Introduction:

As application such as Internet of Things are increasing day by day, need of having secured communication/data transmission between embedded systems are a must requirement. Microcontrollers are the key component of most of the embedded systems and systems are build around these devices. Few higher end microcontrollers come with dedicated inbuilt hardware for the data encryption and decryption and thus provide the data security feature. But all applications may not call for these types of dedicated microcontrollers due to the cost and minimal requirements. Low-cost microcontrollers such as TM4C123GH6PM even though are having interfaces such as SPI, I2C, CAN, UART etc, does not come with dedicated hardware for data encryption and decryption. These types of microcontrollers need to implement the data encryption decryption in software considering the resource constraints like Time, Memory etc.

Advanced Encryption Standard (AES) is a widely used encryption standard, based on design principle known as substitution permutation network, which considered to be highly secure electronic data transfer. As we discussed before, few microcontrollers come with AES support in hardware itself. In this project we are implementing AES standard for encryption and decryption as C library which can be used for small embedded applications. Implemented library is demonstrated using a UART based console program using TM4C123GH6PM Tiva board.

Details of steps included in the AES Encryption and Decryption are explained briefly in the coming section along with specific implementation details.

AES Encryption Decryption

AES is a subset of the **Rijndael** block cipher developed by two Belgian cryptographers, Vincent Rijmen and Joan Daemen, who submitted a proposal to NIST during the AES selection process. It was the winner of this competition and thus named “AES”, for advanced encryption Standard by 2001. It is now the most widely used symmetric key encryption in the world.

Terminology

There are terms that are frequently used throughout this report that need to be clarified.

Block: AES is a block cipher. This means that the number of bytes that it encrypts is fixed. AES can currently encrypt blocks of 16 bytes at a time; no other block sizes are presently a part of the AES standard. If the bytes being encrypted are larger than the specified block, then AES is executed concurrently. This also means that AES must encrypt a minimum of 16 bytes. If the plain text is smaller than 16 bytes, then it must be padded. Simply said the block is a reference to the bytes that are processed by the algorithm.

State: Defines the current condition (state) of the *block*. That is the block of bytes that are currently being worked on. The state starts off being equal to the block, however it changes as each round of the algorithms executes. Plainly said this is the block in progress.

AES Encryption:

The Advanced Encryption Standard (AES) is an algorithm used to encrypt and decrypt data to protect the data when it is transmitted electronically. The algorithm described by AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting the data.

The AES algorithm allows for the use of cipher keys that are 128, 192, or 256 bits (16, 24 or 32 byte) long to protect data in 16-byte blocks. The main reason behind the security offered by the AES is due to its bigger key length itself which makes the no of keys to be checked very huge for brute force attacks.

AES is an iterated block encryption mechanism, which means same operations are repeated many times on a particular message block. One iteration of this operations is called a round. All round except final round are similar and specific bytes derived from initially provided key is used in each round. The process by which these extended keys generating from initially provided key is called key expansion.

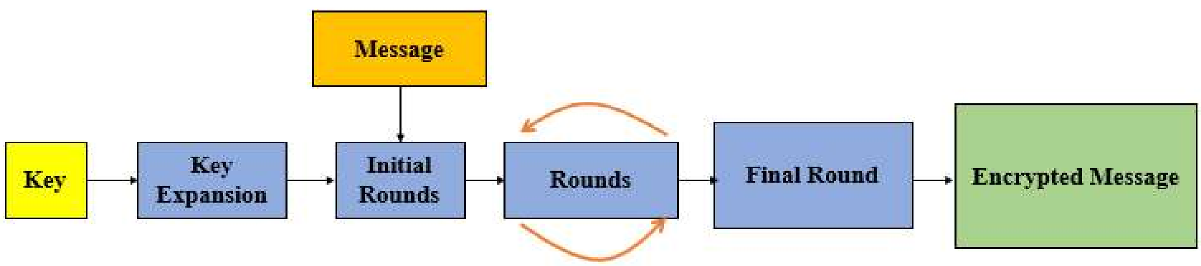


Figure 1: AES encryption flow diagram

The number of rounds the operations are repeated depends up on the key length. Below table gives the details of the no of round and length of expanded key including the initial key specified by the standard for each of the key lengths.

Key length	No of Rounds (Excluding initial round)	Expanded Key length
16 Byte	10	176 Byte
24 Byte	12	208 Byte
32 Byte	14	240 Byte

A specific round contains the following operations or steps.

1) Sub Byte Step
2) Shift Row or Rotation
3) Mix Column
4) Add Round Key

Every round except last round is having all these steps while last round is not having Mix column step included in that.

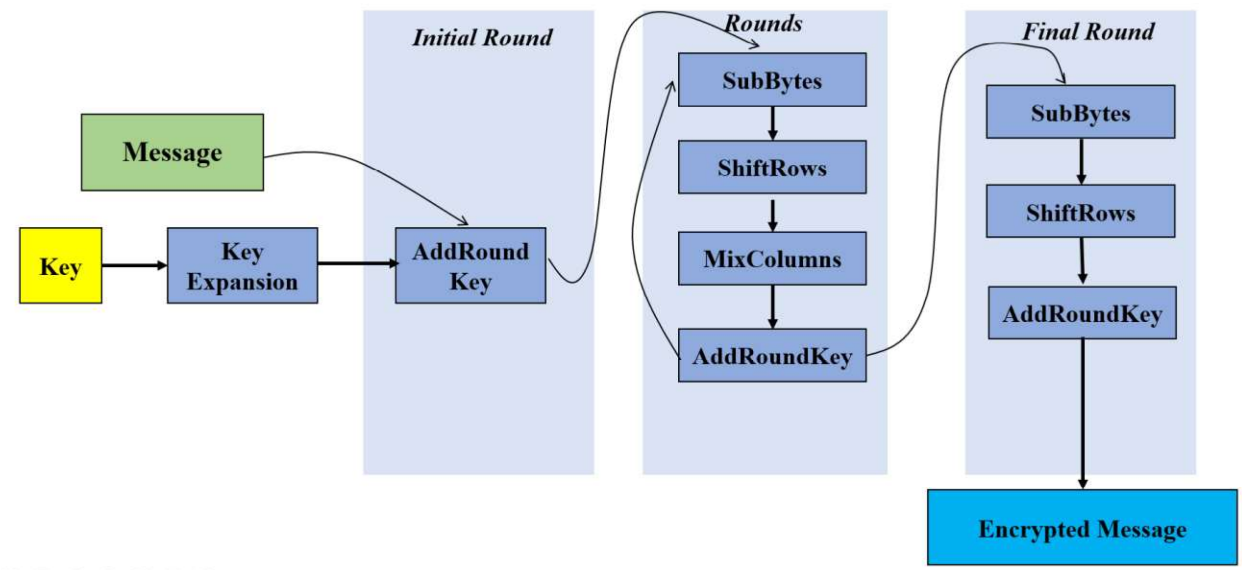


Figure 2 Detailed block diagram of AES encryption.

Detailed block diagram of AES encryption with steps included in each round is given in Figure 2.

Steps included in each step is explained some detail in coming sections.

Key Expansion

Key expansion is the routine or the process by which the initial key is expanded and keys for each round is calculated. This is done by following few fixed steps.

In case of 16-byte key, AES key expansion algorithm takes as input a four-word (16-byte) key and produces a linear array of 44 words (176 bytes). This is sufficient to provide a four-word round key for the initial AddRoundKey stage and each of the

10 rounds of the cipher. Similar operation only done in case of 24 and 32 byte key lengths also.

Let's say that we have the four words of the round key for the i th round:

w[i] ,w[i+1], w[i+2], w[i+3]

For these to serve as the round key for the i th round, i must be a multiple of 4. These will obviously serve as the round key for the (i/4)th round.

For example, w[4], w[5], w[6], w[7] is the round key for round 1, the sequence of words w[8], w[9], w[10], w[11] the round key for round 2, and so on.

Now we need to determine the words:

w[i+4],w[i+5], w[i+6], w[i+7] from the words w[i], w[i+1], w[i+2], w[i+3]

w[i+5] = w[i+4] xor w[i+1]

w[i+6] = w[i+5] xor w[i+2]

w[i+7] = w[i+6] xor w[i+3]

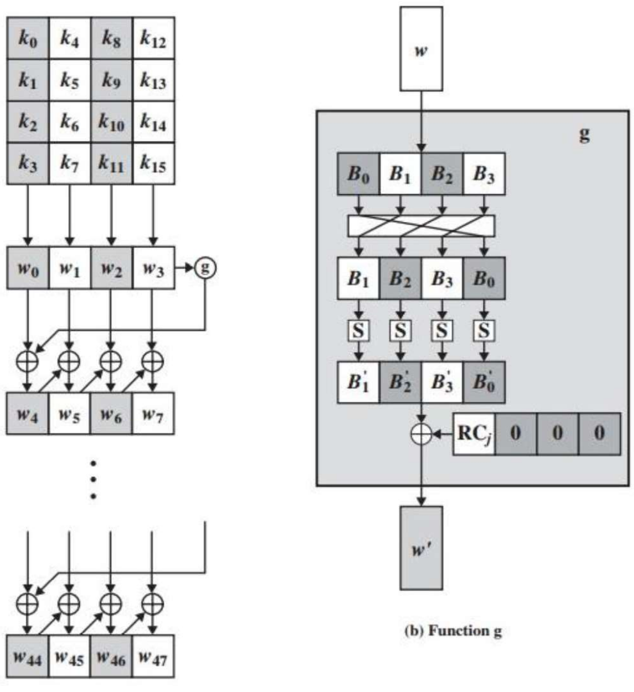
w[i+4] is the beginning word of each 4-word grouping in the key expansion. The beginning word of each round key is obtained by:

w[i+4] = w[i] xor g(w[i+3])

That is, the first word of the new 4-word grouping is to be obtained by XOR'ing the first word of the last grouping with what is returned by applying a function g() to the last word of the previous 4-word grouping.

The key is copied into the first four words of the expanded key. The remainder of the expanded key is filled in four words at a time. Each added word **w**[i] depends on the immediately preceding word, **w**[i - 1], and the word four positions back, w[i - 4].

In three out of four cases, a simple XOR is used. For a word whose position in the **w** array is a multiple of 4, a more complex function is used. The following figure illustrates the generation of the expanded key, using the symbol g to represent that complex function. The function g consists of the following steps:



(a) Overall algorithm

(b) Function g

1.**<u>RotWord :</u>**

Performs a one-byte circular left shift on a word. This means that an input word [B0, B1, B2, B3] is transformed into [B1, B2, B3, B0].

2.**SubWord** :

Perform a byte substitution for each byte of the word returned by the previous step by using the same 16 × 16 lookup table as used in the SubBytes step of the encryption rounds.

3.**Rcon:**

The result of steps 1 and 2 is XORed with a round constant, Rcon[j]. The round constant is a word whose three rightmost bytes are always zero. Therefore, XOR'ing with the round constant amounts to XOR'ing with just its leftmost byte.

The round constant for the i th round is denoted Rcon[i]. Since, by specification, the three rightmost bytes of the round constant are zero, we can write it as shown below. The left hand side of the equation below stands for the round constant to be used in the i th round. The right hand side of the equation says that the rightmost three bytes of the round constant are zero.

Rcon[i] = (RC[i], 0x00, 0x00, 0x00)

The only non-zero byte in the round constants, RC[i], obeys the following recursion:

RC[1] = 0x01

RC[i] = 0x02 × RC[i − 1]

The values of Rcon[i] in hexadecimal are

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
RC[i]	**01**	**02**	**04**	**08**	**10**	**20**	**40**	**80**	**1B**	**36**

The addition of the round constants is for the purpose of destroying any symmetries that may have been introduced by the other steps in the key expansion algorithm.

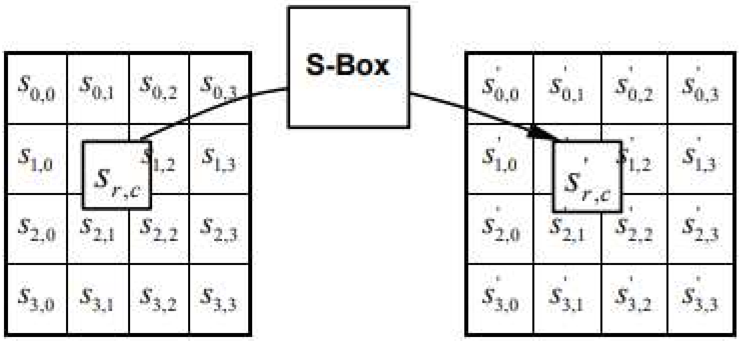
The key expansion algorithm ensures that AES has no **weak keys**. A weak key is a key that reduces the security of a cipher in a predictable manner.

The AES expansion key algorithm designed to be resistant to known cryptanalytic attacks. The inclusion of a round-dependent round constant eliminates the symmetry or similarity, between the ways in which round keys are generated in different rounds.

Sub Bytes

This step consists of using a 16 × 16 lookup table to find a replacement byte for a given byte in the input state array. This is a byte-by-byte substitution using a rule that stays the same in all encryption rounds.

Let xin be a byte of the state array for which we seek a substitute byte xout. We can write xout = f(xin).



The goal of the substitution step is to reduce the correlation between the input bits and the output bits at the byte level. The bit scrambling part of the substitution step ensures that the substitution cannot be described in the form of evaluating a simple mathematical function.

AES S-box

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

Similarly, during decryption, each value of state is replaced with the corresponding Inverse S-Box value:

Inverse S-box

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

Shift Row :

During the shiftrow operation we perform circular shift of each row. The matrix is formed vertically but shifted horizontally.

The ShiftRows transformation consists of

(i) not shifting the first row of the state array at all

(ii) circularly shifting the second row by one byte to the left

(iii) circularly shifting the third row by two bytes to the left.

(iv) circularly shifting the last row by three bytes to the left.

Before

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

After

0	4	8	12
5	9	13	1
10	14	2	6
15	3	7	11

For decryption, the corresponding step shifts the rows in exactly the opposite fashion. The first row is left unchanged, the second row is shifted to the right by one byte, the third row to the right by two bytes, and the last row to the right by three bytes, all shifts being circular.

Before

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

After

0	4	8	12
13	1	5	9
10	14	2	6
7	11	15	3

Mix Column:

Each byte in a column is replaced by two times that byte, plus three times the the next byte, plus the byte that comes next, plus the byte that follows.
[Note that by 'two times' and 'three times', we mean multiplications in $GF(2^8)$ by the bit patterns 000000010 and 00000011, respectively.]

The predefined matrix is multiplied with the State matrix to form the matrix multiplication and followed by the Galois field multiplication using the L Table and E Table.

State

B1	B5	B6	B13
B2	B6	B10	B14
B3	B7	B11	B15
B4	B8	B12	B16

Predefined Matrix

2	3	1	1
1	2	3	1
1	1	2	3
3	1	1	2

The first result byte is calculated by multiplying 4 values of the state column against 4 values of the first row of the matrix. The result of each multiplication is then XORed to produce 1 Byte.

*B1 = (B1 * 2) XOR (B2 * 3) XOR (B3 * 1) XOR (B4 * 1)*

The second result byte is calculated by multiplying the same 4 values of the state column against 4 values of the second row of the matrix. The result of each multiplication is then XORed to produce 1 Byte.

*B2 = (B1 * 1) XOR (B2 * 2) XOR (B3 * 3) XOR (B4 * 1)*

The third result byte is calculated by multiplying the same 4 values of the state column against 4 values of the third row of the matrix. The result of each multiplication is then XORed to produce 1 Byte.

*B3 = (B1 * 1) XOR (B2 * 1) XOR (B3 * 2) XOR (B4 * 3)*

The fourth result byte is calculated by multiplying the same 4 values of the state column against 4 values of the fourth row of the matrix. The result of each multiplication is then XORed to produce 1 Byte.

*B4 = (B1 * 3) XOR (B2 * 1) XOR (B3 * 1) XOR (B4 * 2)*

Similarly done for other columns also.

<u>Gallois Field Multiplication</u>:

The result of the multiplication is simply the result of a lookup of the **L** table, followed by the addition of the results, followed by a look up to the **E** table. The addition is a regular mathematical addition represented by +, not a bitwise AND.

E Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	01	03	05	0F	11	33	55	FF	1A	2E	72	96	A1	F8	13	35
1	5F	E1	38	48	D8	73	95	A4	F7	02	06	0A	1E	22	66	AA
2	E5	34	5C	E4	37	59	EB	26	6A	BE	D9	70	90	AB	E6	31
3	53	F5	04	0C	14	3C	44	CC	4F	D1	68	B8	D3	6E	B2	CD
4	4C	D4	67	A9	E0	3B	4D	D7	62	A6	F1	08	18	28	78	88
5	83	9E	B9	D0	6B	BD	DC	7F	81	98	B3	CE	49	DB	76	9A
6	B5	C4	57	F9	10	30	50	F0	0B	1D	27	69	BB	D6	61	A3
7	FE	19	2B	7D	87	92	AD	EC	2F	71	93	AE	E9	20	60	A0
8	FB	16	3A	4E	D2	6D	B7	C2	5D	E7	32	56	FA	15	3F	41
9	C3	5E	E2	3D	47	C9	40	C0	5B	ED	2C	74	9C	BF	DA	75
A	9F	BA	D5	64	AC	EF	2A	7E	82	9D	BC	DF	7A	8E	89	80
B	9B	B6	C1	58	E8	23	65	AF	EA	25	6F	B1	C8	43	C5	54
C	FC	1F	21	63	A5	F4	07	09	1B	2D	77	99	B0	CB	46	CA
D	45	CF	4A	DE	79	8B	86	91	A8	E3	3E	42	C6	51	F3	0E
E	12	36	5A	EE	29	7B	8D	8C	8F	8A	85	94	A7	F2	0D	17
F	39	4B	DD	7C	84	97	A2	FD	1C	24	6C	B4	C7	52	F6	01

L Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		00	19	01	32	02	1A	C6	4B	C7	1B	68	33	EE	DF	03
1	64	04	E0	0E	34	8D	81	EF	4C	71	08	C8	F8	69	1C	C1
2	7D	C2	1D	B5	F9	B9	27	6A	4D	E4	A6	72	9A	C9	09	78
3	65	2F	8A	05	21	0F	E1	24	12	F0	82	45	35	93	DA	8E
4	96	8F	DB	BD	36	D0	CE	94	13	5C	D2	F1	40	46	83	38
5	66	DD	FD	30	BF	06	8B	62	B3	25	E2	98	22	88	91	10
6	7E	6E	48	C3	A3	B6	1E	42	3A	6B	28	54	FA	85	3D	BA
7	2B	79	0A	15	9B	9F	5E	CA	4E	D4	AC	E5	F3	73	A7	57
8	AF	58	A8	50	F4	EA	D6	74	4F	AE	E9	D5	E7	E6	AD	E8
9	2C	D7	75	7A	EB	16	0B	F5	59	CB	5F	B0	9C	A9	51	A0
A	7F	0C	F6	6F	17	C4	49	EC	D8	43	1F	2D	A4	76	7B	B7
B	CC	BB	3E	5A	FB	60	B1	86	3B	52	A1	6C	AA	55	29	9D
C	97	B2	87	90	61	BE	DC	FC	BC	95	CF	CD	37	3F	5B	D1
D	53	39	84	3C	41	A2	6D	47	14	2A	9E	5D	56	F2	D3	AB
E	44	11	92	D9	23	20	2E	89	B4	7C	B8	26	77	99	E3	A5
F	67	4A	ED	DE	C5	31	FE	18	0D	63	8C	80	C0	F7	70	07

All numbers being multiplied using the Mix Column function converted to HEX will form a maximum of 2-digit Hex number. We use the first digit in the number on the vertical index and the second number on the horizontal index. If the value being multiplied is composed of only one digit, we use 0 on the vertical index.

For example, if the two Hex values being multiplied are AF * 8 we first lookup **L** (AF) index which returns B7 and then lookup **L** (08) which returns 4B.

Once the L table lookup is complete, we can then simply add the numbers together. The only trick being that if the addition result is greater than FF, we subtract FF from the addition result.

For example AF+B7= 166. Because 166 > FF, we perform: 166-FF which gives us 67.

The last step is to look up the addition result on the E table. Again, we take the first digit to look up the vertical index and the second digit to look up the horizontal index.

For example, E (67) =F0

Therefore, the result of multiplying AF * 8 over a Galois Field is F0.

During decryption: Similar steps are followed as stated above, the difference being that we use a different predefined matrix so matrix multiplication.

State

B1	**B5**	**B6**	**B13**
B2	**B6**	**B10**	**B14**
B3	**B7**	**B11**	**B15**
B4	**B8**	**B12**	**B16**

Predefined Matrix

0E	0B	0D	09
09	0E	0B	0D
0D	09	0E	0B
0B	0D	09	0E

B1 = (B1 * 0E) XOR (B2 * 0B) XOR (B3 * 0D) XOR (B4 * 09)

B2 = (B1 * 09) XOR (B2 * 0E) XOR (B3 * 0E) XOR (B4 * 0D)

B3 = (B1 * 0D) XOR (B2 * 09) XOR (B3 * 0E) XOR (B4 * 0B)

B4 = (B1 * 0B) XOR (B2 * 0D) XOR (B3 * 09) XOR (B4 * 0E

AES Encryption Example: -

One complete calculation of AES encryption for a specific message block is included in this section with results in each stage.

For this example, the plaintext, key, and resulting ciphertext are
Plaintext: 0123456789abcdeffedcba9876543210
Key: 0f1571c947d9e8590cb7add6af7f6798

Ciphertext: ff0b844a0853bf7c6934ab4364148fb9

In the table below the left-hand column shows the four round-key words generated for each round. The right-hand column shows the steps:

Key Expansion Steps:-

Key Words	Auxiliary Function
w0 = 0f 15 71 c9	RotWord(w3) = 7f 67 98 af = x1
w1 = 47 d9 e8 59	SubWord(x1) = d2 85 46 79 = y1
w2 = 0c b7 ad	Rcon(1) = 01 00 00 00
w3 = af 7f 67 98	y1 ⊕ Rcon(1) = d3 85 46 79 = z1
w4 = w0 ⊕ z1 = dc 90 37 b0	RotWord(w7) = 81 15 a7 38 = x2
w5 = w4 ⊕ w1 = 9b 49 df e9	SubWord(x4) = 0c 59 5c 07 = y2
w6 = w5 ⊕ w2 = 97 fe 72 3f	Rcon(2) = 02 00 00 00
w7 = w6 ⊕ w3 = 38 81 15 a7	y2 ⊕ Rcon(2) = 0e 59 5c 07 = z2
w8 = w4 ⊕ z2 = d2 c9 6b b7	RotWord(w11) = ff d3 c6 e6 = x3
w9 = w8 ⊕ w5 = 49 80 b4 5e	SubWord(x2) = 16 66 b4 83 = y3
w10 = w9 ⊕ w6 = de 7e c6 61	Rcon(3) = 04 00 00 00
w11 = w10 ⊕ w7 = e6 ff d3 c6	y3 ⊕ Rcon(3) = 12 66 b4 8e = z3
w12 = w8 ⊕ z3 = c0 af df 39	RotWord(w15) = ae 7e c0 b1 = x4
w13 = w12 ⊕ w9 = 89 2f 6b 67	SubWord(x3) = e4 f3 ba c8 = y4
w14 = w13 ⊕ w10 = 57 51 ad 06	Rcon(4) = 08 00 00 00
w15 = w14 ⊕ w11 = b1 ae 7e c0	y4 ⊕ Rcon(4) = ec f3 ba c8 = 4

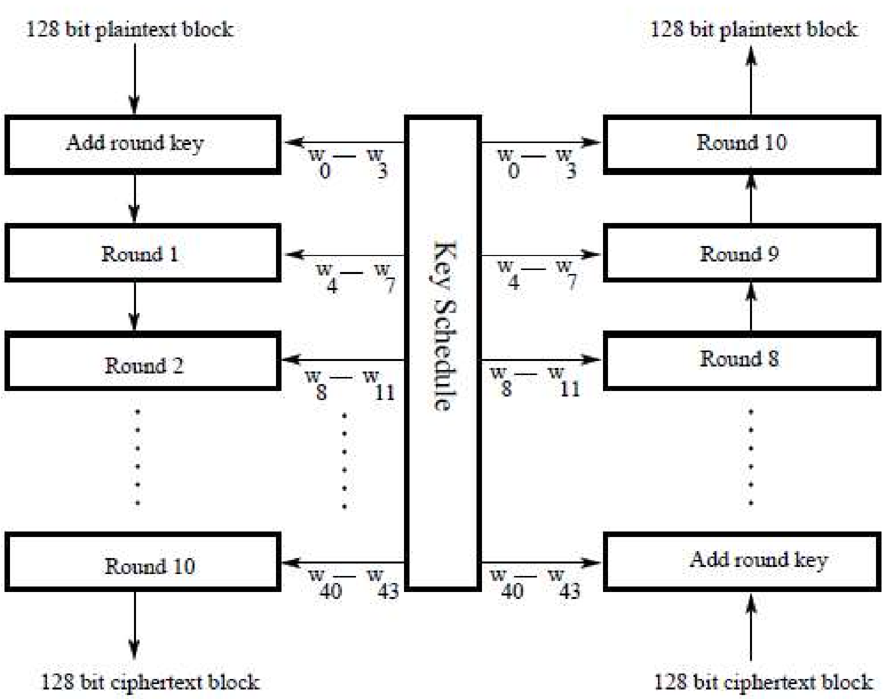
Key Words	Auxiliary Function
w16 = w12 ⊕ z4 = 2c 5c 65 f1	RotWord(w19) = 8c dd 50 43 = x5
w17 = w16 ⊕ w13 = a5 73 0e 96	SubWord(x4) = 64 c1 53 1a = y5
w18 = w17 ⊕ w14 = f2 22 a3 90	Rcon(5) = 10 00 00 00
w19 = w18 ⊕ w15 = 43 8c dd 50	y5 ⊕ Rcon(5) = 74 c1 53 1a = z5
w20 = w16 ⊕ z5 = 58 9d 36 eb	RotWord (w23) = 40 46 bd 4c = x6
w21 = w20 ⊕ w17 = fd ee 38 7d	SubWord (x5) = 09 5a 7a 29 = y6
w22 = w21 ⊕ w18 = 0f cc 9b ed	Rcon(6) = 20 00 00 00
w23 = w22 ⊕ w19 = 4c 40 46 bd	y6 ⊕ Rcon(6) = 29 5a 7a 29 = z6
w24 = w20 ⊕ z6 = 71 c7 4c c2	RotWord (w27) = a5 a9 ef cf = x7
w25 = w24 ⊕ w21 = 8c 29 74 bf	SubWord (x6) = 06 d3 bf 8a = y7
w26 = w25 ⊕ w22 = 83 e5 ef 52	Rcon (7) = 40 00 00 00
w27 = w26 ⊕ w23 = cf a5 a9 ef	y7 ⊕ Rcon(7) = 46 d3 df 8a = z7
w28 = w24 ⊕ z7 = 37 14 93 48	RotWord (w31) = 7d a1 4a f7 = x8
w29 = w28 ⊕ w25 = bb 3d e7 f7	SubWord (x7) = ff 32 d6 68 = y8
w30 = w29 ⊕ w26 = 38 d8 08 a5	Rcon (8) = 80 00 00 00
w31 = w30 ⊕ w27 = f7 7d a1 4a	y8 ⊕ Rcon(8) = 7f 32 d6 68 = z8
w32 = w28 ⊕ z8 = 48 26 45 20	RotWord (w35) = be 0b 38 3c = x9
w33 = w32 ⊕ w29 = f3 1b a2 d7	SubWord (x8) = ae 2b 07 eb = y9
w34 = w33 ⊕ w30 = cb c3 aa 72	Rcon (9) = 1B 00 00 00
w35 = w34 ⊕ w32 = 3c be 0b 3	y9 ⊕ Rcon (9) = b5 2b 07 eb = z9
w36 = w32 ⊕ z9 = fd 0d 42 cb	RotWord (w39) = 6b 41 56 f9 = x10
w37 = w36 ⊕ w33 = 0e 16 e0 1c	SubWord (x9) = 7f 83 b1 99 = y10
w38 = w37 ⊕ w34 = c5 d5 4a 6e	Rcon (10) = 36 00 00 00
w39 = w38 ⊕ w35 = f9 6b 41 56	y10 ⊕ Rcon (10) = 49 83 b1 99 = z10
w40 = w36 ⊕ z10 = b4 8e f3 52	
w41 = w40 ⊕ w37 = ba 98 13 4e	
w42 = w41 ⊕ w38 = 7f 4d 59 20	
w43 = w42 ⊕ w39 = 86 26 18 76	

AES Encryption Steps:

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

f9 e9 8f 2b 1b 34 2f 08 4f c9 85 49 bf bf 81 89	99 1e 73 f1 af 18 15 30 84 dd 97 3b 08 08 0c a7	99 1e 73 f1 18 15 30 af 97 3b 84 dd a7 08 08 0c	31 30 3a c2 ac 71 8c c4 46 65 48 eb 6a 1c 31 62	fd 0e c5 f9 0d 16 d5 6b 42 e0 4a 41 cb 1c 6e 56
cc 3e ff 3b a1 67 59 af 04 85 02 aa a1 00 5f 34	4b b2 16 e2 32 85 cb 79 f2 97 77 ac 32 63 cf 18	4b b2 16 e2 85 cb 79 32 77 ac f2 97 18 32 63 cf	4b 86 8a 36 b1 cb 27 5a fb f2 f2 af cc 5a 5b cf	b4 8e f3 52 ba 98 13 4e 7f 4d 59 20 86 26 18 76
ff 08 69 64 0b 53 34 14 84 bf ab 8f 4a 7c 43 b9				

As AES is a symmetric cipher, decryption is taking same step of encryption except in opposite direction.



AES Encryption AES Decryption

Algorithm Implementation

As part of this mini project, we have implemented the AES algorithm encryption and decryption as a C library for low end microcontrollers especially aiming the TIVA C board having TM4C123GH6PM as the microcontroller. All three key lengths (16, 24 and 32 bytes) are supported in this library and the preferred key length can be selected by changing the macro definition **#define AES_BIT 128 in** aes_fun.h file by appropriately changing to 128/192/256 for 16/24/32 Byte key length.

While implementing the algorithm, we have tried to optimize the implementation in such a way that time taken by the algorithm execution is minimum. Few of the optimization done are,

- Encryption/Decryption steps are combined wherever possible. For example, Substitution of bytes and shift row operation are done together. Thus, time taken for function call, stacking and return are minimized.
- Wherever possible, loops are unrolled and written explicitly. This may increase the code size by little amount but the jump instruction and condition calculation etc can be minimized and thus time can be optimized.
- Memory allocation of expanded key are done based on the **#define AES_BIT Macro** thus unwanted memory allocation for lower key length are avoided.
- Conditional compilation is used to handle the special case of the different key lengths rather than mixing all conditions and checking in the code. This will help to reduce the code size and optimize for the specific key length selected.
- Use of computer architecture for optimum use of instructions. TM4C123GH6PM is a 32 bit microcontroller, thus shift row operations are done using 32 bit operation so that 4 byte can be calculated same time.

Library provides two types of function to user. One is basic set of function which will work on one single block of message ie 16 bytes of message. Second set of functions support longer message encryption and decryption based on standard block cipher modes such as ECB (Electronic Code Book) and CBC (Cipher Block

Chaining). Basic functions can be used by user to implement other modes such as CFB(Cipher Feedback), PCBC(Propagating Cipher Block Chaining etc).

Basic Encryption Decryption functions

Basic encryption function supported by the library are explained in this section. These basic functions are Key Expansion, and encryption decryption function for single block of message.

1) **void Key_Expansion(unsigned char* key)**

This function receives starting address of initial bytes of key as input argument. Using this function will calculate rest bytes required depending upon the **AES_BIT Macro** defined in aes_fun.h file and will store those bytes to memory location followed by the predefined key.

2) **Void AES_Encrypt(unsigned char* Message, unsigned char* Result)**

This function receives the starting address of 16 byte message to be encrypted and starting address to where the encrypted message to be stored.

Function calculates the encrypted message by going through the required steps and number of rounds based on the key length defined using the macro and once computed the result will be written to the starting address given in input argument.

3) **Void AES_Decrypt(unsigned char* Message, unsigned char* Result)**

This function receives the starting address of 16 byte message to be decrypted and starting address to where the decrypted message to be stored.

Once the decryption is over the result will be written to the location starting from address given in input argument.

These three basic functions can be used by user for implementing AES encryption in TM4C123GH6PM without worrying about the AES algorithm or its internal way of working.

C library provided by this mini project gives two files, aes_fun.h which is the header file having declarations of these functions and aes_fun.c file which is having the definitions of these functions. Internal functions used to implement the algorithm etc is also included in these files.

It is also possible for user to just change the macro and small piece of conditional compilation code so that a custom AES implementation with different number of rounds etc can be implemented. User can experiment on that by changing appropriate portions in .c and .h files as per their requirement.

Memory requirement and timing details

S Box, Sinv_Box, E table and L table are saved as look up table in memory. Thus the AES library is having a minimum memory requirement. That is listed below.

S_Box	256 Byte
Sinv_Box	256 Byte
E table	256 Byte
L Table	256 Byte
State (Temporary variable)	16 Byte
State_Temp (Temporary variable)	16 Byte
Key size	176 Byte/ 208Byte/ 240Byte
Total	1232/1264/1296 Byte depending up on whether 16/24/32 Byte keys are selected

Time Taken for Encryption and Decryption

Time taken for one block of message to be encrypted and decrypted using this implementation was measured using systick timer. Time taken with different compiler optimization flags were measured and tabulated and given below.

All the below timings were calculated using 16MHz clock frequency. Thus timing should be recalculated proportionally when different clock frequency are used.

Key Length	No optimization		O1		O3	
	Encrypt	Decrypt	Encrypt	Decrypt	Encrypt	Decrypt
16 Byte	~1.6 ms	~2.6ms	.54 ms	~.9ms	~.35 ms	~.43ms
24 Byte	~1.8ms	~3.2ms	~.66 ms	~1.1ms	~.42ms	~.53ms
32 Byte	~2.2ms	~3.8ms	~0.78 ms	~1.3 ms	~.5ms	~0.62ms

It is evident from the table that even without any compiler optimization, library functions can be directly used for low data rate applications where data is generated in order of 8KSPS and all.

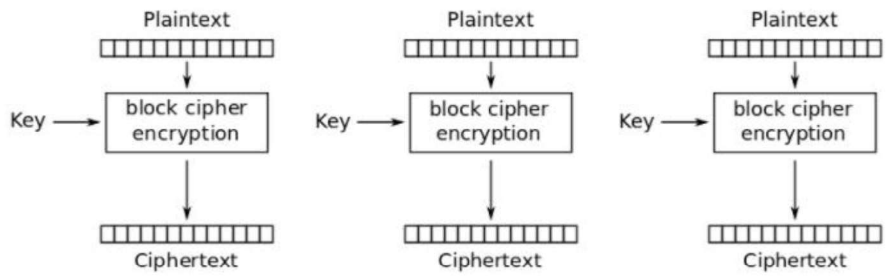
AES Encryption and Decryption for longer Messages

Some time user may have to encrypt messages of longer length which may or may not be of multiple of 16 bytes. In those cases, AES being a block cipher, we need to divide the message into different block and do encryption on each of that block. There are some standard modes being used in block ciphers for encrypting long length messages based up on the way encryption decryption is done. Some of the commonly used techniques are ECB(Electronic codebook), CBC(Cipher block chaining), CFB(Cipher feedback etc). Library supports ECB and CBC modes. If any other modes to be implemented user can use basic encryption decryption function explained above to do the same.

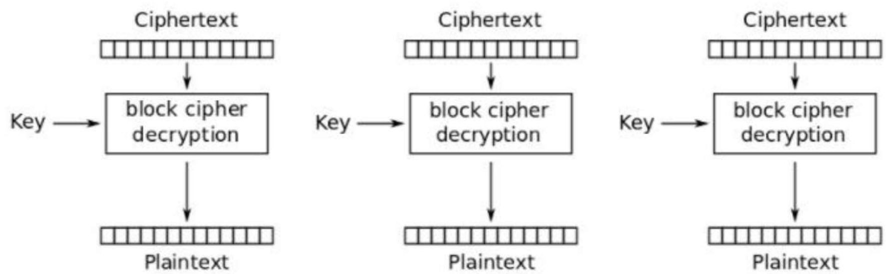
Another thing to done while encrypting longer message is padding of message to multiple of 16 bytes if it is not. This is done automatically in these supported modes of ECB and CBC.

Electronic codebook Encryption decryption

Electronic codeblock is simplest technique where input message is divided into blocks and do the encryption one by one. Encryption can be even done in parallel as there is no interdependency between the message that is encrypted.



Electronic Codebook (ECB) mode encryption



Electronic Codebook (ECB) mode decryption

- **Void Encrypt_Message_ECB(char * Message, int Message_length, char* Result)**

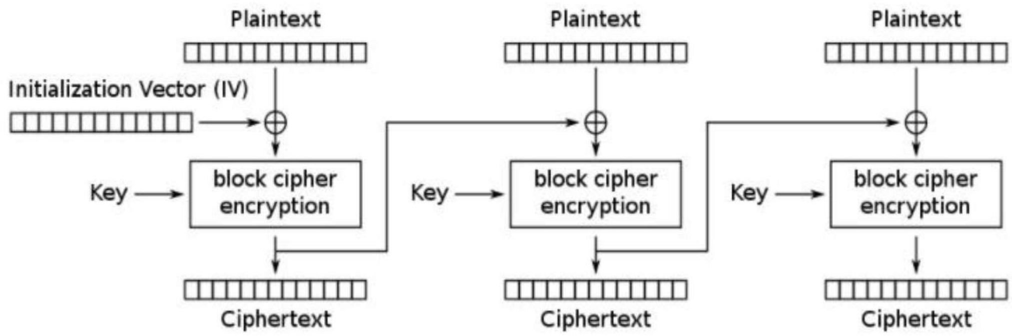
This function takes the starting address of message to encrypted along with message length and starting address where result to be stored as argument. Function will pad the message with 0x00 if message length is not multiple of 16 bytes and divide the message to blocks of 16 byte and do the encryption block by block.

- **Void Decrypt_Message_ECB(char * Message, int Message_length, char* Result)**

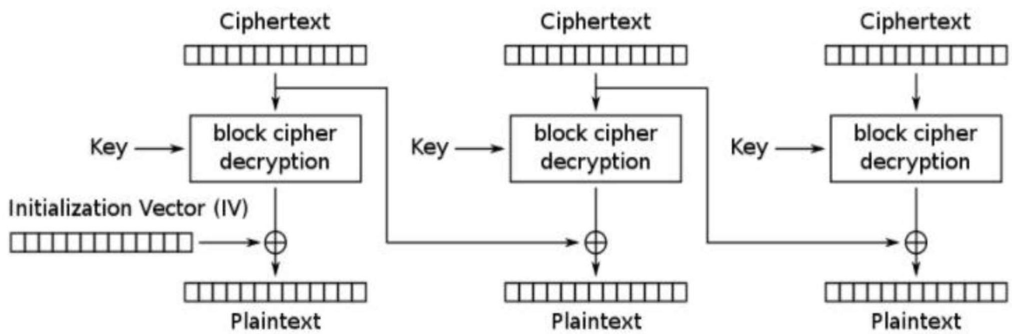
This function takes the starting address of message to decrypted along with message length and starting address where result to be stored as argument. As this is decryption message length should be multiple of 16 anyway. Function will decrypt block by block and store the result to address given as argument.

Cipher Block Chaining(CBC) Encryption decryption

In CBC mode, each block of plain text is XORed with the previous ciphertext block before being encrypted. This way, each ciphertext block depends on all plaintext blocks processed up to that point. To make each message unique, an initialization vector must be used in the first block.



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

- **Void Encrypt_Message_CBC(char * Message, int Message_length, char* Result)**

This function takes the starting address of message to encrypted along with message length and starting address where result to be stored as argument. Function will pad the message with 0x00 if message length is not multiple of 16 bytes and divide the message to blocks of 16 byte and do the encryption block by block.

- **Void Decrypt_Message_CBC(char * Message, int Message_length, char* Result)**

This function takes the starting address of message to decrypted along with message length and starting address where result to be stored as argument. As this is decryption message length should be multiple of 16 anyway. Function will decrypt block by block and store the result to address given as argument.

CBC mode allows better diffusion of message compared to ECB. But it requires additional 16 byte of initialization vector and will take up that space also. IV can be changed in aes_fun.c.

Code Organization: -

Main.c :- This file contains the UART console program written for testing for AES encryption decryption. UART initialization, systick initialization, command reception formatting etc are done in this file.

aes_fun.c :- This file contains the definitions of the AES related functionality.

aes_fun.h :- This file contains the declaration of function related to AES.

Testing and Results

To test the implementation in TIVA board, A UART console program was written where encryption and decryption of messages can be done through console commands.

Commands included are given below with their format.

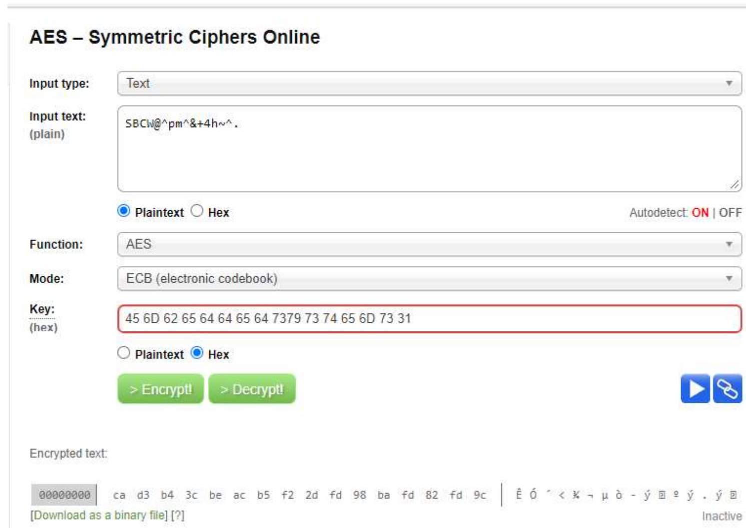
1) *Encrypt STR/HEX message in string or hex format*:- this command can be used to provide the message to be encrypted in string or Hex format. But message length should be fixed to 16 byte in this case. Tiva Board will reply with encrypted message and time taken for encrypting this 16 byte this 16 byte in terms of count taken by the systick counter. This count multiplied with 62.5ns will give absolute time taken.
2) *Decrypt STR/HEX message in string or hex format:*- this command can be used to provide the message to be decrypted in string or Hex format. But the message length should be fixed to 16 byte in this case. Tiva Board will reply with decrypted message and time taken for decrypting this 16 byte in terms of count taken by the systick counter. This count multiplied with 62.5ns will give absolute time taken.
3) *Encrypt ECB message in string*: - this command can be used to provide the message to be encrypted in ECB mode in string format. Message length can be more than 16 characters in this case.
4) *Decrypt ECB message in hex*: - this command can be used to provide the message to be encrypted in ECB mode in hex format. Message length can be more than 16 characters in this case.
5) *Encrypt CBC message in string*: - this command can be used to provide the message to be encrypted in CBC mode in string format. Message length can be more than 16 characters in this case.
6) *Decrypt CBC message in hex*: - this command can be used to provide the message to be encrypted in CBC mode in hex format. Message length can be more than 16 characters in this case.

In order to verify the results obtained we used a online AES encryption decryption tool, http://aes.online-domain-tools.com/. Some of the results obtained is given below.

Result 1:- **(Key length 16 byte)**

- Key :- Embeddedsystems1
- Message : SBCW@^pm^&+4h~^.

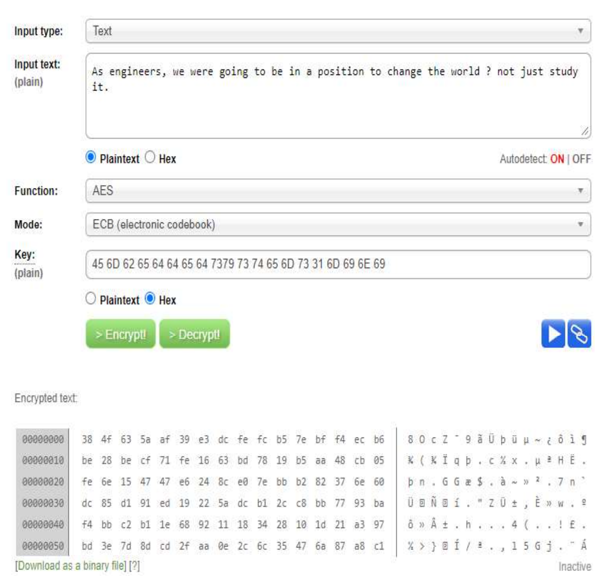
```
encrypt ecb SBCW@^pm^&+4h~^.
ENCRYPTECBSBCW@^pm^&+4h~^.

Message Recieved for encryption

SBCW@^pm^&+4h~^.
53424357405e706d5e262b34687e5e2e


0d000000000000000000000000000000

Message Encrypted: ECB Mode

ÊÓ´<¾¬µò-ýºýý
cad3b43cbeacb5f22dfd98bafd82fd9c


00000000000000000000000000000000
```



Result 2:- ECB Encryption (24 Byte)

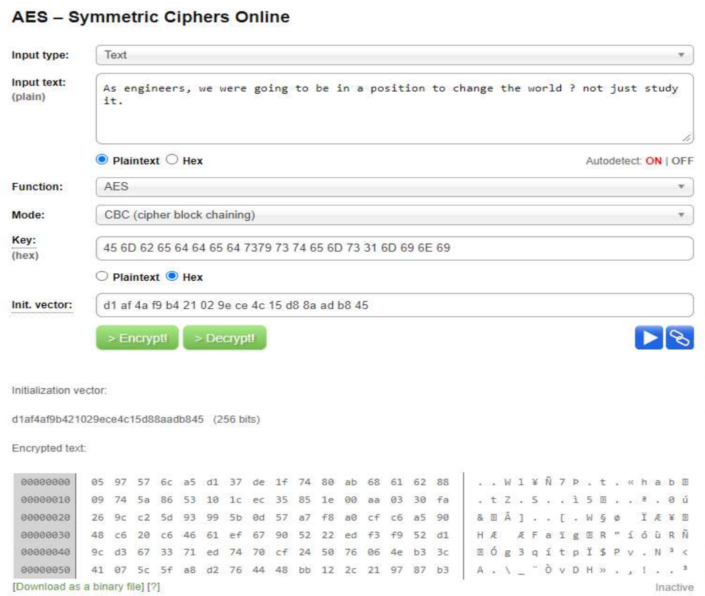
- key = Embeddedsystems1miniproj
- message: As engineers, we were going to be in a position to change the world ? not just study it.

```
ENCRYPTECBAs engineers, we were going to be in a position to change the world ? not just study it.

Message Recieved for encryption

As engineers, we
417320656e67696e656572732c207765

 were going to b
207765726520676f696e6720746f2062

e in a position
6520696e206120706f736974696f6e20

to change the wo
746f206368616e676520746865207766f

rld ? not just s
726c64203f206e6f74206a7573742073

tudy it.
747564792069742e0000000000000000

Message Encrypted: ECB Mode

8OcZ¯9ãÜþüµ~¿ôì¶
384f635aaf39e3dcfefcb57ebff4ecb6

¾(¾Ïqþc½xµªHË
be28becf71fe1663bd7819b5aa48cb05

þnGGæ$à~»²7n`
fe6e154747e6248ce07ebbb282376e60

ÜÑí"ZÜ±,È»wº
dc85d191ed19225adcb12cc8bb7793ba

ô»Â±h4(!£
f4bbc2b11e689211183428101d21a397
```



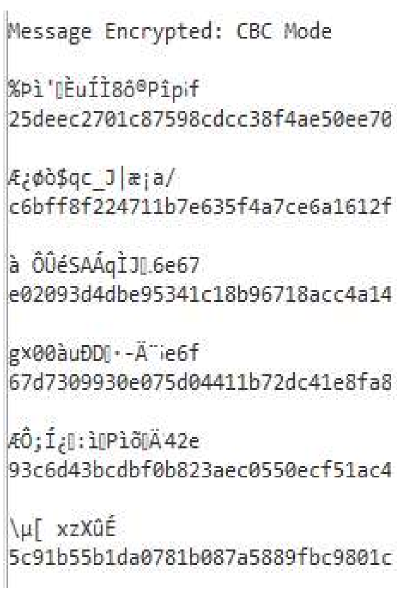
Result 3:- CBC Encryption (24 Byte) `IV[]={0xd1, 0xaf, 0x4a, 0xf9, 0xb4, 0x21, 0x02, 0x9e, 0xce, 0x4c, 0x15, 0xd8, 0x8a, 0xad, 0xb8, 0x45 };`

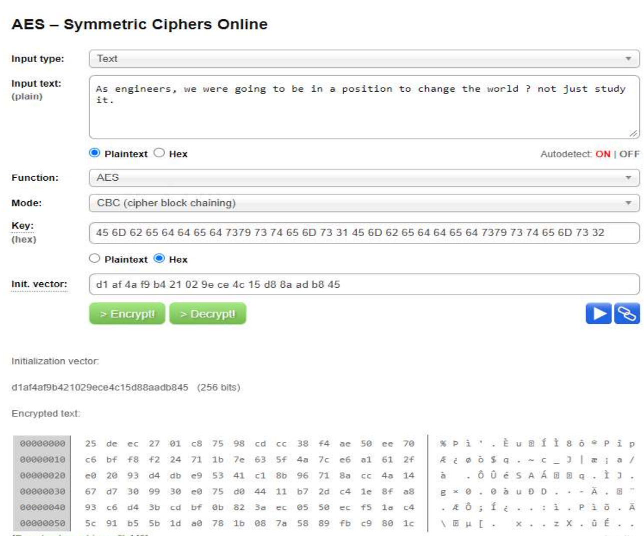
- key = Embeddedsystems1miniproj
- message: As engineers, we were going to be in a position to change the world ? not just study it.



Result 4:- CBC Encryption (32 Byte)

- key = Embeddedsystems1Embeddedsystems2
- message: As engineers, we were going to be in a position to change the world ? not just study it.





Conclusion

AES encryption decryption algorithm is implemented using a c library from scratch. Simple optimization techniques like loop unrolling, combining the functions wherever possible, minimal function call etc were tried while implementing the algorithm and to minimize the time taken for encryption and decryption.

Time taken for encryption and decryption were measured with different compiler optimization and it was found that the implementation is suitable for a moderate data rate upto 10KSPS. Further optimization can be studied seeing the disassembly code and other techniques etc. Other modes of block cipher encryption technique like CFB, PCBC also can be added to the library.