AT10764: Software Library for AES-128 Encryption and Decryption

Application Note

Introduction

Advanced Encryption Standard (AES) is a specification for encryption of electronic data established by National Institute of Standards and Technology (NIST) in 2001 as Federal Information Processing Standards (FIPS) 197. This is a symmetric block cipher algorithm used for the encryption and decryption of electronic data. This application note explains the C implementation of AES encryption and decryption algorithm.

Features

The application note covers the following features.

- Compliant with FIPS Publication 197, Advanced Encryption Standard (AES)
- AES encryption and decryption algorithm
- 128 bit cryptographic key supported
- Five confidentiality modes of operation of AES specified in FIPS
  - Electronic Codebook mode (ECB)
  - Cipher Block Chaining mode(CBC)
  - Cipher Feedback mode (CFB)
  - Output Feedback mode (OFB)
  - Counter mode (CTR)
- 8, 16, 32, 64, and 128-bit data sizes possible in CFB mode

Table of Contents

1 Glossary

The following terms are used throughout this application note document:

Table 1-1. Glossary

Glossary	Description
AES	Advanced Encryption Standard
Cipher	Series of transformations that convert plain text to cipher text using the Cipher Key
Cipher Key	Secret cryptographic key that is used by the Key Expansion Routine to generate a set of Round Keys
Cipher Text	Data output from the Cipher or input to the Inverse Cipher
Inverse Cipher	Series of transformations that converts cipher text to plain text using the Cipher Key
Key Expansion	Routine used to generate a series of Round Keys from the Cipher Key
Nk	Number of 32-bit words comprising the Cipher Key. For AES-128, Nk = 4.
Plain Text	Data input to the Cipher or output from the Inverse Cipher
Round Key	Values derived from the Cipher Key using the Key Expansion Routine
State	Intermediate Cipher result
S-box	Non-linear substitution table used in several byte substitution transformation and in the Key Expansion Routine to perform one-for-one substitution of a byte value.

2 Hardware and Software Setup

The application note has been tested in the following test setup:

- SAM D21 Xplained Pro Board
- Atmel Studio 6.2 SP2 build 1563

3 AES Algorithm

Federal Information Processing Standards Publications (FIPS PUBS) are issued by the National Institute of standards and Technology (NIST). The Advanced Encryption Standard Algorithm (AES) specifies the FIPS approved (FIPS Pub. 197) cryptographic algorithm that can be used to protect electronic data.

AES is a symmetric key algorithm that operates on 128-bit block of input data for a specified number of times. The symmetric key means that same key is used for both encryption and decryption. Encryption process converts the data to unintelligible form called cipher text. Decryption process converts the data back to its original form called plain text from the cipher text.

The key size used for an AES encryption and decryption can be 128, 192, or 256 bits for a fixed input block size of 128 bits. The size of the key determines the number of rounds to be performed on an input block of data. The number of rounds of repetition is as follows:

- 10 rounds of repetition for 128-bit keys
- 12 rounds of repetition for 192-bit keys
- 14 rounds of repetition for 256-bit keys

The following sections explains the AES encryption and decryption algorithm. For more details on the AES standard, refer to the AES standard document.

3.1 AES Encryption

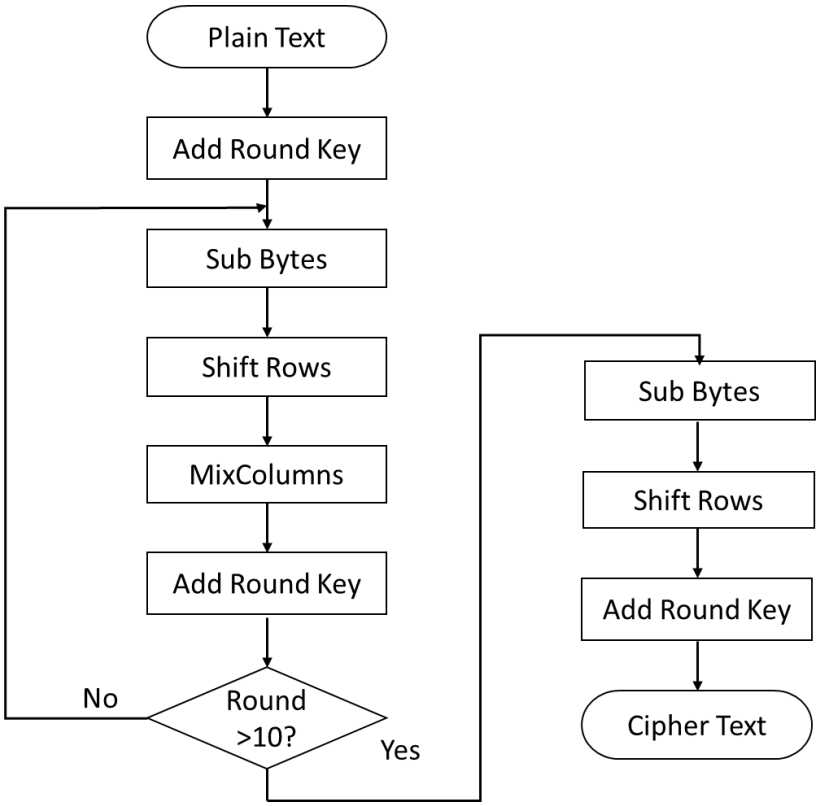
The AES-128 encryption process involves 10 rounds of encryption along with an initial round for the 128 bit data encryption. To begin with, the 128-bit key is expanded into a set of eleven 128-bit round keys using the Key expansion routine. Each of this keys is used for the rounds, finally resulting in the cipher text output.

The initial round in the AES Encryption comprises of the **Add Round key** step in which the plain text is XOR'ed with the Cipher Key. The subsequent 9 rounds goes through four different byte oriented transformations as listed below.

1. Byte substitution using S-box (**Sub Byte**)
2. Shifting rows of the state array by different offsets (**Shift Rows**)
3. Mixing the data within each column **(Mix Columns**)
4. Adding Round key to the state (**Add Round Key**)

In the 10^{th} round, the above steps are repeated excluding the Mix Columns step. Following sections explain each of them in detail. The overall process of AES encryption is illustrated in the figure below:

Figure 3-1. AES Encryption Block diagram



The 128-bit input block of data is processed byte-by-byte and mapped into a 4x4 byte matrix for processing convenience as per the AES standard. Each block of input and the intermediate inputs between the different rounds is mapped into a 4x4 state matrix as shown in the figure below:

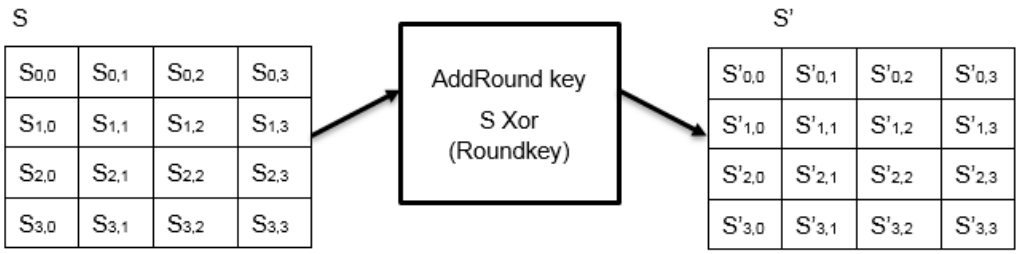
Figure 3-2.

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$

3.1.1 AddRoundKey Transformation

In this transformation, the Round key is added to the State by bitwise XOR operation. The Round key for each round has to be generated from the Key Expansion which is explained in the coming sections.

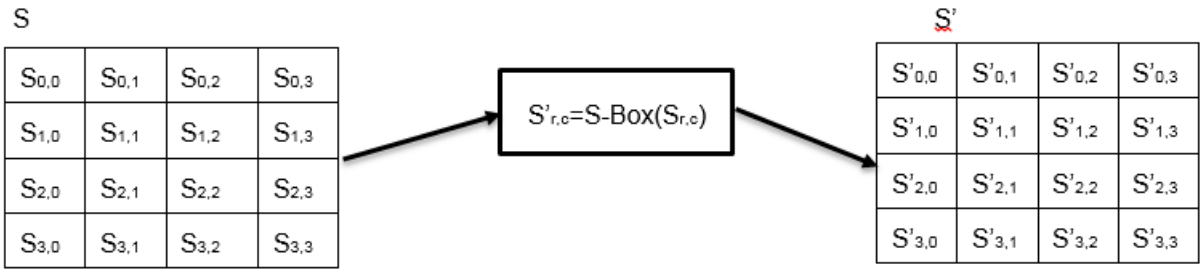
Figure 3-3. AddRoundKey Transformation



3.1.2 SubBytes Transformation

The SubBytes Transformation is a non-linear byte substitution that operates independently on each byte of the state using a substitution table(S-Box). The figure below illustrates the S-Box transformation of a matrix of bytes.

Figure 3-4. SubBytes Transformation



The SubBytes transformation applies S-Box to each byte in the state. Figure shows the S-Box substitution values for the AES-128 Encryption.

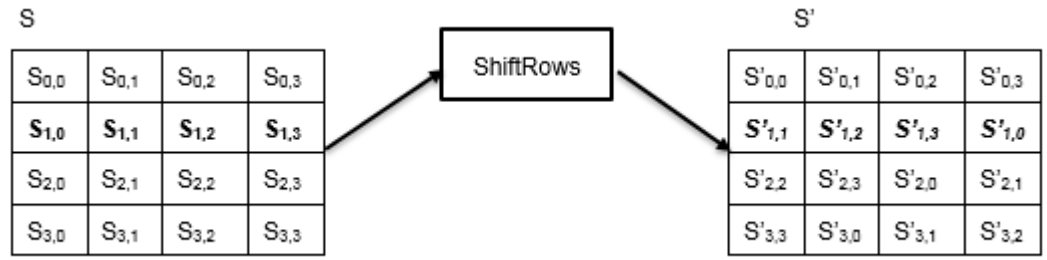
Figure 3-5. S-Box for AES Encryption

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

3.1.3 ShiftRows Transformation

In this transformation, the bytes in the last three rows of the state are cyclically shifted left over different offsets. The elements in the first row are not shifted. The below representation demonstrates the ShiftRows transformation applied on the state array.

Figure 3-6. ShiftRows Transformation

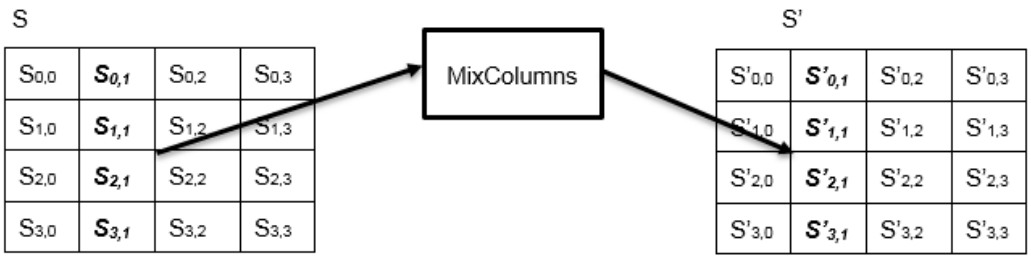


3.1.4 MixColumns Transformation

MixColumns transformation operates column by column on the state matrix applying polynomial transformation on each column. The columns are considered as four-term polynomials over GF(2^8) and multiplied modulo x^4+1 with a fixed polynomial a(x), given by:

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$$

Figure 3-7. MixColumns Transformation



3.1.5 C Implementation

The AES Encryption is implemented in the function **aes_cipher()** in the aes.c. The steps SubBytes and ShiftRows are combined in a single function to optimize the code.

To begin with, one block of the plain text is copied into an intermediate **State** array. In Round 0, the state array is XOR-ed with the Round key 0. The subsequent 9 Rounds are implemented as in the below code snippet:

```
//Round 1 to 9
for (round = 1; round < 10; round ++){
    /* Sub Bytes and Shift Rows */
    SubBytes_ShiftRows(state);
    /* Mix Columns */
    MixColumnns(state);
    /* Add RoundKey */
    AddRoundKey(state,round);
}
```

The final round is performed excluding the MixColumns step to give the Cipher text output.

3.2 Key Expansion

The AES algorithm gets the Cipher Key from the user and uses the Key Expansion Routine to generate the Set of Round keys known as the Key Schedule. For the AES-128 encryption and decryption, the Key Expansion Routine generates a set of eleven 128-bit Round keys from the 128-bit Cipher Key.

The first 4 words (128 bits of first round key) of the expanded key schedule are kept same as the actual cipher key. The remaining 40 words (for remaining 10 round keys) are calculated using the 3 operations – Rotate, Rcon and S-Box.

Rotate In this step, elements in each column in the round key matrix rotated vertically clockwise by 1 byte.

Rcon: This step involves XOR with the Round constant for the words in the position of multiple of Nk.

S-Box: This takes the 4-byte input and applies the S-Box.

3.2.1 C implementation

The key expansion routine is implemented in the function **KeyExpansion** in the aes.c. The cipher key is retained as the key for Round 0. The subsequent bytes in the key schedule are computed by using the Rotate, Rcon and S-box transformations. The overall implementation of the Key expansion routine is as follows:

```
//Retain the initial for round 0
for ( i = 0; i < 16; i++){
    round_key[ i ] = key[ i ];
}

// Compute Key schedule of block size for each round
for ( i = 1; i < (AES_NUM_OF_ROUNDS + 1); i++){
    temp = round_key[ i*16 - 4 ];
    round_key[i*16 +  0] = sbox[ round_key[i*16 - 3] ] ^ round_key[(i-1)*16 + 0] ^ RCon[ i ];
    round_key[i*16 +  1] = sbox[ round_key[i*16 - 2] ] ^ round_key[(i-1)*16 + 1];
    round_key[i*16 +  2] = sbox[ round_key[i*16 - 1] ] ^ round_key[(i-1)*16 + 2];
    round_key[i*16 +  3] = sbox[ temp ] ^ round_key[ (i-1)*16 + 3 ];

    round_key[i*16 +  4] = round_key[(i-1)*16 + 4] ^ round_key[i*16 + 0];
    round_key[i*16 +  5] = round_key[(i-1)*16 + 5] ^ round_key[i*16 + 1];
    round_key[i*16 +  6] = round_key[(i-1)*16 + 6] ^ round_key[i*16 + 2];
    round_key[i*16 +  7] = round_key[(i-1)*16 + 7] ^ round_key[i*16 + 3];

    round_key[i*16 +  8] = round_key[(i-1)*16 + 8] ^ round_key[i*16 + 4];
    round_key[i*16 +  9] = round_key[(i-1)*16 + 9] ^ round_key[i*16 + 5];
    round_key[i*16 + 10] = round_key[(i-1)*16 +10] ^ round_key[i*16 + 6];
    round_key[i*16 + 11] = round_key[(i-1)*16 +11] ^ round_key[i*16 + 7];

    round_key[i*16 + 12] = round_key[(i-1)*16 +12] ^ round_key[i*16 + 8];
    round_key[i*16 + 13] = round_key[(i-1)*16 +13] ^ round_key[i*16 + 9];
    round_key[i*16 + 14] = round_key[(i-1)*16 +14] ^ round_key[i*16 +10];
    round_key[i*16 + 15] = round_key[(i-1)*16 +15] ^ round_key[i*16 +11];
}
```

3.3 AES Decryption

The AES-128 decryption process involves similar number of rounds as the AES-128 Encryption process with corresponding inverse transformations.

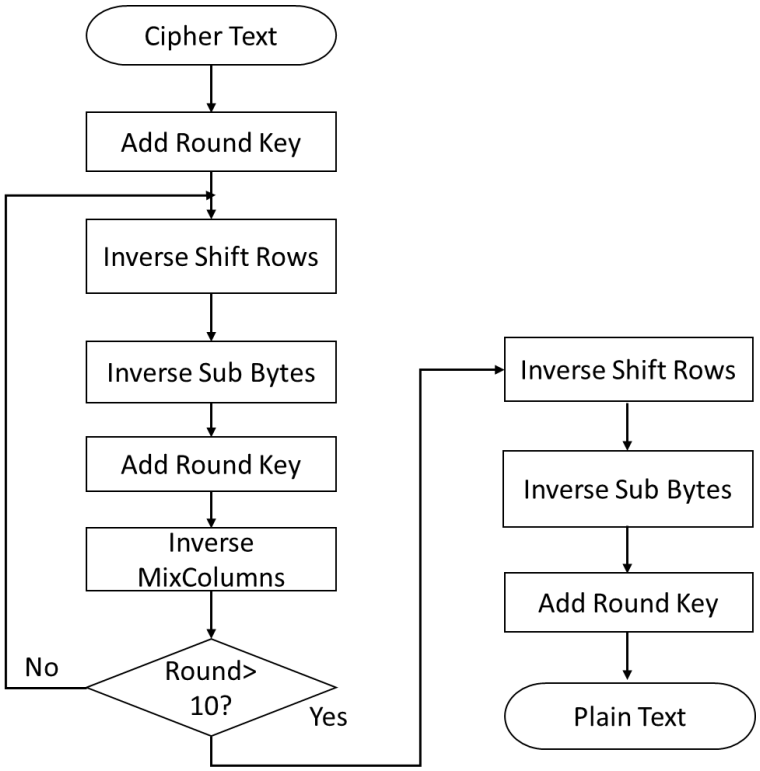
The initial round includes only the AddRoundKey step which is the same as in AES-128 Encryption.

The inverse transformations for the subsequent rounds are as below:

1. Inverse Shift Rows
2. Inverse SubBytes
3. Add Round Key
4. Inverse Mix Columns

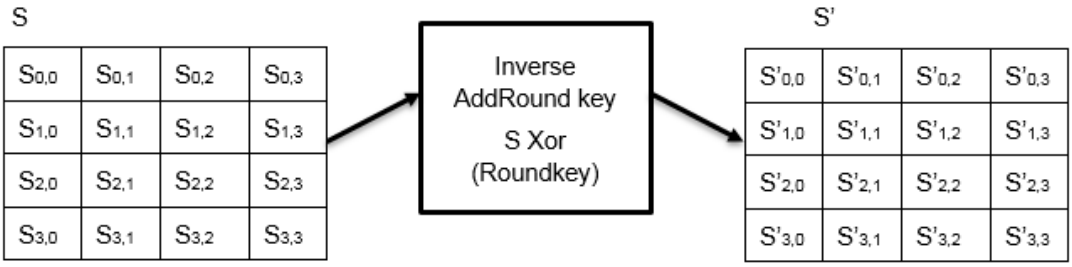
In the 10th round, the above steps are repeated excluding the Mixcolumns step. The overall process of AES decryption is illustrated in the figure below:

Figure 3-8. AES Decryption Block diagram



3.3.1 Inverse of AddRoundKey

This is same as the AddRoundKey transformation as this involves only XOR operation between an input and output blocks.



3.3.2 Inverse SubBytes Transformation

This is the inverse of the SubBytes Transformation, in which the inverse S-Box is applied to each element in the State. The inverse S-Box used in this transformation is as below:

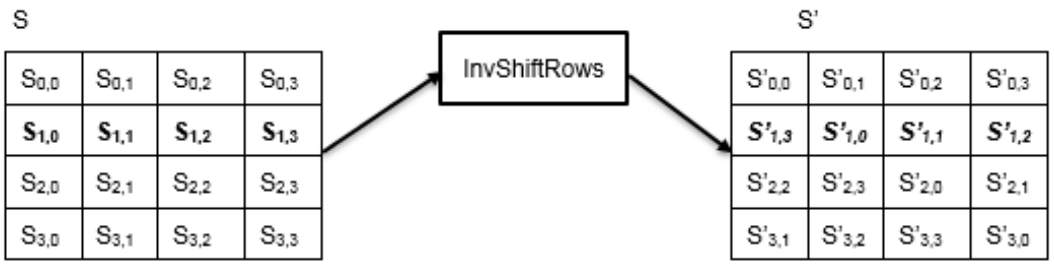
Figure 3-9. Inverse S-Box

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

3.3.3 Inverse ShiftRows Transformation

This is the inverse of the ShiftRows transformation. The bytes in the last three rows are cyclically shifted to the right over a different number of bytes.

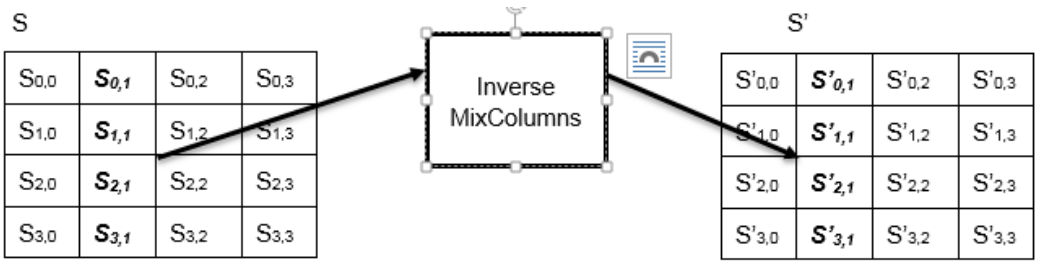
Figure 3-10. Inverse ShiftRows Transformation



3.3.4 Inverse MixColumns Transformation

This is an inverse of MixColumns transformation. This operates on a column-by-column basis on the state matrix applying polynomial transformation.

Figure 3-11. Inverse MixColumns Transformation



3.3.5 C Implementation

The AES decryption is implemented in the function **aes_inverse_cipher()** in the aes.c. The steps inverse SubBytes and inverse ShiftRows are combined in a single function to optimize the code.

To begin with, one block of the cipher text is copied into an intermediate **State** array. In Round 0, the state array is XOR-ed with the Round key 0. The subsequent 9 Rounds are implemented as in the below code snippet:

```
// Round 9 to 1
for(round = 9; round>0; round--){
    /* Inverse Sub Bytes and Shift Rows */
    Inv_ShiftRows_SubBytes(state);
    /* Inverse Add Round Key */
    Inv_AddRoundKey(state, round);
    /* Inverse Mix Columns */
    Inv_MixColumns(state, round);
}
```

The final round is performed excluding the MixColumns step to give the plain text output.

4 Block Cipher Modes of Operation

In FIPS SP800-38A, NIST recommends five confidentiality modes of operation listed below for use with an underlying symmetric key block cipher algorithm. Mode of operation is an algorithm that describes how to repeatedly apply a cipher's single-block operation to encrypt data larger than a block.

- Electronic Codebook mode (ECB)
- Cipher Block Chaining mode (CBC)
- Cipher Feedback mode (CFB)
- Output Feedback mode (OFB)
- Counter mode (CTR)

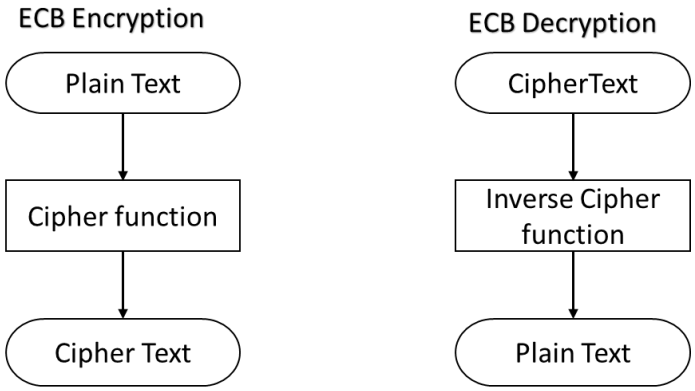
The following sections explain the algorithm of each modes and their C implementation in this application note. The implementation is done in crypt.c/.h file.

The standard document (FIPS Pub 800-38A) should be referred for detailed description of mathematic description of each modes.

4.1 Electronic Codebook Mode

In this mode, the input is divided in to separate block of 128 bits. Each block is encrypted/decrypted independently. In ECB encryption, the cipher function is directly applied to each input block resulting in a cipher text. In ECB decryption, the inverse cipher function is applied to each block to retrieve the plain text. It is illustrated in the Figure 4-1.

Figure 4-1. Electronic Codebook Mode



ECB is the simplest mode of all. As it involves direct encryption of each block, it does not provide better confidentiality and so it is not recommended for cryptographic protocols.

4.1.2 C Implementation of ECB mode

As mentioned, the implementation is very direct that input is divided into each block of 128 bits. The number of input blocks (input_block_size) are calculated from the size of the input plain text. Forward and inverse cipher function is applied to them to get the cipher text and plain text respectively as below in the loop for the input_block_size.

```
void ecb_encrypt(uint8_t *plainText, uint8_t *cipherText, uint32_t size)
{
    uint32_t input_block_size = 0;
    uint8_t block_count = 0;

    //Calculate the number of input blocks
    input_block_size = size/AES_BLOCK_SIZE;

    for(block_count = 0;block_count < input_block_size; block_count++){
        //Perform forward cipher for the input blocks
        aes_cipher(plainText+(block_count * AES_BLOCK_SIZE),
                           cipherText+(block_count * AES_BLOCK_SIZE));
    }

}
```

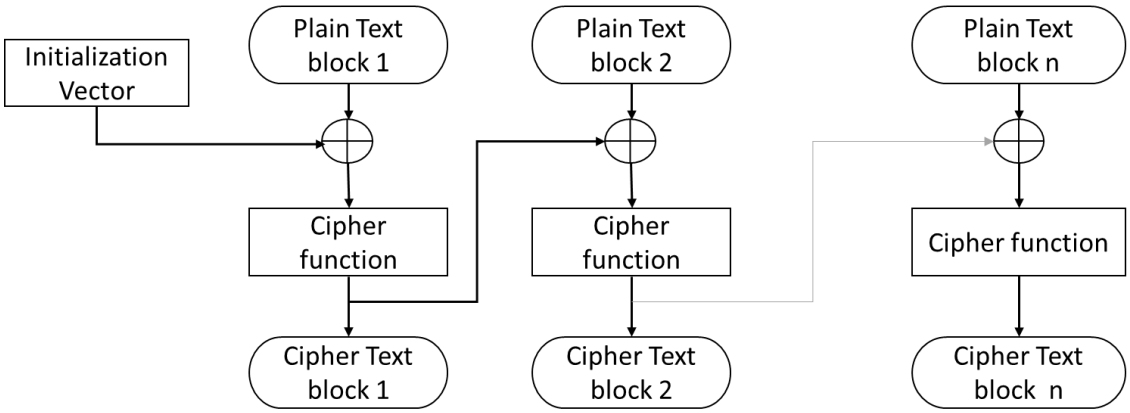
4.2 The Cipher Block Chaining Mode (CBC)

As the name implies, this mode features chaining of plain text with previous cipher text blocks. This mode requires an Initialization Vector (IV) to combine with the first block. The IV is not secret but should be unpredictable. I.e. for any given plaintext, it must not be possible to predict the IV that will be associated to the plaintext in advance of the generation of the IV. Various methods of generation of Initialization vector is explained in Appendix C of FIPS Pub 800-38A.

4.2.1 CBC Encryption

In the CBC Encryption, the first block of plain text is XOR-ed with IV. This forms the input block for cipher function which results in the first cipher text block. This cipher text then XOR-ed with the next plain text block and goes through forward cipher to form the second cipher text. Thus further cipher text blocks are formed by applying forward cipher after XOR-ing the respective block of plain text with the previous block's cipher text. Because of this chaining, one bit change in the plain text or IV could affect all following cipher text blocks.

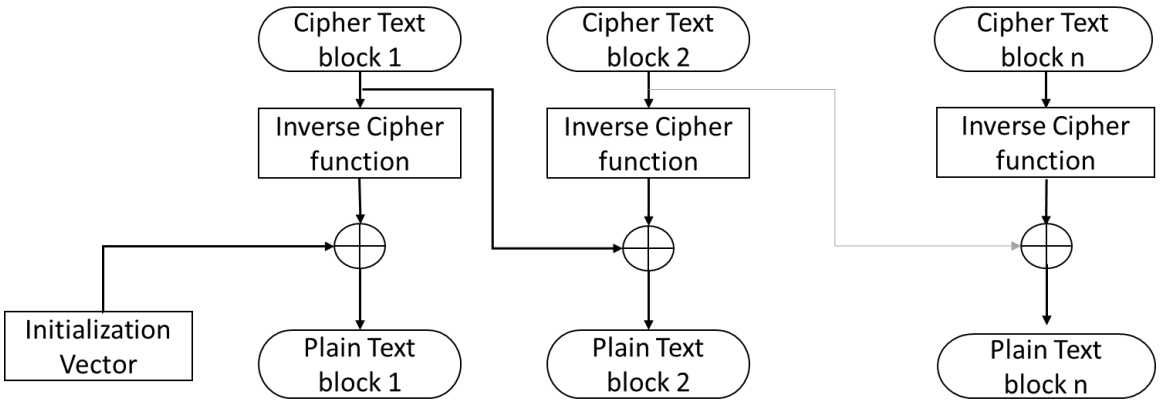
Figure 4-2. CBC Encryption



4.2.2 CBC Decryption

In CBC decryption, the inverse cipher is applied over input cipher text and the resulting output block is XOR-ed with IV to get the first block of plain text. The second cipher text block goes through inverse cipher function and the resulting output block is XOR-ed with previous cipher text block to get the second plain text block. Further plain text blocks are retrieved by applying inverse cipher over respective cipher text block and then XOR-ed the resulting output block with its previous cipher text block.

Figure 4-3. CBC Decryption



4.2.3 C Implementation of CBC mode

For encryption:

The input_block_size is calculated to get the number of input blocks. The initials input block is formed by XOR between IV and plain text as below

```
//Calculate input block size
input_block_size = (size / AES_BLOCK_SIZE );
```

Forward cipher is applied to the input block to get the corresponding cipher text.

```
for(block_count = 0;block_count < input_block_size; block_count++){

    //Sending initialization vector to the cipher function to get further output blocks
    aes_cipher(plainText +(block_count * AES_BLOCK_SIZE),
                    cipherText +(block_count * AES_BLOCK_SIZE));
```

The next input block is formed by XOR of next block of plain text with the previously formed cipher text. This is done until last block is reached.

```
if(block_count != (input_block_size -1)){

    //Move the previous cipher block to feedback for next round
    for(byte_count = 0; byte_count < AES_BLOCK_SIZE; byte_count++){
        // XOR plain text with previous cipher text to form input for next block
        plainText[byte_count + ((block_count + 1)*AES_BLOCK_SIZE)] =
                    plainText[byte_count + ((block_count + 1)*AES_BLOCK_SIZE)] ^
                    cipherText[byte_count + (block_count* AES_BLOCK_SIZE)];
```

For Decryption:

To retrieve the first plain text block, the inverse cipher function is applied to the first block of cipher text. The first block of plain text is formed by XOR of resulting output block with IV as below

```
//Round0
aes_inverse_cipher(cipherText, plainText);


for(byte_count = 0; byte_count < AES_BLOCK_SIZE; byte_count++){
    /* Retrieve the first block of input data by XOR round 0
     * output with Initialization vector
     */
    plainText[byte_count] = plainText[byte_count] ^ init_vector[byte_count];
}
```

The further plain texts are formed by XOR of resulting output block with previous cipher text for specified number of blocks.

```
for(block_count = 1;block_count < input_block_size; block_count++){

    //Perform inverse cipher for the input blocks
    aes_inverse_cipher(cipherText+(block_count * AES_BLOCK_SIZE),
            plainText+(block_count * AES_BLOCK_SIZE));

    for(byte_count = 0; byte_count < AES_BLOCK_SIZE; byte_count++){
        /* XOR output of previous inverse cipher function with previous cipher text to
         * form the input for next block
         */
        plainText[byte_count + (block_count * AES_BLOCK_SIZE)] =
                plainText[byte_count + (block_count * AES_BLOCK_SIZE)] ^
                cipherText[byte_count + ( (block_count - 1) * AES_BLOCK_SIZE)];
```

4.3 Cipher Feedback mode

The Cipher Feedback mode features the feedback of the successive cipher text elements into the input block of the forward cipher function to generate output blocks that are XOR-ed with the plain text to produce the cipher text.

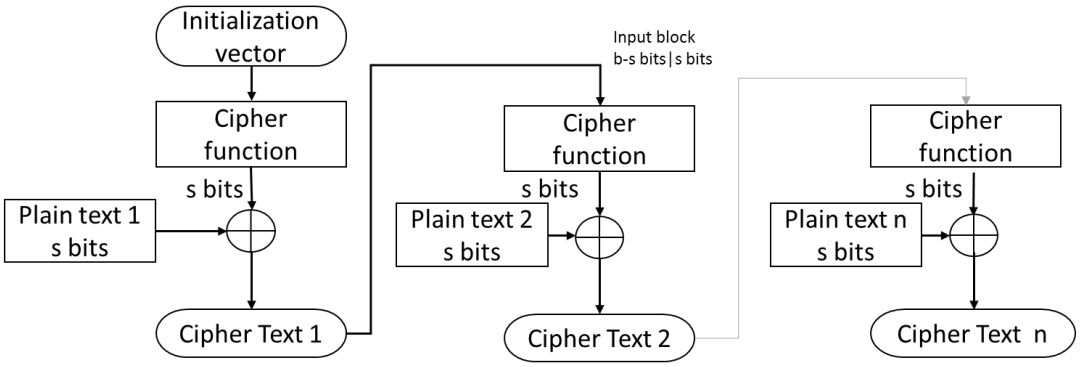
This mode requires a unique Initialization vector in addition to the plaintext and cipher key. The CFB mode also needs an integer parameter ***s***, where ***s*** should be less than or equal to 128 bits. In the CFB mode, each plain text and cipher text block contain s bits. Correspondingly the modes are named as 1-bit CFB, 8-bit CFB, 64-bit CFB, 128-bit CFB, etc.

4.3.1 CFB Encryption

This Initialization vector (IV) is passed as the first input block to the AES Forward Cipher function to produce the first output block. To generate the cipher text output, the first ***s*** bits in the plain text are XOR-ed with the first ***s*** bits in the output block. The remaining ***b-s*** bits in the output block are discarded (b=128).

To generate the second input block, the ***b-s*** LSBs in the IV are concatenated with the s bits from the cipher text output of the first block. The process is repeated with successive input blocks until the entire plaintext is converted into cipher text output. The encryption process of the CFB mode can be illustrated as below:

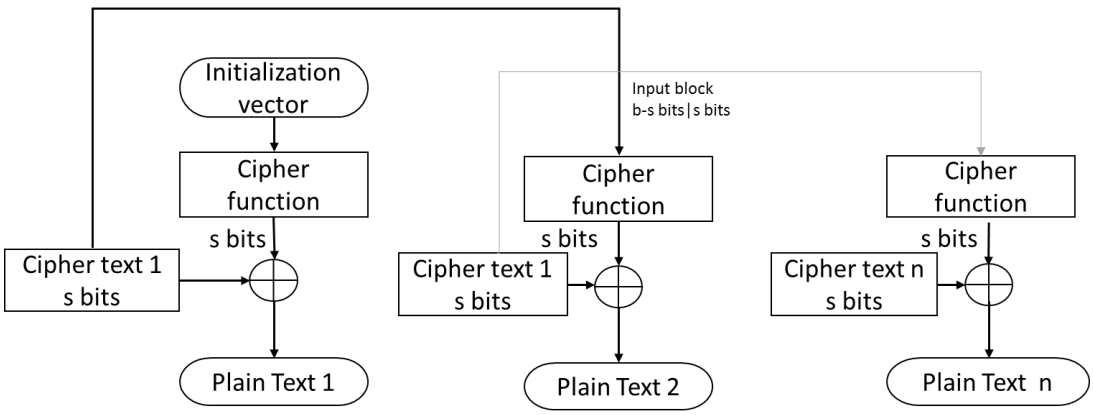
Figure 4-4. CFB Encryption



4.3.2 CFB Decryption

In CFB Decryption, the Initialization vector if given as first input block, and subsequent input blocks are generated by concatenating ***b-s*** bits of IV with ***s*** bits from the cipher text. The input block is applied to the forward cipher function to obtain the output block. Then the ***s*** bits of the cipher text are XOR-ed with ***s*** bits of the ciphertext. The Decryption process of the CFB mode can be illustrated as below.

Figure 4-5. CFB Decryption



4.3.3 C Implementation

The CFB encryption and the decryption functions have been implemented in the crypt.c.

Encryption:

The encryption function takes the plaintext, array for cipher text, initialization vector, CFB mode in bits and size of the plain text as inputs. To start with, the number of input blocks is calculated from the size parameter. Also, the cfb_byte is calculated from the CFB mode. This is because all the operations are performed in byte level throughout the code.

The initialization vector is given to the aes_cipher function for generating the first output block. Then, depending upon the CFB mode, certain number (cfb_byte) of bytes in the plain text are XOR-ed with the output block to get the cipher text output. The implementation of the first encryption block is as below:

```
 for(byte_count = 0; byte_count < AES_BLOCK_SIZE; byte_count++){
     //Moving initial vector to a temp array
    init_vector_temp[byte_count] = init_vector[byte_count];
}

//Forward cipher of initial vector 1
aes_cipher(init_vector_temp, cipherText);

//XOR-ing plain text 's' bits with output block s bits
for(byte_count = 0; byte_count < cfb_byte; byte_count++){
    cipherText[byte_count] = plainText[byte_count] ^ cipherText[byte_count];
}
```

For the subsequent blocks, the input block (init_vector_temp) is generated by concatenating the bits from the IV and the cipher text and given to the forward cipher function. Then the cipher text is generated by XOR-ing the bytes of plain text and output block. The implementation of the encryption is as follows:

```
  //Encryption of remaining blocks
  for(block_count = 1;block_count < ((AES_BLOCK_SIZE * input_block_size) / cfb_byte); block_count++){

      index1 = 0;
      //Preparation of Initialization vector for the next round
      for (byte_count = 0; byte_count < AES_BLOCK_SIZE; byte_count++){

          if(block_count < ((AES_BLOCK_SIZE/cfb_byte))+1){

              /* For the first few blocks where initialization vector and cipher output
               * is used to generate the init_vector_temp
               */
              if(byte_count < (AES_BLOCK_SIZE - (block_count + cfb_byte - 1))){
                   init_vector_temp[byte_count] =
                          init_vector[byte_count + (cfb_byte + block_count-1)];
              }else{
                  init_vector_temp [byte_count] = cipherText[index1++];
              }
              if (block_count == (AES_BLOCK_SIZE / cfb_byte)){
                  index2_flag = true;
              }

          }else{

              /* For the remaining blocks where init_vector_temp is generated from the cipher
               * output from the previous block. AES Standard FIPS SP800-38A explains more on
               * generation of init_vector for each block in different CFB modes
               */
              init_vector_temp[byte_count] =  cipherText[(cfb_byte * index2) + byte_count];

          }

      }

      if (index2_flag == true){
          index2 = index2+1;
      }

      //Sending initialization vector to the cipher function to get 16 byte output block
      aes_cipher(init_vector_temp, (cipherText + (block_count * cfb_byte)));

      for(byte_count = 0; byte_count < cfb_byte; byte_count++){
          //XOR-ing cipher text 's' bits with output block s bits
          cipherText[(byte_count + (block_count * cfb_byte))] =
                  plainText[(byte_count + (block_count * cfb_byte))] ^
                  cipherText[byte_count + (block_count * cfb_byte)];
       }

  } //End of all blocks
```

Decryption:

The decryption process is similar to the encryption where the first step is to generate the input block from the initialization vector and the cipher text array. Then the input block is applied to the forward cipher function to get the output block. The ***s*** bits of the output block are then XOR-ed with the cipher text to get the plain text output. The implementation of the decryption function is as follows:

Decryption of first block:

```
for(byte_count = 0; byte_count<AES_BLOCK_SIZE; byte_count++){
    //Moving initial vector to a temp array
    init_vector_temp[byte_count] = init_vector[byte_count];
}

//Forward cipher of initial vector 1
aes_cipher(init_vector_temp, plainText);

//XOR-ing cipher text 's' bits with output block s bits
for(byte_count = 0; byte_count < cfb_byte; byte_count++){
    plainText[byte_count] = cipherText[byte_count] ^ plainText[byte_count];
}
```

Decryption of remaining blocks:

```
 //Decryption of remaining blocks
 for(block_count = 1;block_count < ((AES_BLOCK_SIZE * input_block_size) / cfb_byte); block_count++){

     index1 = 0;
     //Preparation of Initialization vector for the next round
     for (byte_count = 0; byte_count < AES_BLOCK_SIZE; byte_count++){

         if(block_count < ((AES_BLOCK_SIZE/cfb_byte))+1){

             /* For the first few blocks where initialization vector and cipher output
              * is used to generate the init_vector_temp
              */
             if(byte_count < (16 - (block_count + cfb_byte -1))){
                 init_vector_temp[byte_count] =
                         init_vector[byte_count + (cfb_byte + block_count - 1)];
             }else{
                 init_vector_temp [byte_count] = cipherText[index1++];
             }
             if (block_count == (AES_BLOCK_SIZE / cfb_byte)){
                 index2_flag = true;
             }

         }else{

             /* For the remaining blocks where init_vector_temp is generated from the cipher
              * output from the previous block. AES Standard FIPS SP800-38A explains more on
              * generation of init_vector for each block in different CFB modes
              */
             init_vector_temp[byte_count] =  cipherText[(cfb_byte * index2) + byte_count];

         }

     }
     if (index2_flag == true){
         index2 = index2+1;
     }

     //Sending initialization vector to the cipher function to get 16 byte output block
     aes_cipher(init_vector_temp,  (plainText+(block_count * cfb_byte)));

     //XOR-ing cipher text 's' bits with output block s bits
     for(byte_count = 0; byte_count < cfb_byte; byte_count++){
         plainText[(byte_count + (block_count * cfb_byte))] =
                 cipherText[(byte_count + (block_count * cfb_byte))]^
                 plainText[(byte_count + (block_count * cfb_byte))];
     }

 }
```

4.4 Output Feedback Mode

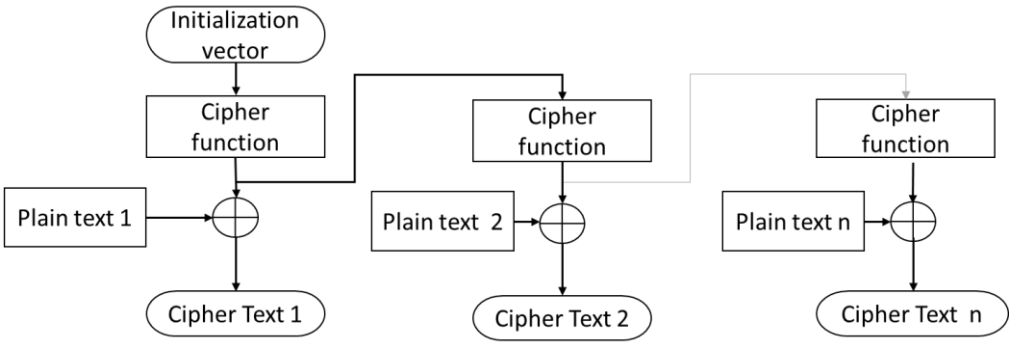
The Output Feedback mode features the feedback of the output blocks of the forward cipher function in each block into the input block of the forward cipher function of the successive blocks. Like the CFB and CBC mode, this mode also requires a unique Initialization vector in addition to the plaintext and cipher key. In addition, the OFB mode requires that the IV is a nonce, that is, the IV must be unique for each execution of the mode with the given key.

4.4.1 OFB Encryption

As in the CFB mode, the IV is given as the first input block. This is applied to the forward cipher function to generate the first output block. The output block is XOR-ed with the plain text to generate the first block of cipher text output.

The first output block is then given to the forward cipher function to generate the second output block, which is then XOR-ed with the plain text to generate the cipher text second block. The second output block is then applied to the forward cipher function to generate the third output block and so on. This can be represented as in the block diagram below.

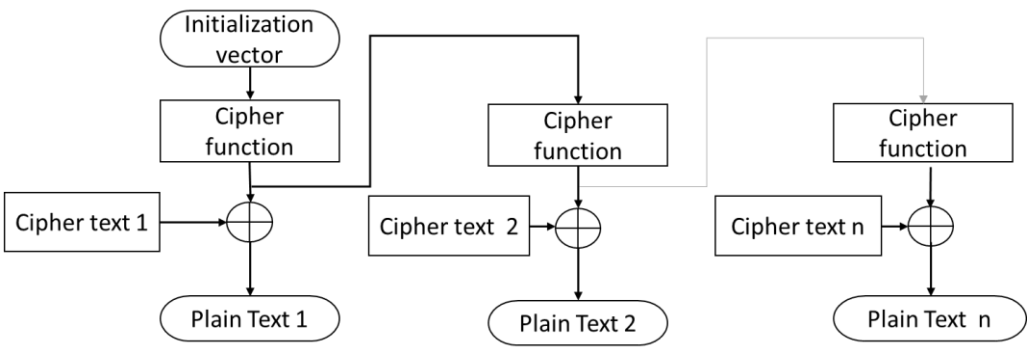
Figure 4-6. OFB Encryption



4.4.2 OFB Decryption

In OFB Decryption, the IV is given as the first input block to the forward cipher function to generate the first output block. The first output block is XOR-ed with the cipher text to generate the first plain text block. The first output block is given to the Forward cipher function to generate the second output block, then XOR-ed with the cipher text to generate the second block in plain text and so on. The OFB decryption can be illustrated as below.

Figure 4-7. OFB Decryption



The OFB mode requires a unique IV for every message that is ever encrypted under the given key. If the same IV is used for the encryption of more than one message, then the confidentiality of those messages may be compromised.

4.4.3 C Implementation

The OFB mode has been implemented in the crypt.c file.

The OFB encryption function takes the plain text array, array for the cipher text, initialization vector, and the size as inputs. The number of blocks in the input plain text data is calculated from the size parameter.

For the encryption of the first block, the initialization vector is applied to the aes_cipher function to obtain the first output block, which is then XOR-ed with the plain text to generate the cipher text.

Similar process is applicable for the decryption. The implementation of OFB encryption and decryption is as follows:

```
void ofb_encrypt(uint8_t *plainText, uint8_t *cipherText, uint8_t *init_vector, uint32_t size)
{
    uint32_t input_block_size = 0;
    uint8_t block_count = 0, byte_count = 0;
    uint8_t input_block[16] = {0};

    //Calculate the number of blocks
    input_block_size = (size / AES_BLOCK_SIZE );

    for(byte_count = 0; byte_count < AES_BLOCK_SIZE; byte_count++){
        //Moving initial vector to a temp array
        input_block[byte_count] = init_vector[byte_count];
    }

    //Encryption of remaining blocks
    for(block_count = 0;block_count < input_block_size; block_count++){

        //Sending initialization vector to the cipher function to get 16 byte output block
        aes_cipher(input_block,  (cipherText+(block_count * AES_BLOCK_SIZE)));

        //XOR cipher output with  plain text to complete the encryption for a block
        for(byte_count = 0; byte_count < AES_BLOCK_SIZE; byte_count++){
            cipherText[byte_count+ (block_count * AES_BLOCK_SIZE)] =
                    plainText[byte_count + (block_count * AES_BLOCK_SIZE)]^
                    cipherText[byte_count+ (block_count * AES_BLOCK_SIZE)];
        }
    }
}
```

```
void ofb_decrypt(uint8_t *cipherText, uint8_t *plainText, uint8_t *init_vector, uint32_t size)
{
    uint32_t input_block_size = 0;
    uint8_t block_count = 0, byte_count = 0;
    uint8_t input_block[16] = {0};

    //Calculate the number of input blocks
    input_block_size = size / AES_BLOCK_SIZE;

    for(byte_count = 0; byte_count<AES_BLOCK_SIZE; byte_count++){
        //Moving initial vector to a temp array
        input_block[byte_count] = init_vector[byte_count];
    }

    //Decryption of remaining blocks
    for(block_count = 0;block_count < input_block_size; block_count++){

        //Sending initialization vector to the cipher function to get 16 byte output block
        aes_cipher(input_block, (plainText+(block_count * AES_BLOCK_SIZE)));

        //XOR cipher output with previous cipher text to decrypt the input data
        for(byte_count = 0; byte_count < AES_BLOCK_SIZE; byte_count++){
            plainText[byte_count + (block_count * AES_BLOCK_SIZE)] =
                    cipherText[byte_count + (block_count * AES_BLOCK_SIZE)]^
                    plainText[byte_count + (block_count * AES_BLOCK_SIZE)];
        }
    }

}
```

4.5 The Counter Mode (CTR)

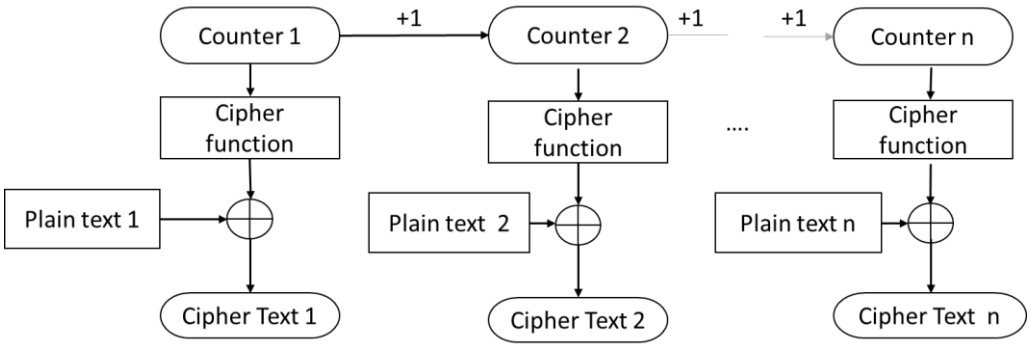
The Counter (CTR) mode is a confidentiality mode that features the application of the forward cipher to a set of input blocks, called counters, to produce a sequence of output blocks that are exclusive-ORed with the plaintext to produce the ciphertext, and vice versa.

In both CTR encryption and CTR decryption, the forward cipher functions can be performed in parallel. Similarly, the plaintext block that corresponds to any particular ciphertext block can be recovered independently from the other plaintext blocks if the corresponding counter block can be determined. Moreover, the forward cipher functions can be applied to the counters prior to the availability of the plaintext or ciphertext data. The sequence of counters must have the property that each block in sequence is different from every other block. Various methods of counter generation are explained in Appendix B of FIPS Pub 800-38A.

4.5.1 CTR Encryption

Encryption involves the application of forward cipher over a block of data called counters which is then XOR-ed with plain text to generate the respective cipher text.

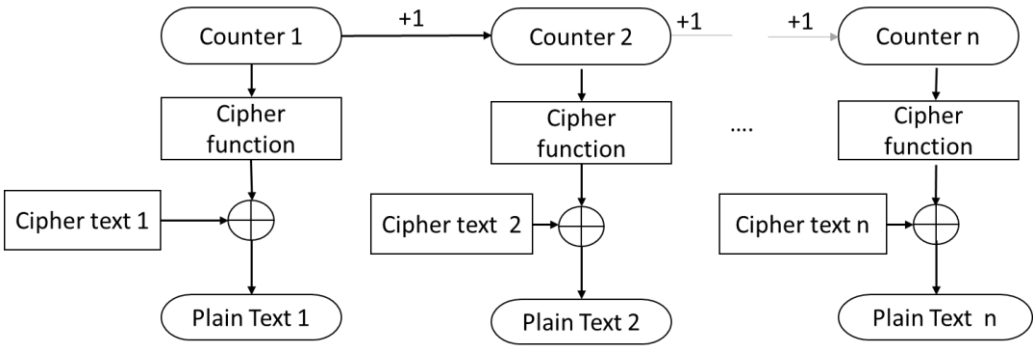
Figure 4-8. Counter mode Encryption



4.5.2 CTR Decryption

For decryption, the forward cipher is applied over counters as in encryption. Then the cipher text is XOR-ed with resulting output block to retrieve the plain text.

Figure 4-9. Counter mode Decryption



4.5.3 Generation of counter blocks

This section explain the counter generation method used in this application note. As mentioned earlier AES – CTR requires unique value for each block and communicate this value for the decryption. This is more similar to IV except that it keeps changing for each block. Counter is divided in to three field called

- Nonce – 32bit
- Initialization Vector – 64bit
- Counter block – 32bit

The counter is incremented for each block and concatenated with Nonce and IV to form 128 bit vector input for subsequent blocks. As counter is 32-bit, the same key will be repeated after 2^{32} bit rounds. So with 32 bit counter 2^{31} block of data can be encrypted effectively.

4.5.4 C Implementation of CTR mode

The counter block contains:

```
//Counter block structure for AES-CTR mode
typedef struct{
    uint32_t nonce;
    uint64_t i_vector;
    uint32_t counter;
} ctr_blk_t;
```

The IV, nonce and counter are concatenated to form the input block and applied to the forward cipher as below:

```
void ctr_inc_counter(ctr_blk_t *counter)
{
    //Append Nonce to 1st 4 bytes of counter block
    counter_block[0] = ((counter->nonce) >> 24) & 0xFF;
    counter_block[1] = ((counter->nonce) >> 16) & 0xFF;
    counter_block[2] = ((counter->nonce) >> 8) & 0xFF;
    counter_block[3] = ((counter->nonce) >> 0) & 0xFF;
    //Append IV to next 8 bytes of counter block
    counter_block[4] = ((counter->i_vector) >> 56) & 0xFF;
    counter_block[5] = ((counter->i_vector) >> 48) & 0xFF;
    counter_block[6] = ((counter->i_vector) >> 40) & 0xFF;
    counter_block[7] = ((counter->i_vector) >> 32) & 0xFF;
    counter_block[8] = ((counter->i_vector) >> 24) & 0xFF;
    counter_block[9] = ((counter->i_vector) >> 16) & 0xFF;
    counter_block[10] = ((counter->i_vector) >> 8) & 0xFF;
    counter_block[11] = ((counter->i_vector) >> 0) & 0xFF;
    //Increment next counter
    counter->counter += 1;
    //Append Initial Counter value to remaining 4 bytes of counter block
    counter_block[12] = ((counter->counter) >> 24) & 0xFF;
    counter_block[13] = ((counter->counter) >> 16) & 0xFF;
    counter_block[14] = ((counter->counter) >> 8) & 0xFF;
    counter_block[15] = ((counter->counter) >> 0) & 0xFF;
}
```

For CTR mode, both encryption and decryption follows the same as explained earlier. So there is only one function for both encryption and decryption.

```
void ctr_encrypt_decrypt (uint8_t *input_block , uint8_t *output_block, ctr_blk_t *counter, uint32_t size)
{
    uint32_t block_count = 0, input_block_size = 0;
    uint8_t byte_count = 0;

    //Calculate input block size
    input_block_size = (size / AES_BLOCK_SIZE );

    //Move the initial counter to local counter block
    //Append Nonce to first 4 bytes of counter block
    counter_block[0] = ((counter->nonce) >> 24) & 0xFF;
    counter_block[1] = ((counter->nonce) >> 16) & 0xFF;
    counter_block[2] = ((counter->nonce) >> 8) & 0xFF;
    counter_block[3] = ((counter->nonce) >> 0) & 0xFF;
    //Append IV to next 8 bytes of counter block
    counter_block[4] = ((counter->i_vector) >> 56) & 0xFF;
    counter_block[5] = ((counter->i_vector) >> 48) & 0xFF;
    counter_block[6] = ((counter->i_vector) >> 40) & 0xFF;
    counter_block[7] = ((counter->i_vector) >> 32) & 0xFF;
    counter_block[8] = ((counter->i_vector) >> 24) & 0xFF;
    counter_block[9] = ((counter->i_vector) >> 16) & 0xFF;
    counter_block[10] = ((counter->i_vector) >> 8) & 0xFF;
    counter_block[11] = ((counter->i_vector) >> 0) & 0xFF;
    //Append Initial Counter value to remaining 4 bytes of counter block
    counter_block[12] = ((counter->counter) >> 24) & 0xFF;
    counter_block[13] = ((counter->counter) >> 16) & 0xFF;
    counter_block[14] = ((counter->counter) >> 8) & 0xFF;
    counter_block[15] = ((counter->counter) >> 0) & 0xFF;

    //Run Encryption for input blocks
    for(block_count = 0; block_count < input_block_size; block_count++){
        if (0 != block_count)
            ctr_inc_counter(counter);

        //Forward cipher of counter
        aes_cipher(counter_block, output_block + (block_count * AES_BLOCK_SIZE));

        //XOR of plain text with the output block
        for (byte_count =0; byte_count < AES_BLOCK_SIZE; byte_count++){
            output_block[byte_count + (block_count * AES_BLOCK_SIZE)] =
                    input_block[byte_count + (block_count * AES_BLOCK_SIZE)] ^
                    output_block[byte_count + (block_count * AES_BLOCK_SIZE)];

        }

    }

}
```

5 AES -128 Example Implementation

The application note comes with the source containing the AES library and demo example. This section gives short description about the demo application of AES-128 covering all 5 modes.

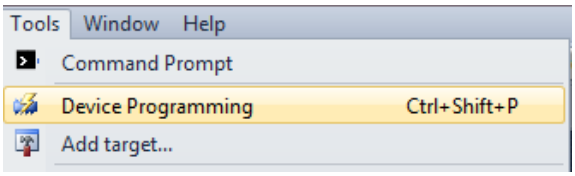
- AES-128 and five confidentiality modes are implemented at two levels. The AES algorithm explained in chapter 3 of this application note is implemented in aes.c/aes.h file.
- The five confidentiality modes are implemented in crypt.c/crypt.h file
- The example is implemented in a way that 64 bytes (i.e. 16 input blocks) of plain text are encrypted and decrypted using all modes separately
- The decrypting message can be viewed in the terminal window
- From the result if the decrypted data is same as the plain text, this conforms the working of each mode
- The modes can be independently enabled or disabled in conf_example.h as below. By default all the modes are enabled

```
/* Set to true to enable respective mode
 * set to false to disable the respective mode
 */
//Enable/Disable ECB mode
#define AES_ECB    true

//Enable/Disable CBC mode
#define AES_CBC    true

//Enable/Disable CFB mode
#define AES_CFB    true

//Enable/Disable OFB mode
#define AES_OFB    true

//Enable/Disable CTR mode
#define AES_CTR    true
```

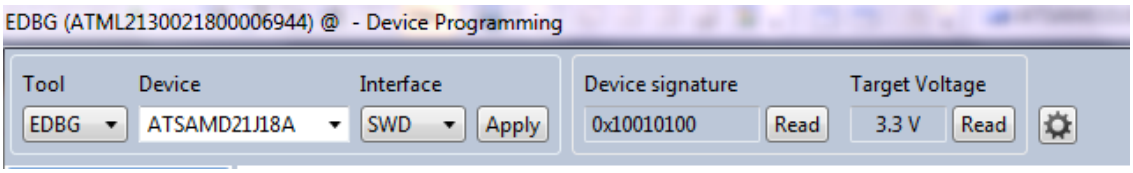
6 Execution of Example in Atmel Studio

The firmware corresponding to this application note comes with the Atmel Software Framework and it can be imported from Atmel Studio as well. The steps below explain the execution of this application.

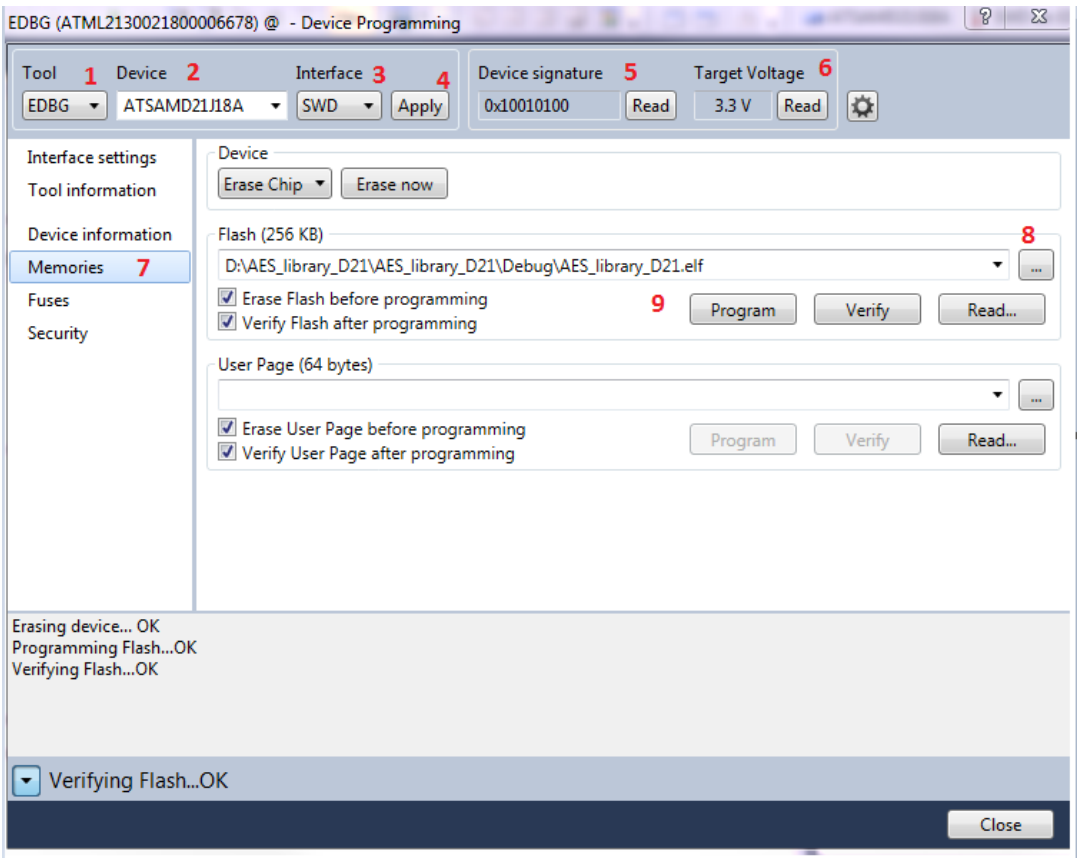
1. Import the example in Atmel Studio from “File → New → Example Project → AES Software Library Demo – SAM D21 Xplained Pro”
2. To build the project, go to Atmel Studio -> Build -> Build Solution
3. Open Terminal window with EDBG COM port with configuration 115200, no parity, one stop bit and hardware control none.
4. Go to ‘Tools -> Device Programming’ Window in Atmel Studio.



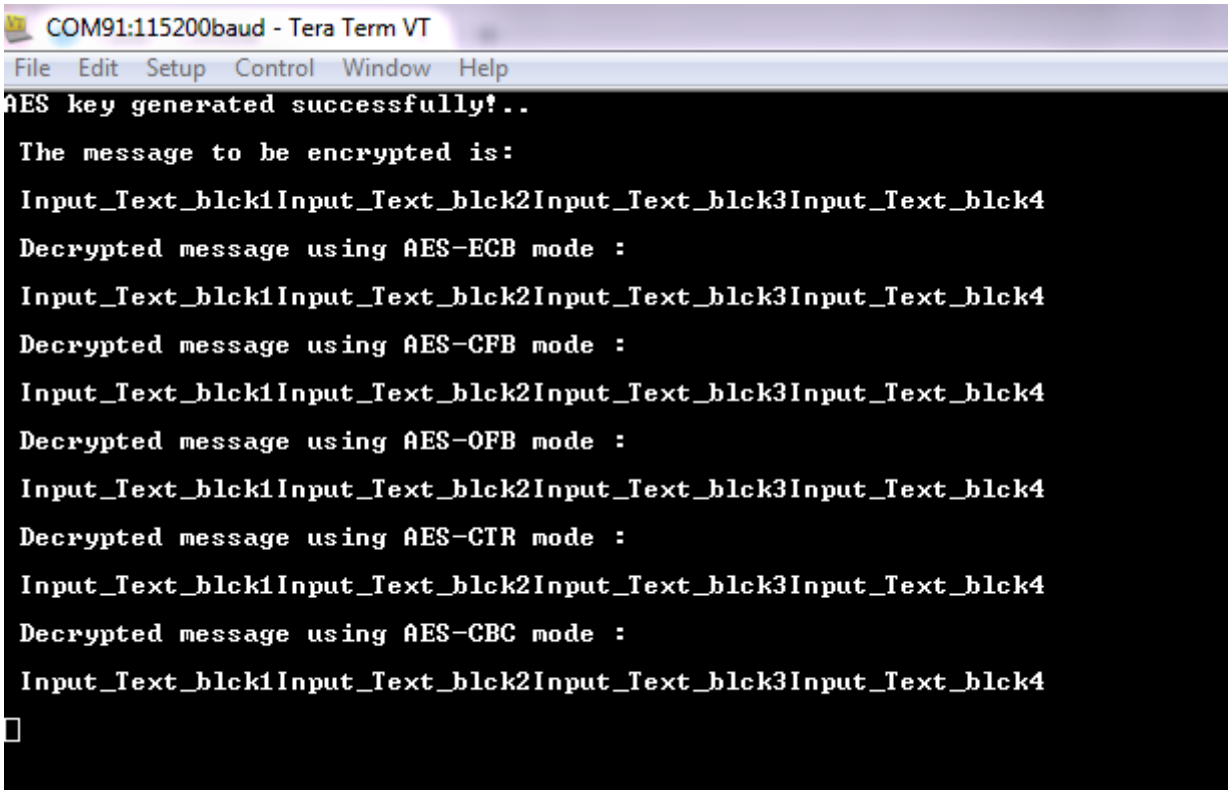
5. Read the device id after selection appropriate tools and device to ensure proper connection



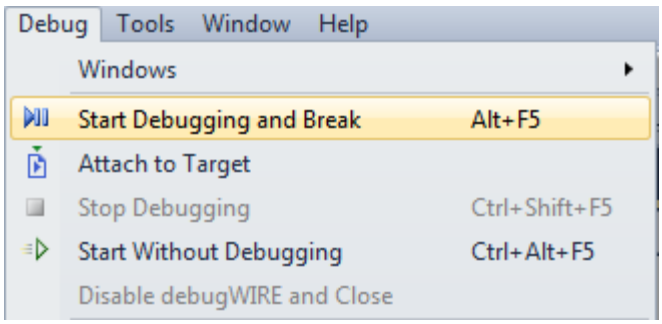
6. Go to ‘Memories’ tab. Select the exe file and click program.



7. In the terminal window the decrypted and actual input message can be viewed.



8. The Application can be debugged or can downloaded without debugging using below shown options.



7 References

7.1 FIPS – 197 Advanced Encryption Standard (AES)

FIPS-197 published by NIST contains the mathematical calculation of AES-128 algorithm for Encryption and Decryption. It can be availed from http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

7.2 FIPS SP 800-38A Recommendation for Block Cipher Modes of Operation

FIPS special publication 800-38A provides detailed description about the five modes of confidentiality. It also explains the generation of Initialization vector and counter used in each modes. It is available at http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

7.3 SAM D Device Datasheet

The device datasheet contains the block diagrams of the peripherals and details about implementing firmware for the device. It also contains the electrical specifications and expected characteristics of the device.

Datasheet is available on www.atmel.com in the Documents section of Atmel SAM D21 product page.

7.4 Hardware Tools User Guide

SAM D21 Xplained Pro board user Guide can be downloadable from http://www.atmel.com/tools/ATSAMD21-XPRO.aspx.

7.5 Atmel Studio

The latest version of Atmel Studio can be downloaded from http://www.atmel.com/tools/atmelstudio.aspx.

8 Revision History

Doc Rev.	Date	Comments
42508A	08/2015	Initial document release.

Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**