Using the STM8L16x AES hardware accelerator

Introduction

The purpose of cryptography is to protect sensitive data to avoid it being read by unauthorized persons. There are many algorithms that implement cryptography. These techniques can be split into:

- Asymmetric cryptography algorithms: These algorithms use a key to encrypt and another key to decrypt messages. RSA and DSA are examples of this type of algorithm.
- Symmetric cryptography algorithms: These algorithms use the same key to encrypt and decrypt messages. Advanced Encryption Standard (AES), Data Encryption Standard (DES) are examples of this type of algorithm.

The Advanced Encryption Standard (AES) specifies a FIPS-approved cryptography algorithm that can be used to protect electronic data. AES exists in three versions: 128-bit, 192-bit and 256-bit.

High density STM8L16x microcontrollers have an embedded AES 128-bit hardware accelerator to off-load the CPU from encryption/decryption tasks. This AES peripheral is a fully compliant implementation of the AES standard as defined by the Federal information processing standards publication (FIPS PUB 197, 2001 November 26).

This application note applies to STM8L16x high-density devices with built-in AES peripheral.

The software supplied with this application note provides an implementation of some commonly used AES chaining modes (ECB, CBC, CFB, OFB and CTR).

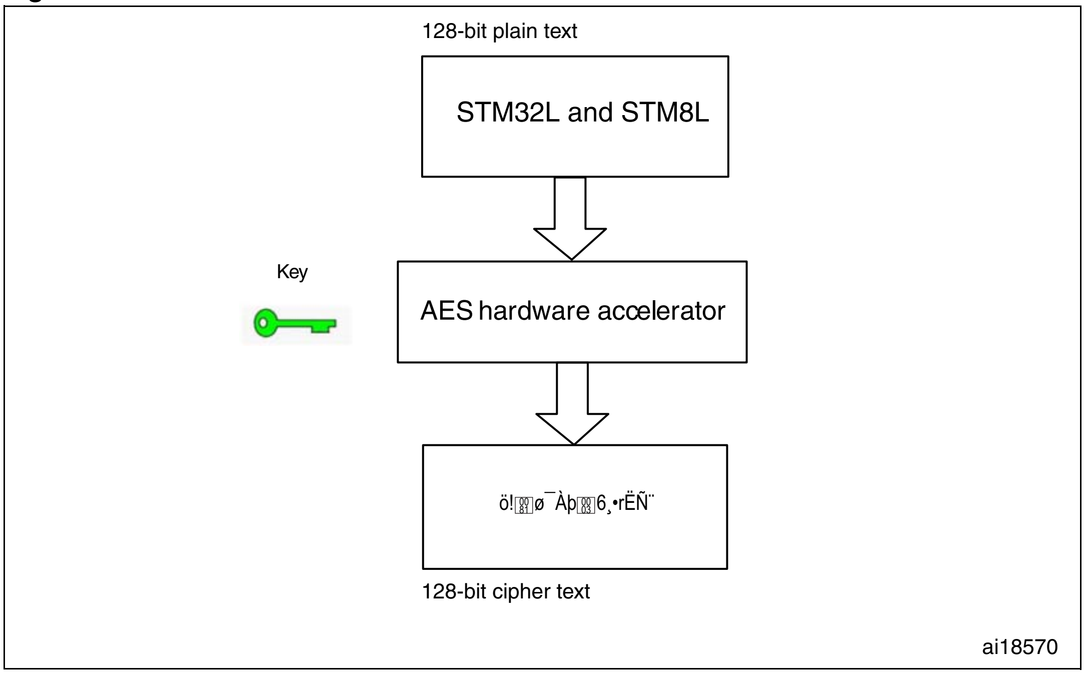
For more detailed information you should refer to the AES section of RM0031 *STM8L15x and STM8L16x microcontroller family reference manual*.

Contents

1 AES hardware accelerator operation modes

The Advanced Encryption Standard (AES) hardware accelerator can be used to encrypt (encipher) and decrypt (decipher) 128-bit blocks using a 128-bit key length. It is a fully compliant implementation of the AES standard as defined by Federal Information Processing Standards publication (FIPS PUB 197, 2001 November 26).

Figure 1. AES hardware accelerator



The main features of the built in AES hardware accelerator STM8L16x devices are:

- Both encryption and decryption with compliance to FIPS using a 128-bit key
- Key derivation for decryption
- 892 clock cycles to encrypt or decrypt one 128-bit block
- Read and write error detection with interrupt capability
- AES Computation Complete interrupt capability
- Automatic data flow control with support of direct memory access (DMA) using 2 channels: one for incoming data, the other for outgoing data.

The AES hardware accelerator provides four modes of operation:

1. Encryption mode
2. Decryption mode (using decryption key)
3. Key derivation mode
4. Key derivation and decryption mode (using encryption key)

Once configured and enabled, the AES goes through three phases:

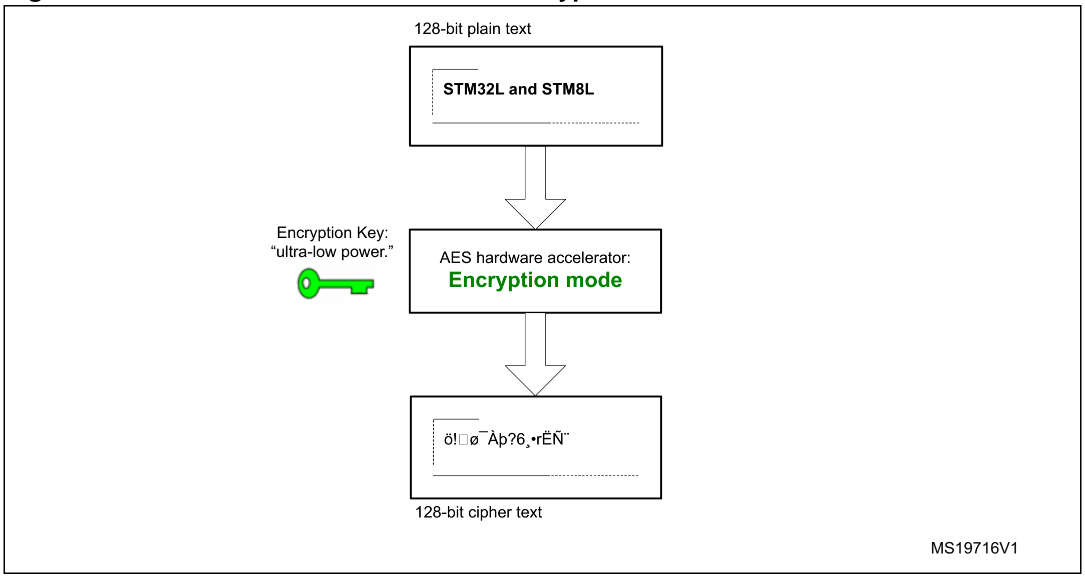
- Input phase: data (plain text, cipher text or key) are written in the AES Data Input Register (AES_DINR). This phase can be carried-out either by the CPU or by Direct Memory Access (DMA).
- Computation phase: the AES hardware accelerator performs the AES operation (encryption, decryption...). Once this phase ends, the Computation Complete Flag is set. An interrupt can be generated if enabled using Computation Complete Interrupt Enable bit (CCIE).
- Output phase: the application can read the Data Output Register (AES_DOUTR). Like the input phase, this task can be performed by the CPU or by DMA.

1.1 AES encryption mode

In this mode, the AES accelerator performs the encryption of a 128-bit plain text using the provided 128-bit key to compute the cipher text.

In the example below, the plain text “STM32L and STM8L” is encrypted using the key “ultra-low power.”. The cipher text, computed by the AES hardware accelerator, in ASCII format is "ö!?ø¯Àþ6¸•rËÑ¨" and in hex format is “22 F6 21 81 F8 AF C0 FE 03 36 B8 95 72 CB D1 A8”.

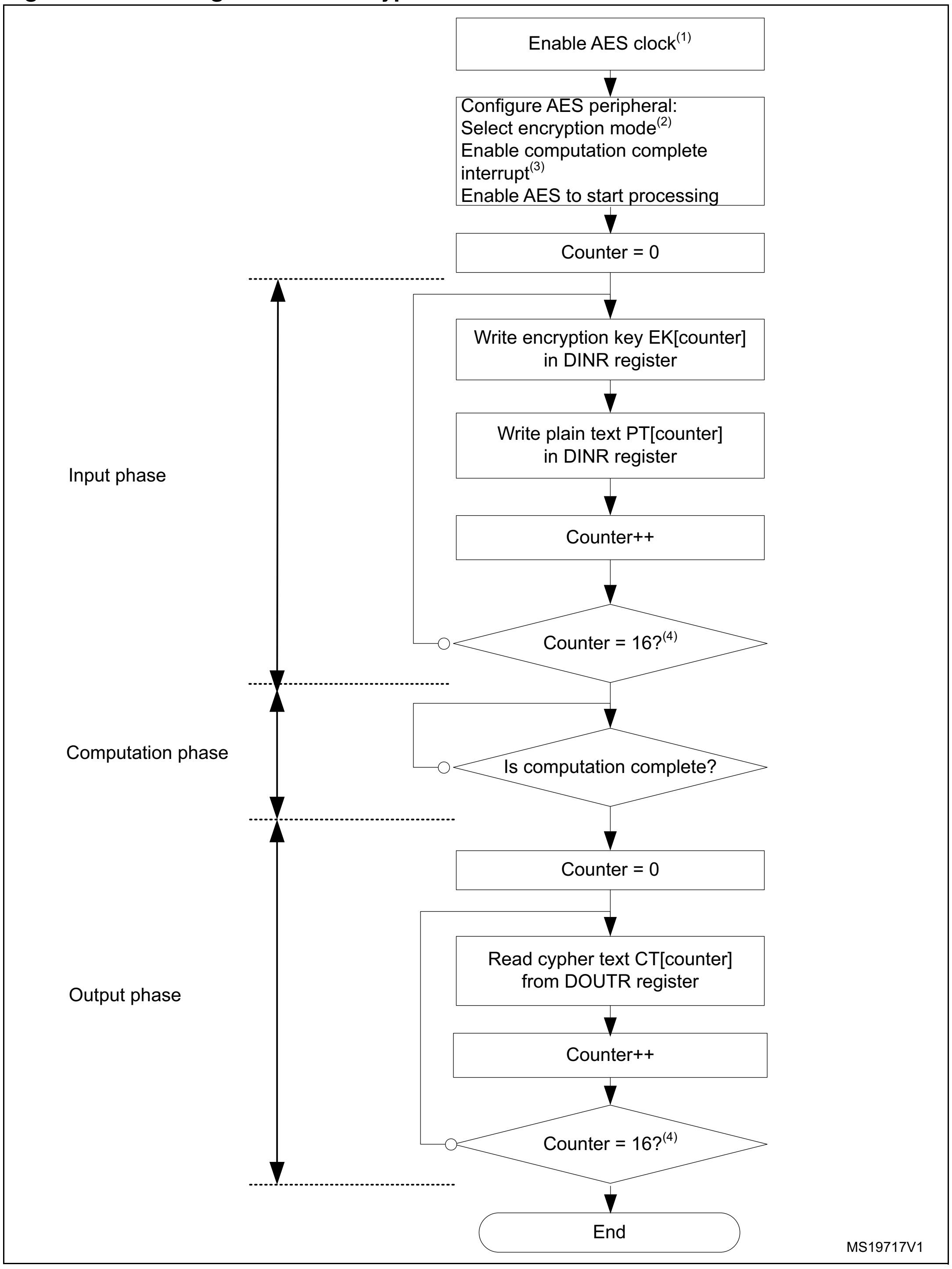
Figure 2. AES hardware accelerator: encryption mode



The AES takes 892 clock cycles to encrypt one 128-bit block.

To perform text encryption, follow the algorithm given in the flowchart below.

Figure 3. AES algorithm in encryption mode


Enable AES clock(1)
Configure AES peripheral:
Select encryption mode(2)
Enable computation complete interrupt(3)
Enable AES to start processing
Counter = 0
Write encryption key EK[counter] in DINR register
Write plain text PT[counter] in DINR register
Counter++
Counter = 16?(4)
Input phase
Computation phase
Is computation complete?
Counter = 0
Read cypher text CT[counter] from DOUTR register
Counter++
Counter = 16?(4)
Output phase
End
MS19717V1

1. The AES peripheral clock must be enabled to allow write access to the AES registers.
2. Mode configuration must be done before enabling the AES.
3. If required, the Computation Complete interrupt can be enabled using the Computation Complete Interrupt Enable (CCIE) bit.
4. The AES hardware accelerator requires the 16 bytes of data (128 bits) to be written/read from MSB (16th data byte) to LSB (1st data byte) that is why the Counter is initialized to 0.

EK is a 16-byte buffer containing the Encryption Key.

PT is a 16-byte buffer containing the Plain Text.

CT is 16-byte buffer containing the Cipher Text.

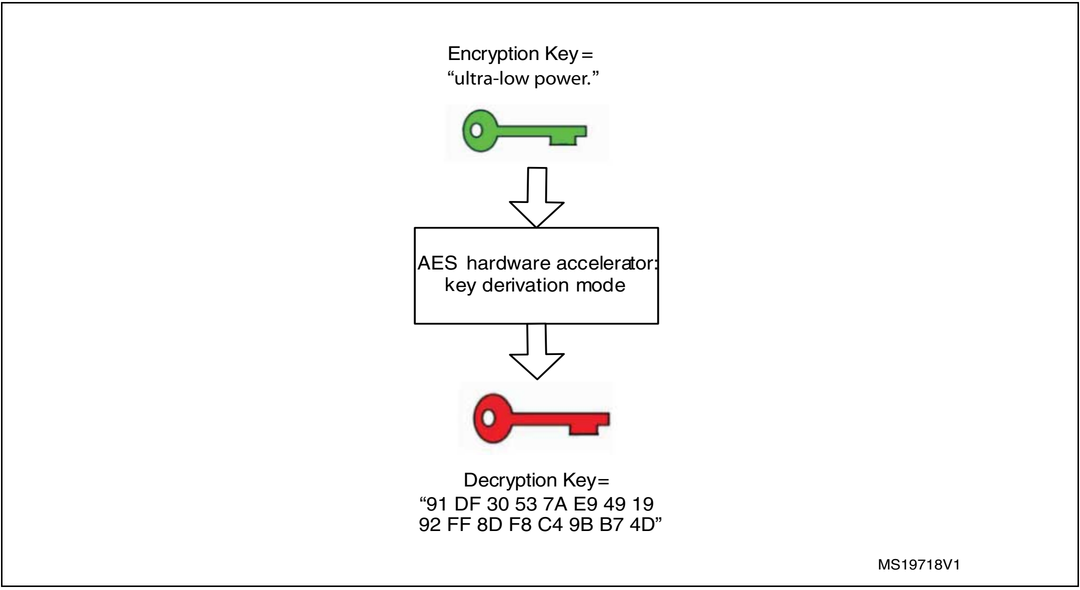
Note: Refer to the AES_EncryptDecrypt example available in the STM8L15x_StdPeriph_Lib firmware package for an example of single block encryption/decryption in C language.

1.2 Key derivation mode

AES is a symmetric cryptography algorithm: Encrypting and decrypting text is performed by the same key. The AES hardware accelerator can compute a derived key from an encryption key that is named “Decryption key”. The decryption key is pre-computed to speed-up the decryption phase.

In this mode, the AES takes the encryption key as input and provides the decryption key in output. This mode takes 320 clock cycles.

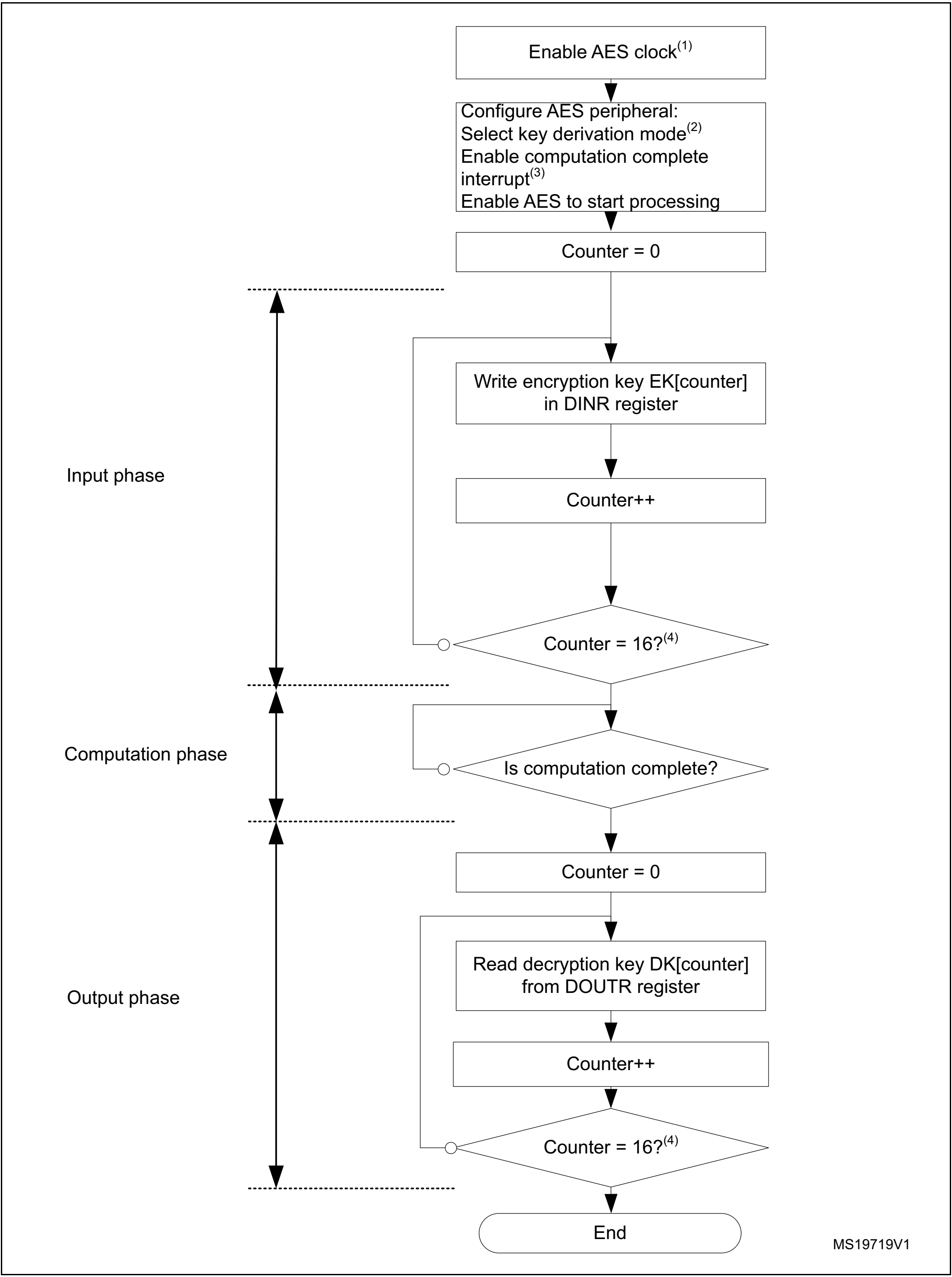
Figure 4. AES hardware accelerator: key derivation mode



In the example above, the decryption key, computed from the encryption key “ultra-low power.”, is “91 DF 30 53 7A E9 49 19 92 FF 8D F8 C4 9B B7 4D” (hex format).

The algorithm given in the figure below provides the steps needed to use the AES hardware accelerator in key derivation mode.

Figure 5. AES algorithm in key derivation mode


Enable AES clock(1)
Configure AES peripheral:
Select key derivation mode(2)
Enable computation complete interrupt(3)
Enable AES to start processing
Counter = 0
Input phase
Write encryption key EK[counter] in DINR register
Counter++
Counter = 16?(4)
Computation phase
Is computation complete?
Counter = 0
Output phase
Read decryption key DK[counter] from DOUTR register
Counter++
Counter = 16?(4)
End
MS19719V1

1. The AES peripheral clock must be enabled to allow write access to the AES registers.
2. Mode configuration must be done before enabling the AES.
3. If required, the Computation Complete interrupt can be enabled using the Computation Complete Interrupt Enable (CCIE) bit.
4. The AES hardware accelerator requires the 16 bytes of data (128 bits) to be written/read from MSB (16th data byte) to LSB (1st data byte) that is why the Counter is initialized to 0.

EK is a 16-byte buffer containing the encryption key.

DK is a 16-byte buffer of 16 bytes containing the decryption key.

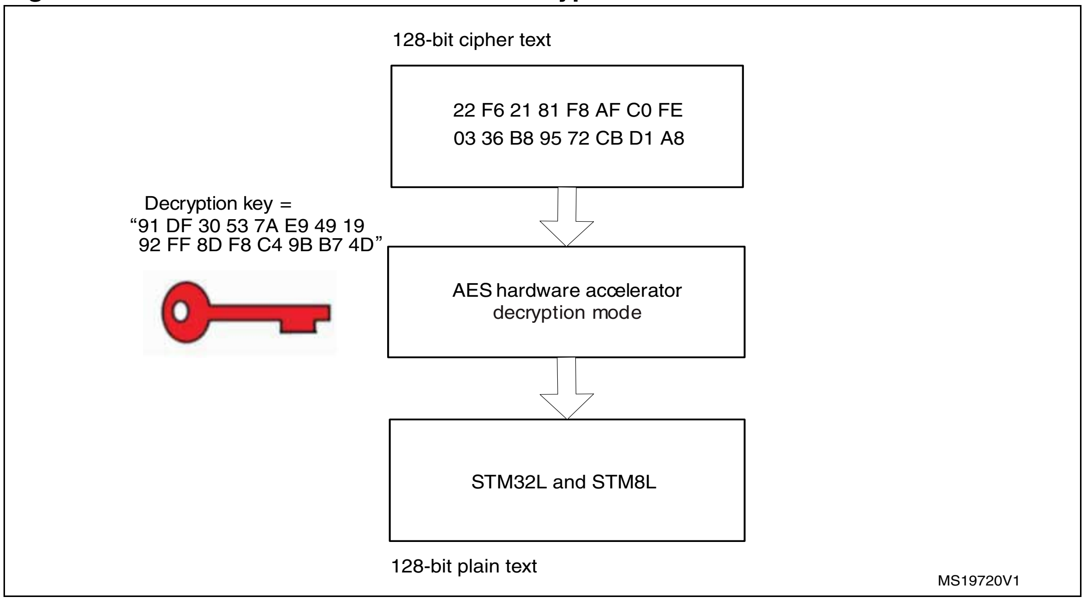
Note: Refer to the AES_KeyDerivation example available in the STM8L15x_StdPeriph_Lib firmware package for an example of implementation in C language.

1.3 AES decryption mode

In this mode, the AES hardware accelerator performs the decryption of a 128-bit cipher text using the provided 128-bit decryption key to compute the plain text. This mode takes 892 clock cycles for a 128-bit block.

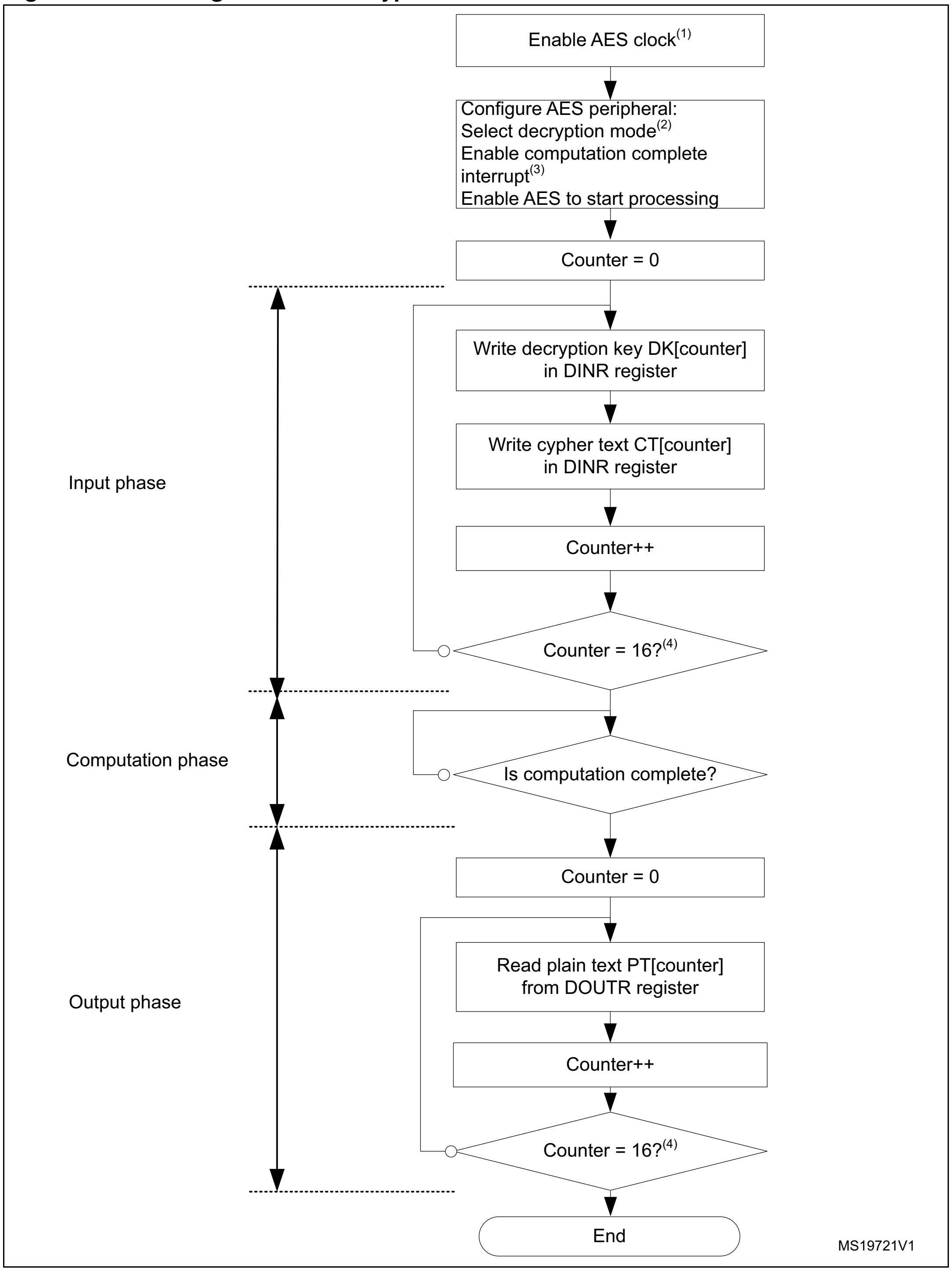
In the example below, the cipher text is “22 F6 21 81 F8 AF C0 FE 03 36 B8 95 72 CB D1 A8” and the decryption key is “91 DF 30 53 7A E9 49 19 92 FF 8D F8 C4 9B B7 4D”. The plain text, computed by AES hardware accelerator, is the original text “STM32L and STM8L”.

Figure 6. AES hardware accelerator: decryption mode



Note: Cipher text and decryption key are in hexadecimal format.

Figure 7. **AES algorithm in decryption mode**


Enable AES clock(1)
Configure AES peripheral:
Select decryption mode(2)
Enable computation complete interrupt(3)
Enable AES to start processing
Counter = 0
Write decryption key DK[counter] in DINR register
Write cypher text CT[counter] in DINR register
Counter++
Counter = 16?(4)
Input phase
Is computation complete?
Computation phase
Counter = 0
Read plain text PT[counter] from DOUTR register
Counter++
Counter = 16?(4)
Output phase
End
MS19721V1

1. The AES peripheral clock must be enabled to allow write access to AES registers
2. Mode configuration must be done before enabling the AES.
3. If required, the Computation Complete interrupt can be enabled using the Computation Complete Interrupt Enable (CCIE) bit.
4. The AES hardware accelerator requires the 16 bytes of data (128 bits) to be written/read from MSB (16th data byte) to LSB (1st data byte) that is why the Counter is initialized to 0.

DK is a 16-byte buffer containing the Decryption Key.

PT is a 16-byte buffer containing the Plain Text.

CT is a 16-byte buffer containing the Cipher Text.

1.4 Key derivation and decryption mode

When this mode is selected, the AES hardware accelerator performs the decryption of a 128-bit cipher text using the provided 128-bit encryption key to compute the plain text. In the example below the cipher text is “22 F6 21 81 F8 AF C0 FE 03 36 B8 95 72 CB D1 A8” and the encryption key is “ultra-low power.”. The plain text, computed by AES hardware accelerator, is the text “STM32L and STM8L”.

Figure 8. AES hardware accelerator: key derivation and decryption mode

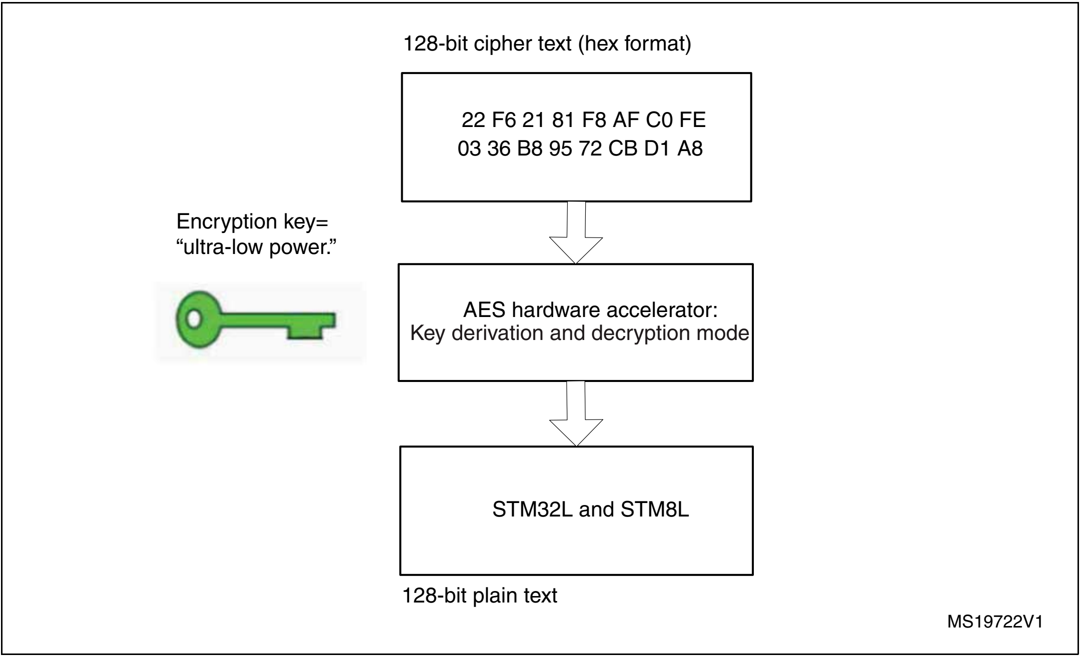
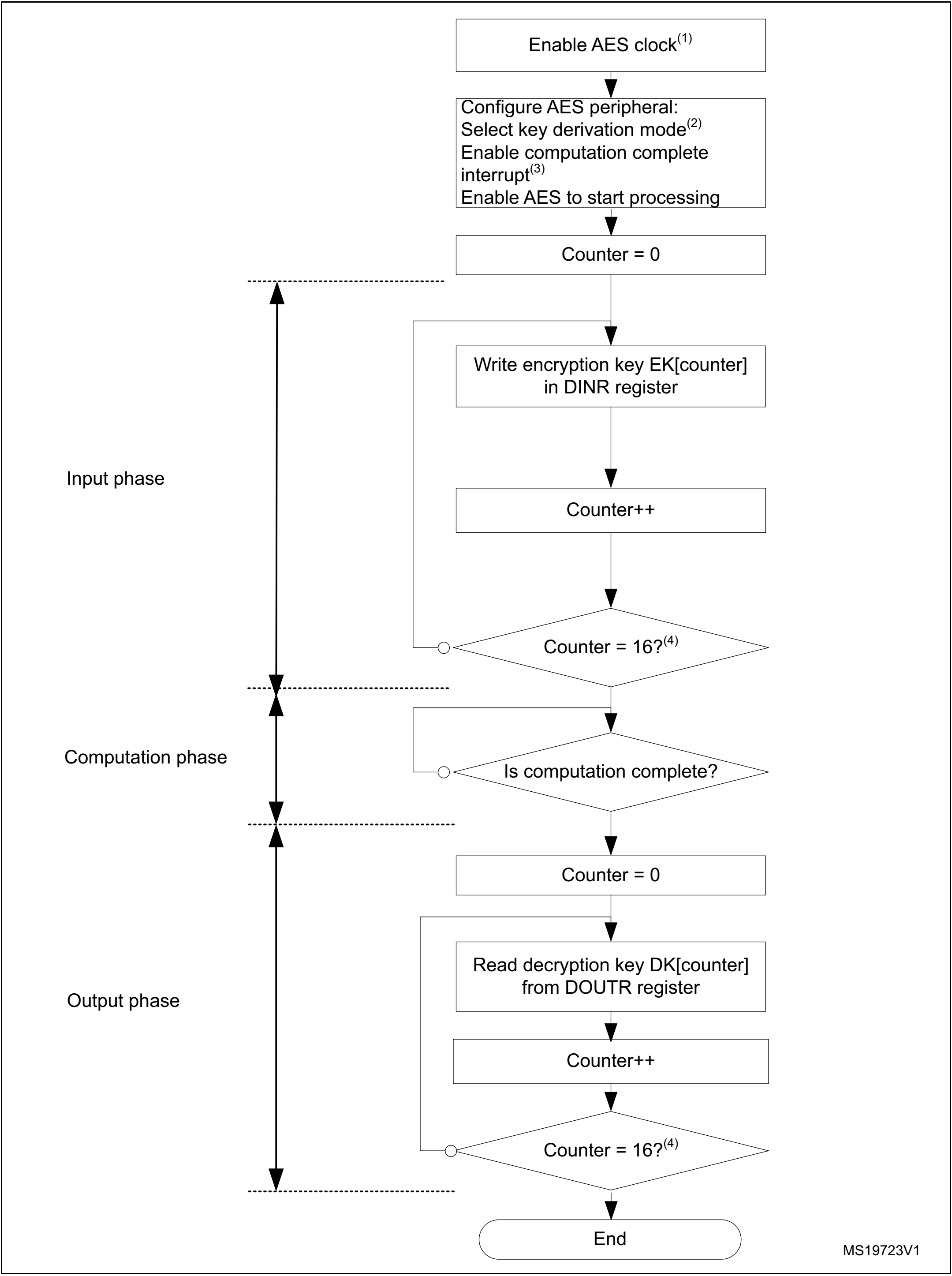


Figure 9. AES algorithm in key derivation and decryption mode


Enable AES clock(1)
Configure AES peripheral:
Select key derivation mode(2)
Enable computation complete interrupt(3)
Enable AES to start processing
Counter = 0
Input phase
Write encryption key EK[counter] in DINR register
Counter++
Counter = 16?(4)
Computation phase
Is computation complete?
Output phase
Counter = 0
Read decryption key DK[counter] from DOUTR register
Counter++
Counter = 16?(4)
End
MS19723V1

1. The AES peripheral clock must be enabled to allow write access to AES registers
2. Mode configuration must be done before enabling the AES
3. If required, the Computation Complete Interrupt can be enabled using the Computation Complete Interrupt Enable (CCIE) bit
4. The AES hardware accelerator requires the 16 bytes of data (128 bits) to be written/read from MSB (16th data byte) to LSB (1st data byte) that is why the Counter is initialized to 0.

EK is a 16-byte buffer containing the Encryption Key.

PT is a 16-byte containing the Plain Text.

CT is 16-byte containing the Cipher Text.

Note: Refer to the AES_EncryptDecrypt example available in the STM8L15x_StdPeriph_Lib firmware package for an example of implementation in C language.

1.5 Decryption mode vs. key derivation and decryption mode

The AES hardware accelerator support two methods for decrypting a cipher text:

- **Key derivation and decryption mode** which requires the encryption key.
- **Decryption mode** which requires the decryption key that must previously be computed using the **key derivation mode**.

The former takes 1228 clock cycles for one 128-bit data block but the latter takes only 892 clock cycles for the same data block length.

Key derivation and decryption modes are useful when the data length is greater than 128-bit. In the first step the decryption key is computed using key derivation mode then data are decrypted using decryption mode.

Example: The length of the data to be encrypted is 160 bytes (128-bit * 10). The formulas below can be used to determine the processing time for both methods:

- **Key derivation and decryption mode** method:

  processing time = number of 128-bit blocks * processing time of single block = 10 * 1228 = 12280 clock cycles
- **Decryption mode + Key derivation mode** method:

  processing time = key derivation processing time + number of 128-bit blocks * processing time of single block = 320 + 10 *892 = 9240 clock cycles

This shows that it is better to use the combined **key derivation and decryption mode** when processing multiple blocks.

2 AES peripheral driver

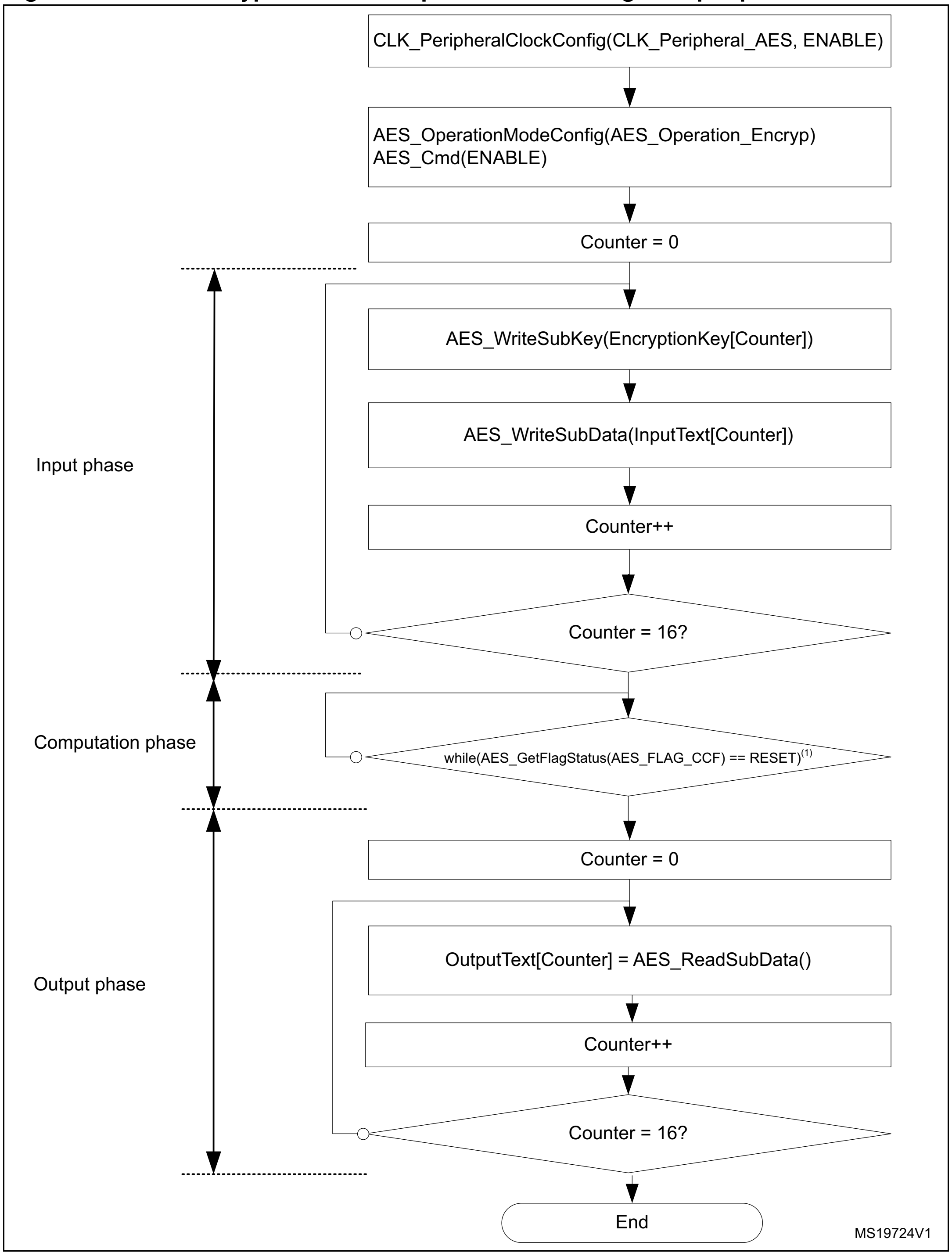
An AES driver written in C language (stm8l15x_aes.c/.h) is available in the STM8L15x_StdPeriph_Lib standard firmware package. This driver provides an abstraction layer to make it easier to use the AES hardware accelerator.

A flowchart of encryption mode using the driver is shown in *Figure 10*.

To help you get started quickly, a set of examples is available in the STM8L15x_StdPeriph_Lib package:

- AES_KeyDerivation: how to get the decryption key from the encryption key
- AES_EncryptionDecryption: how to encrypt and decrypt a single data block
- AES_EncryptionDecryptionBlocks: how to use the DMA peripheral to manage reading/writing data from/to AES data registers

Figure 10. AES encryption mode: implementation using AES peripheral driver


CLK_PeripheralClockConfig(CLK_Peripheral_AES, ENABLE)
AES_OperationModeConfig(AES_Operation_Encryp)
AES_Cmd(ENABLE)
Counter = 0
AES_WriteSubKey(EncryptionKey[Counter])
AES_WriteSubData(InputText[Counter])
Counter++
Counter = 16?
Input phase
while(AES_GetFlagStatus(AES_FLAG_CCF) == RESET)(1)
Computation phase
Counter = 0
OutputText[Counter] = AES_ReadSubData()
Counter++
Counter = 16?
Output phase
End
MS19724V1

1. Once set, Computation Complete Flag (CCF) should be cleared in multiple block processing using the following function AES_ClearFlag(AES_FLAG_CCF).

3 AES block processing

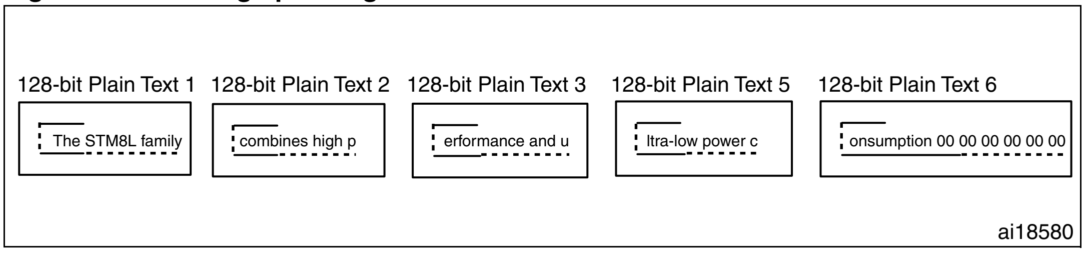
3.1 AES block padding

AES-128 is a 128-bit block cryptography that means that AES encrypts 128-bit plain text providing a 128-bit cipher text. But in general, text to be encrypted/decrypted is of any length so the text must be padded to get an exact multiple of block size (128 bits) before being processed (encrypted or decrypted).

In the following example, the text "The STM8L family combines high performance and ultra-low power consumption" is split into 6 blocks. The last one, since it's size is 10 bytes, is padded by six "00".

Note: There are other methods of message padding than filling with "00".

Figure 11. Message padding with "00"



3.2 AES block chaining: CBC as an example

Encrypting the same plain text using the same key produces the same cipher text, each block is encrypted independently. This could be insecure when encrypting long texts. Chaining modes have been introduced to overcome this limitation. With chaining modes, each block is no longer dependent just on the plain text.

The following are some examples of chaining modes:

- **Electronic CodeBook (ECB)**: This is the simplest mode. Each plain text is encrypted/decrypted separately.
- **Cipher Block Chaining (CBC)**: Each block is XORed with the previous block. This mode requires a 128-bit initialization vector.
- **Cipher Feedback (CFB)**: Each plain text block is XORed with the previous cipher text block before encryption.
- **Output Feedback (OFB)**: The output of the encryption process is used as input for the next block encryption. The output block is XORed with plain text to give the cipher text
- **Counter (CTR)**: A 128-bit counter is encrypted and then XORed with the plain text to give the cipher text. The counter is incremented at each block.

For the purposes of illustration, this section describes block chaining using CBC.

Figure 12 and *Figure 13* below explain the implementation of CBC mode in both encryption and decryption. In CBC mode, each input block is dependent on the previous output block. The input block is XORed with the previous cipher text block before being encrypted. The first block requires an initialization vector.

Figure 12. Encryption using Cipher Block Chaining (CBC) mode

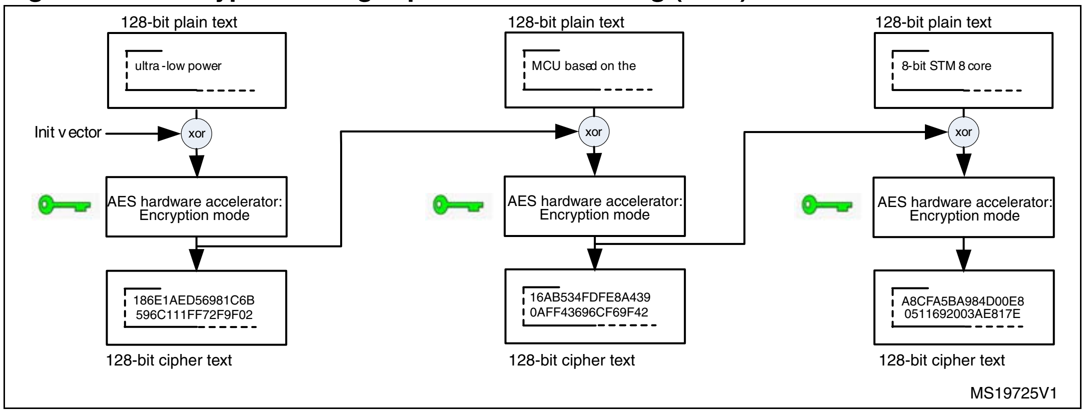
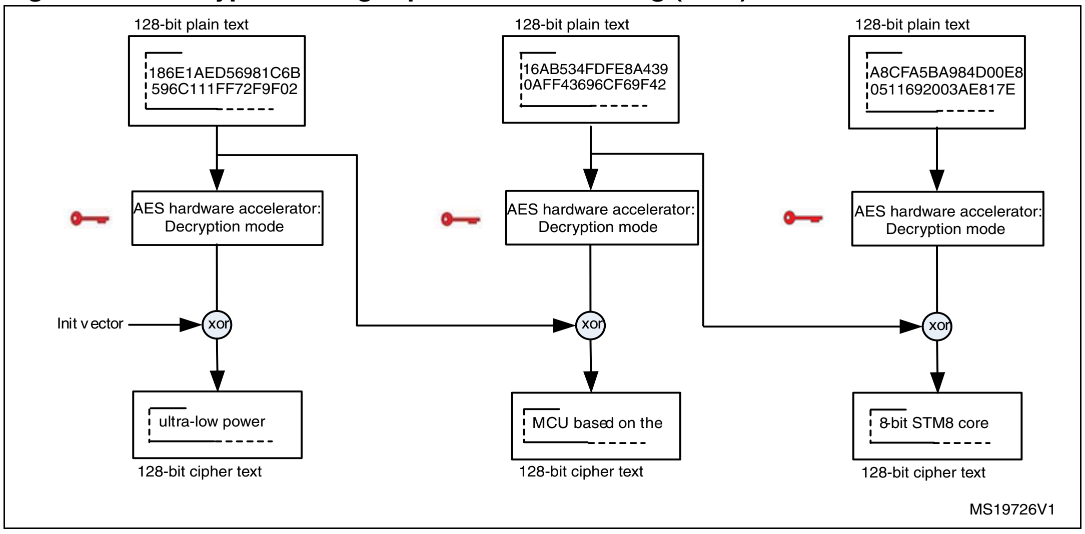

128-bit plain text
ultra-low power
128-bit plain text
MCU based on the
128-bit plain text
8-bit STM 8 core
Init vector
xor
xor
xor
AES hardware accelerator: Encryption mode
AES hardware accelerator: Encryption mode
AES hardware accelerator: Encryption mode
186E1AED56981C6B 596C111FF72F9F02
16AB534FDFE8A439 0AFF43696CF69F42
A8CFA5BA984D00E8 0511692003AE817E
128-bit cipher text
128-bit cipher text
128-bit cipher text
MS19725V1

Figure 13. Decryption using Cipher Block Chaining (CBC) mode


128-bit plain text
186E1AED56981C6B 596C111FF72F9F02
128-bit plain text
16AB534FDFE8A439 0AFF43696CF69F42
128-bit plain text
A8CFA5BA984D00E8 0511692003AE817E
AES hardware accelerator: Decryption mode
AES hardware accelerator: Decryption mode
AES hardware accelerator: Decryption mode
Init vector
xor
xor
xor
ultra-low power
MCU based on the
8-bit STM8 core
128-bit cipher text
128-bit cipher text
128-bit cipher text
MS19726V1

Chaining modes (CBC, CFB, OFB and CTR) are not implemented in hardware. A firmware implementation of block chaining modes (ECB, CBC, CFB, OFB and CTR) using the AES hardware accelerator is provided in the aes_util.c/.h files with this application note.

As an example, the prototype of CBC mode is as follows:

```
AES_Encrypt_CBC(): Encrypt plain text with encryption key providing cipher text using
CBC chaining mode. Encryption status is returned by the function.

ErrorStatus AES_Encrypt_CBC(PlainText, /* address of plain text to be encrypted */

CipherText, /* address of buffer that used to store the cipher text */

EncryptionKey, /* address of key used to encrypt plain text */

InitVector, /* address of initialization vector */

TextSize); /* size of padded plain text */
```

For the implementation of other chaining modes, refer to the aes_util.c/.h files.

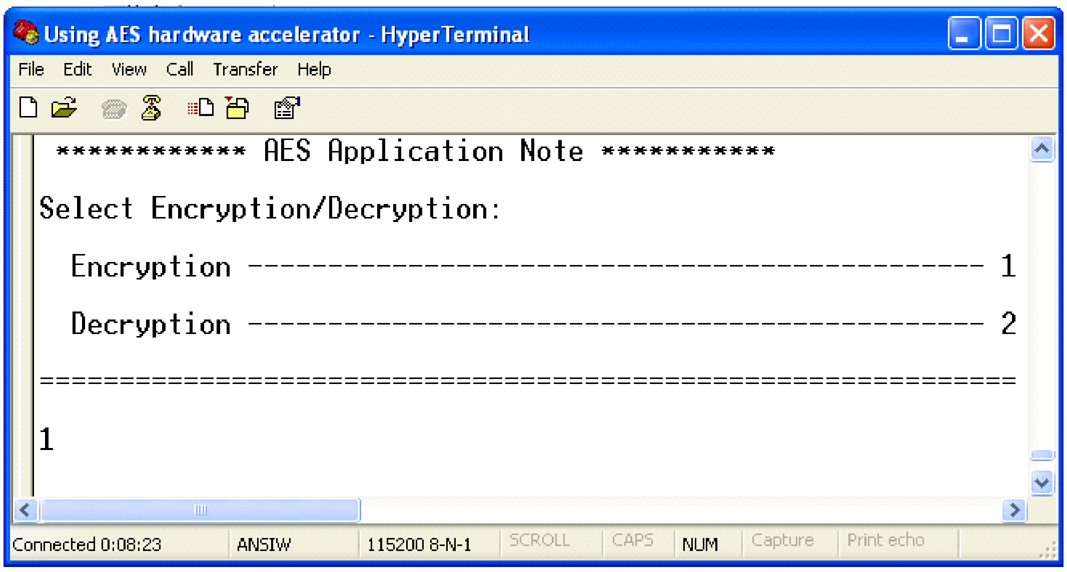
3.3 Using the software implementation of AES chaining modes

An example is available in the firmware associated with this application note to show you how to add a chaining mode implementation to your application.

The example performs both encryption and decryption of a text entered through hyperterminal (or a compatible communication software) using the available chaining modes. The key is stored in the data EEPROM of the STM8L16x device.

Figure 14. below gives a screenshot of encryption/decryption selection.

Figure 14. Encryption / decryption selection

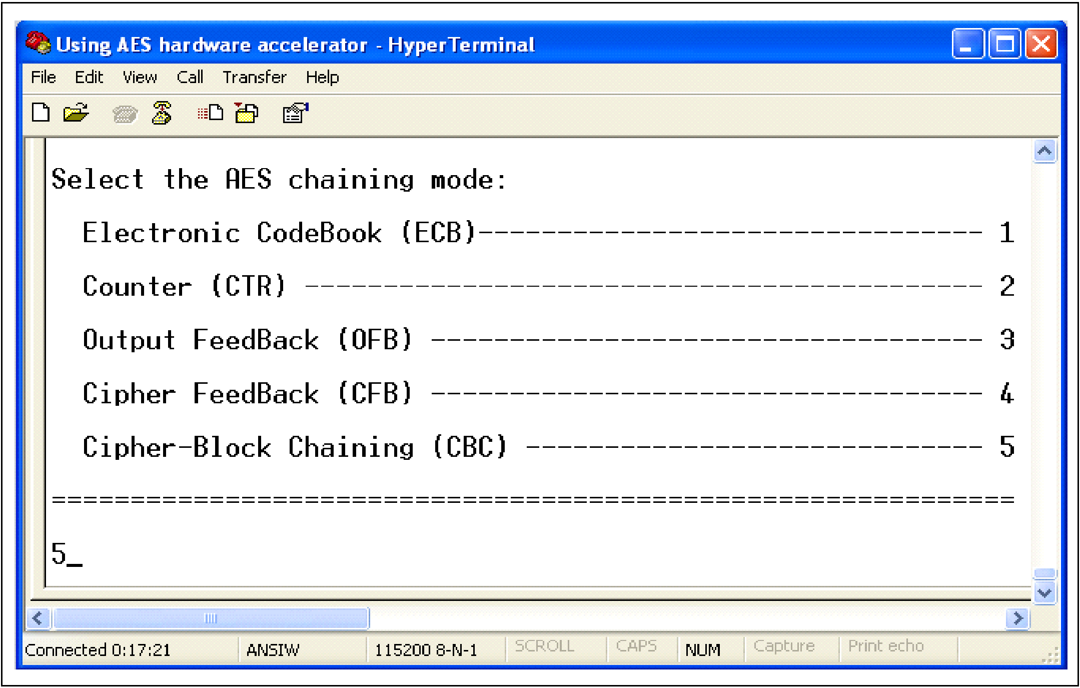

Using AES hardware accelerator - HyperTerminal
File Edit View Call Transfer Help

```
 ************* AES Application Note ***********

Select Encryption/Decryption:

  Encryption -------------------------------------------------- 1

  Decryption -------------------------------------------------- 2

=================================================================

1
```

Connected 0:08:23 ANSIW 115200 8-N-1 SCROLL CAPS NUM Capture Print echo

Figure 15. is a screenshot of block chaining mode selection.

Figure 15. Chaining mode selection



Using AES hardware accelerator - HyperTerminal
File Edit View Call Transfer Help
Connected 0:17:21
ANSIW
115200 8-N-1
SCROLL
CAPS
NUM
Capture
Print echo

```
Select the AES chaining mode:

   Electronic CodeBook (ECB)---------------------------------- 1

   Counter (CTR) --------------------------------------------- 2

   Output FeedBack (OFB) ------------------------------------- 3

   Cipher FeedBack (CFB) ------------------------------------- 4

   Cipher-Block Chaining (CBC) ------------------------------- 5

=============================================================

5_
```

The text received through the RS232 port is saved in a static buffer. The end of transmission occurs when the maximum buffer size has been reached or the carriage return (enter key) has been pressed.

The maximum buffer size is defined in main.h:

#define RX_BUFFER_SIZE (uint16_t) 128

The following code listing (available in stm8l15x_it.c) shows how the end of transmission is detected:

```
if ((RxCounter == RX_BUFFER_SIZE) || (RxBuffer[RxCounter-1] == 0x0D))
  {
    /* Disable the Read data register not empty interrupt */
    USART_ITConfig(EVAL_COM1, USART_IT_RXNE, DISABLE);
      /* Max buffer size had been reached or carriage return had been pressed */
    TransferComplete = 1;
........
.......
```

If the received buffer size is not a multiple of 128-bit, it is padded with the pre-defined pad character. it is defined in main.h

```
#define PAD_CHARACTER          (uint8_t)  0x00
```

The BufferInit() function (in main.c) is available for padding purposes. It checks the size of the entered text and then pads it if necessary.

```
/* Pad the buffer with defined pad character */
    while(counter < (nbrblocks + 1) * 16)
    {
       pText[counter] = PAD_CHARACTER;
       counter++;
    }
```

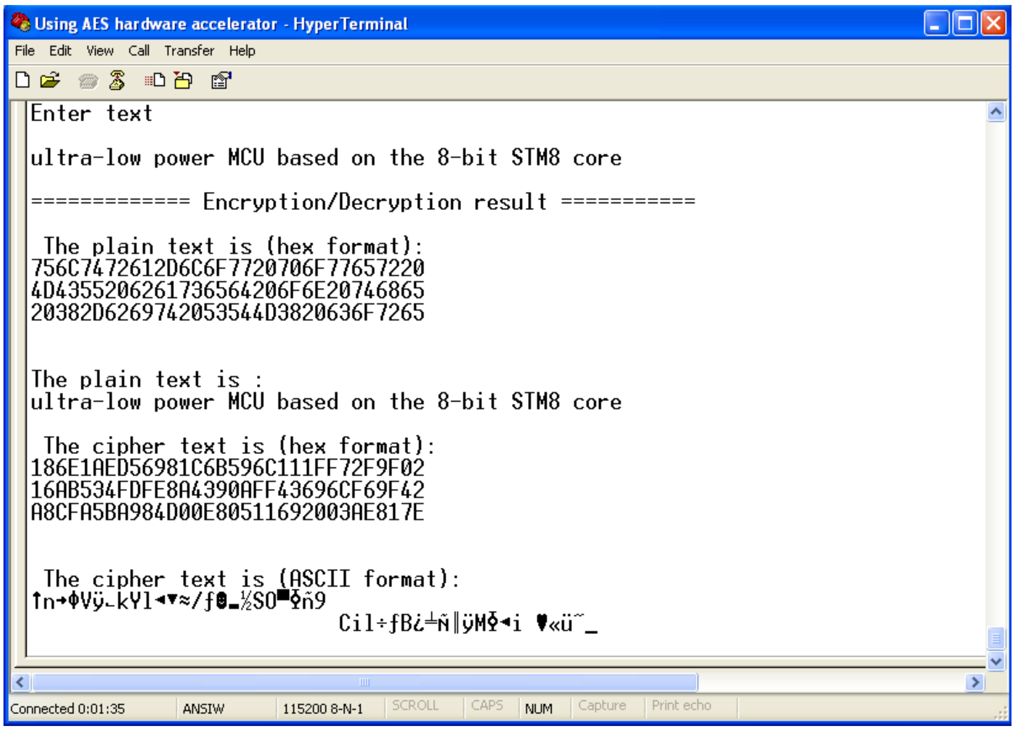
Another buffer is created dynamically to store the result of encryption or decryption (depending on the user selection)

```
if (CryptoMode == CRYPTO_MODE_ENCRYPTION)
     {
       /* Pad the received buffer */
      BufferInit(RxBuffer, &TextSize);
      /* PlainText points at the received buffer */
        PlainText = RxBuffer;
      /* Allocate memory space */
       CipherText = (uint8_t*) malloc(TextSize);
......
......
     else
     {
       /* Pad the received buffer */
      BufferInit(RxBuffer, &TextSize);
      /* CipherText points at the received buffer */
        CipherText = RxBuffer1;
      /* Allocate memory space */
      PlainText = (uint8_t*) malloc(TextSize);
......
......
```

Once the input text is encrypted or decrypted, both plain text and cipher text are displayed. Plain text is displayed in ASCII format and the cipher text is displayed in both ASCII and hex format (because cipher text may include non-printable characters).

Figure 16. shows an example of the display of typed text, plain text and cipher text (ASCII and hex formats).

Figure 16. Encryption/Decryption result display



Using AES hardware accelerator - HyperTerminal
File Edit View Call Transfer Help

```
Enter text

ultra-low power MCU based on the 8-bit STM8 core

============= Encryption/Decryption result ===========

 The plain text is (hex format):
756C7472612D6C6F7720706F77657220
4D43552062617365642O6F6E20746865
20382D62697420535440382O636F7265


The plain text is :
ultra-low power MCU based on the 8-bit STM8 core

 The cipher text is (hex format):
186E1AED56981C6B596C111FF72F9F02
16AB534FDFE8A4390AFF43696CF69F42
A8CFA5BA984D00E80511692003AE817E


 The cipher text is (ASCII format):
↑n→ΦVÿ.kYl◄▼≈/ƒ☻_½SO▀Φñ9
                          Cil÷ƒB¿╧Ñ║ÿMΦ◄i ♥«ü˜_
```

Connected 0:01:35 ANSIW 115200 8-N-1 SCROLL CAPS NUM Capture Print echo

3.4 Some recommendations

The firmware implementation of the AES chaining modes provided with this application note is intended to be used as a starting-point. For many applications, this firmware may be used as is. However, for applications with tough constraints in terms of code size and/or execution speed, this firmware may be tailored and simplified to cover only the required features. Refer to the AES section in the microcontroller's reference manual RM0031 for more details.

The Direct Memory Access (DMA) can be used to handle input and output phases for better performance and free the CPU for other tasks

The encryption key can be stored and protected in EEPROM data memory. Refer to the "Flash program memory and data EEPROM" section in the microcontroller's reference manual RM0031.

4 Conclusion

This application note gives an introduction to the AES hardware accelerator embedded in STM8L16x microcontrollers which can be used for data encryption/decryption with minimal CPU load and memory usage. The main operating modes have been presented in *Section 1 on page 3*.

A description has been given of the software available for the AES with this application note. *Section 2 on page 13* described the peripheral driver with an abstraction layer for easy use of the peripheral, *Section 3 on page 15* described a set of examples that can be used as a quick start for developing your own code and an in-depth example using AES block chaining modes.

5 Revision history

Table 1. Document revision history

Date	Revision	Changes
02-Dec-2010	1	Initial release.
01-Jul-2011	2	Updated figures *1*, *2*, *3*, *4*, *5*, *6*, *7*, *8*, *9*, *10*, *12*, *13*, and *16*.