AES Implementation in C

Project structure

```
.
├── .vscode
│   └── cmake-kits.json
├── CMakeLists.txt
├── include
│   └── aes.h
├── src
│   └── aes.c
├── tests
│   └── test_aes.c
├── toolchain-clang.cmake
└── toolchain-gcc.cmake
```

Here is a breakdown of the project's directory structure and the purpose of each file:

- **`/`**
  - **`CMakeLists.txt`** : The main configuration file for the CMake build system. It defines the project's name, language, compiler options, and targets (libraries and executables).
  - **`toolchain-clang.cmake`** : A CMake toolchain file for configuring the project to be built with the Clang compiler.
  - **`toolchain-gcc.cmake`** : A CMake toolchain file for configuring the project to be built with the GCC compiler.
- **`/.vscode/`**
  - **`cmake-kits.json`** : A configuration file for the Visual Studio Code editor that defines the available CMake kits (compilers and toolchains) for building the project.
- **`/include/`**
  - **`aes.h`** : The main header file for the AES library. It defines the public API, including function prototypes, constants, and data types for the encryption and decryption functions.
- **`/src/`**

- `aes.c` : The main source file for the AES library. It contains the implementation of the AES encryption and decryption algorithms, including all the necessary helper functions and lookup tables.

- `/tests/`
  - `test_aes.c` : The source file for the unit tests. It uses the CTest framework to verify the correctness of the AES implementation by comparing the output of the encryption and decryption functions against known test vectors from the FIPS-197 standard.

Build and run the tests

This project now uses CMake as the build system. Use the following commands to configure, build and run the tests.

```
mkdir -p build
cmake -S . -B build -DCMAKE_EXPORT_COMPILE_COMMANDS=ON -DCMAKE_BUILD_TYPE=Release
cmake --build build --config Release
ctest --test-dir build
```

Introduction to cryptography:

1. Introduction to cryptography
2. Introduction to the Advanced Encryption Standard
3. Description of the AES algorithm
4. AES operations: SubBytes, ShiftRow, MixColumn and AddRoundKey
5. The Rijndael Key Schedule
6. The Key Expansion
7. Implementation: The Key Schedule
8. Implementation: AES Encryption
9. AES Decryption

Serge Vaudenay, in his book "A classical introduction to cryptography", writes:

> *Cryptography is the science of information and communication security.*

Cryptography is the science of secret codes, enabling the confidentiality of communication through an insecure channel. It protects against unauthorized parties by preventing unauthorized alteration of use. Generally speaking, it uses an cryptographic system to transform a plaintext into a ciphertext, using most of the time a key.

One has to notice that there exist certain cipher that don't need a key at all. A famous example is ROT13 (abbreviation from Rotation 13), a simple Caesar-cipher that obscures text by replacing each letter with the letter thirteen places down in the alphabet. Since our alphabet has 26 characters, it is enough to encrypt the ciphertext again to retrieve the original message.

Let me just mention briefly that there are secure public-key ciphers, like the famous and very secure Rivest-Shamir-Adleman (commonly called RSA) that uses a public key to encrypt a message and a secret key to decrypt it.

Cryptography is a very important domain in computer science with many applications. The most famous example of cryptography is certainly the Enigma machine, the legendary cipher machine used by the German Third Reich to encrypt their messages, who's security breach ultimately led to the defeat of their submarine force.

Before continuing, please read carefully the legal issues involving cryptography as in several countries even the domestic use of cryptography is prohibited:

> Cryptography has long been of interest to intelligence gathering agencies and law enforcement agencies. Because of its facilitation of privacy, and the diminution of privacy attendant on its prohibition, cryptography is also of considerable interest to civil rights supporters. Accordingly, there has been a history of controversial legal issues surrounding cryptography, especially since the advent of inexpensive computers has made possible widespread access to high quality cryptography.
>
> In some countries, even the domestic use of cryptography is, or has been, restricted. Until 1999, France significantly restricted the use of cryptography domestically. In China, a license is still required to use cryptography. Many countries have tight restrictions on the use of cryptography. Among the more restrictive are laws in Belarus, Kazakhstan, Mongolia, Pakistan, Russia, Singapore, Tunisia, Venezuela, and Vietnam.[31]
>
> In the United States, cryptography is legal for domestic use, but there has been much conflict over legal issues related to cryptography. One particularly important issue has been the export of cryptography and cryptographic software and hardware. Because of the importance of cryptanalysis in World War II and an expectation that cryptography would continue to be important for national security, many western governments have, at some point, strictly regulated export of cryptography. After World War II, it was illegal in the US to sell or distribute encryption technology overseas; in fact, encryption was classified as a munition, like tanks and nuclear weapons.[32] Until the advent of the personal computer and the Internet, this was not especially problematic. Good cryptography is indistinguishable from bad cryptography for nearly all users, and in any case, most of the cryptographic techniques generally available were slow and error prone whether good or bad. However, as the Internet grew and computers became more widely available, high quality

> encryption techniques became well-known around the globe. As a result, export controls came to be seen to be an impediment to commerce and to research.

Introduction to the Advanced Encryption Standard:

The Advanced Encryption Standard, in the following referenced as AES, is the winner of the contest, held in 1997 by the US Government, after the Data Encryption Standard was found too weak because of its small key size and the technological advancements in processor power. Fifteen candidates were accepted in 1998 and based on public comments the pool was reduced to five finalists in 1999. In October 2000, one of these five algorithms was selected as the forthcoming standard: a slightly modified version of the Rijndael.

The Rijndael, whose name is based on the names of its two Belgian inventors, Joan Daemen and Vincent Rijmen, is a Block cipher, which means that it works on fixed-length group of bits, which are called *blocks*. It takes an input block of a certain size, usually 128, and produces a corresponding output block of the same size. The transformation requires a second input, which is the secret key. It is important to know that the secret key can be of any size (depending on the cipher used) and that AES uses three different key sizes: 128, 192 and 256 bits.

To encrypt messages longer than the block size, a mode of operation is chosen, which I will explain at the very end of this tutorial, after the implementation of AES.

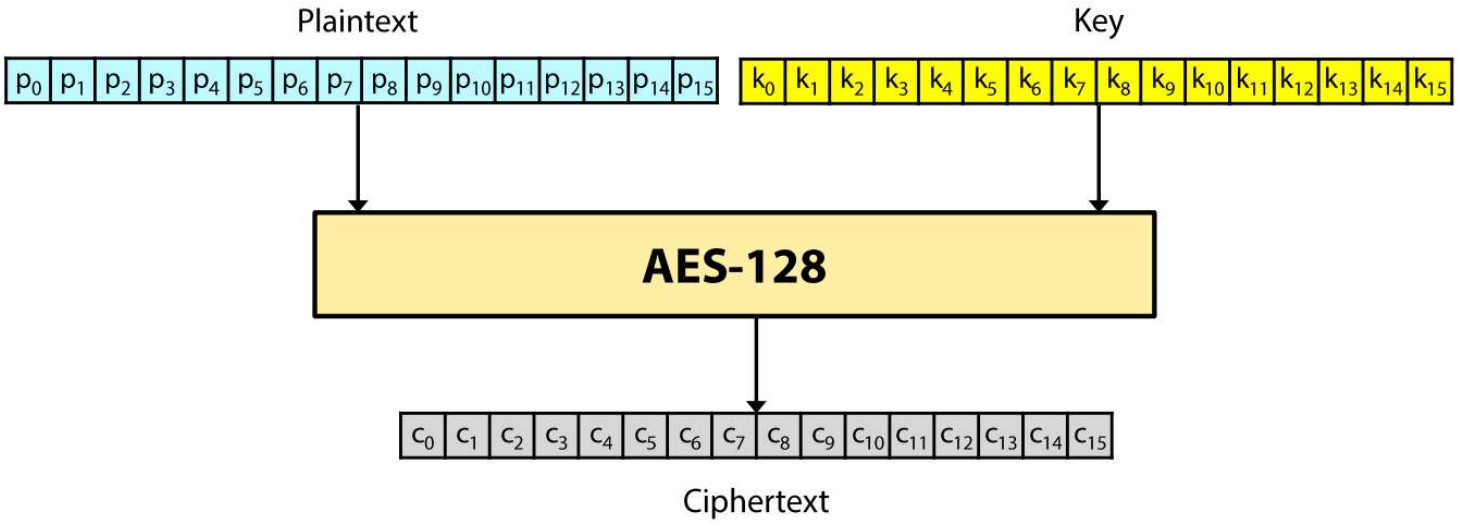
While AES supports only block sizes of 128 bits and key sizes of 128, 192 and 256 bits, the original Rijndael supports key and block sizes in any multiple of 32, with a minimum of 128 and a maximum of 256 bits.

Further readings:

Unlike DES, which is based on an Feistel network, AES is a substitution-permutation network, which is a series of mathematical operations that use substitutions (also called S-Box) and permutations (P-Boxes) and their careful definition implies that each output bit depends on every input bit.

Description of the Advanced Encryption Standard algorithm

AES is an iterated block cipher with a fixed block size of 128 and a variable key length. The different transformations operate on the intermediate results, called *state*. The state is a rectangular array of bytes and since the block size is 128 bits, which is 16 bytes, the rectangular array is of dimensions 4x4. (In the Rijndael version with variable block size, the row size is fixed to four and the number of columns vary. The number of columns is the block size divided by 32 and denoted Nb). The cipher key is similarly pictured as a rectangular array with four rows. The number of columns of the cipher key, denoted Nk, is equal to the key length divided by 32.



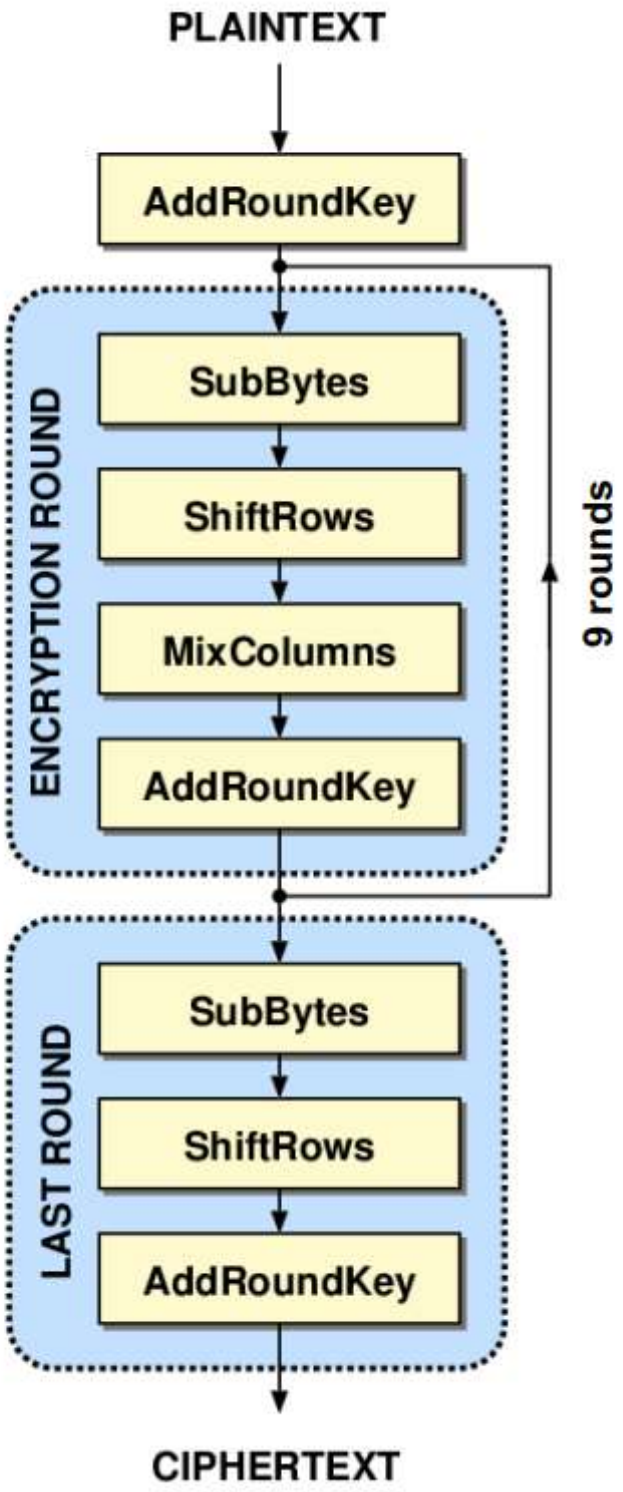

```
A state:
-----------------------------
| a0,0 | a0,1 | a0,2 | a0,3 |
| a1,0 | a1,1 | a1,2 | a1,3 |
| a2,0 | a2,1 | a2,2 | a2,3 |
| a3,0 | a3,1 | a3,2 | a3,3 |
-----------------------------

A key:
-----------------------------
| k0,0 | k0,1 | k0,2 | k0,3 |
| k1,0 | k1,1 | k1,2 | k1,3 |
| k2,0 | k2,1 | k2,2 | k2,3 |
| k3,0 | k3,1 | k3,2 | k3,3 |
-----------------------------
```

It is very *important* to know that the cipher input bytes are mapped onto the the state bytes in the order a0,0, a1,0, a2,0, a3,0, a0,1, a1,1, a2,1, a3,1 ... and the bytes of the cipher key are mapped onto the array in the order k0,0, k1,0, k2,0, k3,0, k0,1, k1,1, k2,1, k3,1 ... At the end of the cipher operation, the cipher output is extracted from the state by taking the state bytes in the same order. AES uses a variable number of rounds, which are fixed: A key of size 128 has 10 rounds. A key of size 192 has 12 rounds. A key of size 256 has 14 rounds. During each round, the following operations are applied on the state:

1. SubBytes: every byte in the state is replaced by another one, using the Rijndael S-Box
2. ShiftRow: every row in the 4x4 array is shifted a certain amount to the left
3. MixColumn: a linear transformation on the columns of the state
4. AddRoundKey: each byte of the state is combined with a round key, which is a different key for each round and derived from the Rijndael key schedule

In the final round, the MixColumn operation is omitted. The algorithm looks like the following (pseudo-C):

```
AES(state, CipherKey)
{
    KeyExpansion(CipherKey, ExpandedKey);
    AddRoundKey(state, ExpandedKey);
    for (i = 1; i < Nr; i++)
    {
        Round(state, ExpandedKey + Nb*i);
    }
    FinalRound(state, ExpandedKey + Nb * Nr);
}
```



Observations:

- The cipher key is expanded into a larger key, which is later used for the actual operations
- The roundKey is added to the state before starting the with loop
- The FinalRound() is the same as Round(), apart from missing the MixColumns() operation.
- During each round, another part of the ExpandedKey is used for the operations
- The ExpandedKey shall ALWAYS be derived from the Cipher Key and never be specified directly.

AES operations: SubBytes, ShiftRow, MixColumn and AddRoundKey

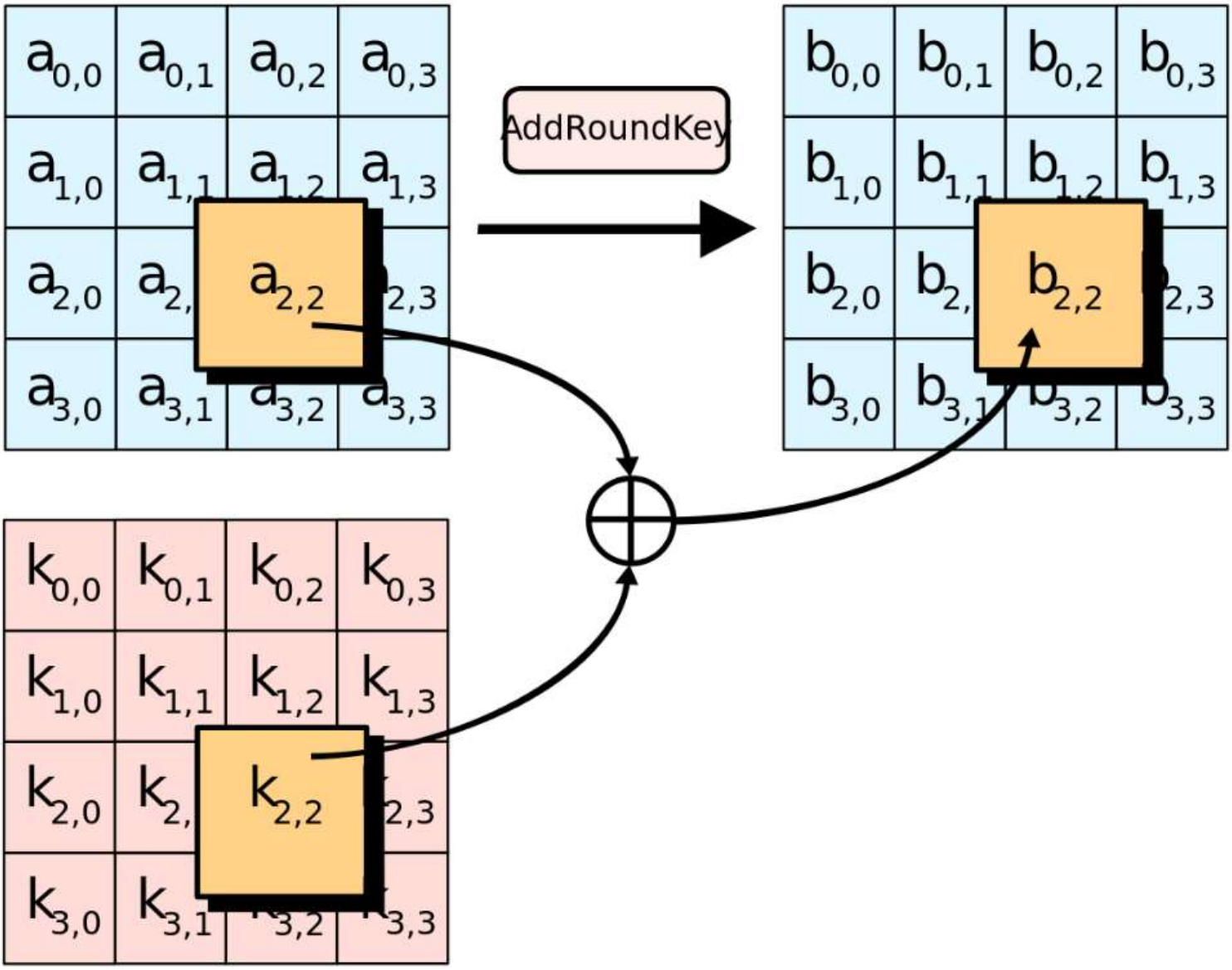
Operation	Description	Inverse Operation
SubBytes	A non-linear byte substitution using the S-Box.	InvSubBytes (uses the inverse S-Box).
ShiftRow	A cyclic left shift of the rows (0, 1, 2, and 3 positions).	InvShiftRows (cyclic right shift).
MixColumn	A linear transformation (matrix multiplication) on the columns over the Galois Field (GF(2^8)). Omitted in the final round.	InvMixColumns (uses an inverse matrix).
AddRoundKey	A simple bitwise XOR of the state with a derived Round Key.	AddRoundKey (XOR is its own inverse).

The AddRoundKey operation:

In this operation, a Round Key is applied to the state by a simple bitwise XOR. The Round Key is derived from the Cipher Key by the means of the key schedule. The Round Key length is equal to the block key length (=16 bytes).

```
----------------------------      ----------------------------   ----------------------------
| a0,0 | a0,1 | a0,2 | a0,3 |      | k0,0 | k0,1 | k0,2 | k0,3 |   | b0,0 | b0,1 | b0,2 | b0,3 |
| a1,0 | a1,1 | a1,2 | a1,3 |  XOR | k2,0 | k2,1 | k2,2 | k2,3 | = | b2,0 | b2,1 | b2,2 | b2,3 |
| a2,0 | a2,1 | a2,2 | a2,3 |      | k1,0 | k1,1 | k1,2 | k1,3 |   | b1,0 | b1,1 | b1,2 | b1,3 |
| a3,0 | a3,1 | a3,2 | a3,3 |      | k3,0 | k3,1 | k3,2 | k3,3 |   | b3,0 | b3,1 | b3,2 | b3,3 |
----------------------------      ----------------------------   ----------------------------

where: b(i,j) = a(i,j) XOR k(i,j)
```

A graphical representation of this operation can be seen below:



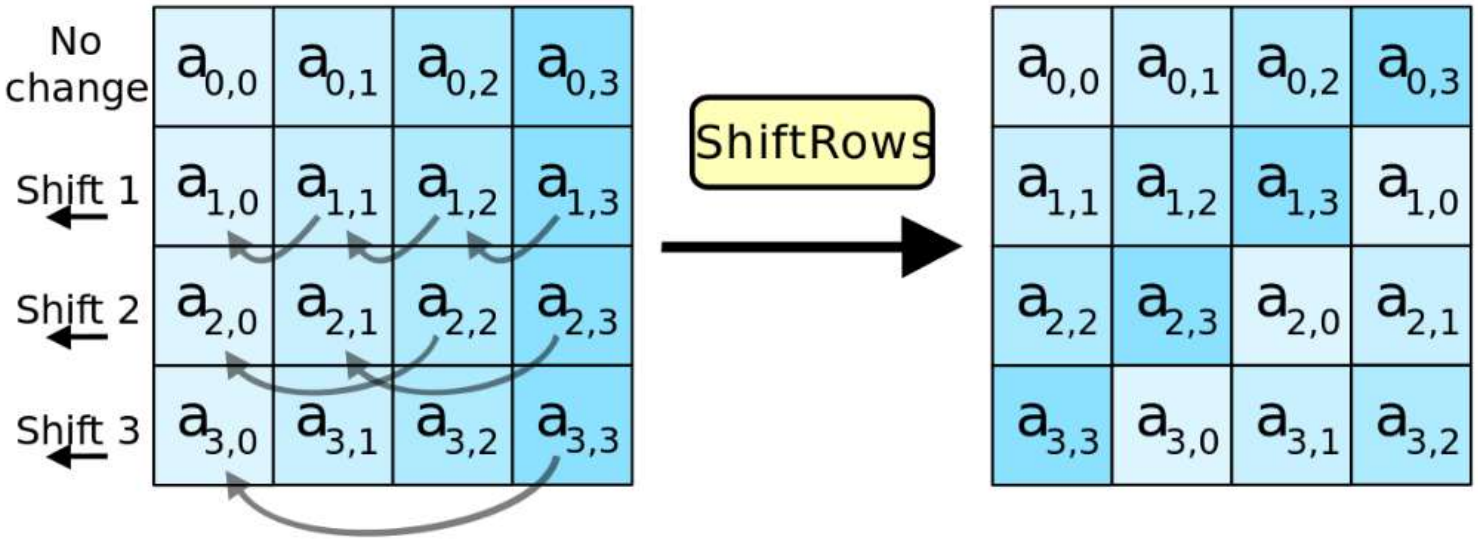
The ShiftRow operation:

In this operation, each row of the state is cyclically shifted to the left, depending on the row index.

- The 1st row is shifted 0 positions to the left.
- The 2nd row is shifted 1 positions to the left.
- The 3rd row is shifted 2 positions to the left.
- The 4th row is shifted 3 positions to the left.

```
-----------------------------    -----------------------------
| a0,0 | a0,1 | a0,2 | a0,3 |    | a0,0 | a0,1 | a0,2 | a0,3 |
| a1,0 | a1,1 | a1,2 | a1,3 | -> | a1,1 | a1,2 | a1,3 | a1,0 |
| a2,0 | a2,1 | a2,2 | a2,3 |    | a2,2 | a2,3 | a2,0 | a2,1 |
| a3,0 | a3,1 | a3,2 | a3,3 |    | a3,3 | a3,0 | a3,1 | a3,2 |
-----------------------------    -----------------------------
```

A graphical representation of this operation can be found below:



Please note that the inverse of ShiftRow is the same cyclically shift but this time to the right. It will be needed later for decoding.

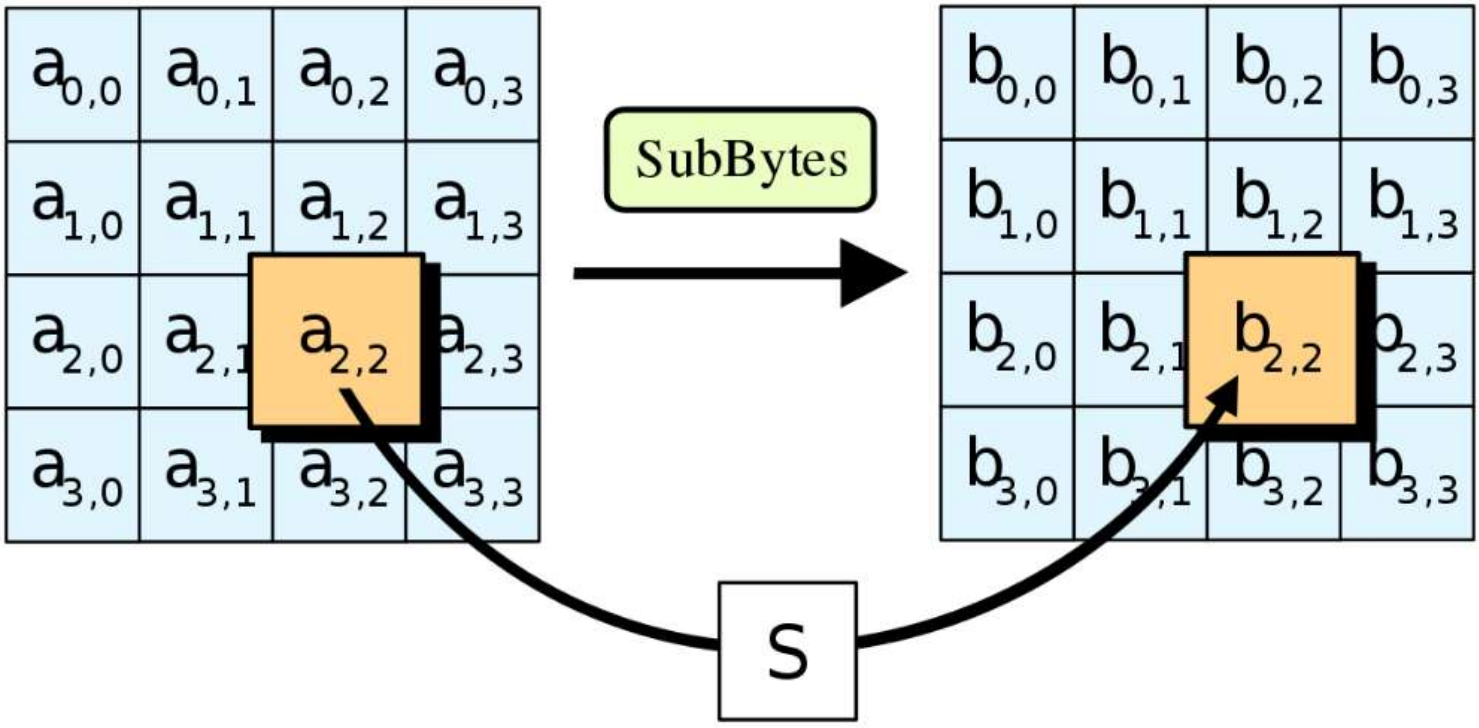
The SubBytes operation:

The SubBytes operation is a non-linear byte substitution, operating on each byte of the state independently. The substitution table (S-Box) is invertible and is constructed by the composition of two transformations:

1. Take the multiplicative inverse in Rijndael's finite field
2. Apply an affine transformation which is documented in the Rijndael documentation.

Since the S-Box is independent of any input, pre-calculated forms are used, if enough memory (256 bytes for one S-Box) is available. Each byte of the state is then substituted by the value in the S-Box whose index corresponds to the value in the state:

```
a(i,j) = SBox[a(i,j)]
```



Please note that the inverse of SubBytes is the same operation, using the inversed S-Box, which is also precalculated.

The MixColumn operation:

I will keep this section very short since it involves a lot of very advance mathematical calculations in the Rijndael's finite field. All you have to know is that it corresponds to the matrix multiplication with:

```
2 3 1 1
1 2 3 1
1 1 2 3
3 1 1 2
```

and that the addition and multiplication operations are a little different from the normal ones.

You can skip this part if you are not interested in the math involved:

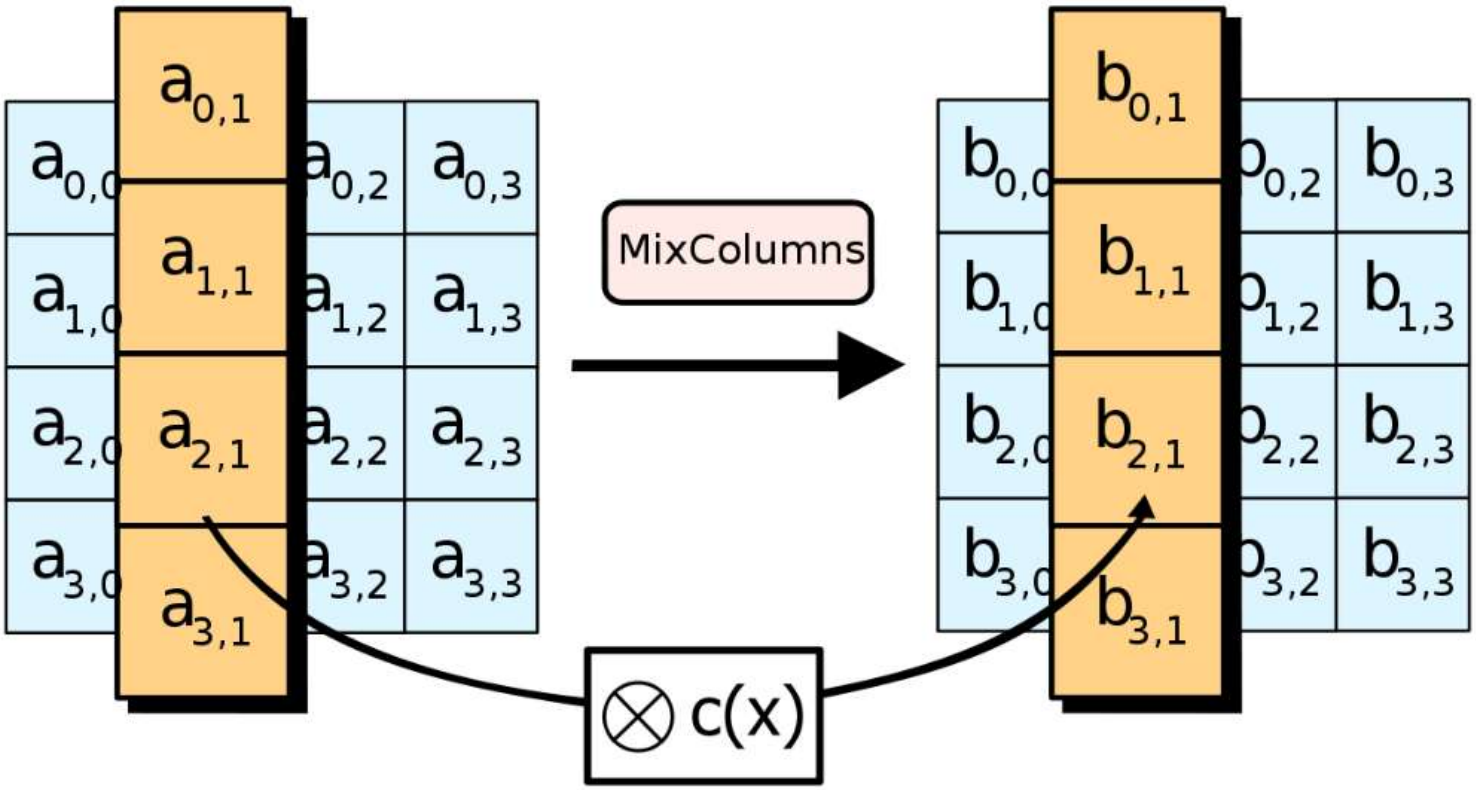
> Addition and Substraction:
>
> Addition and subtraction are performed by the exclusive or operation. The two operations are the same; there is no difference between addition and subtraction.
>
> Multiplication in Rijndael's galois field is a little more complicated. The procedure is as follows:
>
> - Take two eight-bit numbers, a and b, and an eight-bit product p

- Set the product to zero.
- Make a copy of a and b, which we will simply call a and b in the rest of this algorithm

```
Run the following loop eight times:

    1. If the low bit of b is set, exclusive or the product p by
       the value of a
    2. Keep track of whether the high (eighth from left) bit of a
       is set to one
    3. Rotate a one bit to the left, discarding the high bit, and
       making the low bit have a value of zero
    4. If a's hi bit had a value of one prior to this rotation,
       exclusive or a with the hexadecimal number 0x1b
    5. Rotate b one bit to the right, discarding the low bit, and
       making the high (eighth from left) bit have a value of
       zero.
```

- The product p now has the product of a and b

Thanks to Sam Trenholme for writing this explanation.



The Rijndael Key Schedule:

The Key Schedule is responsible for expanding a short key into a larger key, whose parts are used during the different iterations. Each key size is expanded to a different size:

- An 128 bit key is expanded to an 176 byte key.
- An 192 bit key is expanded to an 208 byte key.
- An 256 bit key is expanded to an 240 byte key.

There is a relation between the cipher key size, the number of rounds and the ExpandedKey size. For an 128-bit key, there is one initial AddRoundKey operation plus there are 10 rounds and each round needs a new 16 byte key, therefor we require 10+1 RoundKeys of 16 byte, which equals 176 byte. The same logic can be applied to the two other cipher key sizes. The general formula is that:

ExpandedKeySize = (nbrRounds+1) * BlockSize

The Key Schedule is made up of iterations of the Key schedule core, which works on 4-byte *words*. The core uses a certain number of operations, which are explained here:

Rotate:

The 4-byte word is cyclically shifted 1 byte to the left:

```
--------------------    --------------------
| 1d | 2c | 3a | 4f | -> | 2c | 3a | 4f | 1d |
--------------------    --------------------
```

Rcon:

This section is again extremely mathematical and I recommend everyone who is interested to read this description. Just note that the Rcon values can be pre-calculated, which results in a simple substitution (a table lookup) in a fixed Rcon table (again, Rcon can also be calculated on-the-fly if memory is a design constraint.)

S-Box:

The Key Schedule uses the same S-Box substitution as the main algorithm body.

The Key Schedule Core:

Now that we know what the operations are, let me show you the key schedule core (in pseudo-C):

```
keyScheduleCore(word)
{
```

```
    Rotate(word);
    SBoxSubstitution(word);
    word[0] = word[0] XOR RCON[i];
}
```

In the above code, word has a size of 4 bytes and i is the iteration counter from the Key Schedule

The Key Expansion:

First, let me show you the keyExpansion function as you can find it in the Rijndael documentation (there are 2 version, one for key size 128, 192 and one for key size 256):

```
KeyExpansion(byte Key[4*Nk] word W[Nb*(Nr+1)])
{
    for(i = 0; i < Nk; i++)
        W[i] = (Key[4*i],Key[4*i+1],Key[4*i+2],Key[4*i+3]);
    for(i = Nk; i < Nb * (Nr + 1); i++)
    {
        temp = W[i - 1];
        if (i % Nk == 0)
            temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
        W[i] = W[i - Nk] ^ temp;
    }
}
```

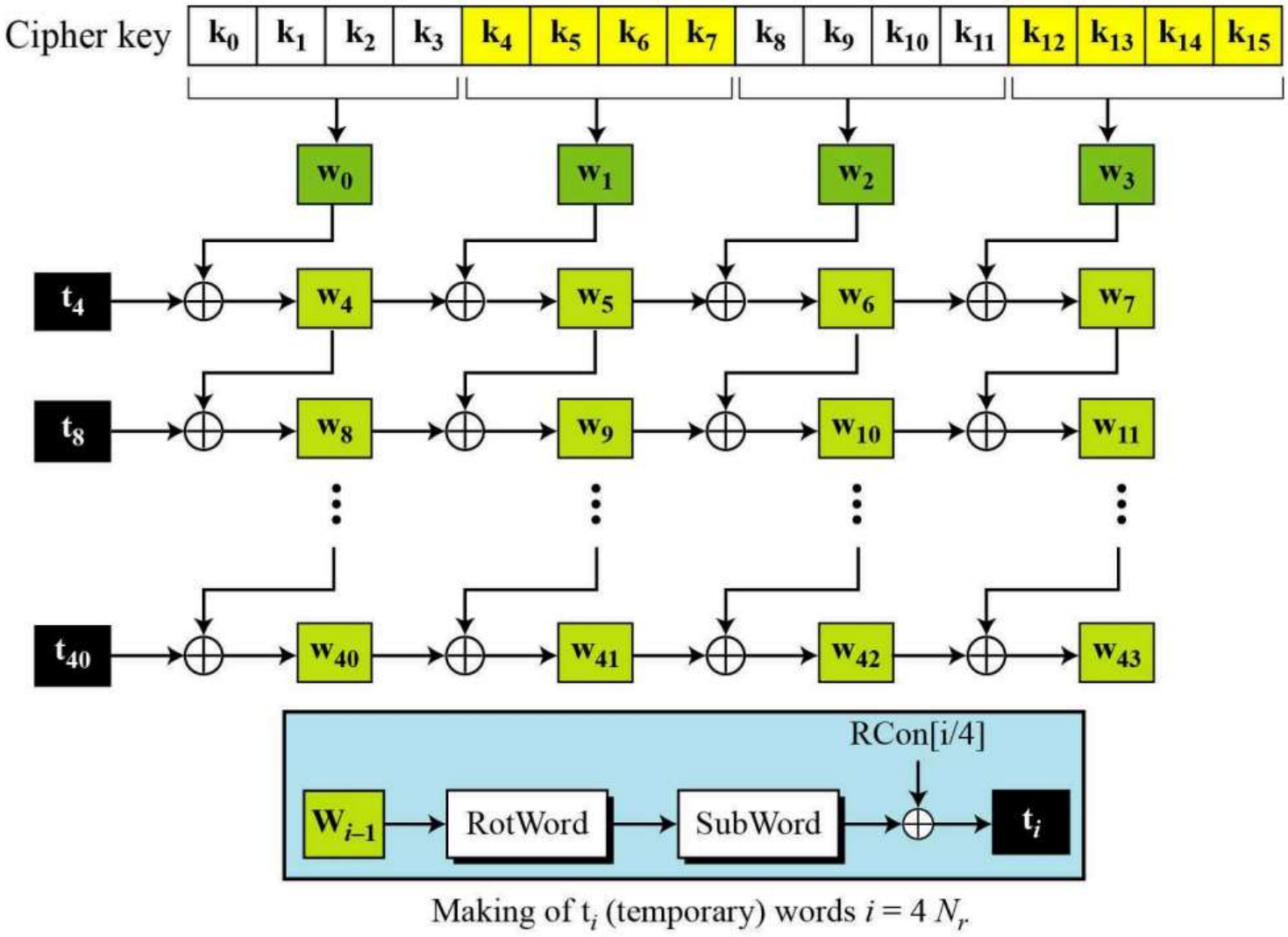
- Nk is the number of columns in the cipher key (128-bit -> 4, 192-bit -> 5, 256-bit -> 6)
- W is of type "word", which is 4-bytes

Let me try to explain this in an easier understandable way:

- The first n bytes of the expanded key are simply the cipher key (n = the size of the encryption key)
- The rcon value i is set to 1
- Until we have enough bytes of expanded key, we do the following to generate n more bytes of expanded key (please note once again that "n" is used here, this varies depending on the key size)
    i. we do the following to generate four bytes
        - we use a temporary 4-byte word called t
        - we assign the previous 4 bytes to t
        - we perform the key schedule core on t, with i as rcon value
        - we increment i
        - we XOR t with the 4-byte word n bytes before in the expandedKey (where n is once either either 16,24 or 32 bytes)

ii. we do the following x times to generate the next x*4 bytes of the expandedKey (x = 3 for n=16,32 and x = 5 for n=24)
    - we assign the previous 4-byte word to t
    - we XOR t with the 4-byte word n bytes before in the expandedKey (where n is once either either 16,24 or 32 bytes)
iii. if n = 32 (and ONLY then), we do the following to generate 4 more bytes
    - we assign the previous 4-byte word to t
    - We run each of the four bytes in t through Rijndael's S-box
    - we XOR t with the 4-byte word 32 bytes before in the expandedKey
iv. if n = 32 (and ONLY then), we do the following three times to generate twelve more bytes
    - we assign the previous 4-byte word to t
    - we XOR t with the 4-byte word 32 bytes before in the expandedKey

- We now have our expandedKey



Making of t_i (temporary) words $i = 4\ N_r$

Don't worry if you still have problems understanding the Key Schedule, you'll see that the implementation isn't very hard. What you should note is that:

- the part in red is only for cipher key size = 32
- for n=16, we generate: 4 + 3*4 bytes = 16 bytes per iteration
- for n=24, we generate: 4 + 5*4 bytes = 24 bytes per iteration
- for n=32, we generate: 4 + 3*4 + 4 + 3*4 = 32 bytes per iteration

The implementation of the key schedule is pretty straight forward, but since there is a lot of code repetition, it is possible to optimize the loop slightly and use the modulo operator to check when the additional operations have to be made.

Implementation: The Key Schedule

We will start the implementation of AES with the Cipher Key expansion. As you can read in the theoretical part above, we intend to enlarge our input cipher key, whose size varies between 128 and 256 bits into a larger key, from which different RoundKeys can be derived.

I prefer to implement the helper functions (such as rotate, Rcon or S-Box first), test them and then move on to the larger loops. If you are not a fan of bottom-up approaches, feel free to start a little further in this tutorial and move your way up, but I felt that my approach was the more logical one here.

Implementation: General comments

Even though some might think that integers were the best choice to work with, since their 32 bit size best corresponds one *word*, I strongly discourage you from using integers. You wrongly assume that integers, or more specifically the "int" type, always has 4 bytes. However, the required ranges for signed and unsigned int are identical to those for signed and unsigned short. On compilers for 8 and 16 bit processors (including Intel x86 processors executing in 16 bit mode, such as under MS-DOS), an int is usually 16 bits and has exactly the same representation as a short. On compilers for 32 bit and larger processors (including Intel x86 processors executing in 32 bit mode, such as Win32 or Linux) an int is usually 32 bits long and has exactly the same representation as a long. | For this very reason, we will be using unsigned chars, since the size of an char (which is called CHAR_BIT and defined in limits.h) is required to be at least 8. Jack Klein wrote:

Almost all modern computers today use 8 bit bytes (technically called octets, but there are still some in production and use with other sizes, such as 9 bits. Also some processors (especially Digital Signal Processors) cannot efficiently access memory in smaller pieces than the processor's word size. There is at least one DSP I have worked with where CHAR_BIT is 32. The char types, short, int and long are all 32 bits.

Since we want to keep our code as portable as possible and since it is up to the compiler to decide if the default type for char is signed or not, we will specify unsigned char throughout the entire code.

Implementation: S-Box

The S-Box values can either be calculated on-the-fly to save memory or the pre-calculated values can be stored in an array. Since I assume that every machine my code runs on will have at least 2x 256bytes (there are 2 S-Boxes, one for the encryption and one for the decryption) we will store the values in an array. Additionally, instead of accessing the values immediately from our program, I'll wrap a little function around which makes for a more readable code and would allow us to add additional code later on. Of course, this is a matter of taste, feel free to access the array immediately.

	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

Here's the code for the 2 S-Boxes, it's only a table-lookup that returns the value in the array whose index is specified as a parameter of the function:

```
unsigned char sbox[256] =   {
//0     1    2      3     4    5     6     7      8    9     A      B    C     D     E     F
0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, /
0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, /
0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, /
```

```
0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, /
0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, /
0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, /
0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, /
0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, /
0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, /
0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, /
0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, /
0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, /
0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, /
0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, /
0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, /
0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 };

unsigned char rsbox[256] =
{ 0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb
, 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb
, 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e
, 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25
, 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92
, 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84
, 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06
, 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b
, 0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73
, 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e
, 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b
, 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4
, 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f
, 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef
, 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61
, 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d

unsigned char getSBoxValue(unsigned char num)
{
    return sbox[num];
}

unsigned char getSBoxInvert(unsigned char num)
{
    return rsbox[num];
}
```

Implementation: Rotate

From the theoretical part, you should know already that Rotate takes a word (a 4-byte array) and rotates it 8 bit to the left. Since 8 bit correspond to one byte and our array type is character (whose size is one byte), rotating 8 bit to the left corresponds to shifting cyclically the array values one to the left.

Here's the code for the Rotate function:

```
/* Rijndael's key schedule rotate operation
 * rotate the word eight bits to the left
 *
 * rotate(1d2c3a4f) = 2c3a4f1d
 *
 * word is an char array of size 4 (32 bit)
 */
void rotate(unsigned char *word)
{
    unsigned char c;
    int i;

    c = word[0];
    for (i = 0; i < 3; i++)
        word[i] = word[i+1];
    word[3] = c;
}
```

Implementation: Rcon

Same as with the S-Box, the Rcon values can be calculated on-the-fly but once again I decide to store them in an array since they only require 255 bytes of space. To keep in line with the S-Box implementation, I write a little access function.

Here's the code for Rcon:

```
unsigned char Rcon[255] = {

0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,
0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,
0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,
0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d,
0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab,
0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25,
0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01,
0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d,
0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa,
0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a,
0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02,
0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,
0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef,
0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
```

```
0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f,
0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5,
0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33,
0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb};

unsigned char getRconValue(unsigned char num)
{
    return Rcon[num];
}
```

Implementation: Key Schedule Core

The implementation of the Key Schedule Core from the pseudo-C is pretty easy. All the code does is apply the operations one after the other on the 4-byte word. The parameters are the 4-byte word and the iteration counter, on which Rcon depends.

```
void core(unsigned char *word, int iteration)
{
    int i;

    /* rotate the 32-bit word 8 bits to the left */
    rotate(word);

    /* apply S-Box substitution on all 4 parts of the 32-bit word */
    for (i = 0; i < 4; ++i)
    {
        word[i] = getSBoxValue(word[i]);
    }

    /* XOR the output of the rcon operation with i to the first part (leftmost) only */
    word[0] = word[0]^getRconValue(iteration);
}
```

Implementation: Key Expansion

The Key Expansion is where it all comes together. As you can see in the pretty big list in the theory about the Rijndael Key Expansion, we need to apply several operations a number of times, depending on they key size. As the key size can only take a very limited number of values, I decided to implement it as an enumeration type. Not only does that limit the key size to only three possible values, it also makes the code more readable.

```
enum keySize{
    SIZE_16 = 16,
    SIZE_24 = 24,
```

```
    SIZE_32 = 32
    };
```

Our key expansion function basically needs only two things:

- the input cipher key
- the output expanded key

Since in C, it is not possible to know the size of an array passed as pointer to a function, we'll add the cipher key size (of type "enum keySize") and the expanded key size (of type size_t) to the parameter list of our function. The prototype looks like the following:

```
void expandKey(unsigned char *expandedKey, unsigned char *key, enum keySize, size_t expandedKeySi
```

While implementing the function, I try to follow the details in the theoretical list as close as possible. As I already explained, since several parts of the code are repeated, I'll try to get rid of the code repetition and use conditions to see when I need to use a certain operation.

Instead of writing:

```
while (expanded_key_size < required_key_size)
{
    key_schedule_core(word);
    for (i=0; i<4; i++)
        some_operation();
}
```

I'll use a different structure:

```
while (expanded_key_size < required_key_size)
{
    if (expanded_key_size%key_size == 0)
        key_schedule_core(word);
    some_operation();
}
```

This structure comes down to the same thing, but allows me to be more flexible when it comes to add the 256-bit cipherkey version that has those additional steps.

Let me show you the keyexpansion function and give explanations later on:

```
/* Rijndael's key expansion
 * expands an 128,192,256 key into an 176,208,240 bytes key
 *
 * expandedKey is a pointer to an char array of large enough size
 * key is a pointer to a non-expanded key
 */

void expandKey(unsigned char *expandedKey,
               unsigned char *key,
               enum keySize size,
               size_t expandedKeySize)
{
    /* current expanded keySize, in bytes */
    int currentSize = 0;
    int rconIteration = 1;
    int i;
    unsigned char t[4] = {0};   // temporary 4-byte variable

    /* set the 16,24,32 bytes of the expanded key to the input key */
    for (i = 0; i < size; i++)
        expandedKey[i] = key[i];
    currentSize += size;

    while (currentSize < expandedKeySize)
    {
        /* assign the previous 4 bytes to the temporary value t */
        for (i = 0; i < 4; i++)
        {
            t[i] = expandedKey[(currentSize - 4) + i];
        }

        /* every 16,24,32 bytes we apply the core schedule to t
         * and increment rconIteration afterwards
         */
        if(currentSize % size == 0)
        {
            core(t, rconIteration++);
        }

        /* For 256-bit keys, we add an extra sbox to the calculation */
        if(size == SIZE_32 && ((currentSize % size) == 16)) {
            for(i = 0; i < 4; i++)
                t[i] = getSBoxValue(t[i]);
        }

        /* We XOR t with the four-byte block 16,24,32 bytes before the new expanded key.
         * This becomes the next four bytes in the expanded key.
         */
```

```
        for(i = 0; i < 4; i++) {
            expandedKey[currentSize] = expandedKey[currentSize - size] ^ t[i];
            currentSize++;
        }
    }
}
```

As you can see, I never use inner loops to repeat an operation, the only inner loops are to iterate over the 4 parts of the temporary array t. I use the modulo operator to check if I need to apply the operation:

- *if(currentSize % size == 0)*: whenever we have have created n bytes of expandedKey (where n is the cipherkey size), we run the key expansion core once
- *if(size == SIZE_32 && ((currentSize % size) == 16))*: if we are expanding an 32-bit cipherkey and if we have already generated 16 bytes (as I explained above, in the 32-bit version we run the first loop only 3 times, which generates 12 bytes + the 4 bytes from the core), we add one additional S-Box substitution

Implementation: Using the Key Expansion

Finally, we can test our newly created key expansion. I won't calculate the expandedKey size just yet but rather give it a fixed value (the calculation requires the number of rounds which isn't needed at this point). Here's the code that would expand a given cipher key:

```
/* the expanded keySize */
int expandedKeySize = 176;

/* the expanded key */
unsigned char expandedKey[expandedKeySize];

/* the cipher key */
unsigned char key[16] = {0};

/* the cipher key size */
enum keySize size = SIZE_16;

int i;

expandKey(expandedKey, key, size, expandedKeySize);

printf("Expanded Key:n");
for (i = 0; i < expandedKeySize; i++)
{
        printf("%2.2x%c", expandedKey[i], (i%16) ? 'n' : ' ');
}
```

Of course, this code uses several constants that will be generated automatically once we implement the body of the AES encryption.

Here are several test results:

The Key Expansion of an 128-bit key consisting of null characters (like the example above):

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
62 63 63 63 62 63 63 63 62 63 63 63 62 63 63 63
9b 98 98 c9 f9 fb fb aa 9b 98 98 c9 f9 fb fb aa
90 97 34 50 69 6c cf fa f2 f4 57 33 0b 0f ac 99
ee 06 da 7b 87 6a 15 81 75 9e 42 b2 7e 91 ee 2b
7f 2e 2b 88 f8 44 3e 09 8d da 7c bb f3 4b 92 90
ec 61 4b 85 14 25 75 8c 99 ff 09 37 6a b4 9b a7
21 75 17 87 35 50 62 0b ac af 6b 3c c6 1b f0 9b
0e f9 03 33 3b a9 61 38 97 06 0a 04 51 1d fa 9f
b1 d4 d8 e2 8a 7d b9 da 1d 7b b3 de 4c 66 49 41
b4 ef 5b cb 3e 92 e2 11 23 e9 51 cf 6f 8f 18 8e
```

The Key Expansion of an 192-bit key consisting of null characters:

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 62 63 63 63 62 63 63 63
62 63 63 63 62 63 63 63 62 63 63 63 62 63 63 63
9b 98 98 c9 f9 fb fb aa 9b 98 98 c9 f9 fb fb aa
9b 98 98 c9 f9 fb fb aa 90 97 34 50 69 6c cf fa
f2 f4 57 33 0b 0f ac 99 90 97 34 50 69 6c cf fa
c8 1d 19 a9 a1 71 d6 53 53 85 81 60 58 8a 2d f9
c8 1d 19 a9 a1 71 d6 53 7b eb f4 9b da 9a 22 c8
89 1f a3 a8 d1 95 8e 51 19 88 97 f8 b8 f9 41 ab
c2 68 96 f7 18 f2 b4 3f 91 ed 17 97 40 78 99 c6
59 f0 0e 3e e1 09 4f 95 83 ec bc 0f 9b 1e 08 30
0a f3 1f a7 4a 8b 86 61 13 7b 88 5f f2 72 c7 ca
43 2a c8 86 d8 34 c0 b6 d2 c7 df 11 98 4c 59 70
```

The Key Expansion of an 256-bit key consisting of null characters:

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
62 63 63 63 62 63 63 63 62 63 63 63 62 63 63 63
aa fb fb fb aa fb fb fb aa fb fb fb aa fb fb fb
6f 6c 6c cf 0d 0f 0f ac 6f 6c 6c cf 0d 0f 0f ac
7d 8d 8d 6a d7 76 76 91 7d 8d 8d 6a d7 76 76 91
53 54 ed c1 5e 5b e2 6d 31 37 8e a2 3c 38 81 0e
96 8a 81 c1 41 fc f7 50 3c 71 7a 3a eb 07 0c ab
```

```
9e aa 8f 28 c0 f1 6d 45 f1 c6 e3 e7 cd fe 62 e9
2b 31 2b df 6a cd dc 8f 56 bc a6 b5 bd bb aa 1e
64 06 fd 52 a4 f7 90 17 55 31 73 f0 98 cf 11 19
6d bb a9 0b 07 76 75 84 51 ca d3 31 ec 71 79 2f
e7 b0 e8 9c 43 47 78 8b 16 76 0b 7b 8e b9 1a 62
74 ed 0b a1 73 9b 7e 25 22 51 ad 14 ce 20 d4 3b
10 f8 0a 17 53 bf 72 9c 45 c9 79 e7 cb 70 63 85
```

Implementation: AES Encryption

To implement the AES encryption algorithm, we proceed exactly the same way as for the key expansion, that is, we first implement the basic helper functions and then move up to the main loop. The functions take as parameter a *state*, which is, as already explained, a rectangular 4x4 array of bytes. We won't consider the state as a 2-dimensional array, but as a 1-dimensional array of length 16.

Implementation: subBytes

There isn't much to say about this operation, it's a simple substitution with the S-Box value:

```
void subBytes(unsigned char *state)
{
    int i;
    /* substitute all the values from the state with the value in the SBox
     * using the state value as index for the SBox
     */
    for (i = 0; i < 16; i++)
        state[i] = getSBoxValue(state[i]);
}
```

Implementation: shiftRows

I decided to split this function in two parts, not that it wasn't possible to do it all in one go, but simply because it was easier to read and debug. The shiftRows function iterates over all the rows and then call shiftRow with the correct offset. shiftRow does nothing but to shift a 4-byte array by the given offset.

```
void shiftRows(unsigned char *state)
{
    int i;
    /* iterate over the 4 rows and call shiftRow() with that row */
    for (i = 0; i < 4; i++)
        shiftRow(state+i*4, i);
}
```

```
void shiftRow(unsigned char *state, unsigned char nbr)
{
    int i, j;
    unsigned char tmp;
    /* each iteration shifts the row to the left by 1 */
    for (i = 0; i < nbr; i++)
    {
        tmp = state[0];
        for (j = 0; j < 3; j++)
            state[j] = state[j+1];
        state[3] = tmp;
    }
}
```

This implementation is the least efficient but the easiest to understand. A very simple improvement would be, since the first row isn't shifted, to have the iterator in shiftRows start at 1 instead of

1.

Implementation: addRoundKey

This is the part that involves the roundKey we generate during each iteration. We simply XOR each byte of the key to the respective byte of the state.

```
void addRoundKey(unsigned char *state, unsigned char *roundKey)
{
    int i;
    for (i = 0; i < 16; i++)
        state[i] = state[i] ^ roundKey[i] ;
}
```

Implementation: mixColumns

mixColumns is probably the most difficult operation of the 4. It involves the galois addition and multiplication and processes columns instead of rows (which is unfortunate since we use a linear array that represents the rows).

First of all, we need a function that multiplies two number in the galois field. I didn't bother to implement this one from scratch and used the one provided by Sam Trenholme instead:

```
unsigned char galois_multiplication(unsigned char a, unsigned char b)
{
    unsigned char p = 0;
    unsigned char counter;
```

```
    unsigned char hi_bit_set;
    for(counter = 0; counter < 8; counter++) {
        if((b & 1) == 1)
            p ^= a;
        hi_bit_set = (a & 0x80);
        a <<= 1;
        if(hi_bit_set == 0x80)
            a ^= 0x1b;
        b >>= 1;
    }
    return p;
}
```

Once again, I decided to split the function in 2 parts, the first one would generate a column and then call mixColumn, which would then apply the matrix multiplication.

```
void mixColumns(unsigned char *state)
{
    int i, j;
    unsigned char column[4];

    /* iterate over the 4 columns */
    for (i = 0; i < 4; i++)
    {
        /* construct one column by iterating over the 4 rows */
        for (j = 0; j < 4; j++)
        {
            column[j] = state[(j*4)+i];
        }

        /* apply the mixColumn on one column */
        mixColumn(column);

        /* put the values back into the state */
        for (j = 0; j < 4; j++)
        {
            state[(j*4)+i] = column[j];
        }
    }
}
```

The mixColumn is simply a galois multiplication of the column with the 4x4 matrix provided in the theory. Since an addition corresponds to a XOR operation and we already have the multiplication function, the implementation is rather simple:

```
void mixColumn(unsigned char *column)
{
    unsigned char cpy[4];
    int i;
    for(i = 0; i < 4; i++)
    {
        cpy[i] = column[i];
    }
    column[0] = galois_multiplication(cpy[0],2) ^
                galois_multiplication(cpy[3],1) ^
                galois_multiplication(cpy[2],1) ^
                galois_multiplication(cpy[1],3);

    column[1] = galois_multiplication(cpy[1],2) ^
                galois_multiplication(cpy[0],1) ^
                galois_multiplication(cpy[3],1) ^
                galois_multiplication(cpy[2],3);

    column[2] = galois_multiplication(cpy[2],2) ^
                galois_multiplication(cpy[1],1) ^
                galois_multiplication(cpy[0],1) ^
                galois_multiplication(cpy[3],3);

    column[3] = galois_multiplication(cpy[3],2) ^
                galois_multiplication(cpy[2],1) ^
                galois_multiplication(cpy[1],1) ^
                galois_multiplication(cpy[0],3);
}
```

Once again, this function could be optimized (like using memcpy instead of the loop) but I left the formulas in their unsimplified form to make them easier to read.

Implementation: AES round

As you can see in the theory, one AES round does nothing but to apply all four operations on the state consecutively.

```
void aes_round(unsigned char *state, unsigned char *roundKey)
{
    subBytes(state);
    shiftRows(state);
    mixColumns(state);
    addRoundKey(state, roundKey);
}
```

Implementation: the main AES body

Now that we have all the small functions, the main loop gets really easy. All we have to do is take the state, the expandedKey and the number of rounds as parameters and then call the operations one after the other. A little function called createRoundKey() is used to copy the next 16 bytes from the expandedKey into the roundKey, using the special mapping order.

```
void createRoundKey(unsigned char *expandedKey, unsigned char *roundKey)
{
    int i,j;
    /* iterate over the columns */
    for (i = 0; i < 4; i++)
    {
        /* iterate over the rows */
        for (j = 0; j < 4; j++)
            roundKey[(i+(j*4))] = expandedKey[(i*4)+j];
    }
}


void aes_main(unsigned char *state, unsigned char *expandedKey, int nbrRounds)
{
    int i = 0;

    unsigned char roundKey[16];

    createRoundKey(expandedKey, roundKey);
    addRoundKey(state, roundKey);

    for (i = 1; i < nbrRounds; i++) {
        createRoundKey(expandedKey + 16*i, roundKey);
        aes_round(state, roundKey);
    }

    createRoundKey(expandedKey + 16*nbrRounds, roundKey);
    subBytes(state);
    shiftRows(state);
    addRoundKey(state, roundKey);
}
```

Implementation: AES encryption

Finally, all we have to do is put it all together. Our parameters are the input plaintext, the key of size keySize and the output. First, we calculate the number of rounds based on they keySize and then the expandedKeySize based on the number of rounds. Then we have to map the 16 byte input plaintext in the correct order to the 4x4 byte state (as explained above), expand the key using our key schedule,

encrypt the state using our main AES body and finally unmap the state again in the correct order to get the 16 byte output ciphertext.

Sounds complicated, but you'll see that the code really isn't:

```
char aes_encrypt(unsigned char *input,
                 unsigned char *output,
                 unsigned char *key,
                 enum keySize size)
{
    /* the expanded keySize */
    int expandedKeySize;

    /* the number of rounds */
    int nbrRounds;

    /* the expanded key */
    unsigned char *expandedKey;

    /* the 128 bit block to encode */
    unsigned char block[16];

    int i,j;

    /* set the number of rounds */
    switch (size)
    {
        case SIZE_16:
            nbrRounds = 10;
            break;
        case SIZE_24:
            nbrRounds = 12;
            break;
        case SIZE_32:
            nbrRounds = 14;
            break;
        default:
            return UNKNOWN_KEYSIZE;
            break;
    }

    expandedKeySize = (16*(nbrRounds+1));
    if ((expandedKey = malloc(expandedKeySize * sizeof(char))) == NULL)
    {
        return MEMORY_ALLOCATION_PROBLEM;
    }

    /* Set the block values, for the block:
```

```
     * a0,0 a0,1 a0,2 a0,3
     * a1,0 a1,1 a1,2 a1,3
     * a2,0 a2,1 a2,2 a2,3
     * a3,0 a3,1 a3,2 a3,3
     * the mapping order is a0,0 a1,0 a2,0 a3,0 a0,1 a1,1 ... a2,3 a3,3
     */

    /* iterate over the columns */
    for (i = 0; i < 4; i++)
    {
        /* iterate over the rows */
        for (j = 0; j < 4; j++)
            block[(i+(j*4))] = input[(i*4)+j];
    }

    /* expand the key into an 176, 208, 240 bytes key */
    expandKey(expandedKey, key, size, expandedKeySize);

    /* encrypt the block using the expandedKey */
    aes_main(block, expandedKey, nbrRounds);

    /* unmap the block again into the output */
    for (i = 0; i < 4; i++)
    {
        /* iterate over the rows */
        for (j = 0; j < 4; j++)
            output[(i*4)+j] = block[(i+(j*4))];
    }
    return 0;
}
```

In the above code, UNKNOWN_KEYSIZE and MEMORY_ALLOCATION_PROBLEM are macros to some predefined error codes that I can use to check if everything was ok. The code shouldn't be too complicated and the comments should be enough to understand everything.