CBC, CFB, OFB, CTR)
- Authenticated encryption (EAX, CCM, OCB)

The basic mode of operation is ECB: 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. Indeed, 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. This means that each block depends on the previously encrypted data, making everything interdependent.

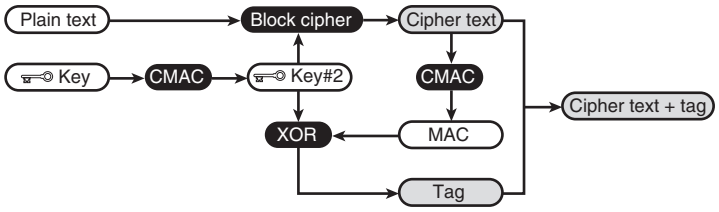
The first block is encrypted using a random **Initialization Vector** (IV). While this vector can be transmitted in clear text, the same vector shall never be used more than once with the same key. It is likely that a manufacturer will produce more than one firmware upgrade for a product in its lifetime. 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 5-2.** CBC Mode of Operation




Lastly, **authenticated encryption** modes 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 have less overhead than applying a MAC and a symmetric cipher separately.

**Figure 5-3.** EAX Mode of Operation




*5.2.1.3 Performances on an AT91SAM7XC*

Several of the aforementioned ciphers and modes have been tested using different implementations on a AT91SAM7XC. This section presents the results obtained.

The **AES** cipher has been tested using three different implementations. The first one uses the **libTomCrypt** library, freely available from http://libtomcrypt.com. The second one is based on a standard implementation provided by Paulo Baretto and Vincent Rijmen, while the last one uses the hardware peripheral provided by the SAM7XC chip.

**Table 5-2.** Performance Measurement of AES (128-bit key and 128-bit blocks)

| Encryption Mode | Implementation Source | Size Overhead (bytes) | Decryption Time for a 128 KB File (ms) |
|---|---|---|---|
| ECB | libTomCrypt | 7,280 | 1089.9 |
| | Reference implementation | 2,744 | 2654.9 |
| | Hardware acceleration | 364 | 19.4 |
| CBC | libTomCrypt | 7,372 | 1206.4 |
| | Reference implementation | 2,804 | 2624.4 |
| | Hardware acceleration | 432 | 19.4 |
| CTR | libTomCrypt | 7,520 | 1278 |
| | Reference implementation | 2,084 | 2674.6 |
| | Hardware acceleration | 432 | 19.4 |

**Triple-DES** was tested using the libTomCrypt implementation as well as the hardware acceleration of the SAM7XC.

**Table 5-3.** Performance Measurement of Triple-DES (168-bit key and 64-bit blocks)

| Encryption Mode | Implementation Source | Size Overhead (bytes) | Decryption Time for a 128 KB File (ms) |
|---|---|---|---|
| ECB | libTomCrypt | 6,280 | 2998 |
| | Hardware acceleration | 344 | 61.2 |
| CBC | libTomCrypt | 6,376 | 3110.6 |
| | Hardware acceleration | 424 | 61.2 |

### 5.2.2 Hash Functions

A hash function has three defining characteristics:

- Output length
- Security
- Size & speed

The **output length** of a hash needs to be large enough. It must make it almost impossible to find collisions, i.e., two different files having the same digest. This also prevent someone from finding a piece of data producing a specific hash. In practice, most modern hash functions will have at least a 160-bit output (like SHA-1). Note that the standard is quickly moving towards 512 bits.

But 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 mean it becomes completely cracked; most attacks are still not feasible without gigantic computational power. Still, newer designs which are considered secure for the moment should be preferred over depreciated algorithms.

When deciding on a hash function, its **size & 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.

Table 5-4 details several commonly used hash algorithms.

**Table 5-4.** Hash Algorithms

| Algorithm | Hash 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 |

### 5.2.3 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.

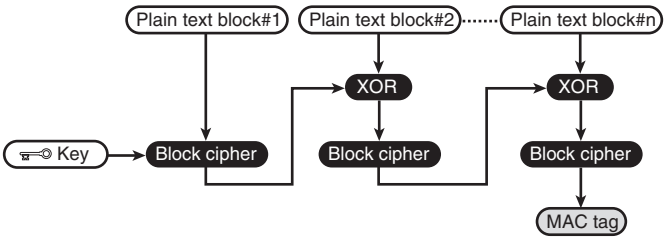
Here are the different types of (secure) MACs available:

- 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

For example, if you are already using a hash function (to check the firmware integrity at startup), then using a HMAC will not have a high overhead.

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 5-4.** CMAC Message Authentication Code




### 5.2.4 Digital Signature Algorithms

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

- the Digital Signature Standard (DSS)
- a system based on the Rivest-Shamir-Adleman (RSA) public-key algorithm

**DSS** is, as its name implies, 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.

The most popular method is using **RSA** with a padding scheme. RSA is originally a public-key (or asymmetric) encryption algorithm, and does not directly support encrypting an input which is smaller than the key length. Since a key is often more than 1024 bits long, the resulting digest of a hash function falls in this category.

Therefore, digital signing with RSA requires the use of a **padding scheme**. The goal is to append additional data (or **pad**) to the message to encrypt, making the system secure. There are three commonly used padding schemes:

- Full-domain hashing
- Optimal Asymmetric Encryption Padding (OEAP)
- Probabilistic Signature Scheme (PSS)

**Full-domain hashing** is not really a padding scheme, since it involves using a hash function having an output size equals to the RSA key length. That way, the message digest can represented any value between 0 and $2^k$ (k being the key length used), making the system secure. However, it is not really usable in practice since currently no hash function can produce an output longer than 512-bits.

**OEAP** and **PSS** both rely on adding a quantity of random data to convert RSA into a probabilistic encryption scheme. These two algorithms, combined with RSA, show strong security properties (which are provable). The drawback is that they require the use of a random number generator to add *salt* (randomness) to the message. However, this is only needed to sign a message, not to verify it; it has no impact on the code size of the decryption algorithm.

### 5.2.5 Pseudo-random Number Generators

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.

Note however 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 no 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, nonces, 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

### 5.2.6 Available Implementations

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

#### *5.2.6.1 Libraries in C*

**libTomCrypt** (http://libtomcrypt.com/)

- Symmetric ciphers: AES, DES/Triple-DES, Blowfish, RC6, Twofish
- Modes of operation: ECB, CBC, OFB, CFB, CTR, EAX, OCB, CCM
- Hash functions: MD5, SHA-1, SHA-256, TIGER-192, RIPEMD-160, WHIRLPOOL
- Message authentication codes: HMAC, CMAC, PMAC
- Digital signatures: DSA, RSA/OEAP, RSA/PSS
- Pseudo-random number generators: Yarrow, Fortuna

**Catacomb** (http://www.excessus.demon.co.uk/misc-hacks/#catacomb)

- Symmetric ciphers: Blowfish, DES/Triple-DES, AES, Serpent, Twofish
- Modes of operation: ECB, CBC, CFB, OFB, CTR
- Hash functions: MD5, SHA-1, SHA-256, Tiger
- Message authentication codes: HMAC
- Digital signatures: DSA, RSA/OEAP, RSA/PSS

**OpenSSL crypto** (http://www.openssl.org/)

- Symmetric ciphers: Blowfish, DES/Triple-DES
- Modes of operation: ECB, CBC, CFB, OFB
- Hash functions: MD5, RIPEMD-160, SHA-1
- Message authentication codes: HMAC
- Digital signatures: DSA, RSA/OEAP

#### 5.2.6.2 *Libraries in C++*

**Crypto++** (http://www.eskimo.com/~weidai/cryptlib.html)

- Symmetric ciphers: AES, RC6, Twofish, Serpent, DES/3DES, Blowfish
- Modes of operation: ECB, CBC, CFB, OFB, CTR
- Hash functions: MD5, HAVAL, RIPEMD-160, Tiger, SHA-1, SHA-256, WHIRLPOOL
- Message authentication codes: HMAC, CMAC
- Digital signatures: RSA/OAEP, RSA/PSS, DSA
- Pseudo-random numbers generators: Blum Blum Shub

#### 5.2.6.3 *Separate Implementations in C*

Brian Gladman (http://fp.gladman.plus.com/cryptography_technology/index.htm)

- Symmetric ciphers: AES, Serpent
- Modes of operation: ECB, CBC, CFB, OFB, CTR, CCM, EAX
- Hash functions: SHA-1, SHA-256
- Message authentication codes: HMAC, OMAC

## 5.3 Error Detection Codes

Since AT91 microcontrollers are based on a 32-bit architecture, it seems logical to implement codes which are at least as long. They will not cost more in terms of speed and size than an 8-bit or 16-bit variant.

Originally, simple **checksums** where 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 a slightly less reliable than CRC-32 but significantly faster algorithm. 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 once.

## 5.4 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. Several formats include additional

information, such as linking addresses (Motorola s-record, Intel® .hex) or debug information (Executable and Linking Format .elf).

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.

## 5.5 Target Chips

Some of the chips in the AT91SAM family have different IPs, e.g., for the Flash controller. This means that they are programmed differently; therefore, several versions of the code must be written to accommodate all the microcontrollers. This is already done with the “Basic” series of applications offered by Atmel for the AT91SAM family.

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.

A good “first-chip” candidate seems to be the AT91SAM7XC, since it embeds cryptographic accelerators. This will allow testing of both the software and hardware version of AES (and DES/3DES) without using two different microcontrollers.

## Atmel Corporation

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

## Regional Headquarters

*Europe*
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
Tel: (41) 26-426-5555
Fax: (41) 26-426-5500

*Asia*
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

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

## Atmel Operations

*Memory*
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

*Microcontrollers*
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
Tel: (33) 2-40-18-18-18
Fax: (33) 2-40-18-19-60

*ASIC/ASSP/Smart Cards*
Zone Industrielle
13106 Rousset Cedex, France
Tel: (33) 4-42-53-60-00
Fax: (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
Tel: (44) 1355-803-000
Fax: (44) 1355-242-743

*RF/Automotive*
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
Tel: (49) 71-31-67-0
Fax: (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

*Biometrics/Imaging/Hi-Rel MPU/*
*High-Speed Converters/RF Datacom*
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
Tel: (33) 4-76-58-30-00
Fax: (33) 4-76-58-34-80


*Literature Requests*
www.atmel.com/literature