AB SOFTWARE DEVELOPMENT

Written By: Adam Berent

Advanced Encryption Standard by Example

V.1.5

1.0 Preface

The following document provides a detailed and easy to understand explanation of the implementation of the AES (RIJNDAEL) encryption algorithm. The purpose of this paper is to give developers with little or no knowledge of cryptography the ability to implement AES.

2.0 Terminology

There are terms that are frequently used throughout this paper 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 has to 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.

XOR Refers to the bitwise operator **Exclusive Or**. XOR operates on the individual bits in a byte in the following way:

```
0 XOR 0 = 0
1 XOR 0 = 1
1 XOR 1 = 0
0 XOR 1 = 1
```

For example the Hex digits D4 XOR FF

```
        11010100
XOR     11111111
=       00101011 (Hex 2B)
```

Another interesting property of the XOR operator is that it is reversible. So Hex 2B XOR FF = D4

Most programming languages have the XOR operator built in.

Programming Language	XOR Operator
C	^
C++	^
C#	^
Java	^
Visual Basic	XOR

HEX: Defines a notation of numbers in base 16. This simply means that; the highest number that can be represented in a single digit is 15, rather than the usual 9 in the decimal (base 10) system.

Hex to Decimal table:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

For example using the above table HEX D4 = DEC 212

All of the tables and examples in this paper are written in HEX. The reason for this is that a single digit of Hex represents exactly 4 bits. This means that a single byte can always be represented by 2 HEX digits. This also makes it very useful in creating lookup tables where each HEX digit can represent a table index.

3.0 AES Brief History

Effective May 26, 2002 the National Institute of Science and Technology (NIST) has selected a block cipher called RIJNDAEL (named after its creators Vincent Rijmen and Joan Daemen) as the symmetric key encryption algorithm to be used to encrypt sensitive but unclassified American federal information.

RIJNDAEL was originally a variable block (16, 24, 32 bytes) and variable key size (16, 24, 32 bytes) encryption algorithm. NIST has however decided to define AES with a block size of 16 bytes while keeping their options open to future changes.

4.0 AES Algorithm

AES is an iterated symmetric block cipher, which means that:

- AES works by repeating the same defined steps multiple times.
- AES is a secret key encryption algorithm.
- AES operates on a fixed number of bytes

AES as well as most encryption algorithms is reversible. This means that almost the same steps are performed to complete both encryption and decryption in reverse order. The AES algorithm operates on bytes, which makes it simpler to implement and explain.

This key is expanded into individual sub keys, a sub keys for each operation round. This process is called KEY EXPANSION, which is described at the end of this document.

As mentioned before AES is an iterated block cipher. All that means is that the same operations are performed many times on a fixed number of bytes. These operations can easily be broken down to the following functions:

ADD ROUND KEY
BYTE SUB
SHIFT ROW
MIX COLUMN

An iteration of the above steps is called a round. The amount of rounds of the algorithm depends on the key size.

Key Size (bytes)	Block Size (bytes)	Rounds
16	16	10
24	16	12
32	16	14

The only exception being that in the last round the **Mix Column** step is not performed, to make the algorithm reversible during decryption.

4.1 Encryption

AES encryption cipher using a 16 byte key.

Round	Function
–	Add Round Key(State)
0	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
1	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
2	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
3	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
4	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
5	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
6	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
7	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
8	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
9	Add Round Key(Shift Row(Byte Sub(State)))

AES encryption cipher using a 24 byte key.

Round	Function
–	Add Round Key(State)
0	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
1	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
2	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
3	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
4	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
5	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
6	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
7	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
8	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
9	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
10	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
11	Add Round Key(Shift Row(Byte Sub(State)))

AES encryption cipher using a 32 byte key.

Round	Function
–	Add Round Key(State)
0	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
1	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
2	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
3	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
4	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
5	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
6	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
7	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
8	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))

9	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
10	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
11	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
12	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
13	Add Round Key(Shift Row(Byte Sub(State)))

4.2 Decryption

AES decryption cipher using a 16 byte key.

Round	Function
–	Add Round Key(State)
0	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
1	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
2	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
3	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
4	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
5	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
6	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
7	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
8	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
9	Add Round Key(Byte Sub(Shift Row(State)))

AES decryption cipher using a 24 byte key.

Round	Function
–	Add Round Key(State)
0	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
1	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
2	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
3	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
4	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
5	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
6	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
7	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
8	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
9	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
10	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
11	Add Round Key(Byte Sub(Shift Row(State)))

AES decryption cipher using a 32 byte key.

Round	Function
–	Add Round Key(State)
0	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
1	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
2	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
3	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
4	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
5	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
6	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
7	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
8	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
9	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
10	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
11	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
12	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
13	Add Round Key(Byte Sub(Shift Row(State)))

5.0 AES Cipher Functions

5.1 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. So once the first 16 bytes are XORed against the first 16 bytes of the expanded key then the expanded key bytes 1-16 are never used again. The next time the Add Round Key function is called bytes 17-32 are XORed against the state.

The first time Add Round Key gets executed

State	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR
Exp Key	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The second time Add Round Key is executed

State	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR
Exp Key	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

And so on for each round of execution.

The method for deriving the expanded key is described in section 6.0

5.2 Byte Sub

During encryption each value of the state is replaced with the corresponding SBOX value

AES S-Box Lookup Table

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
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

For example HEX 19 would get replaced with HEX D4

During decryption each value in the state is replaced with the corresponding inverse of the SBOX

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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

For example HEX D4 would get replaced with HEX 19

5.3 Shift Row

Arranges the state in a matrix and then performs a circular shift for each row. This is not a bit wise shift. The circular shift just moves each byte one space over. A byte that was in the second position may end up in the third position after the shift. The circular part of it specifies that the byte in the last position shifted one space will end up in the first position in the same row.

In Detail:

The state is arranged in a 4x4 matrix (square)

The confusing part is that the matrix is formed vertically but shifted horizontally. So the first 4 bytes of the state will form the first bytes in each row.

So bytes 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Will form a matrix:

1 5 9 13
2 6 10 14
3 7 11 15
4 8 12 16

Each row is then moved over (shifted) 1, 2 or 3 spaces over to the right, depending on the row of the state. First row is never shifted

Row1 0
Row2 1
Row3 2
Roe4 3

The following table shows how the individual bytes are first arranged in the table and then moved over (shifted).

Blocks 16 bytes long:

From	To
1 5 9 13	**1 5 9 13**
2 6 10 14	**6 10 14 2**

```
3  7 11 15      11 15  3  7
4  8 12 16      16  4  8 12
```

During decryption the same process is reversed and all rows are shifted to the left:

```
From             To

1  5  9 13       1  5  9 13
2  6 10 14      14  2  6 10
3  7 11 15      11 15  3  7
4  8 12 16       8 12 16  4
```

5.4 Mix Column

This is perhaps the hardest step to both understand and explain. There are two parts to this step. The first will explain which parts of the state are multiplied against which parts of the matrix. The second will explain how this multiplication is implemented over what's called a Galois Field

5.4.1 Matrix Multiplication

The sate is arranged into a 4 row table (as described in the Shift Row function).

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 (16 total multiplications). The results of these multiplications are XORed together to produce only 4 result bytes for the next state. There fore 4 bytes input, 16 multiplications 12 XORs and 4 bytes output. The multiplication is performed one matrix row at a time against each value of a state column.

Multiplication Matrix

2	**3**	**1**	**1**
1	**2**	**3**	**1**
1	**1**	**2**	**3**
3	**1**	**1**	**2**

16 byte State

b1	**b5**	**b9**	**b13**
b2	**b6**	**b10**	**b14**
b3	**b7**	**b11**	**b15**
b4	**b8**	**b12**	**b16**

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)
```

This procedure is repeated again with the next column of the state, until there are no more state columns.

Putting it all together:

The first column will include state bytes 1-4 and will be multiplied against the matrix in the following manner:

```
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)
```

(b1= specifies the first byte of the state)

The second column will be multiplied against the second row of the matrix in the following manner.

```
b5 = (b5 * 2) XOR (b6*3) XOR (b7*1) XOR (b8*1)
b6 = (b5 * 1) XOR (b6*2) XOR (b7*3) XOR (b8*1)
b7 = (b5 * 1) XOR (b6*1) XOR (b7*2) XOR (b8*3)
b8 = (b5 * 3) XOR (b6*1) XOR (b7*1) XOR (b8*2)
```

And so on until all columns of the state are exhausted.

5.4.2 Galois Field Multiplication

The multiplication mentioned above is performed over a Galois Field. The mathematics behind this is beyond the scope of this paper. This section will instead concentrate on the implementation of the multiplication which can be done quite easily with the use of the following two tables in (HEX).

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

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.

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 then 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

There fore the result of multiplying AF * 8 over a Galois Field is F0

Two last exceptions are that:

- Any number multiplied by one is equal to its self and does not need to go through the above procedure. For example: FF * 1 = FF
- Any number multiplied by zero equals zero

5.4.3 Mix Column Inverse

During decryption the Mix Column the multiplication matrix is changed to:

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

Other then the change to the matrix table the function performs the same steps as during encryption.

5.4.4 Mix Column Example

The following examples are denoted in HEX.

5.4.4.1 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
```

5.4.4.2 Mix Column During Decryption

```
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
```

6.0 AES Key Expansion

Prior to encryption or decryption the key must be expanded. The expanded key is used in the **Add Round** Key function defined above.

Each time the Add Round Key function is called a different part of the expanded key is XORed against the state. In order for this to work the Expanded Key must be large enough so that it can provide key material for every time the Add Round Key function is executed. The Add Round Key function gets called for each round as well as one extra time at the beginning of the algorithm.

There fore the size of the expanded key will always be equal to:

16 * (number of rounds + 1).

The 16 in the above function is actually the size of the block in bytes. This provides key material for every byte in the block during every round +1

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

Since the key size is much smaller than the size of the sub keys, the key is actually "stretched out" to provide enough key space for the algorithm.

The key expansion routine executes a maximum of 4 consecutive functions. These functions are:

ROT WORD
SUB WORD
RCON
EK
K

An iteration of the above steps is called a round. The amount of rounds of the key expansion algorithm depends on the key size.

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

The first bytes of the expanded key are always equal to the key. If the key is 16 bytes long the first 16 bytes of the expanded key will be the same as the original key. If the key size is 32 bytes then the first 32 bytes of the expanded key will be the same as the original key.

Each round adds 4 bytes to the Expanded Key. With the exception of the first rounds each round also takes the previous rounds 4 bytes as input operates and returns 4 bytes.

One more important note is that not all of the 4 functions are always called in each round. The algorithm only calls all 4 of the functions every:

4 Rounds for a 16 byte Key
6 Rounds for a 24 byte Key
8 Rounds for a 32 byte Key

The rest of the rounds only a **K** function result is XORed with the result of the **EK** function. There is an exception of this rule where if the key is 32 bytes long an additional call to the **Sub Word** function is called every 8 rounds starting on the 13th round.

6.1 AES Key Expansion Functions

Rot Word (4 bytes)

This does a circular shift on 4 bytes similar to the Shift Row Function.

1,2,3,4 to **2,3,4,1**

Sub Word (4 bytes)

This step applies the S-box value substitution as described in **Bytes Sub** function to each of the **4** bytes in the argument.

Rcon((Round/(KeySize/4))-1)

This function returns a 4 byte value based on the following table

```
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
```

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

EK(Offset)

EK function returns 4 bytes of the Expanded Key after the specified offset. For example if offset is 0 then EK will return bytes 0,1,2,3 of the Expanded Key

K(Offset)

K function returns 4 bytes of the Key after the specified offset. For example if offset is 0 then **K** will return bytes 0,1,2,3 of the Expanded Key

6.2 AES Key Expansion Algorithm

Since the expansion algorithm changes depending on the length of the key, it is extremely difficult to explain in writing. This is why the explanation of the Key Expansion Algorithm is provided in a table format.

There are 3 tables, one for each AES key sizes (16, 24, and 32). Each table has 3 fields:

Field	*Description*
Round	A counter representing the current step in the key expansion algorithm, think of this as a loop counter
Expanded Key Bytes	Expanded key bytes effected by the result of the function(s)
Function	The function(s) that will return the 4 bytes written to the effected expanded key bytes

Notice that most numbers that change in following tables match the current round number. This makes implementation in code much easier as these numbers can easily be replaced with loop variables.

16 byte Key Expansion

Each round (except rounds 0, 1, 2 and 3) will take the result of the previous round and produce a 4 byte result for the current round. Notice the first 4 rounds simply copy the total of 16 bytes of the key

Round	Expanded Key Bytes	Function
0	0 1 2 3	**K**(0)
1	4 5 6 7	**K**(4)
2	8 9 10 11	**K**(8)
3	12 13 14 15	**K**(12)
4	16 17 18 19	**Sub Word**(**Rot Word**(**EK**((4-1)*4))) XOR **Rcon**((4/4)-1) XOR **EK**((4-4)*4)
5	20 21 22 23	**EK**((5-1)*4)XOR **EK**((5-4)*4)
6	24 25 26 27	**EK**((6-1)*4)XOR **EK**((6-4)*4)
7	28 29 30 31	**EK**((7-1)*4)XOR **EK**((7-4)*4)
8	32 33 34 35	**Sub Word**(**Rot Word**(**EK**((8-4)*4))) XOR **Rcon**((8/4)-1) XOR **EK**((8-4)*4)
9	36 37 38 39	**EK**((8-1)*4)XOR **EK**((9-4)*4)
10	40 41 42 43	**EK**((10-1)*4)XOR **EK**((10-4)*4)
11	44 45 46 47	**EK**((11-1)*4)XOR **EK**((11-4)*4)
12	48 49 50 51	**Sub Word**(**Rot Word(EK**((12-4)*4))) XOR **Rcon**((12/4)-1) XOR **EK**((12-4)*4)
13	52 53 54 55	**EK**((13-1)*4)XOR **EK**((13-4)*4)
14	56 57 58 59	**EK**((14-1)*4)XOR **EK**((14-4)*4)
15	60 61 62 63	**EK**((15-1)*4)XOR **EK**((15-4)*4)
16	64 65 66 67	**Sub Word**(**Rot Word**(**EK**((16-4)*4))) XOR **Rcon**((16/4)-1) XOR **EK**((16-4)*4)
17	68 69 70 71	**EK**((17-1)*4)XOR **EK**((17-4)*4)
18	72 73 74 75	**EK**((18-1)*4)XOR **EK**((18-4)*4)
19	76 77 78 79	**EK**((19-1)*4)XOR **EK**((19-4)*4)
20	80 81 82 83	**Sub Word**(**Rot Word**(**EK**((20-4)*4))) XOR **Rcon**((20/4)-1) XOR **EK**((20-4)*4)
21	84 85 86 87	**EK**((21-1)*4)XOR **EK**((21-4)*4)
22	88 89 90 91	**EK**((22-1)*4)XOR **EK**((22-4)*4)
23	92 93 94 95	**EK**((23-1)*4)XOR **EK**((23-4)*4)
24	96 97 98 99	**Sub Word**(**Rot Word**(**EK**((24-4)*4))) XOR **Rcon**((24/4)-1) XOR **EK**((24-4)*4)
25	100 101 102 103	**EK**((25-1)*4)XOR **EK**((25-4)*4)
26	104 105 106 107	**EK**((26-1)*4)XOR **EK**((26-4)*4)
27	108 109 110 111	**EK**((27-1)*4)XOR **EK**((27-4)*4)
28	112 113 114 115	**Sub Word**(**Rot Word**(**EK**((28-4)*4))) XOR **Rcon**((28/4)-1) XOR **EK**((28-4)*4)
29	116 117 118 119	**EK**((29-1)*4)XOR **EK**((29-4)*4)
30	120 121 122 123	**EK**((30-1)*4)XOR **EK**((30-4)*4)
31	124 125 126 127	**EK**((31-1)*4)XOR **EK**((31-4)*4)
32	128 129 130 131	**Sub Word**(**Rot Word**(**EK**((32-4)*4))) XOR **Rcon**((32/4)-1) XOR **EK**((32-4)*4)
33	132 133 134 135	**EK**((33-1)*4)XOR **EK**((33-4)*4)
34	136 137 138 139	**EK**((34-1)*4)XOR **EK**((34-4)*4)
35	140 141 142 143	**EK**((35-1)*4)XOR **EK**((35-4)*4)
36	144 145 146 147	**Sub Word**(**Rot Word**(**EK**((36-4)*4))) XOR **Rcon**((36/4)-1) XOR **EK**((36-4)*4)
37	148 149 150 151	**EK**((37-1)*4)XOR **EK**((37-4)*4)
38	152 153 154 155	**EK**((38-1)*4)XOR **EK**((38-4)*4)
39	156 157 158 159	**EK**((39-1)*4)XOR **EK**((39-4)*4)
40	160 161 162 163	**Sub Word**(**Rot Word**(EK((40-4)*4))) XOR **Rcon**((40/4)-1) XOR **EK**((40-4)*4)
41	164 165 166 167	**EK**((41-1)*4)XOR **EK**((41-4)*4)
42	168 169 170 171	**EK**((42-1)*4)XOR **EK**((42-4)*4)
43	172 173 174 175	**EK**((43-1)*4)XOR **EK**((43-4)*4)

24 byte Key Expansion

Each round (except rounds 0, 1, 2, 3, 4 and 5) will take the result of the previous round and produce a 4 byte result for the current round. Notice the first 6 rounds simply copy the total of 24 bytes of the key.

Round	Expanded Key Bytes	Function
0	0 1 2 3	**K**(0)
1	4 5 6 7	**K**(4)
2	8 9 10 11	**K**(8)
3	12 13 14 15	**K**(12)
4	16 17 18 19	**K**(16)
5	20 21 22 23	**K**(20)
6	24 25 26 27	**Sub Word**(**Rot Word**(**EK**((6-1)*4))) XOR **Rcon**((6/6)-1) XOR **EK**((6-6)*4)
7	28 29 30 31	**EK**((7-1)*4)XOR **EK**((7-6)*4)
8	32 33 34 35	**EK**((8-1)*4)XOR **EK**((8-6)*4)
9	36 37 38 39	**EK**((9-1)*4)XOR **EK**((9-6)*4)
10	40 41 42 43	**EK**((10-1)*4)XOR **EK**((10-6)*4)
11	44 45 46 47	**EK**((11-1)*4)XOR **EK**((11-6)*4)
12	48 49 50 51	**Sub Word**(**Rot Word**(**EK**((12-1)*4))) XOR **Rcon**((12/6)-1) XOR **EK**((12-6)*4)
13	52 53 54 55	**EK**((13-1)*4)XOR **EK**((13-6)*4)
14	56 57 58 59	**EK**((14-1)*4)XOR **EK**((14-6)*4)
15	60 61 62 63	**EK**((15-1)*4)XOR **EK**((15-6)*4)
16	64 65 66 67	**EK**((16-1)*4)XOR **EK**((16-6)*4)
17	68 69 70 71	**EK**((17-1)*4)XOR **EK**((17-6)*4)
18	72 73 74 75	**Sub Word**(**Rot Word**(**EK**((18-1)*4))) XOR **Rcon**((18/6)-1) XOR **EK**((18-6)*4)
19	76 77 78 79	**EK**((19-1)*4)XOR **EK**((19-6)*4)
20	80 81 82 83	**EK**((20-1)*4)XOR **EK**((20-6)*4)
21	84 85 86 87	**EK**((21-1)*4)XOR **EK**((21-6)*4)
22	88 89 90 91	**EK**((22-1)*4)XOR **EK**((22-6)*4)
23	92 93 94 95	**EK**((23-1)*4)XOR **EK**((23-6)*4)
24	96 97 98 99	**Sub Word**(**Rot Word**(**EK**((24-1)*4))) XOR **Rcon**((24/6)-1) XOR **EK**((24-6)*4)
25	100 101 102 103	**EK**((25-1)*4)XOR **EK**((25-6)*4)
26	104 105 106 107	**EK**((26-1)*4)XOR **EK**((26-6)*4)
27	108 109 110 111	**EK**((27-1)*4)XOR **EK**((27-6)*4)
28	112 113 114 115	**EK**((28-1)*4)XOR **EK**((28-6)*4)
29	116 117 118 119	**EK**((29-1)*4)XOR **EK**((29-6)*4)
30	120 121 122 123	**Sub Word**(**Rot Word**(**EK**((30-1)*4))) XOR **Rcon**((30/6)-1) XOR **EK**((30-6)*4)
31	124 125 126 127	**EK**((31-1)*4)XOR **EK**((31-6)*4)
32	128 129 130 131	**EK**((32-1)*4)XOR **EK**((32-6)*4)
33	132 133 134 135	**EK**((33-1)*4)XOR **EK**((33-6)*4)
34	136 137 138 139	**EK**((34-1)*4)XOR **EK**((34-6)*4)
35	140 141 142 143	**EK**((35-1)*4)XOR **EK**((35-6)*4)
36	144 145 146 147	**Sub Word**(**Rot Word**(**EK**((36-1)*4))) XOR **Rcon**((36/6)-1) XOR **EK**((36-6)*4)
37	148 149 150 151	**EK**((37-1)*4)XOR **EK**((37-6)*4)
38	152 153 154 155	**EK**((38-1)*4)XOR **EK**((38-6)*4)
39	156 157 158 159	**EK**((39-1)*4)XOR **EK**((39-6)*4)
40	160 161 162 163	**EK**((40-1)*4)XOR **EK**((40-6)*4)
41	164 165 166 167	**EK**((41-1)*4)XOR **EK**((41-6)*4)
42	168 169 170 171	**Sub Word**(**Rot Word**(**EK**((42-1)*4))) XOR **Rcon**((42/6)-1) XOR **EK**((42-6)*4)
43	172 173 174 175	**EK**((43-1)*4)XOR **EK**((43-6)*4)
44	176 177 178 179	**EK**((44-1)*4)XOR **EK**((44-6)*4)
45	180 181 182 183	**EK**((45-1)*4)XOR **EK**((45-6)*4)
46	184 185 186 187	**EK**((46-1)*4)XOR **EK**((46-6)*4)
47	188 189 190 191	**EK**((47-1)*4)XOR **EK**((47-6)*4)
48	192 193 194 195	**Sub Word**(**Rot Word**(**EK**((48-1)*4))) XOR **Rcon**((48/6)-1) XOR **EK**((48-6)*4)
49	196 197 198 199	**EK**((49-1)*4)XOR **EK**((49-6)*4)
50	200 201 202 203	**EK**((50-1)*4)XOR **EK**((50-6)*4)
51	204 205 206 207	**EK**((51-1)*4)XOR **EK**((51-6)*4)

32 byte Key Expansion

Each round (except rounds 0, 1, 2, 3, 4, 5, 6 and 7) will take the result of the previous round and produce a 4 byte result for the current round. Notice the first 8 rounds simply copy the total of 32 bytes of the key.

Round	Expanded Key Bytes	Function
0	0 1 2 3	**K**(0)
1	4 5 6 7	**K**(4)
2	8 9 10 11	**K**(8)
3	12 13 14 15	**K**(12)
4	16 17 18 19	**K**(16)
5	20 21 22 23	**K**(20)
6	24 25 26 27	**K**(24)
7	28 29 30 31	**K**(28)
8	32 33 34 35	**Sub Word**(**Rot Word**(**EK**((8-1)*4))) XOR **Rcon**((8/8)-1) XOR **EK**((8-8)*4)
9	36 37 38 39	**EK**((9-1)*4)XOR **EK**((9-8)*4)
10	40 41 42 43	**EK**((10-1)*4)XOR **EK**((10-8)*4)
11	44 45 46 47	**EK**((11-1)*4)XOR **EK**((11-8)*4)
12	48 49 50 51	**Sub Word(EK**((12-1)*4))XOR **EK**((12-8)*4)
13	52 53 54 55	**EK**((13-1)*4)XOR **EK**((13-8)*4)
14	56 57 58 59	**EK**((14-1)*4)XOR **EK**((14-8)*4)
15	60 61 62 63	**EK**((15-1)*4)XOR **EK**((15-8)*4)
16	64 65 66 67	**Sub Word**(**Rot Word**(**EK**((16-1)*4))) XOR **Rcon**((16/8)-1) XOR **EK**((16-8)*4)
17	68 69 70 71	**EK**((17-1)*4)XOR **EK**((17-8)*4)
18	72 73 74 75	**EK**((18-1)*4)XOR **EK**((18-8)*4)
19	76 77 78 79	**EK**((19-1)*4)XOR **EK**((19-8)*4)
20	80 81 82 83	**Sub Word(EK**((20-1)*4))XOR **EK**((20-8)*4)
21	84 85 86 87	**EK**((21-1)*4)XOR **EK**((21-8)*4)
22	88 89 90 91	**EK**((22-1)*4)XOR **EK**((22-8)*4)
23	92 93 94 95	**EK**((23-1)*4)XOR **EK**((23-8)*4)
24	96 97 98 99	**Sub Word**(**Rot Word**(**EK**((24-1)*4))) XOR **Rcon**((24/8)-1) XOR **EK**((24-8)*4)
25	100 101 102 103	**EK**((25-1)*4)XOR **EK**((25-8)*4)
26	104 105 106 107	**EK**((26-1)*4)XOR **EK**((26-8)*4)
27	108 109 110 111	**EK**((27-1)*4)XOR **EK**((27-8)*4)
28	112 113 114 115	**Sub Word(EK**((28-1)*4))XOR **EK**((28-8)*4)
29	116 117 118 119	**EK**((29-1)*4)XOR **EK**((29-8)*4)
30	120 121 122 123	**EK**((30-1)*4)XOR **EK**((30-8)*4)
31	124 125 126 127	**EK**((31-1)*4)XOR **EK**((31-8)*4)
32	128 129 130 131	**Sub Word**(**Rot Word**(**EK**((32-1)*4))) XOR **Rcon**((32/8)-1) XOR **EK**((32-8)*4)
33	132 133 134 135	**EK**((33-1)*4)XOR **EK**((33-8)*4)
34	136 137 138 139	**EK**((34-1)*4)XOR **EK**((34-8)*4)
35	140 141 142 143	**EK**((35-1)*4)XOR **EK**((35-8)*4)
36	144 145 146 147	**Sub Word(EK**((36-1)*4))XOR **EK**((36-8)*4)
37	148 149 150 151	**EK**((37-1)*4)XOR **EK**((37-8)*4)
38	152 153 154 155	**EK**((38-1)*4)XOR **EK**((38-8)*4)
39	156 157 158 159	**EK**((39-1)*4)XOR **EK**((39-8)*4)
40	160 161 162 163	**Sub Word**(**Rot Word**(**EK**((40-1)*4))) XOR **Rcon**((40/8)-1) XOR **EK**((40-8)*4)
41	164 165 166 167	**EK**((41-1)*4)XOR **EK**((41-8)*4)
42	168 169 170 171	**EK**((42-1)*4)XOR **EK**((42-8)*4)
43	172 173 174 175	**EK**((43-1)*4)XOR **EK**((43-8)*4)
44	176 177 178 179	**Sub Word(EK**((44-1)*4))XOR **EK**((44-8)*4)
45	180 181 182 183	**EK**((45-1)*4)XOR **EK**((45-8)*4)
46	184 185 186 187	**EK**((46-1)*4)XOR **EK**((46-8)*4)
47	188 189 190 191	**EK**((47-1)*4)XOR **EK**((47-8)*4)
48	192 193 194 195	**Sub Word**(**Rot Word**(**EK**((48-1)*4))) XOR **Rcon**((48/8)-1) XOR **EK**((48-8)*4)
49	196 197 198 199	**EK**((49-1)*4)XOR **EK**((49-8)*4)
50	200 201 202 203	**EK**((50-1)*4)XOR **EK**((50-8)*4)
51	204 205 206 207	**EK**((51-1)*4)XOR **EK**((51-8)*4)
52	208 209 210 211	**Sub Word(EK**((52-1)*4))XOR **EK**((52-8)*4)
53	212 213 214 215	**EK**((53-1)*4)XOR **EK**((53-8)*4)
54	216 217 218 219	**EK**((54-1)*4)XOR **EK**((54-8)*4)
55	220 221 222 223	**EK**((55-1)*4)XOR **EK**((55-8)*4)
56	234 225 226 227	**Sub Word**(**Rot Word**(**EK**((56-1)*4))) XOR **Rcon**((56/8)-1) XOR **EK**((56-8)*4)
57	228 229 230 231	**EK**((57-1)*4)XOR **EK**((57-8)*4)
58	232 233 234 235	**EK**((58-1)*4)XOR **EK**((58-8)*4)

59	236 237 238 239	**EK**((59-1)*4)XOR **EK**((59-8)*4)

7.0 Conclusion

The above document provides you with only the basic information needed to implement the AES encryption algorithm. The mathematics and design reasons behind AES were purposely left out. For more information on these topics I suggest you visit the Rijndael Home Page at:

http://www.esat.kuleuven.ac.be/~rijmen/rijndael/

8.0 References

FIPS 197, "Advanced Encryption Standard"
Advanced Encryption Standard (AES) http://www.ratchkov.com/vpn/aes/aes.html
RIJNDAEL http://www.cs.mcgill.ca/~kaleigh/computers/crypto_rijndael.html
The Laws of Cryptography http://www.cs.utsa.edu/~wagner/laws/