# Atmel AVR231: AES Bootloader

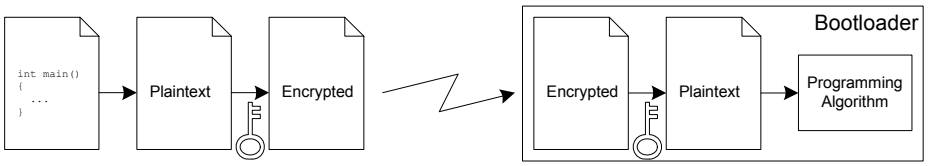
## Features

- **Fits Atmel® AVR® Microcontrollers with bootloader capabilities and at least 1kB SRAM**
- **Enables secure transfer of firmware and sensitive data to an AVR based application**
- **Includes easy-to-use configurable example applications:**
  - **Encrypting binary files and data**
  - **Creating target bootloaders**
  - **Downloading encrypted files to target**
- **Implements the Advanced Encryption Standard (AES):**
  - **128-, 192-, and 256-bit keys**
- **AES Bootloader fits into 2kB**
- **Typical update times of a 64kB application, 115200 baud, 3.69MHz target frequency:**
  - **AES128: 27 seconds**
  - **AES192: 30 seconds**
  - **AES256: 33 seconds**

## 1 Introduction

This application note describes how firmware can be updated securely on AVR microcontrollers with bootloader capabilities. The method uses the Advanced Encryption Standard (AES) to encrypt the firmware.

**Figure 1-1.** Overview.




Electronic designs with microcontrollers always need to be equipped with firmware, be it a portable music player, a hairdryer or a sewing machine. As many electronic designs evolve rapidly there is a growing need for being able to update products that have already been shipped or sold. It may prove difficult to make changes to hardware, especially if the product has already reached the end customer, but the firmware can easily be updated on products based on Flash microcontrollers, such as the AVR.

Many AVR microcontrollers are configured such that it is possible to implement a bootloader able to receive firmware updates and to reprogram the Flash memory on demand. The program memory space is divided in two sections: the Bootloader Section (BLS) and the Application Section. Both sections have dedicated lock bits for read and write protection so that the bootloader code can be secured in the BLS while still being able to update the code in the application area. Hence, the update algorithm in the BLS can easily be secured against outside access.

**8-bit Atmel Microcontrollers**

**Application Note**

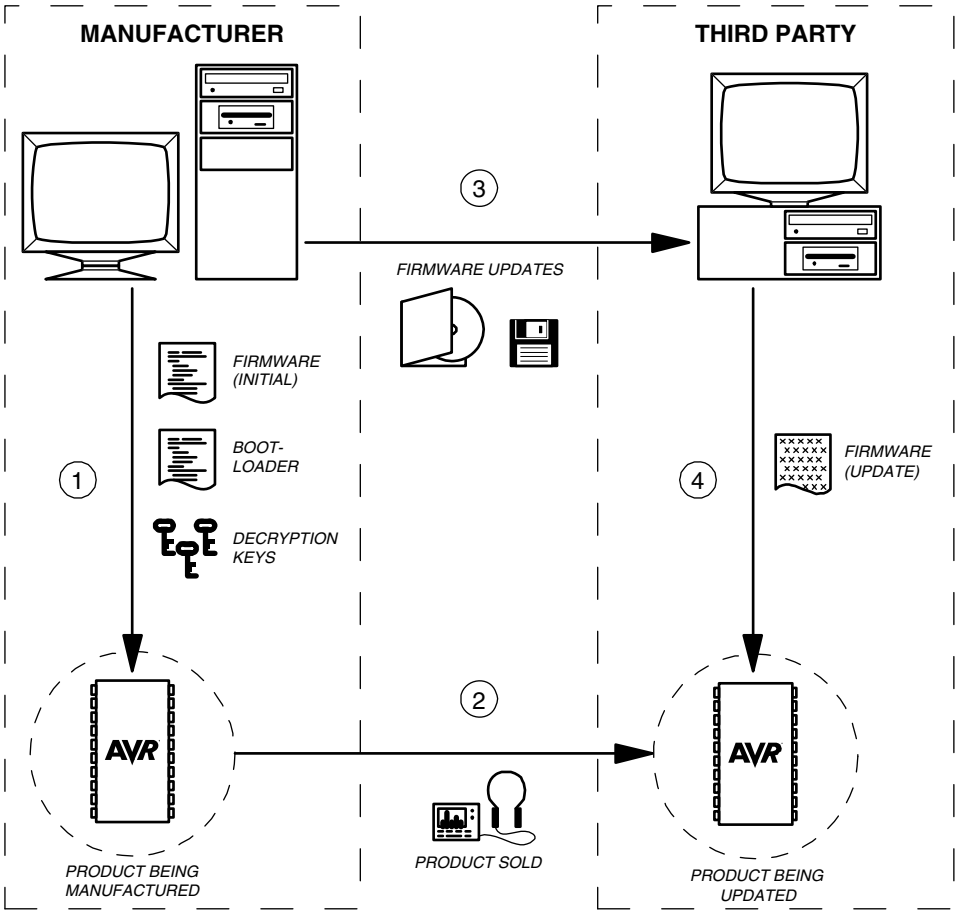
Rev. 2589E-AVR-03/12

The problem remains with the firmware, which typically is not secure before it has been programmed into Flash memory and lock bits have been set. This means that if the firmware needs to be updated in the field, it will be open for unauthorized access from the moment it leaves the programming bench or manufacturer's premises.

This application note shows how data to be transferred to Flash and EEPROM memories can be secured at all times by using cryptographic methods. The idea is to encrypt the data before it leaves the programming bench and decrypt it only after it has been downloaded to the target AVR. This procedure does not prevent unauthorized copying of the firmware, but the encrypted information is virtually useless without the proper decryption keys. Decryption keys are only stored in one location outside the programming environment: Inside the AVR. The keys cannot be regenerated from the encrypted data. The only way to gain access to the data is by using the proper keys.

Figure 1-2 shows an example of how a product is first manufactured, loaded with initial firmware, sold and later updated with a new revision of the firmware.

**Figure 1-2.** An example of the typical production and update procedure.



Notes:
1. During manufacturing, the microcontroller is first equipped with bootloader, decryption keys and application firmware. The bootloader takes care of receiving the actual application and programming it into Flash memory, while keys are required for decrypting the incoming data. Lock bits are set to secure the firmware inside the AVR.
2. The product is then shipped to a distributor or sold to the end customer. Lock bit settings continue to keep the firmware secured inside the AVR.
3. A new release of the firmware is completed and there is a need to update products, which already have been distributed. The firmware is therefore encrypted and shipped to the distributor. The encrypted firmware is useless without decryption keys and therefore even local copies of the software (for example, on the hard drive of the distributor) do not pose a security hazard.
4. The distributor upgrades all units in stock and those returned by customers (for example, during repairs). The encrypted firmware is downloaded to the AVR and decrypted once inside the microcontroller. Lock bit settings continue to keep the updated firmware secured inside the AVR.

# 2 Cryptography overview

The term cryptography is used when information is locked and made unavailable using keys. Unlocking information can only be achieved using the correct keys.

Algorithms based on cryptographic keys are divided in two classes; symmetric and asymmetric. Symmetric algorithms use the same key for encryption and decryption while asymmetric algorithms use different keys. AES is a symmetric key algorithm.

## 2.1 Encryption

Encryption is the method of encoding a message or data so that its contents are hidden from outsiders. The plaintext message or data in its original form may contain information the author or distributor wants to keep secret, such as the firmware for a microcontroller. For example, when a microcontroller is updated in the field it may prove difficult to secure the firmware against illicit copying attempts and reverse engineering. Encrypting the firmware will render it useless until it is decrypted.

## 2.2 Decryption

Decryption is the method of retrieving the original message or data and typically cannot be performed without knowing the proper key. Keys can be stored in the bootloader of a microcontroller so that the device can receive encrypted data, decrypt it and reprogram selected parts of the Flash or EEPROM memory. Decryption keys cannot be retrieved from the encrypted data and cannot be read from AVR microcontrollers if lock bits have been programmed accordingly.

# 3 AES implementation

This section is not intended to be a detailed description of the AES algorithm or its history. The intention is rather to describe the AVR-specific implementations for the various parts of the algorithm. Since memory is a scarce resource in embedded applications, the focus has been on saving code memory. The bootloader application will never be run the same time as the main code, and it is therefore not important to save data memory (RAM) as long as the data memory requirements do not exceed the capacity of the microcontroller.

If not interested in the AES implementation itself, the reader can skip right to Chapter 4, Software implementation and usage on page 13 without loss of continuity.

In the following subsections, some basic mathematical operations and their AVR-specific implementations are described. Note that there are some references to finite field theory from mathematics. Knowledge of finite fields is not required to read this document, but the interested reader should study the AES specification.

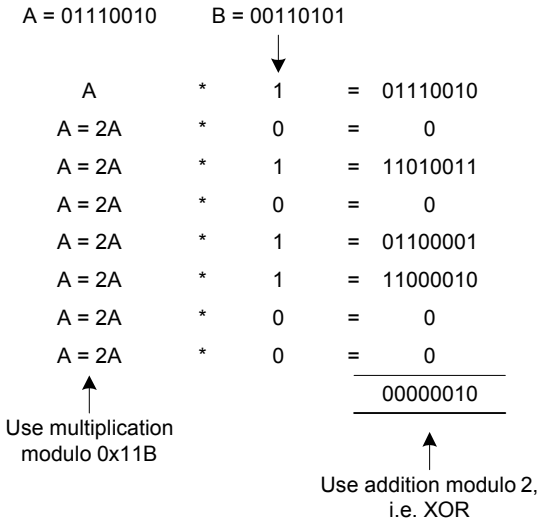
## 3.1 Byte addition

In the AES algorithm, byte addition is defined as addition of individual bits without carry propagation. This is identical to the standard XOR operation. The XOR operation is its own inverse; hence byte subtraction is identical to addition in the AES algorithm. XOR operations are trivial to implement on AVR.

## 3.2 Byte multiplication

In the AES algorithm, byte multiplication is defined as finite field multiplication with modulus 0x11B (binary 1 0001 1011). A suggested implementation is to repetitively multiply the first factor by 2 (modulo 0x11B) and sum up the intermediate results for each bit in the second factor having value 1. An example: If the second factor is 0x1A (binary 0001 1010), then the first, third and fourth intermediate results should be summed. Another example is shown in Figure 3-1. This method uses little memory and is well suited for an 8-bit microcontroller.

**Figure 3-1.** Byte multiplication.




The multiplication algorithm can be described by the following pseudo code:

```
bitmask = 1
tempresult = 0
tempfactor = firstfactor
while bitmask < 0x100
    if bitmask AND secondfactor <> 0
        add tempfactor to tempresult using XOR
    end if
    shift bitmask left once
    multiply tempfactor by 2 modulo 0x11B
end while
return tempresult
```

## 3.3 Multiplicative inverses

To be able to compute finite field multiplicative inverses, that is, 1/x, a trick has been used in this implementation. Using exponentiation and logarithms with a common base, the following identity can be utilized:

**Equation 3-1.** Using exponentiation and logarithms to compute 1/x.

$$a^{-\log_a x} = \frac{1}{x}$$

In this case the base number 3 has been chosen, as it is the simplest primitive root. By using finite field multiplication when computing the exponents and logarithms, the

multiplicative inverse is easy to implement. Instead of computing exponents and logarithms every time, two lookup tables are used. Since the multiplicative inverse is only used when preparing the S-box described in Section 3.4, the memory used for the two lookup tables can be used for other purposes when the S-box has been prepared.

The lookup table computation can be described by the following pseudo code:

```
tempexp = 0
tempnum = 1
do
    exponentiation_table[ tempexp ] = tempnum
    logarithm_table[ tempnum ] = tempexp
    increase tempexp
    multiply tempnum by 3 modulo 0x11B
loop while tempexp < 256
```

## 3.4 S-boxes

The AES algorithm uses the concept of substitution tables or S-boxes. One of the steps of the algorithm is to apply an invertible transformation to a byte. The S-box is the pre-computed results of this transformation for all possible byte values. The transformation consists of two steps: (1) A multiplicative inverse as described in Section 3.3, and (2) a linear transformation according to the following equation, where $a_i$ are the bits of the result and $b_i$ are the bits of the result from step 1.

**Equation 3-2.** Linear transformation used in the S-box.

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

A closer look at the matrix reveals that the operation can be implemented as the sum (using XOR addition) of the original byte, the right-hand vector and the original byte rotated left one, two, three and four times. This method is well suited for an 8-bit microcontroller.

The inverse S-box, used for decryption, has a similar structure and is also implemented using XOR additions and rotations. Refer to the AES specification for the corresponding matrix and to the source code for implementation details.

## 3.5 The ‘State’

The AES algorithm is a block cipher, which means that data is managed in blocks. For the AES cipher, the block size is 16 bytes. The AES block is often organized in a 4x4 array called the ‘State’ or the ‘State array’. The leftmost column of the State holds the first four bytes of the block, from top to bottom, and so on. The reader should also

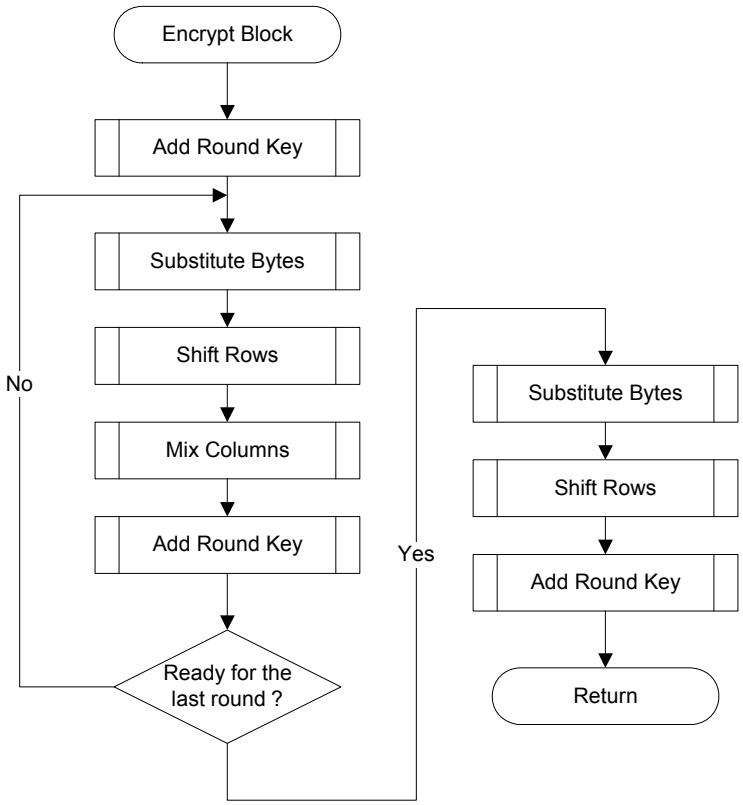
be aware that in the AES specification, four consecutive bytes are referred to as a word.

## 3.6 AES encryption

Before discussing the steps of the encryption process, the concept ‘encryption round’ needs to be introduced. Most block ciphers consist of a few operations that are executed in a loop a number of times. Each loop iteration uses a different encryption key. At least one of the operations in each iteration depends on the key. The loop iterations are referred to as encryption rounds, and the series of keys used for the rounds is called the key schedule. The number of rounds depends on the key size.

The flowchart for the encryption process is shown in Figure 3-2. The following subsections explain the different steps in the process. Each step is implemented as a subroutine for convenience. Using an optimizing compiler removes unnecessary function calls to save code memory.

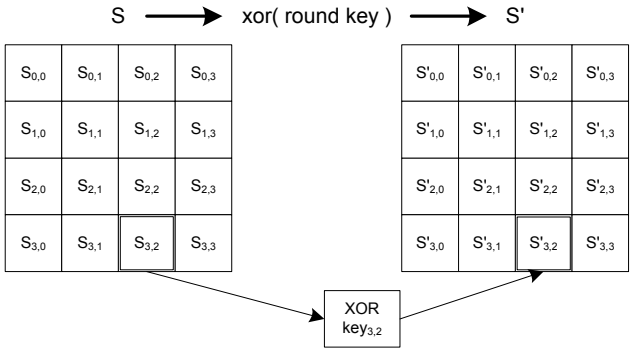
**Figure 3-2.** Encryption flowchart.




### 3.6.1 Add round key

This step uses XOR addition to add the current round key to the current State array. The round key has the same size as the State, that is, 16 bytes or four words. This operation is implemented as a 16-step loop.

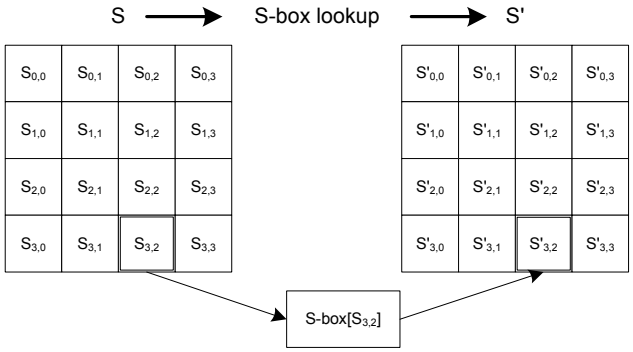
**Figure 3-3.** Adding the round key to the current state.




### 3.6.2 Substitute bytes

This step uses the precalculated S-box lookup table to substitute the bytes in the State. Like Section 3.6.1, this step is also implemented as a 16-step loop.

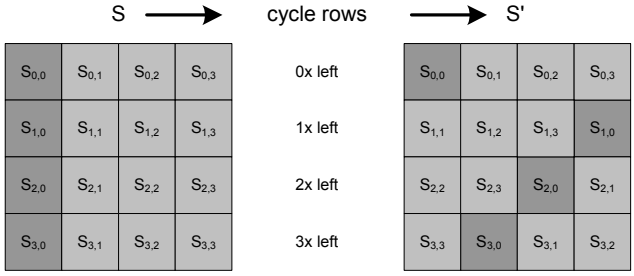
**Figure 3-4.** Substituting the bytes of the current state.




### 3.6.3 Shift rows

This step operates on the rows of the current State. The first row is left untouched, while the last three are cycled left one, two and three times, respectively. To cycle left once, the leftmost byte is moved to the rightmost column, and the three remaining bytes are moved one column to the left. The process is shown in Figure 3-5.

**Figure 3-5.** Cycling the rows of the current state.



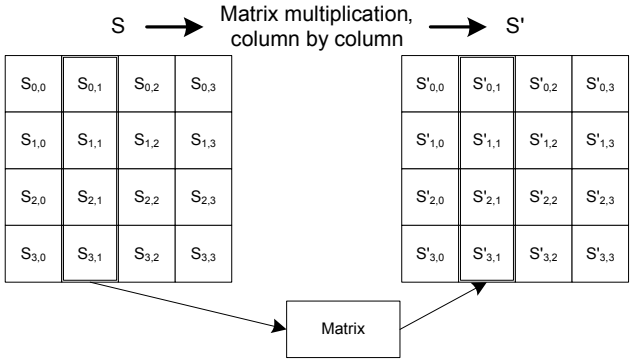

The naive implementation would be to write a subroutine that cycles a row left one time, and then call it the required number of times on each row. However, tests show that implementing the byte shuffling directly, without any loops or subroutines, results

in only a small penalty in code size but a significant gain (3x) in speed. Therefore the direct implementation has been chosen. Please refer to the `ShiftRows()` function in the source code for details.

### 3.6.4 Mix columns

This step operates on the State column by column. Each column is treated as a vector of bytes and is multiplied by a fixed matrix to get the column for the modified State.

**Figure 3-6.** Mixing the columns of the current state.




The operation can be described by the following equation, where $a_i$ are the bytes of the mixed column and $b_i$ are the bytes of the original column. Note that XOR addition and finite field multiplication from sections 3.1 and 3.2 are used.

**Equation 3-3.** Matrix multiplication when mixing one column.

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

This step is implemented directly without any secondary function calls. From the matrix equation one can see that every byte $a_i$ of the mixed column is a combination of the original bytes $b_i$ and their doubles $2b_i$. Please refer to the `MixColumns()` function in the source code for details.

## 3.7 Decryption

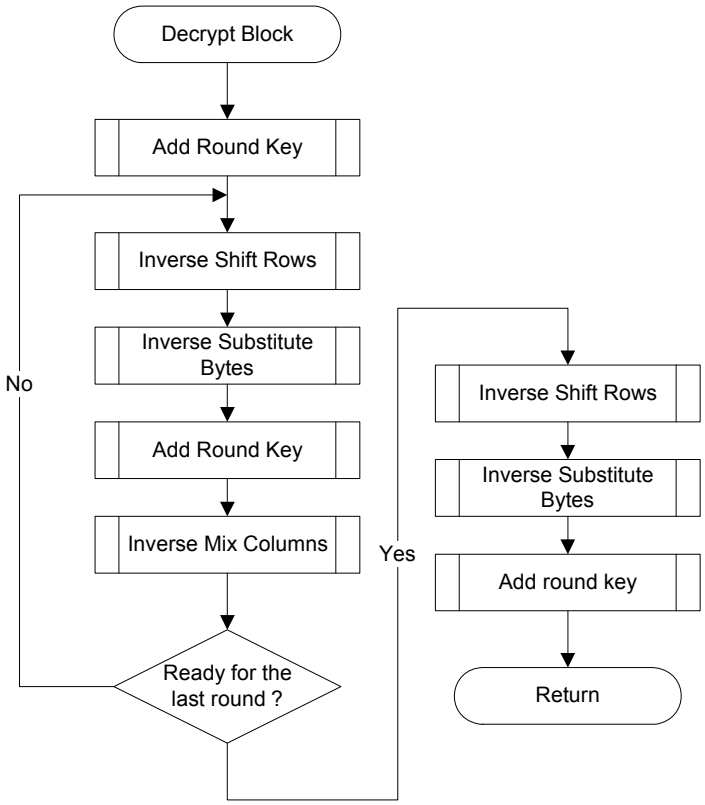
The flowchart for the decryption process is shown in Figure 3-7. The process is very similar to the encryption process, except the order of the steps has changed. All steps except “Add Round Key” have their corresponding inverses. “Inverse Shift Rows” cycles the rows right instead of left. “Inverse Substitute Bytes” uses inverse S-boxes.

“Inverse Mix Columns” also uses an inverse transformation. Please refer to the AES specification for the corresponding matrix and to the source code for implementation details.

NOTE

The key schedule used for decryption is the same as for encryption, but in reverse order.

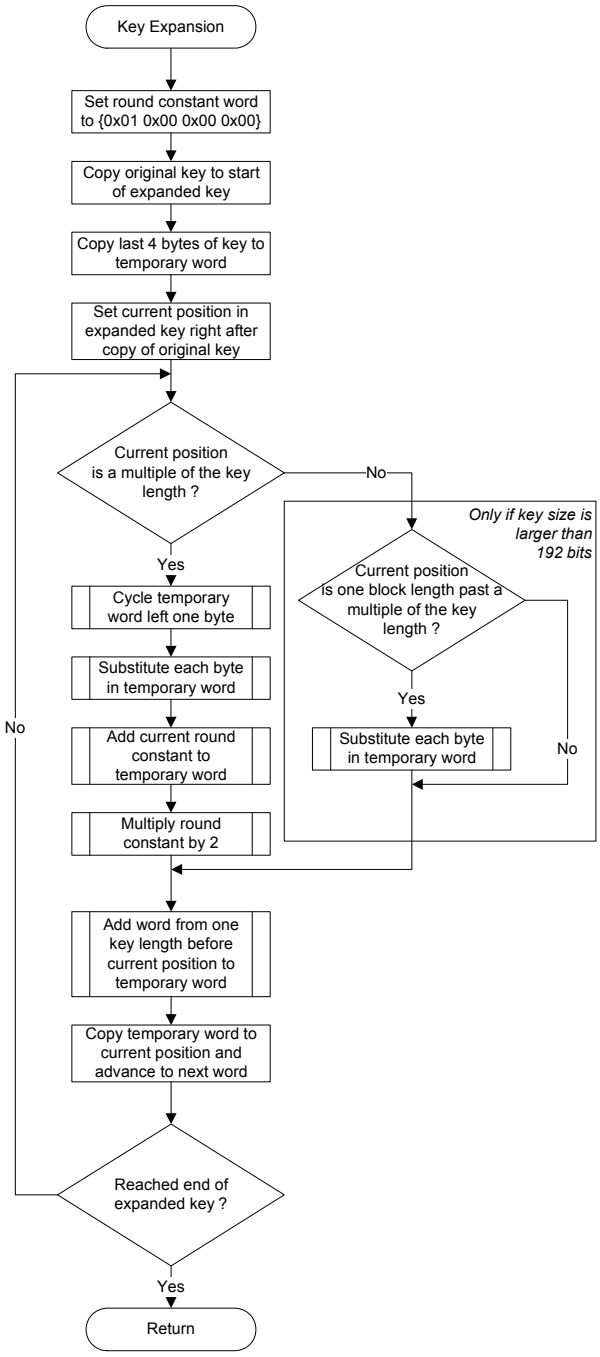
**Figure 3-7.** Decryption flowchart.




## 3.8 Key expansion

Key expansion is the process of generating the key schedule from the original 128-, 196- or 256-bit cipher key. The flowchart for key expansion is shown in Figure 3-8.

**Figure 3-8.** Key expansion flowchart.




The algorithm uses operations already described, such as XOR addition, finite field multiplication, substitution and word cycling. Please refer to the source code for details.

NOTE

The key expansion is identical for both encryption and decryption. Therefore the S-box used for encryption is required even if only decryption is used. In the AVR implementation, the ordinary S-box is computed prior to key expansion, and then its memory is reused when computing the inverse S-box.

## 3.9 Cipher block chaining – CBC

AES is a block cipher, meaning that the algorithm operates on fixed-size blocks of data. The cipher key is used to encrypt data in blocks of 16 bytes. For a known input block and a constant (although unknown) encryption key, the output block will always be the same. This might provide useful information for somebody wanting to attack the cipher system.

There are some methods commonly used which cause identical plaintext blocks being encrypted to different ciphertext blocks. One such method is called Cipher Block Chaining (CBC).

CBC is a method of connecting the cipher blocks so that leading blocks influence all trailing blocks. This is achieved by first performing an XOR operation on the current plaintext block and the previous ciphertext block. The XOR result is then encrypted instead of the plaintext block. This increases the number of plaintext bits one ciphertext bit depends on.

# 4 Software implementation and usage

This section first discusses some important topics for improving system security. These topics motivate many of the decisions in the later software design.

## 4.1 Motivation

This application note presents techniques that can be used when securing a design from outside access. Although no design can ever be fully secured it can be constructed such that the effort required to break the security is as high as possible. There is a significant difference between an unsecured design that a person with basic engineering skills can duplicate and a design that only few, highly skilled intruders can break. In the unsecured case, the design is easily copied and even reverse engineered, violating the intellectual property of the manufacturer and jeopardizing the market potential for the design. In the secured case, the effort required to break the design is so high that most intruders simply focus on developing their own products.

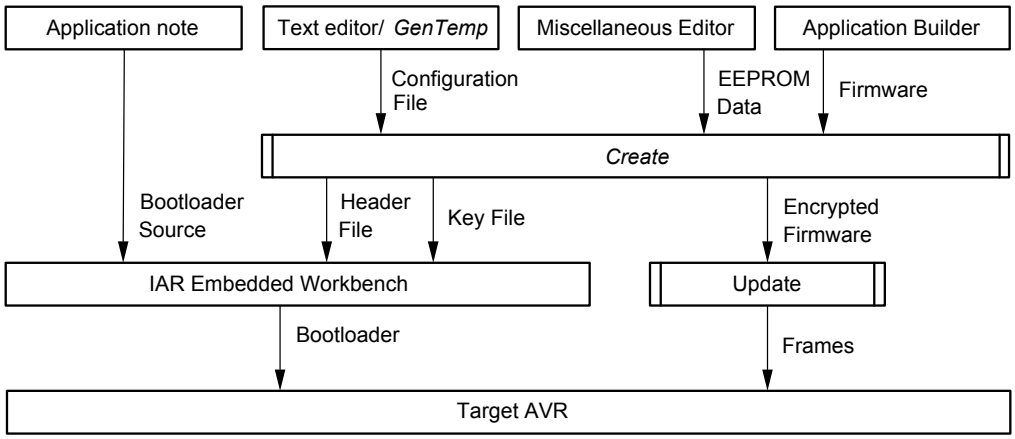
There is only one general rule on how to build a secure system: It should be designed to be as difficult to break as possible. Any mechanism that can be used to circumvent security will be tried during a break attempt. A few examples of what must be considered are given below.

- What will happen if power is removed during a firmware update? What is the state of the microcontroller when power is restored back? Are lock bits and reset vectors set properly at all times?
- Are there any assumptions that can be made on what plaintext data will look like? In order for AES to be broken, there must be a pattern to look for. The attack software will have to be configured to search for a known pattern, such as interrupt vectors at the start of program memory, memory areas padded with zero or one, and so on.
- Is there any feedback that can be derived from the decryption process? Any such feedback can help the attacker. For example, if the decryption algorithm inside the bootloader would give an OK / Not-OK type of signal for each block processed, then this signal could be used as feedback to the attacker.
- Should encrypted frames be sent in another order? If the first frame sent to the bootloader always includes the first block of the encrypted file then the attacker can make some assumptions from this. For example, it can be assumed that the first frame maps program data starting from address zero and that it contains the interrupt vector table. This information helps the attacker to refine the key search. To increase the security of the system, send the frames in random order (the decrypted frames will be mapped to their proper address, anyhow).

## 4.2 Usage overview

This and the following subsections describe how to use and configure the applications. The process is illustrated in Figure 4-1.

**Figure 4-1.** Overview of project flow.




The main steps are as follows:

- Create an application for the target AVR. If required, create an EEPROM layout in a separate file
- Create a configuration file with project dependent information. The application called `gentemp` can be used for creating a file frame
- Run the application called `create`. This will create the header file, key file and the encrypted file
- Using IAR Embedded Workbench®, configure and build the bootloader for the target AVR
- Download bootloader to target AVR and set lock and fuse bits
- Now the encrypted firmware may be downloaded to the AVR at any time

## 4.3 Configuration file

The configuration file contains a list of parameters, which are used to configure the project. The parameters are described in Table 4-1.

**Table 4-1.** Summary of configuration file options.

| Parameter | Description | Default | Required |
|---|---|---|---|
| PAGE_SIZE | Size of AVR Flash page in decimal bytes. This parameter is part dependent. Please see datasheet. | N/A | Yes |
| KEY1 | First part (128-bit) of encryption key in hex. Should be 16 random bytes, with odd-parity bits inserted after every 8$^{th}$ bit, making a total of 18 bytes. | None: No encryption | No, but strongly recommended |
| KEY2 | Second part (64-bit) of encryption key in hex. Should be eight random bytes, with odd-parity bits inserted after every 8$^{th}$ bit, making a total of nine bytes. If omitted, AES128 will be used. | None: Use AES128 | No, but recommended |
| KEY3 | Third part (64-bit) of encryption key in hex. Should be nine random bytes, with odd-parity bits inserted after every 8$^{th}$ bit, making a total of nine bytes. If omitted AES128 or AES192 will be used. | None: Use AES128 or AES192 | No, but recommended |
| INITIAL_VECTOR | Used for chaining cipher blocks. Should be 16 random bytes in hex. | 0 | No, but strongly recommended |
| SIGNATURE | Frame validation data in hex. This can be any four bytes, but it is recommended that the values be chosen at random. | 0 | No |

| Parameter | Description | Default | Required |
|---|---|---|---|
| ENABLE_CRC | Enable CRC checking: YES or NO. If enabled, the whole application section will be overwritten and the application must pass a CRC check before it is allowed to start. | No | No, but recommended |
| MEM_SIZE | Size of application section in target AVR (in decimal bytes). | N/A | Yes, if CRC is used |

The configuration file can be given any valid file name. The name is later given as a parameter to the application that will create the project files. Below is a sample configuration file for Atmel ATmega16. The KEY1 parameter is an example 128-bit key (hex 0123456789ABCDEF0123456789ABCDEF) with parity bits inserted.

```
PAGE_SIZE         = 128
KEY1              = 0111914CE8955B35DE0111914CE8955B35DE
INITIAL_VECTOR    = 00112233445566778899AABBCCDDEEFF
SIGNATURE         = 89ABCDEF
ENABLE_CRC        = YES
MEM_SIZE          = 14336
```

Some of the parameters cannot be set without specific knowledge of the target AVR. Table 4-2 summarizes the features of some present AVR microcontrollers with bootloader functionality. For devices not present in this table, please refer to the data sheet of the device.

**Table 4-2.** AVR feature summary.

| | M8 | M16 | M162 | M169 | M32 | M64 | M128 |
|---|---|---|---|---|---|---|---|
| Flash Size, bytes | 8192 | 16384 | 16384 | 16384 | 32768 | 65536 | 131072 |
| Flash Page size, bytes | 64 | 128 | 128 | 128 | 128 | 256 | 256 |
| Flash Pages | 128 | 128 | 128 | 128 | 256 | 256 | 512 |
| BLS (max), bytes | 2048 | 2048 | 2048 | 2048 | 4096 | 8192 | 8192 |
| BLS Pages | 32 | 16 | 16 | 16 | 32 | 32 | 32 |
| MEM_SIZE, bytes | 6144 | 14336 | 14336 | 14336 | 28672 | 57344 | 122880 |
| PAGE_SIZE, bytes | 64 | 128 | 128 | 128 | 128 | 256 | 256 |

## 4.4 PC application – GenTemp

This application generates a template for the configuration file. The application generates random encryption keys and initial vectors, leaving other parameters for the user to be filled in (such as the Flash page size). It is recommended to always start with creating a template using this application.

The application is used as follows:

```
gentemp FileName.Ext
```

`FileName.Ext` is the name of the configuration file to be created. After the file has been generated it can be edited using any plain text editor.

## 4.5 PC application – create

This application reads information from the configuration file and generates key and header files for the bootloader. It is also used for encrypting the firmware. Typically, the application is run at least twice: (1) To generate key and header files for the bootloader and (2) when new firmware is encrypted.

NOTE

It is very important that the same encryption information (configuration file) is used when generating project files and when encoding the firmware. Otherwise, the bootloader may not have the correct set of encryption keys and cannot decrypt the data. It should also be noted that it is possible to use the information in the configuration file to decrypt the encrypted firmware. Hence, the configuration file must be kept safe at all times and should not be modified after it has been used for the first time.

### 4.5.1 Command line arguments

Table 4-3 shows the available command line arguments.

**Table 4-3.** Summary of command line arguments.

| Argument | Description |
|---|---|
| -c <filename.ext> | Path to configuration file. |
| -d | If set, contents of each Flash page is deleted before writing. Else, previous data will be preserved if not specifically written to. |
| -e <filename.ext> | Path to EEPROM file (data that goes into EEPROM). |
| -f <filename.ext> | Path to Flash file (code that goes into Application Section). |
| -h <filename.ext> | Name of output header file. This file is later included in the bootloader. |
| -k <filename.ext> | Name of output key file. This file is later included in the bootloader. |
| -l [BLB12] [BLB11] [BLB02] [BLB01] | Lock bits to set. These lock bits are set after all data has been transferred and before control is transferred to the updated application. |
| -n | Nonsense. Add random number of nonsense records to encrypted file. As nonsense records are ignored by the bootloader, this setting does not affect the application, only the predictability of the output file. |
| -o <filename.ext> | Output file name. This is the encrypted file that may be distributed and sent to the target when it needs to be updated. |

### 4.5.2 First run

In the first run, typically, only key and header files for the bootloader are generated. The generation of key and header files is requested using command line arguments. For example:

```
create -c Config.txt -h BootLdr.h -k AESKeys.inc
```

The key and header files must be copied to the project directory of the bootloader application and be included into the bootloader code.

NOTE

The bootloader project files are preconfigured to use the file names mentioned above, that is, BootLdr.h and AESKeys.inc. It is recommended these file names are not changed.

### 4.5.3 Subsequent runs

In subsequent runs, the application is used for encoding the firmware. Prior to encryption, the source file must be compiled, assembled and linked into one code segment file and/or one EEPROM segment file. Files must be of type Intel® hex.

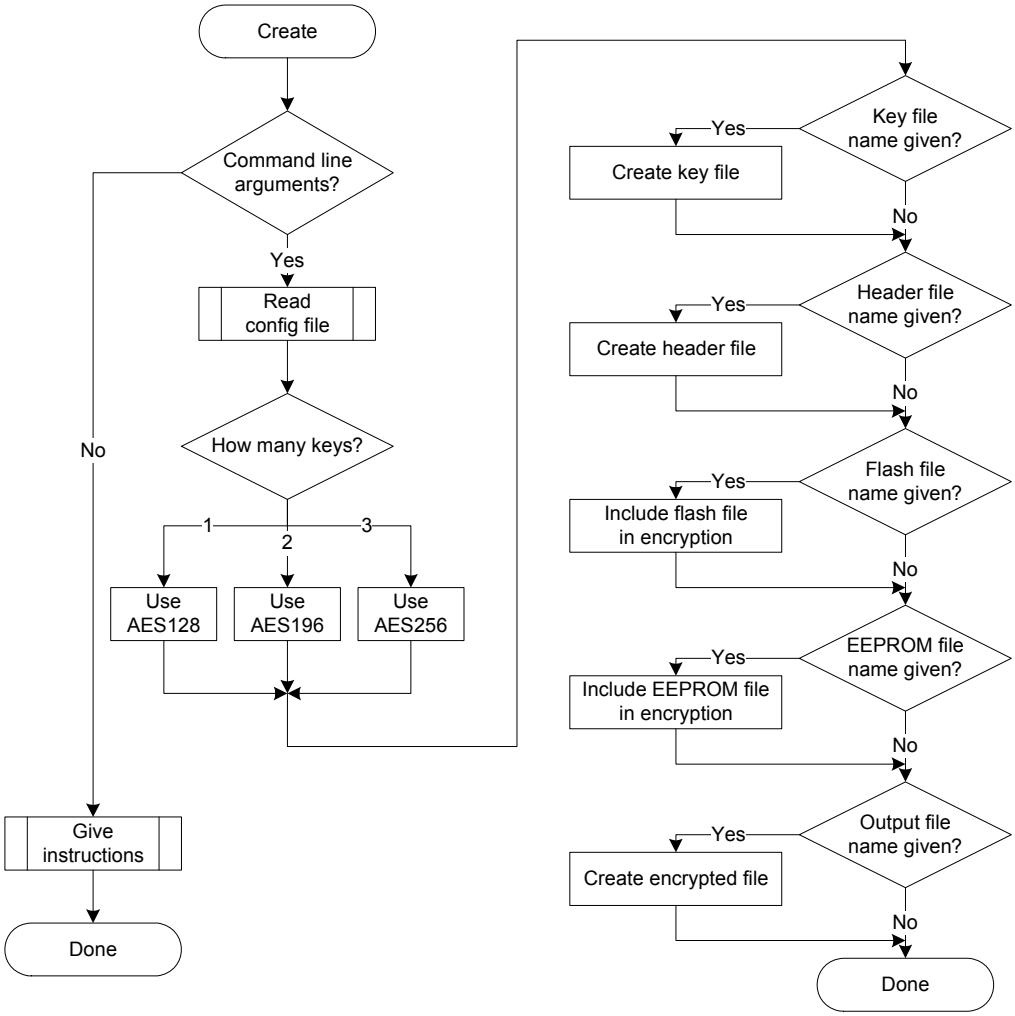
A file name is given at the command prompt and an encrypted file will be generated according to data in the configuration file. For example:

```
create -c Config.txt -e EEPROM.hex -f Flash.hex -o Update.enc -l
BLB11 BLB12
```

The application software and EEPROM data files will be combined into a single encrypted file.

### 4.5.4 Program flow

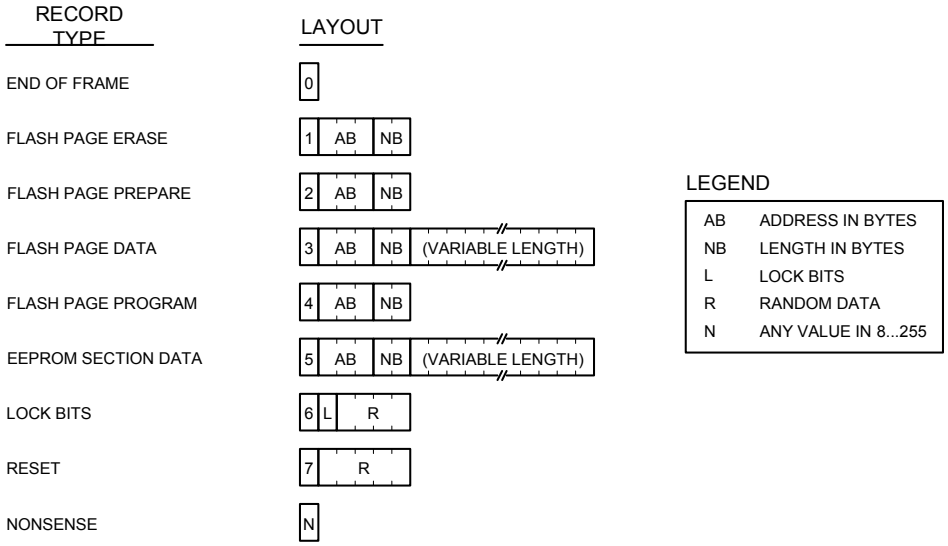
**Figure 4-2.** Flowchart for the `create` application.




### 4.5.5 The encrypted file

The Flash and EEPROM files are encrypted and stored in one target file. Before encryption, however, data is organized into records. There are seven types of records, as illustrated in Figure 4-3.

**Figure 4-3.** Record types for encrypted file.

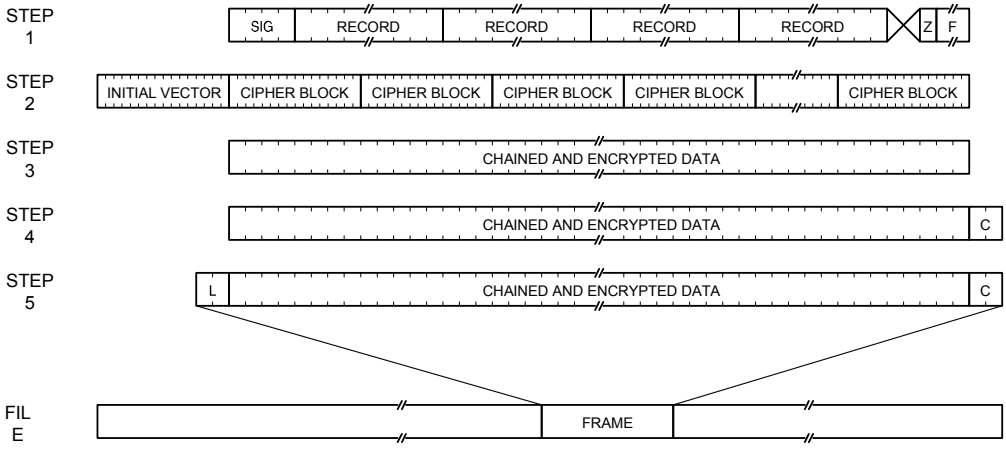



The record type is given as the first byte in the record. The application data is broken down to record types 1, 2, 3, and 4 (that is, erase, prepare, load and write buffer page to Flash). The data for the EEPROM section is formatted into record type 5. Lock bits are sent in record type 6. Record types 0 and 7 are for ending a frame and transmission, respectively.

All other records, that is, those with a record identifier above 7, are of type nonsense. When this option is enabled (see `create` tool), a random number of nonsense records will be placed at random locations in the file.

The output file is created as illustrated in Figure 4-4.

**Figure 4-4.** Creating the encrypted file.




The steps are described below (numbers refer to Figure 4-4):

1. Data is formatted into records, which are then lined up following the frame signature (SIG). A zero (Z) is added to mark the end of the frame and the frame is padded with random data (F) to create a frame size that is a multiple of 16 bytes.

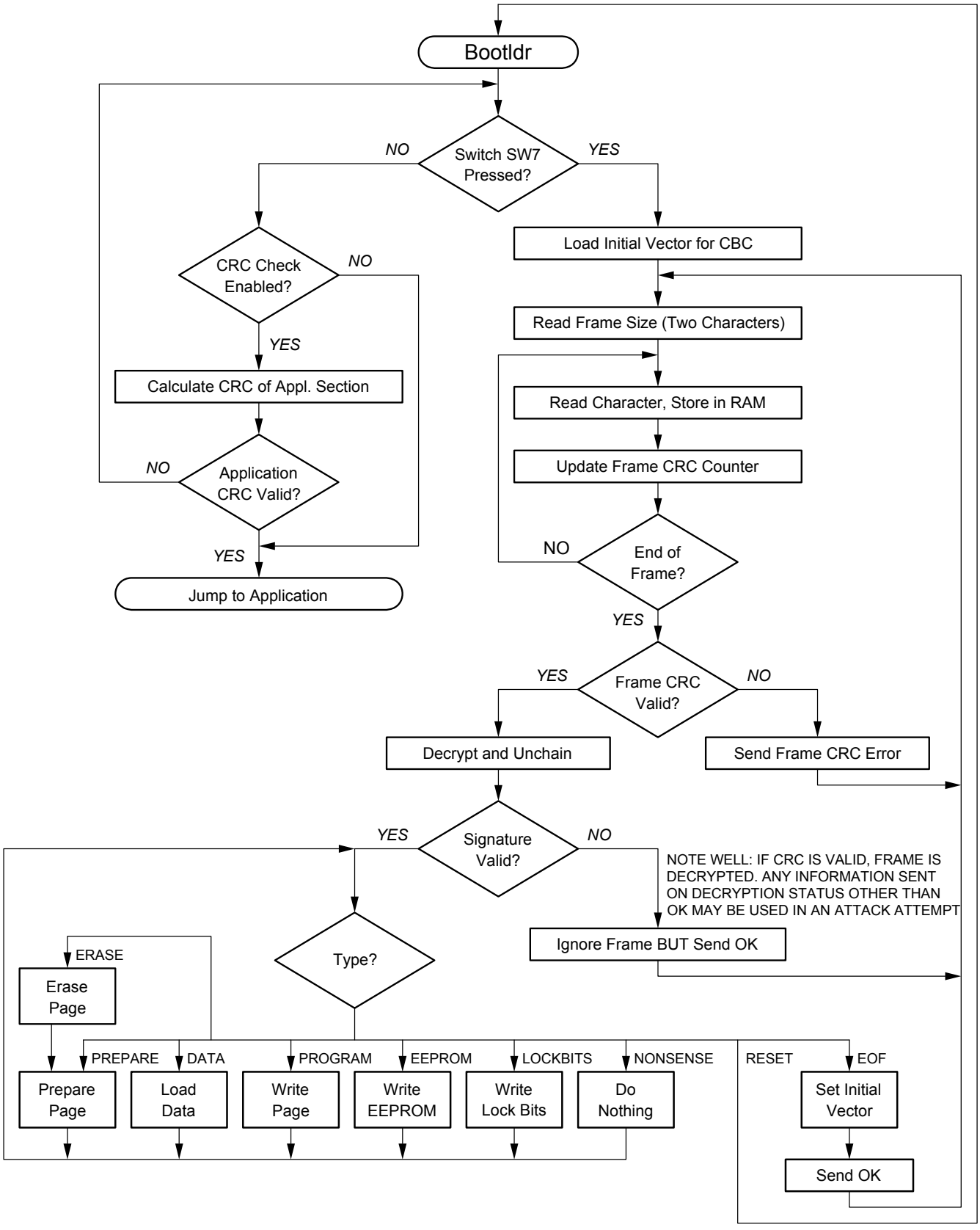
2. The initial vector is attached to the frame. In the first frame, the vector is equal to the one given in the configuration file. In subsequent frames, the initial vector is equal to the last cipher block of the previous frame.
3. The initial vector and cipher blocks are chained and encrypted. The initial vector is then removed from the frame.
4. A CRC-16 checksum (C) is calculated and added to the frame.
5. The length (L) of the frame, excluding the length information, is calculated and saved at the start of the frame.

The frame is written to the output file and the procedure is repeated until all data has been processed.

## 4.6 AVR bootloader

The bootloader must reside in the target AVR before the device can be updated with encrypted firmware. The bootloader communicates with the PC and is capable of programming the EEPROM and the application area of the Flash memory. The bootloader included with this application note has been created using IAR™ Embedded Workbench, version 3.20c, but it can be ported to other C compilers. The program flow is illustrated in the figure below.

**Figure 4-5.** Flowchart for the AVR bootloader.



### 4.6.1 Key and header files

Before the bootloader can be compiled, there are some parameters that need to be set up. To start with, the encryption key and target header files generated by the PC application `create` must be copied to the directory of the bootloader. The files will be included when they are referred to with the `#include` directive inside the bootloader source code.

### 4.6.2 Project files

The application note comes with device specific project files for the following devices:

- Atmel ATmega8
- Atmel ATmega16
- Atmel ATmega162
- Atmel ATmega169
- Atmel ATmega32
- Atmel ATmega64
- Atmel ATmega128

Use the predefined project files with the corresponding AVR. For AVR devices not listed, use the project file for a device that matches the target device as close as possible and modify as described in sections 4.6.3 and 4.6.4.

### 4.6.3 Linker file

The IAR compiler requires a modified linker file because the bootloader will reside in the upper memory area, that is, in the Boot Loader Section (BLS). Linker files have an extension of .xcl and are distributed with the IAR compiler for each AVR device separately. This application note comes with the modified linker file `bootldr.xcl`.

The linker file is defined under “Project” – “Options”, category “XLINK”, tab “Include”, field “XCL file name”.

NOTE

The linker file is already set up in the device specific project files that come with this application note.

### 4.6.4 Other compiler settings

The following settings need to be defined in the dialog window found under “Project” – “Options”. Note that all settings are already defined in the device specific project files.

**Table 4-4.** Required compiler settings.

| Category | Tab | Set to | Example |
|---|---|---|---|
| General | Target | Set “Processor configuration” to match target AVR | -cpu=m8, AT90mega8 |
| | | Set “Memory model” to Small | |
| | | Uncheck “Configure system using dialogs (not in .XCL file)” | |
| | Library configuration | Check “Enable bit definitions in I/O-include files” | |
| AAVR | Preprocessor | Define symbol “INCLUDE_FILE” to match target AVR | INCLUDE_FILE=”iom8.h” |
| | | Define symbol SPMREG to match target | SPMREG=SPMCR |
| | | Define symbol __RAMPZ__ for devices with more than 64kB Flash memory | |

| Category | Tab | Set to | Example |
|---|---|---|---|
| | | Define symbol __MEMSPM__ for devices with SPM control register located above address 0x3F | |
| XLINK | Output | Define output file format such that the file can programmed into the target. Set to Intel-Extended | |
| | #define | Define symbol BOOT_SIZE to match target boot loader section. In hex bytes | BOOT_SIZE=800 |
| | | Define symbol FLASH_SIZE to match Flash size of target. In hex bytes | FLASH_SIZE=2000 |
| | | Define symbol IVT_SIZE to match size of target interrupt vector table. In hex bytes | IVT_SIZE=26 |
| | | Define symbol RAM_SIZE to match amount of RAM on target. In hex bytes | RAM_SIZE=400 |
| | | Define symbol RAM_BASE to match start of SRAM (following I/O area). In hex bytes | RAM_BASE=60 |
| | | Define symbol APP_SRAM_USED to match the total SRAM usage reported from the compile process. In hex bytes | APP_SRAM_USAGE=30A |
| | Includes | Under section “XCL file name”, check “Override default” | |
| | | Under section “XCL file name”, enter file name in box | $PROJ_DIR$\bootldr.xcl |

Table 4-5 summarizes some of the compiler options for currently supported AVR devices. Please note that bootloader start address depends on fuse settings, as explained later (see Table 4-6).

**Table 4-5.** Compiler setting reference.

| | M8 | M16 | M162 | M169 | M32 | M64 | M128 |
|---|---|---|---|---|---|---|---|
| Linker file name | bootldr.xcl | | | | | | |
| BOOT_SIZE | 800 | 800 | 800 | 800 | 800 | 800 | 1000 |
| FLASH_SIZE | 2000 | 4000 | 4000 | 4000 | 8000 | 10000 | 20000 |
| IVT_SIZE | 26 | 54 | 70 | 5C | 58 | 8C | 8C |
| RAM_SIZE | 400 | 400 | 400 | 400 | 800 | 1000 | 1000 |
| RAM_BASE | 60 | 60 | 100 | 100 | 60 | 100 | 100 |
| APP_SRAM_USAGE | 30A | 31E | 31E | 31E | 31E | 41E | 41E |
| SPMREG | SPMCR | SPMCR | SPMCR | SPMCSR | SPMCR | SPMCR | SPMCSR |
| __RAMPZ__ | | | | | | | X |
| __MEMSPM__ | | | | | | X | X |

Note: The symbols __RAMPZ__ and __MEMSPM__ should not be set equal to anything. If required, they should merely be included in the defined symbols list.

### 4.6.5 Installing the bootloader

Compile the boot loader and then download it to the target using Atmel AVR Studio®. Before running the boot loader, the following fuse bits must be configured:

- Size of Boot Loader Section. Set fuse bits so that the section size matches the BOOT_SIZE setting, as described earlier. Note that the BLS is usually given in words, but the BOOT_SIZE parameter is given in bytes

- Boot reset vector. The boot reset vector must be enabled
- Oscillator options. The oscillator fuse bits are device dependent. They may require configuration (affects USART)

NOTE

Please pay special attention in setting oscillator options correctly. Even small misadjustments could result in communication failure.

Table 4-6 lists the recommended fuse bit settings. See datasheet for detailed explanation of device dependent fuse bits.

**Table 4-6.** Recommended fuse bits.

| | M8, M8515, M8535, M16, M162, M169, M32, M64 | M128 |
|---|---|---|
| BOOTSZ1:0 | 0:0 | 0:1 |
| BOOTRST | 0 | 0 |

Note: "0" means programmed, "1" means not programmed.

It is recommended to program lock bits to protect both application memory and the bootloader, but only after fuse bits have been set. Lock bits can be programmed using AVR Studio. BLS lock bits will also be set during firmware update, provided that they have been defined as command line arguments when the firmware is encrypted. The recommended lock bit settings are:

- Memory lock bits: These should be set to prevent unauthorized access to memory. Note that after the memory has been locked it cannot be accessed via in-system programming without erasing the device.
- Protection mode for Boot Loader Section: SPM and LPM should not be allowed to write to or read from the BLS. This will prevent the firmware in the application section to corrupt the bootloader and will keep the decryption keys safe.
- Protection mode for application section: No restrictions should be set for SPM or LPM accessing the application section; otherwise the bootloader cannot program it.

NOTE

It is important to understand that if the device is not properly locked then memory can be accessed via an ISP interface and the whole point of encrypting the firmware is gone.

Table 4-7 lists the recommended lock bit setting for present AVR microcontrollers. See data sheet for detailed explanation of lock bits.

**Table 4-7.** Recommended lock bits.

| | M8, M8515, M8535, M16, M162, M169, M32, M64, M128 |
|---|---|
| BLB12 : BLB11 | 0 0 |
| BLB02 : BLB01 | 1 1 |
| LB2 : LB1 | 0 0 |

## 4.7 PC application – update

This application is used for sending the encrypted file to the target. The data can be sent via a serial port on the PC directly to the USART on the target hardware. The program flow is illustrated in Figure 4-6.

The Update application reads in files generated with the Create application. The file consists of one or more concatenated frames of encrypted data. The application transmits data one frame at a time, pausing in between to wait for a reply from the

bootloader. The next frame is transmitted only after an acknowledgement has been received; otherwise the application will either resend the frame or close communication.

The update application is run from the command prompt. The command prompt arguments are listed in Table 4-8.

**Table 4-8.** Command line arguments for the `update` application.

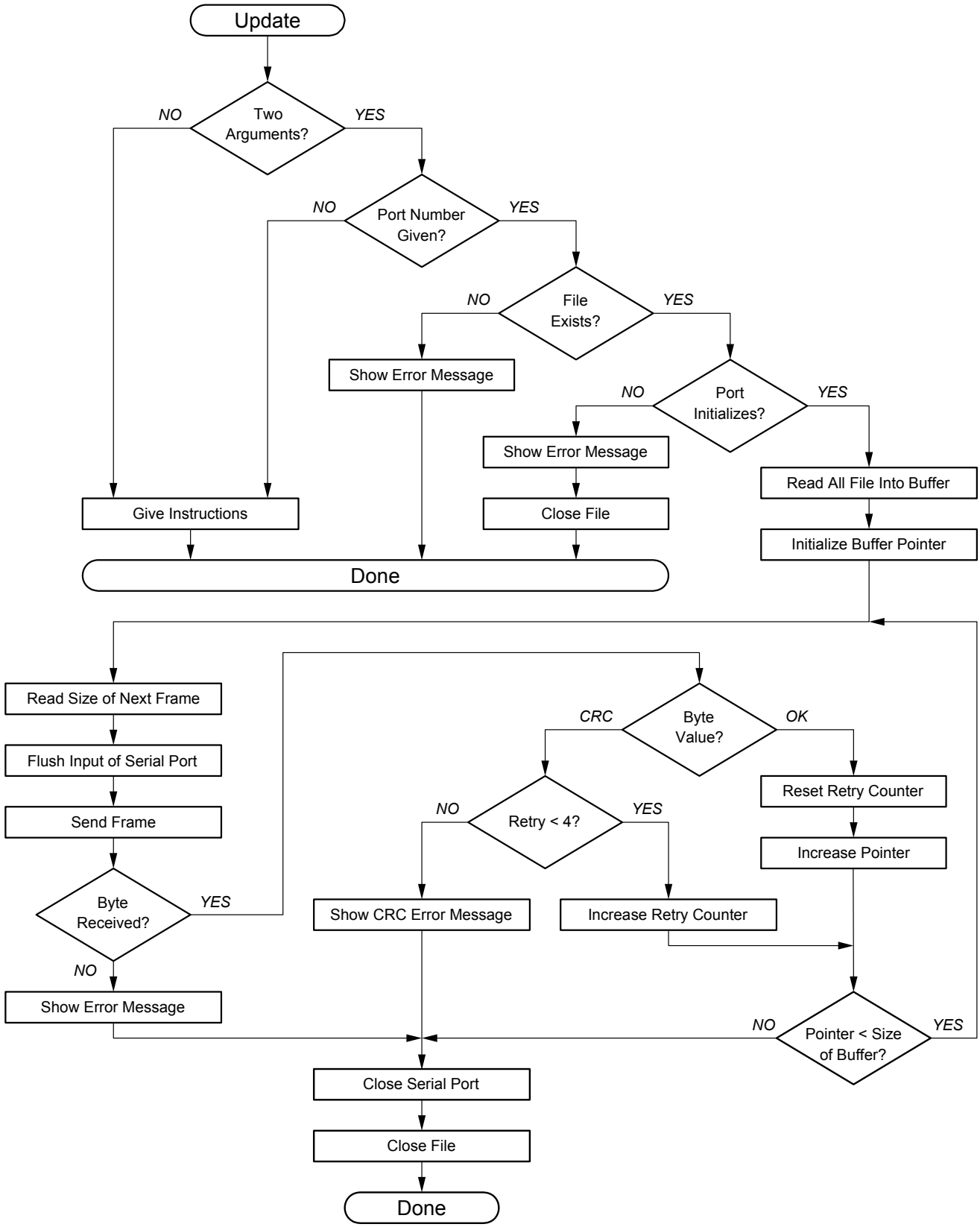
| Argument | Description |
|---|---|
| <filename.ext> | Path to encrypted file to be transferred |
| -COM*n* | Serial port, where *n* is the serial port number |
| -*baudrate* | Baudrate, where *baudrate* is the actual baudrate figure |

For example:

```
update blinky.ext -COM1 -115200
```

It should be noted that the update system only updates those parts of the Flash and EEPROM denoted in the application and EEPROM files. If CRC check of the application section is enabled, or the erase option is selected in the `create` tool, all application memory will be cleared before programming.

**Figure 4-6.** Flowchart for the `update` application.



## 4.8 Hardware setup

The target hardware must be properly set up before the encrypted firmware can be sent to the bootloader. In this application note, it is assumed an Atmel STK®500 is used as the target platform, using an external 3.69MHz crystal oscillator. The STK500 should be configured as follows:

- Connect the STK500 (via connector labeled “RS232 CTRL”) to the PC using a serial cable. Power on the STK500
- Use Atmel AVR Studio to download the boot loader and set fuse and lock bits, as described earlier. Power off the STK500
- Move the serial cable to the connector labeled “RS232 SPARE”
- Connect the device USART pins RXD and TXD to the corresponding pins on the connector labeled “RS232 SPARE”
- Connect PD7 (pin 8 of PORTD) to SW7 (pin 8 of SWITCHES)
- Press and hold down SW7 while switching on the STK500. This will start the boot loader and set it in update mode
- Release switch SW7
- The `update` application on the PC can now be used to send encrypted data to the target

## 4.9 Performance

Sections 4.9.1 and 4.9.2 summarize system performance with respect to execution time and code size.

### 4.9.1 Execution Time

The time required for the target device to receive, decode, and program data depends on the following factors:

- File size. The more data, the longer it takes.
- Baudrate. The higher the transmission speed, the shorter the transmission time.
- Target AVR speed. The higher the clock frequency, the shorter the decoding time.
- Programming time of Flash page. This is a device constant and cannot be altered.
- Keysize. AES128 is faster to decrypt than AES256. In fact, AES192 is slower than AES256. It has something to do with 192 not being a power of 2.
- Miscellaneous settings. For example, CRC check of application section takes a short while.

### 4.9.2 Code size

Using the highest optimization setting for the compiler, the bootloader will fit nicely into 2kB of Flash memory.

It should be noted that if no encryption keys are given, the bootloader is built without AES support. This application note then performs as a standard bootloader system and can be used on any AVR with boot loader support.

## 5 Summary

This application note has presented a method for transferring data securely to an Atmel AVR microcontroller with bootloader capabilities. This document has also highlighted techniques that should be implemented when building a secured system. The following issues should be considered in order to increase the security of an AVR design.

Implement a bootloader that supports downloading in encrypted form. When the bootloader is first installed (during manufacturing) it must be equipped with decryption keys, required for future firmware updates. The firmware can then be distributed in an encrypted form, securing the contents from outsiders.

Use AVR lock bits to secure Application and Boot Loader sections. When lock bits are set to prevent reading from the device, the memory contents cannot be retrieved. If lock bits are not set, there is no use encrypting the firmware.

Encrypt the firmware before distribution. Encrypted software is worthless to any outside entity without the proper decryption keys.

Keep encryption keys safe. Encryption keys should be stored in two places only: in the bootloader, which has been secured by lock bits, and in the firmware development bench at the manufacturer.

Chain encrypt data. When data is chained, each encrypted block depends on the previous block. As a consequence, equal plaintext blocks produce different encrypted outputs.

Avoid standard, predictable patterns in the firmware. Most programs have a common framework and any predictable patterns, such as an interrupt vector table starting with a jump to a low address, only serve to help the intruder. Also avoid padding unused memory areas with a constant number.

Hide the method. There is no need to mention which algorithm is being used or what the key length is. The less the intruder knows about the system, the better. It may be argued that knowing the encryption method fends off some attackers, but knowing nothing about the method increases the effort and may fend off even more.

The bootloader may also be used to erase the application section, if required. Many attack attempts include removing the device from its normal working environment and powering it up in a hacking bench. Detecting, for example, that an LCD is missing or that there are CRC errors in memory, the bootloader may initiate a complete erase of all memory (including the bootloader section and decryption keys).

In applications where it is not feasible or possible to use an external communications channel for updates, the firmware can be stored in one of the Atmel CryptoMemory® devices. The memory can be packaged as a removable smart card, which can easily be inserted in a slot of the device when an upgrade is needed. The microcontroller can check for the presence of a CryptoMemory upon startup and retrieve a firmware upgrade as needed.

Use secure hardware. A strong encryption protocol is useless if the hardware has structural flaws. There are no reported security issues with AVR microcontrollers.

This list can be made much longer but the purpose of it is merely to set the designer off in the right direction. Do not underestimate the wit or endurance of your opponent.

# 6 Literature references


- Menezes, Oorschot & Vanstone, Handbook of Applied Cryptography
- AES Specification, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
- Atmel Corporation, Application Note AVR230 DES Bootloader
- Brian Gladman, AES Implementation Example, http://gladman.plushost.co.uk/oldsite/cryptography_technology/rijndael/index.php

# 7 Table of contents

**Atmel Corporation**
2325 Orchard Parkway
San Jose, CA 95131
USA
**Tel:** (+1)(408) 441-0311
**Fax:** (+1)(408) 487-2600
www.atmel.com

**Atmel Asia Limited**
Unit 01-5 & 16, 19F
BEA Tower, Milennium 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**
16F, Shin Osaki Kangyo Bldg.
1-6-4 Osaki Shinagawa-ku
Tokyo 104-0032
JAPAN
**Tel:** (+81) 3-6417-0300
**Fax:** (+81) 3-6417-0370



2589E-AVR-03/12