EFM32
... the world's most energy friendly microcontrollers

AES Cipher Modes with EFM32

AN0033 - Application Note

Introduction

This application note describes how to implement several cryptographic cipher modes with the Advanced Encryption Standard (AES) on the EFM32 microcontrollers using the built-in AES hardware accelerator. Examples are given for interrupt and DMA driven operation.

This application note includes:

- This PDF document
- Source files (zip)
  - Example C-code
  - Multiple IDE projects

ARM ZERO TINY GECKO LEOPARD GIANT WONDER

SILICON LABS

1 Introduction

The Advanced Encryption Standard (AES) is a symmetric key encryption standard (NIST standard FIPS-197) adopted worldwide. It comprises three block ciphers, AES-128 (128 bit key), AES-192 (192 bit key) and AES-256 (256-bit key), adopted from a larger collection originally published by the Belgian cryptographers Joan Daemen and Vincent Rijmen under the name Rijndael.

The standard has been extensively investigated by cryptanalysts who have not found any potential weakness and AES is today one of the most used symmetric cryptographic algorithms.

AES was introduced as a replacement to its 30 year old predecessor DES, which uses an effective key length of 56 bits only, which can be easily broken by brute force attacks with specialized hardware nowadays. In comparison, even 128-bit long keys (not to mention 192- and 256-bit long keys) cannot be broken or "guessed" within a reasonable time frame with todays hardware, and longer keys are currently used only as a precaution in case algorithm weaknesses will be found in the future.

2 The AES Algorithm

The AES algorithm is hardware friendly, requiring few resources, and fast. It is a block cipher, meaning that it operates on data blocks of fixed length of 128 bits (16 bytes). The input block is called PlainText and the output after an encryption is called CipherText (also 128 bits).The exact transformation is controlled using a second input, the secret key. The security of AES only relies on keeping the key secret, whereas the algorithm itself is fully public. Decryption is similar to encryption; the decryption algorithm receives as input a 128-bit block of CipherText together with the key, and produces the original 128-bit block of PlainText.

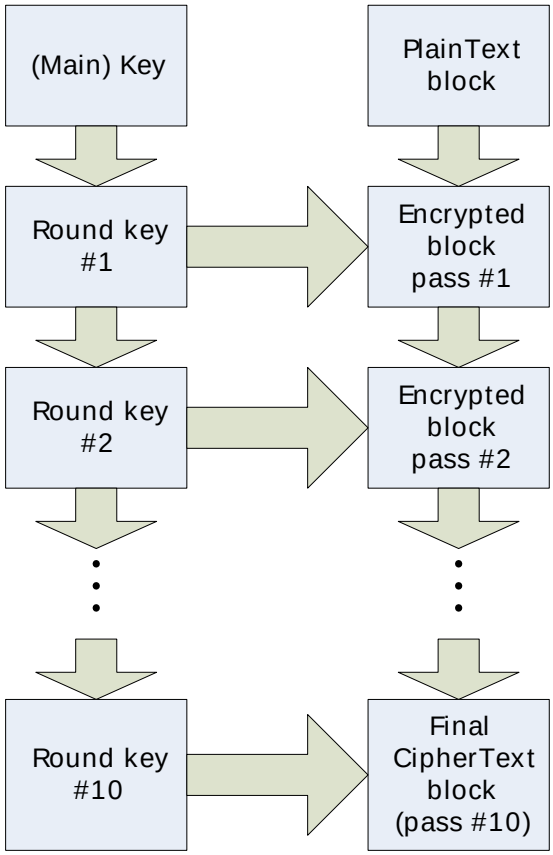
When encrypting or decrypting data which has a length that is not a multiple of 128 bits, the last block must be padded with zeros or other length preserving methods.

The principle of the AES algorithm is that of a product cipher and can be summarized as follows:

- It is a multi-step encryption algorithm, where the initial non-encrypted block is gradually encrypted, such as the input data to be encrypted at each step is the output of the previous step
- The algorithm steps are called rounds. The AES algorithm has a number of 10 (for a 128-bit key), 12 (for a 192-bit key), or 14 rounds (for a 256-bit key)
- Each round uses a different key (thus the term of round key). The 10/12/14 round keys are calculated from the original key prior to executing the actual encoding rounds
- The actual operations performed at in each round are simple byte wise transformations, such as XOR, byte substitutions or byte permutations

Figure 2.1 (p. 3) is descriptive of the general structure of the algorithm in case of a 10-round (128-bit long key) operation:

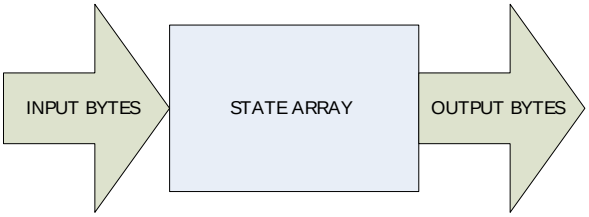
Figure 2.1. AES Rounds



The input bytes (PlainText or CipherText) are organized in groups of four bytes resulting 32-bit words. The words are placed in the matrix in the first column. After the first row is completed, the next column is initiated and the process continues until the input array is depleted.

The input matrix is transferred to the State matrix (Figure 2.2 (p. 4)). The four bytes in each column of the State array form 32-bit words where the row number provides an index for the bytes within each word, thus the State array can be interpreted as a one-dimensional array of 32-bit words.

Figure 2.2. AES State Array



Based on the key length (128-bit, 192-bit or 256-bit) we obtain the key block rounds (10, 12, 14 rounds).

After an initial round key addition, the state array is transformed by implementing a round function (10, 12, 14 rounds). The cipher process uses the following functions:

- Sub Bytes - a non-linear substitution function that operates independently on each byte of the State, using a substitution table (S-box)
- Shift Rows - the bytes in the last three rows of the state array are cyclically shifted. The row number gives the shift number and the first row is not shifted
- Mix Columns - operates on the state column-by-column, treating each column as a four-term polynomial
- Add Round Key - a round key is added to the state using a XOR operation.

The Inverse cipher uses the same functions as the Cipher, but inverted. The order is : inverted shift rows, then inverted sub byte, inverted mixed columns, add round key. At the end of the state process an Output array (cipher text or PlainText) is obtained.

The differences between various modes of creating feedback datapaths from one encryption step to the next result in various variants of the AES algorithm, which will be detailed in what follows.

2.1 Cipher Modes

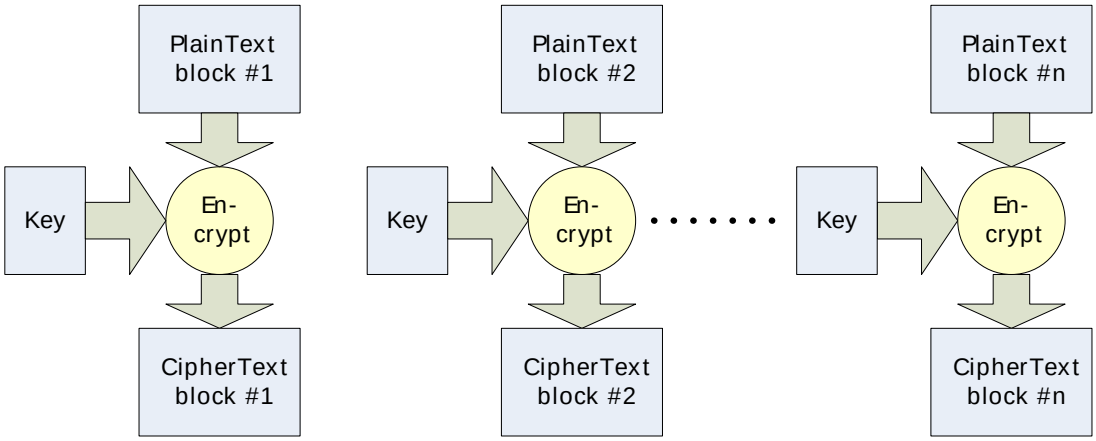
The most straight forward way of using AES is to encrypt your message one block at a time (using the same key) and using the output as it is. This approach has got some security limitations as equal blocks of PlainText will produce equal CipherText, revealing any patterns in the PlainText in the resulting CipherText. This is a general trait for all block ciphers and to mend this issue, several Cipher Modes have been introduced. In these modes each block of CipherText is dependent on parts of the earlier encryptions in addition to the key and PlainText block. Such algorithms then yield different CipherTexts for consecutive blocks of equal PlainText. The following sections introduce the five main cipher modes (ECB, CBC, CFB, OFB and CTR) as defined by NIST Special Publication 800-38a. This application note further explains how to implement these modes using the AES accelerator in the EFM32 microcontrollers.

2.1.1 Electronic Codebook Mode (ECB)

The ECB is by far the simplest, and most vulnerable, AES encryption type, as there are no connections whatsoever between encryption steps. In other words, each plain or CipherText block is encrypted and

decrypted individually with the key value, i.e. there is no feedback input from the result of previous encryption steps. (Figure 2.3 (p. 5)).

Figure 2.3. Electronic Codebook Mode (ECB)



The decryption obviously consists of the inverse transform of the one used during encryption, in each case.

The disadvantage of this method is that identical PlainText blocks are encrypted into identical CipherText blocks; thus, it does not hide data patterns. In some sense, it doesn't provide serious message confidentiality, and it is not recommended for use in cryptographic protocols at all. The tendency to preserve data patterns is a large disadvantage, as all other AES modes result in a certain degree of "pseudo-randomness". This has highly offset the advantage of the implementation being fit for parallel processing.

The ECB mode can also make protocols without integrity protection even more susceptible to replay type of attacks, since each block is decrypted in exactly the same way.

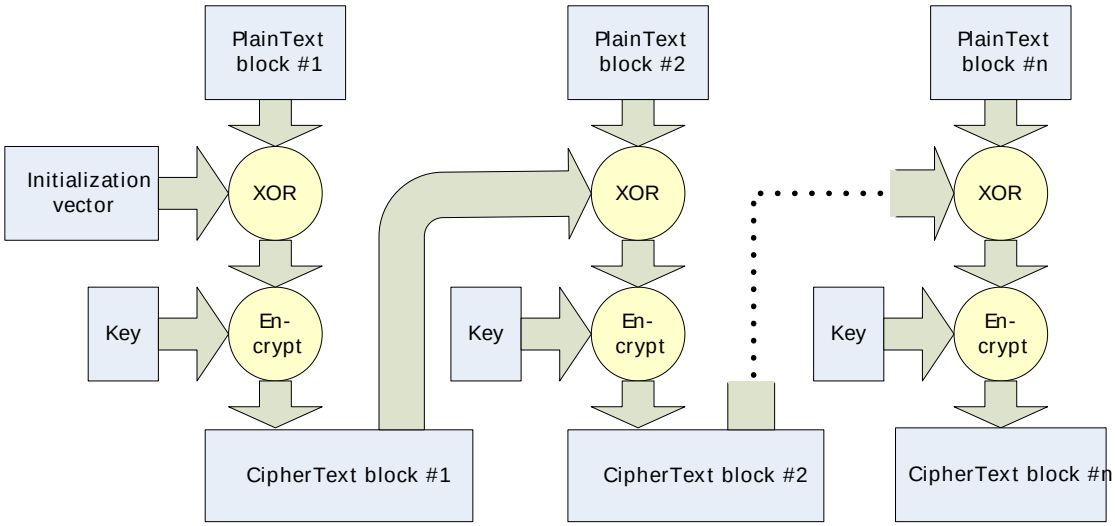
Implementation in embedded environments, albeit nowadays useless in real life encryption applications, has been achieved in the past with both 128-bit and 256-bit wide keys.

2.1.2 Cipher-block Chaining Mode (CBC)

The CBC mode of operation was introduced by IBM in 1976. In the cipher-block chaining (CBC) mode, each block of PlainText is XOR'ed with the previous CipherText block before being encrypted. This way, each CipherText block is dependent on all PlainText blocks processed up to the current point (Figure 2.4 (p. 6)).

At the first step, since there is no prior CipherText block to XOR with, an initialization vector (generally a pseudo-random number) is used in order to make each message unique. The decryption algorithm must know two constants prior to performing decryption: the key and the initialization vector.

Figure 2.4. Cipher-block Chaining Mode (CBC)



The formulas describing the algorithm are the following:

Encryption CBC

$$C_i = E_k(P_i \text{ (xor) } C_{i-1}),\ C_0 = IV \qquad (2.1)$$

while the mathematical formula for CBC decryption is

Decryption CBC

$$P_i = D_k(C_i) \text{ (xor) } C_{i-1},\ C_0 = IV \qquad (2.2)$$

where IV is the initialization vector and E_k, D_k are the encryption and decryption transformations respectively.

CBC is the most commonly used mode of operation. It completely hides information about PlainText (with exception of the length), even the fact that two cipher texts are a result of the same PlainText, on the condition that different initialization vectors are used for different encrypted messages.

Its main drawbacks are that encryption is sequential (i.e., it cannot be parallelized), and that the message must be padded to a multiple of the cipher block size. One way to handle this last issue is through the method known as CipherText stealing.

CipherText stealing is the technique of altering processing of the last two blocks of PlainText, resulting in a reordered transmission of the last two blocks of CipherText and no CipherText expansion. This is accomplished by padding the last PlainText block (which is possibly incomplete) with the high order bits from the second to last CipherText block (stealing the CipherText from the second to last block). The (now full) last block is encrypted, and then exchanged with the second to last CipherText block, which is then truncated to the length of the final PlainText block, removing the bits that were stolen, resulting in CipherText of the same length as the original message size.

It can be seen from the above that in all cases the processing of all but the last two blocks is unchanged.

For simplicity reasons, all code examples presented with the current application note will assume a PlainText message length of an exact multiple of the key length.

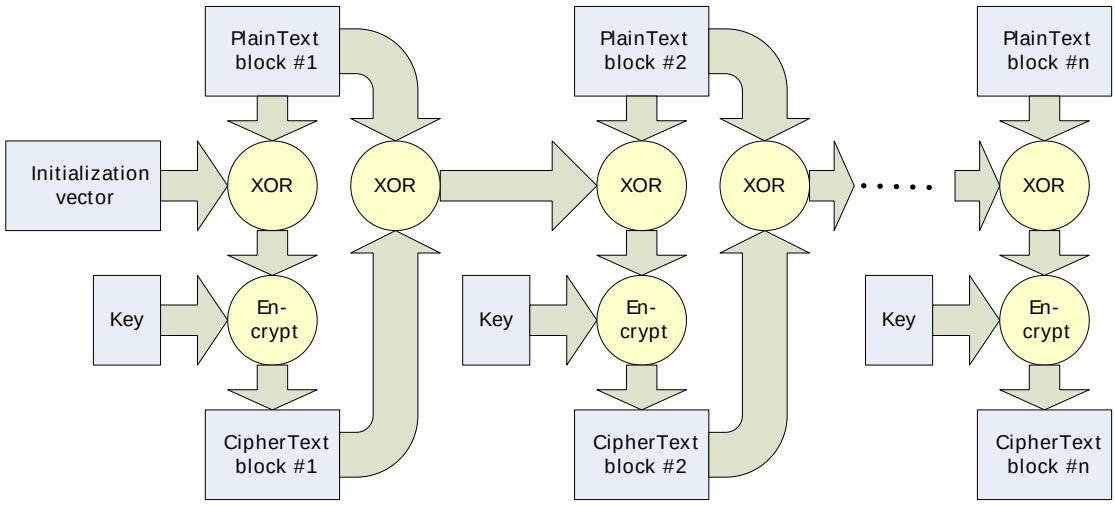
When using CBC, a one-bit change in a PlainText block affects all following CipherText blocks. A PlainText can be recovered from just two adjacent blocks of CipherText. As a consequence, decryption

can be parallelized, and a one-bit change to the CipherText causes complete corruption of the corresponding block of PlainText, and inverts the corresponding bit in the following block of PlainText.

As a consequence, the Propagating Cipher-Block Chaining (PCBC) has been adopted as a more advanced variant of the CBC, which causes small changes in the CipherText to propagate indefinitely when decrypting, as well as when encrypting.

The PCBC is extremely similar to the CBC, with the sole difference being that, at each step, the current PlainText block is XOR'ed not with the former step's CipherText block, but with the result of another XOR, between the PlainText and the CipherText blocks of the previous step (Figure 2.5 (p. 7)).

Figure 2.5. Propagating Cipher Block Chaining Mode (PCBC)



The encryption and decryption algorithms can be represented as follows:

Encryption PCBC

$$C_i = E_k(P_i \text{ (xor) } P_{i-1} \text{ (xor) } C_{i-1}),\ P_0 \text{ (xor) } C_0 = IV \quad (2.3)$$

Decryption PCBC

$$C_i = D_k(C_i) \text{ (xor) } P_{i-1} \text{ (xor) } C_{i-1},\ P_0 \text{ (xor) } C_0 = IV \quad (2.4)$$

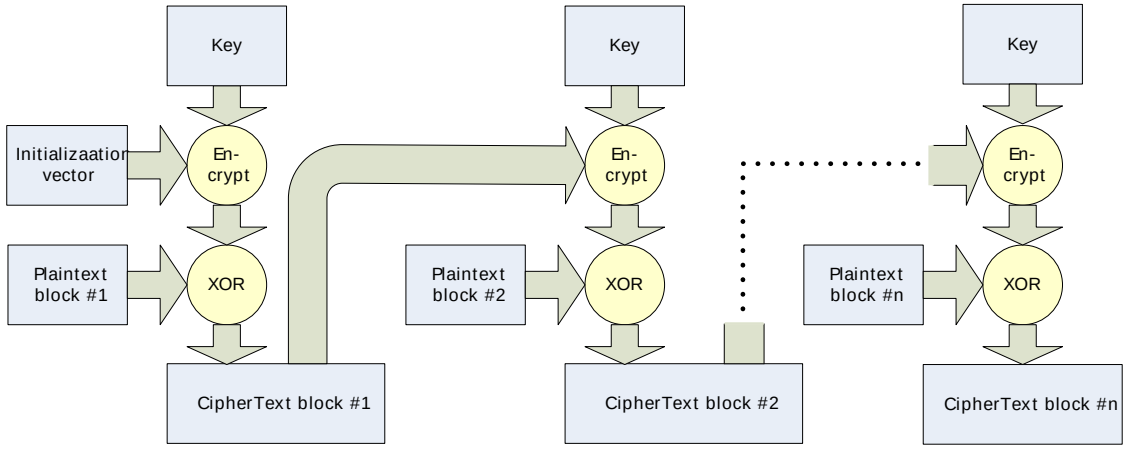
PCBC is used in several computer network authentication protocols, but otherwise is not common. The operations are also sequential and in a message encrypted in PCBC mode, if two adjacent CipherText blocks are exchanged, this does not affect the decryption of subsequent blocks. This can be a weakness in the algorithm for some applications.

2.1.3 Cipher feedback Mode (CFB)

In cryptography, a stream cipher is a symmetric key cipher where PlainText bits are combined with a pseudo random cipher bit stream (key stream), typically by a XOR operation. In a stream cipher the PlainText digits are encrypted one at a time, and the transformation of successive digits varies during the encryption. The encryption of each digit is dependent on the current state. In practice, the "digits" are typically single bits or bytes.

The CFB method is a way of transforming a block cipher into a stream cipher. The way it works is at each step, it initially encrypts the CipherText block result from the previous step with the key, then the result of this operation is XOR'ed with the PlainText block at the current step, the result being the CipherText block of the current step. At the initial step, since there is no previous CipherText block to be encrypted with the key, an initialization vector is used (Figure 2.6 (p. 8)).

Figure 2.6. Cipher Feedback Mode (CFB)



The encryption algorithm can be summarized by the following equations:

CFB Encryption

$$C_i = E_k(C_{i-1} \text{ (xor) } P_i \quad (2.5)$$

CFB Decryption

$$P_i = E_k(C_{i-1} \text{ (xor) } C_i, C_0 \quad (2.6)$$

CFB decryption is almost identical to CBC encryption performed in reverse. The self-synchronizing capabilities of the two are comparable. If one block of CipherText is lost both CBC and CFB will eventually synchronize, but losing only a single byte or bit will corrupt all the decrypted data. To be able to synchronize after the loss of a single byte or bit, a single byte or bit must be encrypted at a time. CFB can be used this way when combined with a shift register as the input for the block cipher. The following method can be applied to use CFB to make a self-synchronizing stream cipher that will synchronize for any multiple of x bits lost.

Encryption:

- Start by initializing a shift register the size of the block size with the initialization vector
- Encrypt the above with the block cipher
- XOR the most significant x bits of the result with x bits of the PlainText to produce x CipherText bits
- Shift these x bits of output into the shift register
- Repeat the process with the next x bits of PlainText

Decryption:

- Start with the initialization vector
- Encrypt
- XOR the most significant x bits of the result with x bits of the CipherText to produce x PlainText bits
- Then shift the x bits of the CipherText into the shift register and repeat

The below equations summarize the algorithm, where S_i is the i-th state of the shift register, a << x is a shifted up x bits, head(a, x) is the x most significant bits of a and n is number of bits of IV:

$$S_0 = IV \quad (2.6)$$

$$S_i = ((S_{i-1} << x) + C_i \bmod 2^n, \quad (2.6)$$

$$C_i = \text{head}(E_k(S_{i-1}),x) \text{ (xor) } P_i \quad (2.6)$$

$$P_i = \text{head}(E_k(S_{i-1}),x) \text{ (xor) } C_i \quad (2.6)$$

If x bits are lost from the CipherText, the cipher will output incorrect PlainText until the shift register once again equals a state it held while encrypting. At this point the cipher is resynchronized and this guarantees a maximum of one block size of output will be garbled.

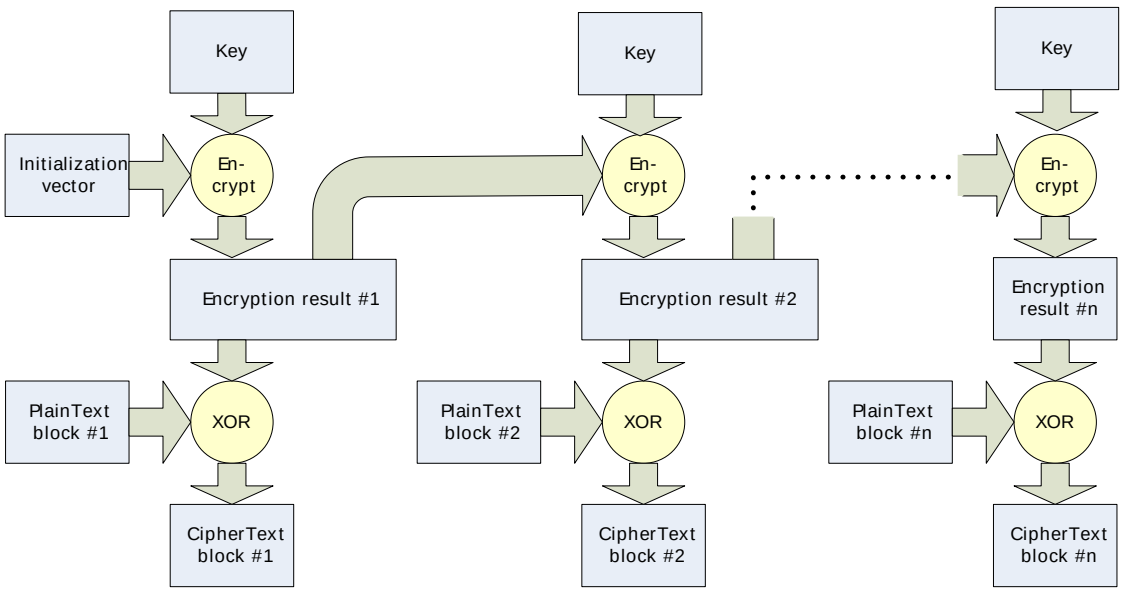
With the CFB method, changes in the PlainText propagate indefinitely in the CipherText. Encryption cannot be parallelized. But, just like the CBC, decryption can be parallelized. When decrypting, a one-bit change in the CipherText affects two PlainText blocks: it results in a one-bit change in the corresponding PlainText block and complete corruption of the following PlainText block. Later PlainText blocks are decrypted normally.

The CFB shares two advantages over CBC mode with the stream cipher modes OFB and CTR (see further): the block cipher is only used in the encrypting direction and the message does not need to be padded to a multiple of the cipher block size (though CipherText stealing can also be used).

2.1.4 Output Feedback Mode (OFB)

The output feedback (OFB) mode transforms a block cipher into a synchronous stream cipher. It generates key stream blocks, which are then XOR'ed with the PlainText blocks to get the CipherText. Just as with other stream ciphers, flipping a bit in the CipherText produces a flipped bit in the PlainText at the same location. This property allows many error correcting codes to function even when applied before encryption.

Figure 2.7. Output Feedback Mode (OFB)



Because of the symmetry of the XOR operation, encryption and decryption are identical:

OFB Algorithm

$$C_i = P_i \text{ (xor) } O_iP_i = C_i \text{ (xor) } O_iO_i = E_k(O_{i-1}, O_0 = IV \quad (2.7)$$

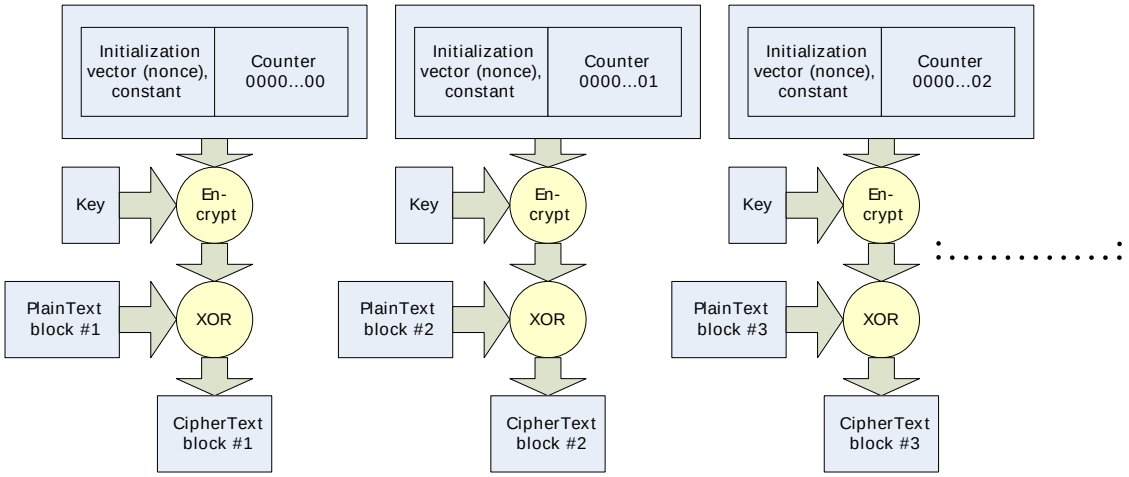
When using the OFB mode, each output feedback block cipher operation depends on all previous ones, thus they can only be performed sequentially. However, because the PlainText or CipherText is only used for the final XOR, the block cipher operations can be performed in advance sequentially, allowing the final step to be performed in parallel once the PlainText or CipherText is available.

2.1.5 Counter Mode (CTR)

CTR mode is also known as Integer Counter Mode (ICM) or Segmented Integer Counter (SIC) mode. This is another encryption mode that turns a block cipher into a stream cipher. The name of this method is given by the fact that it generates the next key stream block by encrypting successive values of a "counter". In practice, the counter can be any function which produces a sequence which is guaranteed not to repeat for a long time, although an actual counter is the most frequently used.

CTR mode has similar characteristics to OFB, but also allows a random access property during decryption. CTR mode is well suited to operation on a multi-processor machine where blocks can be encrypted in parallel. Similarly to earlier and lower performance algorithms, there is no feedback between one encryption step and the next; however input parameters (or "particular keys") to each step are generated using a combination of an initialization vector (sometimes called a nonce) and the actual counter. This combination can be concatenation (the most frequent), addition, XOR-ing, etc.

Figure 2.8. Counter Mode (CTR)



Decryption is a two-step process, the first identical and the second reversed with respect to encryption. First the nonce-counter concatenation is encrypted with the key, then the result is XOR'ed with the CipherText block to result the corresponding PlainText block.

2.1.6 Note on Error Propagation

Before using message authentication codes and authenticated encryption, the "error propagation" was a common selection criterion for a mode of operation vs. another. As mentioned, for example, a one-block error in the transmitted CipherText would result in a one-block error in the reconstructed PlainText for ECB or CTR mode encryption, while in CBC mode such an error would affect two blocks.

Historically there is a dispute between exhibiting such resilience in the face of random errors such as line noise and the argument that the degree of "robustness" implied by error correction increases the scope for attackers to maliciously tamper with a message.

In general, such an error will result in the entire message being rejected. If resistance to random error is desired, error-correcting codes should be applied to the CipherText before the actual transmission. Error-correcting codes are not within the software scope of this application note.

3 EFM32 Implementation

The EFM32 family of microcontrollers includes a hardware AES accelerator (symmetric block cipher engine) that can be used with 128-bit block sizes and 128- or 256-bit keys with little or no CPU intervention. Encrypting or decrypting one 128-bit data block is done in 54 HFCORECLK cycles with 128-bit keys and 75 HFCORECLK cycles with 256-bit keys. On a finished encryption/decryption operation, the AES accelerator can output an interrupt request or a DMA request. For more details, see the Chapter "AES - Advanced Encryption Standard Accelerator" of the device reference manual.

In order to start the encryption or decryption of one data block, the key and data block must be loaded into the KEY and DATA registers respectively. The key for encryption is called PlainKey. After one encryption, the resulting key in the KEY registers is the CipherKey. This key must be loaded into the KEY registers before every decryption. After one decryption, the resulting key will be the PlainKey. The resulting PlainKey/CipherKey is only dependent on the value in the KEY registers before encryption/decryption.

As such, the AES accelerator implements the "encrypt" step present in all method diagrams in Figure 2.3 (p. 5) through Figure 2.8 (p. 10) and the software included in this application note takes care of appropriately supplying the input parameters (keys, nonces, counters, data blocks, etc) and cycling the operations throughout the encryption and decryption of a complete message. For messages whose length is not an exact multiple of the key length, the last data word is padded with trailing zeros.

While ECB and CBC decryption involves at each step the reverse transformation from that used when encrypting, OFB, CFB and CTR decryption involves at each step the same transformation used when encrypting (see formulae).

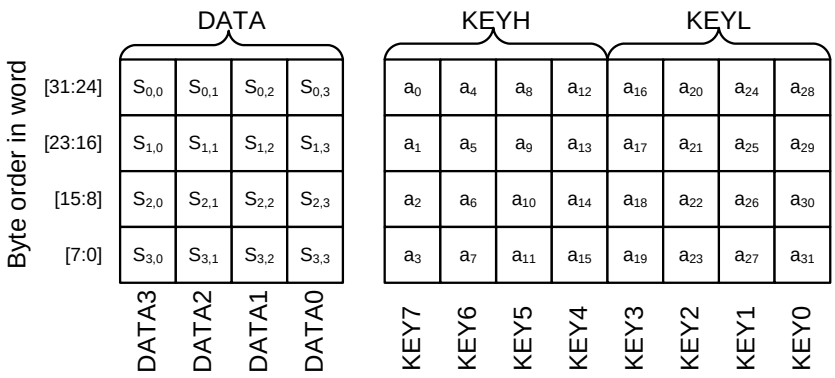
3.1 Data and Key Orientation

The EFM32 AES module implements encryption with either 128-bit or 256-bit keys. It contains a 128-bit DATA (State) register and two 128-bit KEY registers defined as DATA3-DATA0, KEY3-KEY0 (KEYL) and KEY7-KEY4 (KEYH)

- In AES128 mode, the 128-bit key is read from KEYL
- In AES256 mode, the 256-bit key is read from [KEYH:KEYL]

Figure 3.1 (p. 11) presents the key byte order for 256-bit keys, as the more general case (In 128-bit mode a_{16} represents the first byte of the 128-bit key, structured in 16 bytes, a_{16} through a_{31}).

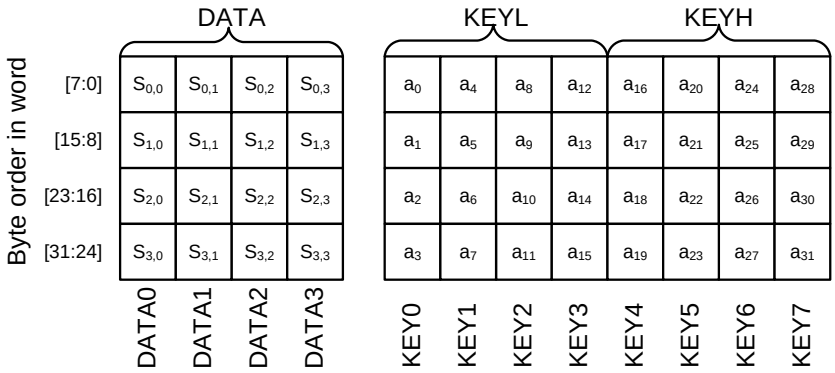
Figure 3.1. AES Data and Key order



Note that the byte ordering above is opposite of the order defined in the AES standard. Some EFM32 devices therefore have an additional control bit (BYTEORDER in AES_CTRL), which allows the an

alternative ordering of the individual bytes within the DATA and KEY registers (Figure 3.2 (p. 12)). The interrupt driven examples in this application note do not use BYTEORDER in order to be compatible with all EFM32 devices. The order of the bytes are instead reversed by software. The BYTEORDER bit is however used for the DMA example as the DMA is not able to reverse the bytes itself.

Figure 3.2. AES Data and Key order (BYTEORDER=1)



The registers DATA3-DATA0 are not memory mapped directly, but can be written/read by accessing AES_DATA or AES_XORDATA. Writing DATA3-DATA0 is done through 4 consecutive writes to AES_DATA (or AES_XORDATA), starting with the word which is to be written to DATA0. Following each write, the words will be automatically word-wise barrel shifted towards the least significant word : DATA3->DATA2->DATA1->DATA0. Similarly, the key registers are not memory mapped directly: KEY3-KEY0 are accessed through AES_KEYLn (n=A, B, C or D), while KEY7-KEY4 are accessed through KEYHn (n=A, B, C or D). 4 consecutive writes to AES_KEYLn rotate its content right through KEY3->KEY2->KEY1->KEY0, whereas 4 consecutive writes to AES_KEYHn rotate its content right through KEY7->KEY6->KEY5->KEY4.

Note that KEYHA, KEYHB, KEYHC and KEYHD are physically the same register, but mapped to four different addresses. The user can choose which of these addresses to use when updating the KEY7-KEY4 and/or KEY3-KEY0 registers.

The PlainText data for encryption should be presented in consecutive blocks to the DATA3-DATA0 registers and the key in the KEY7-KEY4/KEY3-KEY0 as described above, starting with the least significant word. At the end of the block's encryption, the result can be read from AES_DATA, also starting with the least significant word (4 consecutive reads).

4 Software Examples

This application note includes software examples for encryption and decryption in the following cipher modes:

- AES-128 ECB
- AES-256 ECB
- AES-128 CBC
- AES-256 CBC
- AES-128 CFB
- AES-128 OFB
- AES-128 CTR

All the above examples are interrupt driven to allow the CPU to sleep or execute other tasks while the encryption is ongoing. The AES functions used allow in-place RAM-to-RAM encryption for a minimum memory usage. The examples are provided as IDE-projects for the EFM32G890F128, but the projects can be easily ported to other EFM32 devices which include the AES Accelerator peripheral. The data and key used in the examples are taken from the NIST test vectors which are also provided in PDFs.

4.1 AES-128 CBC with DMA Example

In addition to the interrupt driven examples, one example is included which demonstrates AES-128 encryption and decryption with the use of the DMA. This allows the CPU to be fully asleep while large blocks of data are being encrypted or decrypted. For encryption, two DMA channels are used:

1. **READDATA**: This channel reads the finished output data.
2. **WRITEDATA**: This channel writes the input data to the AES. The XORDATA function ensures that the input data is automatically XORed with the previous data.

For decryption, three DMA channels are used:

1. **WRITEPREVDATA**: Writes the input from the previous block operation into the data for the current block using the XORDATA function.
2. **READDATA**: This channel reads the finished output data.
3. **WRITEDATA**: This channel writes the input data to the AES.

It is important that the DMA channels are executed in the order above. The AES peripheral sets the request signals for these channels at the same time when an AES operation has finished. It is therefore important to use lower numbered DMA channels for the first operations, as the DMA priorities according to the channel numbering. This example is written for a EFM32WG990F256 and uses the BYTEORDER function. Note that some devices which includes the BYTEORDER function has an errata that prevents this function from being used with XORDATA writes, which are necessary to implement DMA driven CBC. For more information on DMA operation, please refer to application note an0013.

5 Revision History

5.1 Revision 1.11

2013-09-03

New cover layout

5.2 Revision 1.10

2013-05-08

Added software projects for ARM-GCC and Atollic TrueStudio.

Changed byte order in AES examples.

Added DMA example.

5.3 Revision 1.02

2012-11-12

Adapted software projects to new kit-driver and bsp structure.

5.4 Revision 1.01

2012-04-20

Adapted software projects to new peripheral library naming and CMSIS_V3.

5.5 Revision 1.00

2012-01-02

Initial revision.

A Disclaimer and Trademarks

A.1 Disclaimer

Silicon Laboratories intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Laboratories products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Laboratories reserves the right to make changes without further notice and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Silicon Laboratories shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products must not be used within any Life Support System without the specific written consent of Silicon Laboratories. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Laboratories products are generally not intended for military applications. Silicon Laboratories products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons.

A.2 Trademark Information

Silicon Laboratories Inc., Silicon Laboratories, the Silicon Labs logo, Energy Micro, EFM, EFM32, EFR, logo and combinations thereof, and others are the registered trademarks or trademarks of Silicon Laboratories Inc. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. All other products or brand names mentioned herein are trademarks of their respective holders.

B Contact Information

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701

Please visit the Silicon Labs Technical Support web page:
http://www.silabs.com/support/pages/contacttechnicalsupport.aspx
and register to submit a technical support request.

Table of Contents

1. Introduction 2
2. The AES Algorithm 3
    2.1. Cipher Modes 4
3. EFM32 Implementation 11
    3.1. Data and Key Orientation 11
4. Software Examples 13
    4.1. AES-128 CBC with DMA Example 13
5. Revision History 14
    5.1. Revision 1.11 14
    5.2. Revision 1.10 14
    5.3. Revision 1.02 14
    5.4. Revision 1.01 14
    5.5. Revision 1.00 14
A. Disclaimer and Trademarks 15
    A.1. Disclaimer 15
    A.2. Trademark Information 15
B. Contact Information 16
    B.1. 16

List of Figures

2.1. AES Rounds 3
2.2. AES State Array 4
2.3. Electronic Codebook Mode (ECB) 5
2.4. Cipher-block Chaining Mode (CBC) 6
2.5. Propagating Cipher Block Chaining Mode (PCBC) 7
2.6. Cipher Feedback Mode (CFB) 8
2.7. Output Feedback Mode (OFB) 9
2.8. Counter Mode (CTR) 10
3.1. AES Data and Key order 11
3.2. AES Data and Key order (BYTEORDER=1) 12

List of Equations

2.1. Encryption CBC 6
2.2. Decryption CBC 6
2.3. Encryption PCBC 7
2.4. Decryption PCBC 7
2.5. CFB Encryption 8
2.6. CFB Decryption 8
2.7. OFB Algorithm 9

silabs.com


ZERO
ARM Cortex-M0+
TINY
ARM Cortex-M3
GECKO
ARM Cortex-M3
LEOPARD
ARM Cortex-M3
GIANT
ARM Cortex-M3
WONDER
ARM Cortex-M4