MOTIVATION

- Secure data transmission between embedded devices is very much important in modern times where application like IoTs etc are increasing.
- Higher end Microcontrollers comes with dedicated inbuilt hardware for data encryption and decryption. But low cost microcontrollers such as TM4C123GH6PM etc doesn’t have the dedicated hardware for encryption.
- Thus applications using low-end microcontrollers have to implement the encryption decryption algorithms in software even though it will be memory and time consuming.
- In this project we plan to implement AES(Advanced Encryption Standard) Encryption and Decryption as a software library for TM4C123GH6PM and demonstrate using the UART Console.

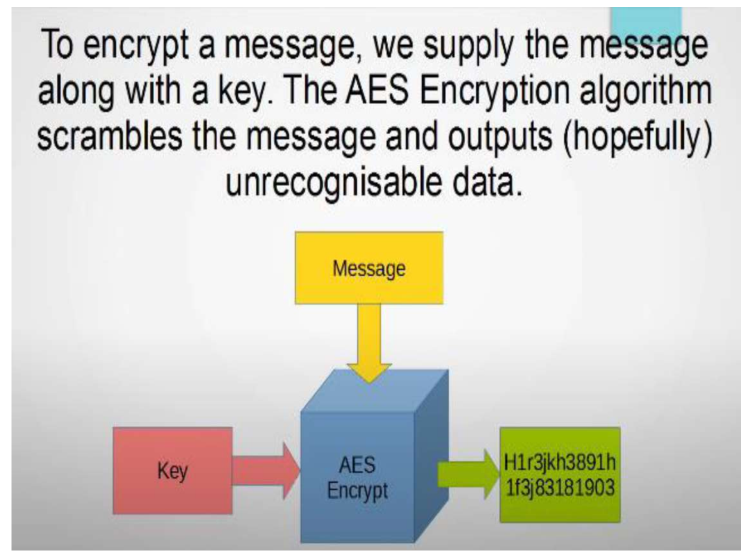
Origins

- It was originally called “Rijndael Cipher” after the names of the developer. It was an entrant in a competition held by NIST(National Institute of Standards and Technology) in 1997, to find a new secure encryption method.

- 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.

AES Encryption

- Symmetric Key Encryption algorithm.
- AES is block cipher which encrypts 128 bits (16 bytes) data at a time.
- These 16 bytes are treated like 4X4 grid
- Here we can select the key length to be 128 bits , 192 bits or 256 bits.
- The size of the key dictates the number of rounds or cycles of scrambling we need to perform.
- Each round we have a modified version of the original key , the modifications performed are called “ AES Key Expansion”

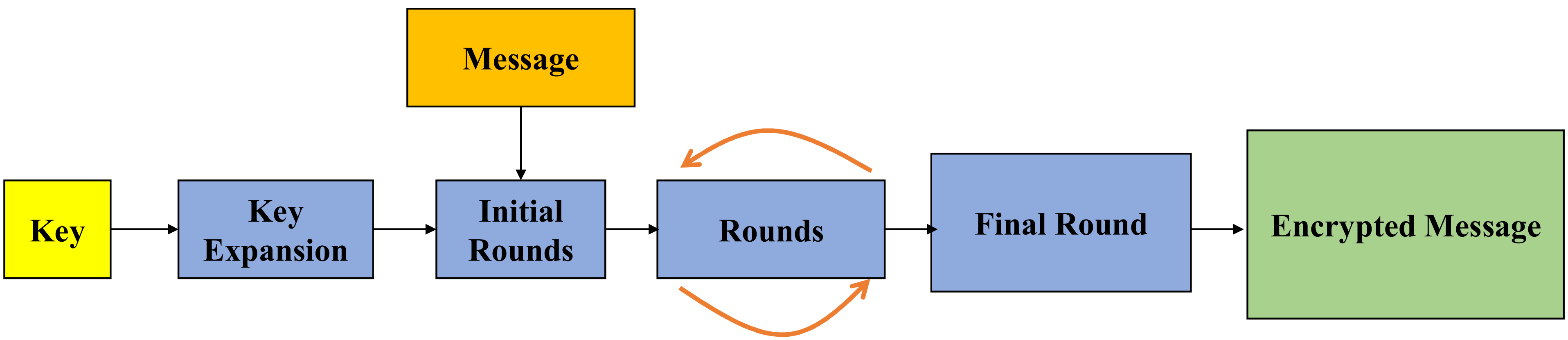


Terminology

- Block : AES is a block cipher which encrypts 128 bits (16 bytes) of data at a time. It treats the 16 bytes as a grid of 4x4. Messages which are longer than 128 bits are broken into blocks of 128 bits. Each block is encrypted separately using the same steps.
  - If the message is not divisible by the block length, then padding is appended. E.g., if the message is 25 bytes, you need 7 bytes of padding to make the message 32 bytes long. 32 is divisible by 16.
- 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 data, however it changes as each round of the algorithms executes. Plainly said this is the block in progress.

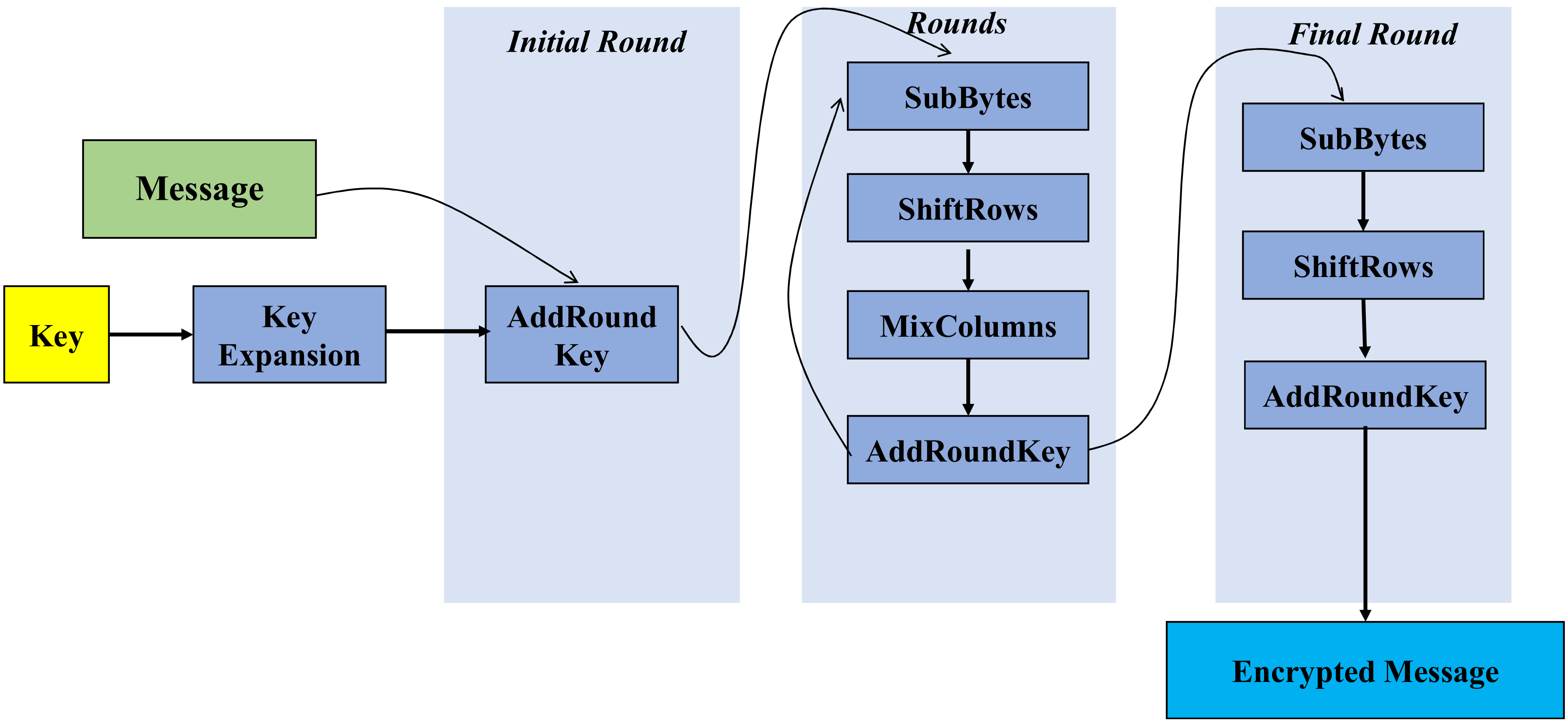
Rounds

- AES is made up of a couple of initialization steps (key expansion and the initial round). Then a series of rounds of encryption are performed using the expanded key. The number of times we repeat the "rounds" step is determined by the size of the key we select.



Most common and fastest	**16 bytes key**	**10 Cycles**
	24 bytes key	**12 Cycles**
Slowest and Most secure	**32 bytes key**	**14 Cycles**

Stages within Rounds



Key and States

- Message is : “this is ciphered” ; Key : “1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,G”

W1	W2	W3	W4
B1	**B5**	**B6**	**B13**
B2	**B6**	**B10**	**B14**
B3	**B7**	**B11**	**B15**
B4	**B8**	**B12**	**B16**

State

t		c	e
h	i	i	r
i	s	p	e
s		h	d

Round Key

1	5	9	D
2	6	A	E
3	7	B	F
4	8	C	G

AES Key Expansion

- Each round we use a modified version of the original key. The modifications we perform are called "AES Key Expansion", or sometimes "Rijndael key expansion".
- Each time the Add Round Key function is called a different part of the expanded key is XORed against the state.
- There fore the size of the expanded key will always be equal to:

  16 * (number of rounds + 1).

Key Size (Bytes)	Block Size (Bytes)	Expanded key (Bytes)
16	16	176
24	16	208
32	16	240

- The key expansion routine executes the following functions. These functions are:

1. ROT WORD
2. SUB WORD
3. RCON

Key Size (bytes)	Block Size (bytes)	Expansion Algorithm Rounds	Expanded Bytes / Round	Rounds Key Copy	Rounds Key Expansion	Expanded Key (bytes)
16	16	44	4	4	40	176
24	16	52	4	6	46	208
32	16	60	4	8	52	240

Rot Word

- Rotation Word does a circular shift of 4 Bytes

W1	W2	W3	W4
B1	B5	B6	B13
B2	B6	B10	B14
B3	B7	B11	B15
B4	B8	B12	B16

B2
B3
B4
B1

$$\underbrace{W1\quad W2\quad W3\quad W4}_{K0}\quad \underbrace{W5\quad W6\quad W7\quad W8}_{K1}\quad \ldots\ldots\quad \underbrace{W41\quad W42\quad W33\quad W44}_{K11}$$

Sub Word

- This Step applies the Sbox value substitution to each of the 4 bytes in the argument.

Rcon

- For example for a 16 byte key Rcon is first called in the 4 th round

(4/(16/4))-1=0

In this case Rcon will return 01000000

- For a 24 byte key Rcon is first called in the 6 th round

(6/(24/4))-1=0

In this case Rcon will also return 01000000

```
Rcon(0)      = 01000000
Rcon(1)      = 02000000
Rcon(2)      = 04000000
Rcon(3)      = 08000000
Rcon(4)      = 10000000
Rcon(5)      = 20000000
Rcon(6)      = 40000000
Rcon(7)      = 80000000
Rcon(8)      = 1B000000
Rcon(9)      = 36000000
Rcon(10)     = 6C000000
Rcon(11)     = D8000000
Rcon(12)     = AB000000
Rcon(13)     = 4D000000
Rcon(14)     = 9A000000
```

Cipher Key = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

for $Nk = 4$, which results in

w_0 = 2b7e1516 w_1 = 28aed2a6 w_2 = abf71588 w_3 = 09cf4f3c

i (dec)	temp	After RotWord()	After SubWord()	Rcon[i/Nk]	After XOR with Rcon	w[i-Nk]	w[i]= temp XOR w[i-Nk]
4	09cf4f3c	cf4f3c09	8a84eb01	01000000	8b84eb01	2b7e1516	a0fafe17
5	a0fafe17					28aed2a6	88542cb1
6	88542cb1					abf71588	23a33939
7	23a33939					09cf4f3c	2a6c7605
8	2a6c7605	6c76052a	50386be5	02000000	52386be5	a0fafe17	f2c295f2
9	f2c295f2					88542cb1	7a96b943
10	7a96b943					23a33939	5935807a
11	5935807a					2a6c7605	7359f67f
12	7359f67f	59f67f73	cb42d28f	04000000	cf42d28f	f2c295f2	3d80477d
13	3d80477d					7a96b943	4716fe3e
14	4716fe3e					5935807a	1e237e44
15	1e237e44					7359f67f	6d7a883b
16	6d7a883b	7a883b6d	dac4e23c	08000000	d2c4e23c	3d80477d	ef44a541
17	ef44a541					4716fe3e	a8525b7f
18	a8525b7f					1e237e44	b671253b
19	b671253b					6d7a883b	db0bad00
20	db0bad00	0bad00db	2b9563b9	10000000	3b9563b9	ef44a541	d4d1c6f8
21	d4d1c6f8					a8525b7f	7c839d87
22	7c839d87					b671253b	caf2b8bc
23	caf2b8bc					db0bad00	11f915bc
24	11f915bc	f915bc11	99596582	20000000	b9596582	d4d1c6f8	6d88a37a
25	6d88a37a					7c839d87	110b3efd
26	110b3efd					caf2b8bc	dbf98641
27	dbf98641					11f915bc	ca0093fd
28	ca0093fd	0093fdca	63dc5474	40000000	23dc5474	6d88a37a	4e54f70e
29	4e54f70e					110b3efd	5f5fc9f3
30	5f5fc9f3					dbf98641	84a64fb2
31	84a64fb2					ca0093fd	4ea6dc4f
32	4ea6dc4f	a6dc4f4e	2486842f	80000000	a486842f	4e54f70e	ead27321
33	ead27321					5f5fc9f3	b58dbad2
34	b58dbad2					84a64fb2	312bf560
35	312bf560					4ea6dc4f	7f8d292f
36	7f8d292f	8d292f7f	5da515d2	1b000000	46a515d2	ead27321	ac7766f3
37	ac7766f3					b58dbad2	19fadc21
38	19fadc21					312bf560	28d12941
39	28d12941					7f8d292f	575c006e
40	575c006e	5c006e57	4a639f5b	36000000	7c639f5b	ac7766f3	d014f9a8
41	d014f9a8					19fadc21	c9ee2589
42	c9ee2589					28d12941	e13f0cc8
43	e13f0cc8					575c006e	b6630ca6

Add Round Key

- Each of the 16 bytes of the state is XORed against each of the 16 bytes of a portion of the expanded key for the current round.

- The Expanded Key bytes are never reused.

- The next time the Add Round Key function is called bytes 17-32 are XORed against the state.

SubBytes

- In the SubBytes step we replace each byte of the state with another byte using a Look-up table called the Rijndael S-box. The S-box consists of 256 byte substitutions arranged in a 16x16 grid.

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

Shift Rows

- The Shift Rows step shifts the rows of the state to the left. The first row is not shifted. The second row is shifted by 1 byte to the left. The third row is shifted by two bytes, and the final row is shifted by 3 bytes.
- As bytes are shifted out on the left, they reappear on the right. This operation is sometimes called rotation.

Shift Rows Example during Encryption

- Best example would be to imagine a 0 to 15 numbers in the state

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

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

Shift Rows Example during Decryption

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

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

In our implementation we are doing Subbyte and shift row together at same time

Mix Column

- This step involves multiplying current state matrix with a predefined matrix to obtain next state. This multiplication is done over a Gallois field. Matrix Multiplication

- The multiplication is performed one column at a time (4 bytes). Each value in the column is eventually multiplied against every value of the matrix .The results of these multiplications are XORed together to produce only 4 result bytes for the next state.

- This second matrix (the one we're multiplying our state by) is predefined. It is a collection of Galois fields, just like our state

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

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

Galois Field Multiplication

- 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 lookup to the E table. The addition is a regular mathematical addition represented by +, not a bitwise AND.

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

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

Mix Column Example During Encryption

- Input = D4 BF 5D 30
- Output(0) = (D4 * 2) XOR (BF*3) XOR (5D*1) XOR (30*1)
  = E(L(D4) + L(02)) XOR E(L(BF) + L(03)) XOR 5D XOR 30
  = E(41 + 19) XOR E(9D + 01) XOR 5D XOR 30
  = E(5A) XOR E(9E) XOR 5D XOR 30
  = B3 XOR DA XOR 5D XOR 30
  = 04

Output(1) = (D4 * 1) XOR (BF*2) XOR (5D*3) XOR (30*1)
= D4 XOR E(L(BF)+L(02)) XOR E(L(5D)+L(03)) XOR 30
= D4 XOR E(9D+19) XOR E(88+01) XOR 30
= D4 XOR E(B6) XOR E(89) XOR 30
= D4 XOR 65 XOR E7 XOR 30
= 66

- Output(2) = (D4 * 1) XOR (BF*1) XOR (5D*2) XOR (30*3)

  = D4 XOR BF XOR E(L(5D)+L(02)) XOR E(L(30)+L(03))

  = D4 XOR BF XOR E(88+19) XOR E(65+01)

  = D4 XOR BF XOR E(A1) XOR E(66)

  = D4 XOR BF XOR BA XOR 50

  = 81

Output(3) = (D4 * 3) XOR (BF*1) XOR (5D*1) XOR (30*2)

  = E(L(D4)+L(3)) XOR BF XOR 5D XOR E(L(30)+L(02))

  = E(41+01) XOR BF XOR 5D XOR E(65+19)

  = E(42) XOR BF XOR 5D XOR E(7E)

  = 67 XOR BF XOR 5D XOR 60

  = E5

Mix Column During Decryption

- The Mix Column the multiplication is changed to

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

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)

- Input 04 66 81 E5
- Output(0) = (04 * 0E) XOR (66*0B) XOR (81*0D) XOR (E5*09)

  = E(L(04)+L(0E)) XOR E(L(66)+L(0B)) XOR E(L(81)+L(0D)) XOR E(L(E5)+L(09))

  = E(32+DF) XOR E(1E+68) XOR E(58+EE) XOR E(20+C7)

  = E(111-FF) XOR E(86) XOR E(146-FF) XOR E(E7)

  = E(12) XOR E(86) XOR E(47) XOR E(E7)

  = 38 XOR B7 XOR D7 XOR 8C

  = D4

Output(1) = (04 * 09) XOR (66*0E) XOR (81*0B) XOR (E5*0D)

= E(L(04)+L(09)) XOR E(L(66)+L(0E)) XOR E(L(81)+L(0B)) XOR E(L(E5)+L(0D))

= E(32+C7) XOR E(1E+DF) XOR E(58+68) XOR E(20+ EE)

= E(F9) XOR E(FD) XOR E(C0) XOR E(10E-FF)

= E(F9) XOR E(FD) XOR E(C0) XOR E(0F)

= 24 XOR 52 XOR FC XOR 35

= BF

- Output(2) = (04 * 0D) XOR (66*09) XOR (81*0E) XOR (E5*0B)

  = E(L(04)+L(0D)) XOR E(L(66)+L(09) XOR E(L(81)+L(0E)) XOR E(L(E5)+(0B))

  = E(32+EE) XOR E(1E+C7) XOR E(58+DF) XOR E(20+68)

  = E(120-FF) XOR E(E5) XOR E(137-FF) XOR E(88)

  = E(21) XOR E(E5) XOR E(38) XOR E(88)

  = 34 XOR 7B XOR 4F XOR 5D

  = 5D

Output(3) = (04 * 0B) XOR (66*0D) XOR (81*09) XOR (E5*0E)

= E(L(04)+L(0B)) XOR E(L(66)+L(0D)) XOR E(L(81)+L(09)) XOR E(L(E5)+L(0E))

= E(32+68) XOR E(1E+EE) XOR E(58+C7) XOR E(20+DF)

= E(9A) XOR E(10C-FF) XOR E(11F-FF) XOR E(FF)

= E(9A) XOR E(0D) XOR E(20) XOR E(FF)

= 2C XOR F8 XOR E5 XOR 01

= 30

Implementation details of AES Algorithm

- AES Encryption and Decryption is implemented for TIVA C Board for all three key lengths (16, 24 and 32 bytes)
- Key length is selected based on the Macro, `#define AES_BIT 128` Changing the bits to 192 or 256 for 24 and 32 byte AES encryption
- Key Expansion (Including Memory allocation for Expanded Key) and No of rounds etc will be decided by this.
- Functions are combined wherever possible (Example: Substitution of bytes and rotation is done together)
- Encryption and decryption is written using minimal function calls and minimal loops
- Loops are unrolled to the maximum possible

Implementation details of AES Algorithm

- Major functions as part of AES Library

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

Function receives starting address of initial bytes of key and calculate the rest bytes required depending upon the `AES_BIT` selected and store to the memory location followed by the predefined key

```
2) void AES_Encrypt(unsigned char* Message,unsigned char* Result);
```

Function receives the starting address of 16 byte message to be encrypted and store the encrypted message to address starting from Result

```
3) void AES_Decrypt(unsigned char* Message,unsigned char* Result);
```

Function receives the starting address of 16 byte message to be decrypted and store the decrypted message to address starting from Result.

Implementation details of AES Algorithm

- Memory/RAM Requirement

  S_Box = 256 Byte

  SInV_Box=256 Byte

  E Table = 256 Byte

  L Table = 256 Byte

  Temporary Variable for calculation state and state_temp = 16 byte each

  Key Size = 176 byte/208byte/240byte

Total Memory Required : 1232 byte/1264 byte /1296 byte based on the AES mode selected

Implementation details of AES Algorithm

Average Time taken by encryption and decryption:

Calculated using Systick timer, Calculated with different compiler optimization levels

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

Calculated with 16 MHz

Implementation details of AES Algorithm : For higher length messages

- Two modes of Encryption and decryption is supported by the library.

1) Electronic code book (ECB) Mode

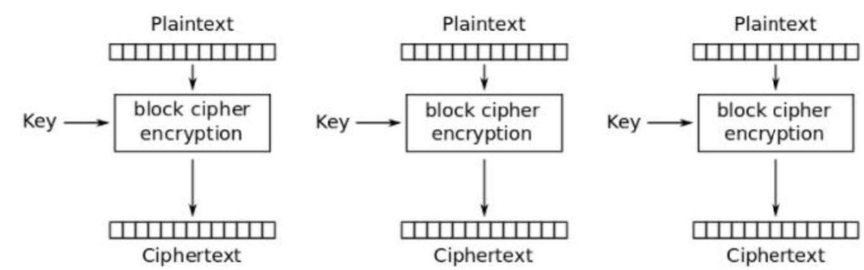
- Simplest method.
- Message is divided in to blocks of 16 and encrypted block wise

```
void Encrypt_Message_ECB(char* Message, int
Message_Length,char* Result);
```

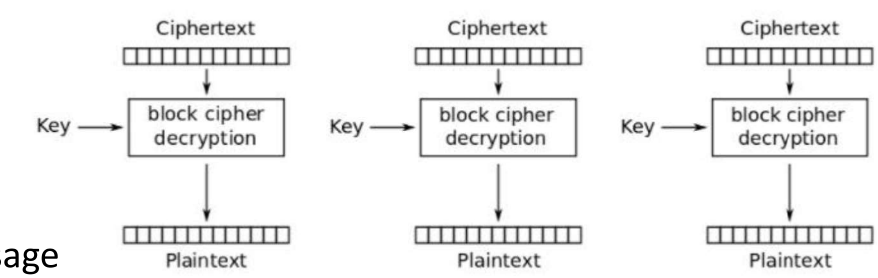
Function receives starting address of message to be encrypted, message length and starting address to where result to be stored.
Padding of message to multiple of 16 will be done by the function.

```
void Decrypt_Message_ECB(char* Message, int
Message_Length,char* Result);
```

Function receives starting address of message to be decrypted, message length and starting address to where result to be stored.
Message length to be multiple of 16 byte.



Electronic Codebook (ECB) mode encryption



Electronic Codebook (ECB) mode decryption

Implementation details of AES Algorithm : For higher length messages

2)Cipher block chaining (CBC)

Diffusion of message more compared to ECB

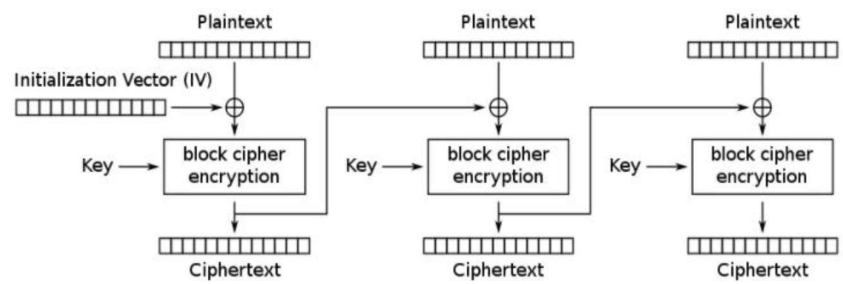
```
void Encrypt_Message_CBC(char* Message, int
Message_Length,char* Result);
```

Function receives starting address of message to be encrypted, message length and starting address to where result to be stored.
Padding of message to multiple of 16 will be done by the function.

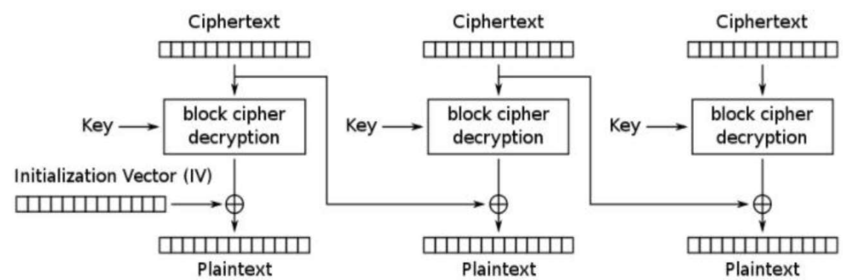
Additional 16 byte required for initialization vector

```
void Decrypt_Message_CBC(char* Message, int
Message_Length,char* Result);
```

Function receives starting address of message to be decrypted, message length and starting address to where result to be stored.
Message length to be multiple of 16 byte.



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

Other Modes of encryption like PCBC, CFB etc can be implemented if required by the user using basic encrypt and decrypt functions

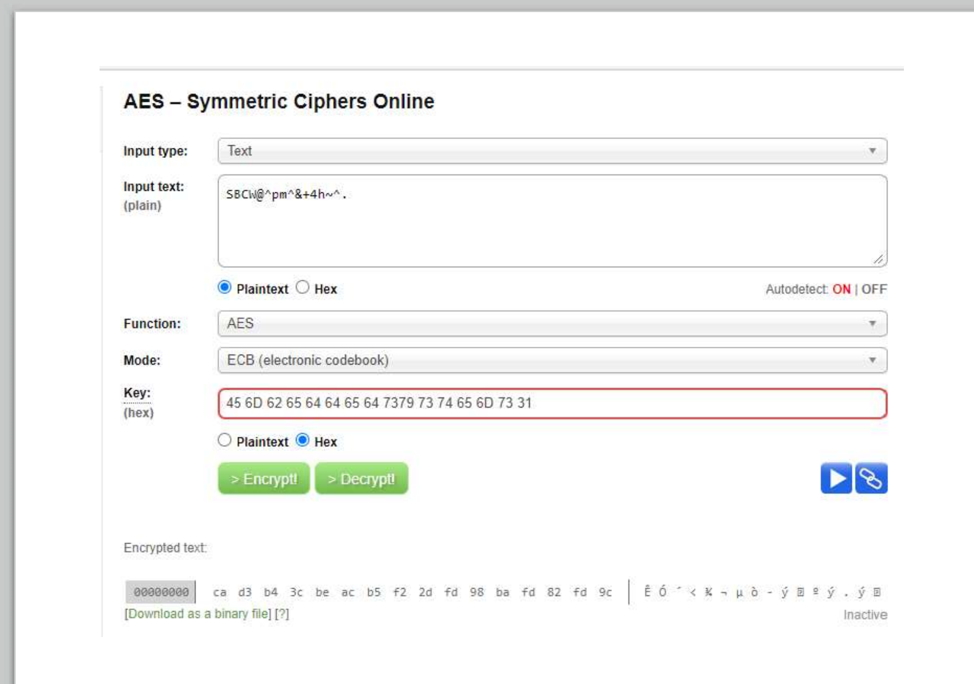
Results

- unsigned char Key[]= {0x45, 0x6D, 0x62, 0x65, 0x64, 0x64, 0x65, 0x64, 0x73, 0x79, 0x73, 0x74, 0x65, 0x6D, 0x73, 0x31}; // key = Embeddedsystems1
- Message : SBCW@^pm^&+4h~^.
- Different test cases have been taken to validate and the results have been compared with online AES encryption tools namely http://aes.online-domain-tools.com/
- The Test Cases are :

1. ECB Encryption and Decryption
2. CBC Encryption and Decryption

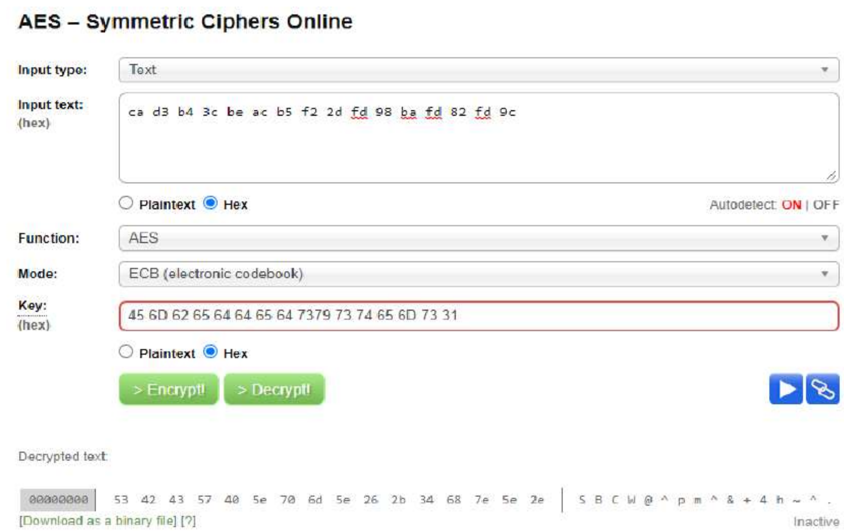
ECB Encryption (Key 16 bytes)

- Message = SBCW@^pm^&+4h~^.
- Output : 9b ac a4 33 b1 c1 57 fb 6b f6 92 5c 27 08 90 43



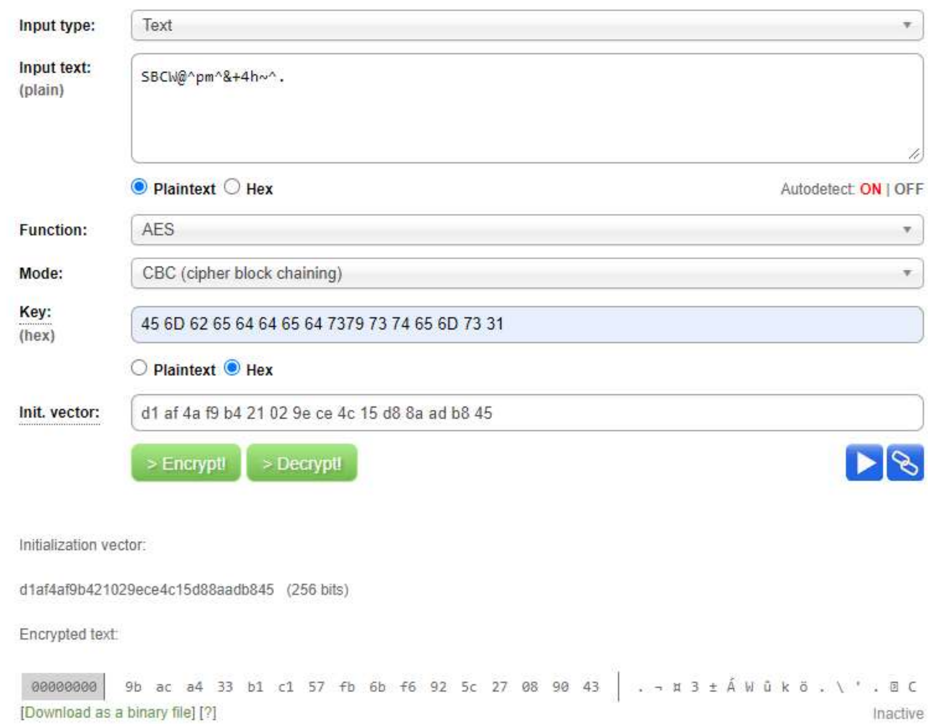
Decryption ECB
Message = ca d3 b4 3c be ac b5 f2 2d fd 98 ba fd 82 fd 9c
Output : SBCW@^pm^&+4h~^.

```
decrypt ecb cad3b43cbeacb5f22dfd98bafd82fd9c
DECRYPTECBcad3b43cbeacb5f22dfd98bafd82fd9c

Message Recieved for decryption

ÊÓ´<¾¬µò-ý˜ºýý
cad3b43cbeacb5f22dfd98bafd82fd9c

Message decrypted: ECB Mode

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

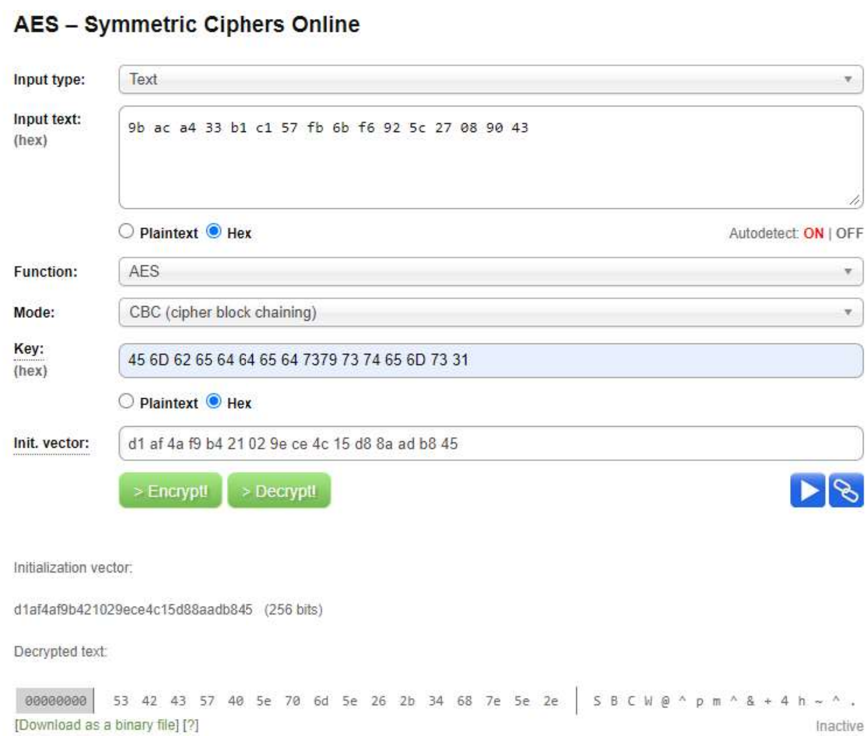


Encryption cbc (16 bytes)
Message : SBCW@^pm^&+4h~^.
Output : 9b ac a4 33 b1 c1 57 fb 6b f6 92 5c 27 08 90 43

```
encrypt cbc SBCW@^pm^&+4h~^.
ENCRYPTCBCSBCW@^pm^&+4h~^.

Message Recieved for encryption

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


0d000000000000000000000000000000

Message Encrypted: CBC Mode

¬¤3±ÁWûkö\Ce70
9baca433b1c157fb6bf6925c27089043

Ê:¿s*ÒAgA/¸ß6000
ca3a80bf732ad241876792412fb8df36
```



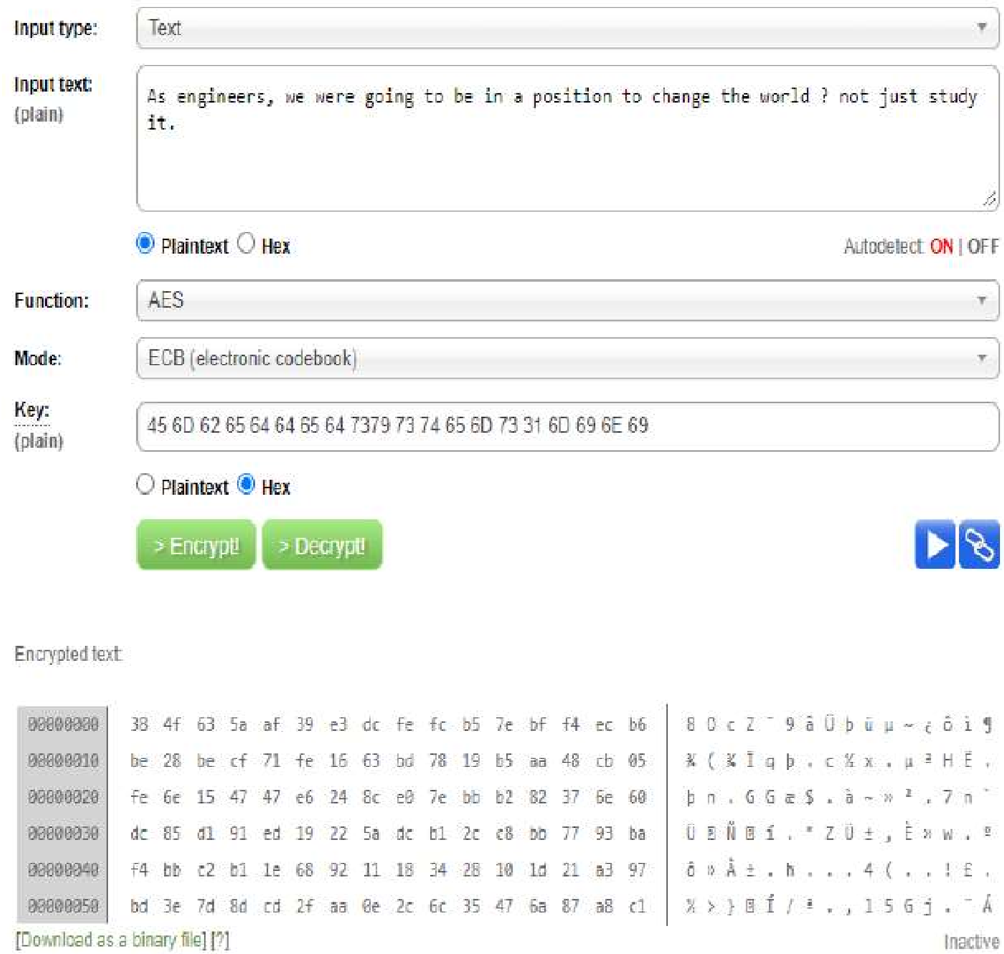
Decryption using CBC (16 bytes)



ECB Encryption 24bytes

Key (24 bytes) : {0x45, 0x6D, 0x62, 0x65, 0x64, 0x64, 0x65, 0x64, 0x73, 0x79, 0x73, 0x74, 0x65, 0x6D, 0x73, 0x31,0x6D, 0x69 ,0x6E, 0x69}; // key = Embeddedsystems1miniproj

```
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
20776572652067696e6720746f2062

e in a position
6520696e206120706f736974696f6e20

to change the wo
746f206368616e676520746865207776f

rld ? not just s
726c64203f206e6f74206a7573742073

tudy it.
747564792069742e0000000000000000

Message Encrypted: ECB Mode

8OcZ¯9ãÜþüµ~¿ôì¶
384f635aaf39e3dcfefcb57ebff4ecb6

¾(¾ÏqþcÅx µªHË
be28becf71fe1663bd7819b5aa48cb05

þnGGæ$à~»²7n`
fe6e154747e6248ce07ebbb282376e60

ÜÑí"ZÜ±,È»w°
dc85d191ed19225adcb12cc8bb7793ba

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



CBC Encryption (24 bytes)

Message Encrypted: CBC Mode

WI¥Ñ7Pt«hab
0597576ca5d137de1f7480ab68616288

tZS i5
09745a8653101cec35851e00aa0330fa

W§ø ÏÆ¥
269cc25d93995b0d57a7f8a0cfc6a590

HÆ ÆFaïgR"íóùRÑ
48c620c64661ef67905222edf3f952d1

Óg3qítpÏ$Pv N³< .
9cd3673371ed7470cf245076064eb33c

A_¯ÒvDH», !³
41075c5fa8d2764448bb122c219787b3

AES – Symmetric Ciphers Online

Input type: Text

Input text: (plain)
As engineers, we were going to be in a position to change the world ? not just study it.

Plaintext / Hex — Autodetect: ON | OFF

Function: AES

Mode: CBC (cipher block chaining)

Key: (hex) 45 6D 62 65 64 64 65 64 7379 73 74 65 6D 73 31 6D 69 6E 69

Plaintext / Hex

Init. vector: d1 af 4a f9 b4 21 02 9e ce 4c 15 d8 8a ad b8 45

\> Encrypt!   > Decrypt!

Initialization vector:

d1af4af9b421029ece4c15d88aadb845 (256 bits)

Encrypted text:

00000000	05	97	57	6c	a5	d1	37	de	1f	74	80	ab	68	61	62	88	. . W l ¥ Ñ 7 Þ . t . « h a b .
00000010	09	74	5a	86	53	10	1c	ec	35	85	1e	00	aa	03	30	fa	. t Z . S . . ì 5 . . . ª . 0 ú
00000020	26	9c	c2	5d	93	99	5b	0d	57	a7	f8	a0	cf	c6	a5	90	& . Â] . . [. W § ø Ï Æ ¥ .
00000030	48	c6	20	c6	46	61	ef	67	90	52	22	ed	f3	f9	52	d1	H Æ Æ F a ï g . R " í ó ù R Ñ
00000040	9c	d3	67	33	71	ed	74	70	cf	24	50	76	06	4e	b3	3c	. Ó g 3 q í t p Ï $ P v . N ³ <
00000050	41	07	5c	5f	a8	d2	76	44	48	bb	12	2c	21	97	87	b3	A . \ _ ¨ Ò v D H » . , ! . . ³

[Download as a binary file] [?]

Inactive

ECB Encryption
32 bytes

Key (32 bytes) : {0x45, 0x6D, 0x62, 0x65, 0x64, 0x64, 0x65, 0x64, 0x73, 0x79, 0x73, 0x74, 0x65, 0x6D, 0x73, 0x31,0x45, 0x6D, 0x62, 0x65, 0x64, 0x64, 0x65, 0x64, 0x73, 0x79, 0x73, 0x74, 0x65, 0x6D, 0x73, 0x32}; // key = Embeddedsystems1Embeddedsystems2

```
Message Encrypted: ECB Mode

&× .²¢û-¨'À♠to b
829726d79f202e1fb2a2fb2da827c00c

©r`Ó`¢1po
a91f7260d38f60a26c008d891b1cbbe4

`vDmâ¢%æoVÝ¶e wo
6076449a6d99e2a21dbde66f569dddb6

ª0ÖÆÝ¯s¢) jus
a9aa0d9db0f382d6c6ddaf967384a229

H¢ÏI·j»YúC[-
8e48a2cf491ab76abb59fa43a3085bad

Áøú%5»õF=' Jypte
c1f8fabd8535bb8df546119c3d8c274a
```

AES – Symmetric Ciphers Online

Input type: Text

Input text: (plain)

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

Plaintext / Hex — Autodetect: ON | OFF

Function: AES

Mode: ECB (electronic codebook)

Key: (plain) 45 6D 62 65 64 64 65 64 73 79 73 74 65 6D 73 31 45 6D 62 65 64 64 65 64 73 79 73 74 65 6D 73 32

Plaintext / Hex

> Encrypt! > Decrypt!

Encrypted text:

```
00000000  82 97 26 d7 9f 20 2e 1f b2 a2 fb 2d a8 27 c0 0c   . . & × .   . . ² ¢ û - ¨ ' À .
00000010  a9 1f 72 60 d3 8f 60 a2 6c 00 8d 89 1b 1c bb e4   © . r ` Ó . ` ¢ l . . . . . » ä
00000020  60 76 44 9a 6d 99 e2 a2 1d bd e6 6f 56 9d dd b6   ` v D . m . â ¢ . ½ æ o V . Ý ¶
00000030  a9 aa 0d 9d b0 f3 82 d6 c6 dd af 96 73 84 a2 29   © ª . . ° ó . Ö Æ Ý ¯ . s . ¢ )
00000040  8e 48 a2 cf 49 1a b7 6a bb 59 fa 43 a3 08 5b ad   . H ¢ Ï I . · j » Y ú C £ . [ .
00000050  c1 f8 fa bd 85 35 bb 8d f5 46 11 9c 3d 8c 27 4a   Á ø ú ½ . 5 » . õ F . . = . ' J
```

[Download as a binary file] [?]

Inactive

CBC Encryption (32 Bytes)

```
Message Encrypted: CBC Mode

%Pì'ÈuÍÌ8ô®Pîp¡f
25deec2701c87598cdcc38f4ae50ee70

Æ¿øò$qc_J|æ¡a/
c6bff8f224711b7e635f4a7ce6a1612f

à ÔÛéSAÁqÌJ.6e67
e02093d4dbe95341c18b96718acc4a14

gx00àuÐD·-Ä¨e6f
67d7309930e075d04411b72dc41e8fa8

ÆÔ;Í¿:ìPìõÄ42e
93c6d43bcdbf0b823aec0550ecf51ac4

\µ[ xzXûÉ
5c91b55b1da0781b087a5889fbc9801c
```

AES – Symmetric Ciphers Online

Input type: Text

Input text: (plain) As engineers, we were going to be in a position to change the world ? not just study it.

● Plaintext ○ Hex — Autodetect: ON | OFF

Function: AES

Mode: CBC (cipher block chaining)

Key: (hex) 45 6D 62 65 64 64 65 64 7379 73 74 65 6D 73 31 45 6D 62 65 64 64 65 64 7379 73 74 65 6D 73 32

○ Plaintext ● Hex

Init. vector: d1 af 4a f9 b4 21 02 9e ce 4c 15 d8 8a ad b8 45

> Encrypt! > Decrypt!

Initialization vector:

d1af4af9b421029ece4c15d88aadb845 (256 bits)

Encrypted text:

00000000	25	de	ec	27	01	c8	75	98	cd	cc	38	f4	ae	50	ee	70	% P ì ' . È u . Í Ì 8 ô ® P î p
00000010	c6	bf	f8	f2	24	71	1b	7e	63	5f	4a	7c	e6	a1	61	2f	Æ ¿ ø ò $ q . ~ c _ J | æ ¡ a /
00000020	e0	20	93	d4	db	e9	53	41	c1	8b	96	71	8a	cc	4a	14	à . Ô Û é S A Á . . q . Ì J .
00000030	67	d7	30	99	30	e0	75	d0	44	11	b7	2d	c4	1e	8f	a8	g × 0 . 0 à u Ð D . · - Ä . . ¨
00000040	93	c6	d4	3b	cd	bf	0b	82	3a	ec	05	50	ec	f5	1a	c4	. Æ Ô ; Í ¿ . . : ì . P ì õ . Ä
00000050	5c	91	b5	5b	1d	a0	78	1b	08	7a	58	89	fb	c9	80	1c	\ . µ [. . x . . z X . û É . .

[Download as a binary file] [?]

Inactive

Thank You