Descição das funções. Tomar atenção ás equações. Sugerido fluxogramas/pseudo-código.

RIOT -> sys -> crypto (.c)  (aes.c e ciphers.c)                RIOT -> sys -> include -> crypto (.h)

# RIOT

"RIOT is a **real-time multi-threading operating system** that supports a range of devices that are typically found in the Internet of Things (IoT): 8-bit, 16-bit and 32-bit microcontrollers.

RIOT is based on the following design principles: energy-efficiency, real-time capabilities, small memory footprint, modularity, and uniform API access, independent of the underlying hardware (this API offers partial POSIX compliance)."

## FEATURES

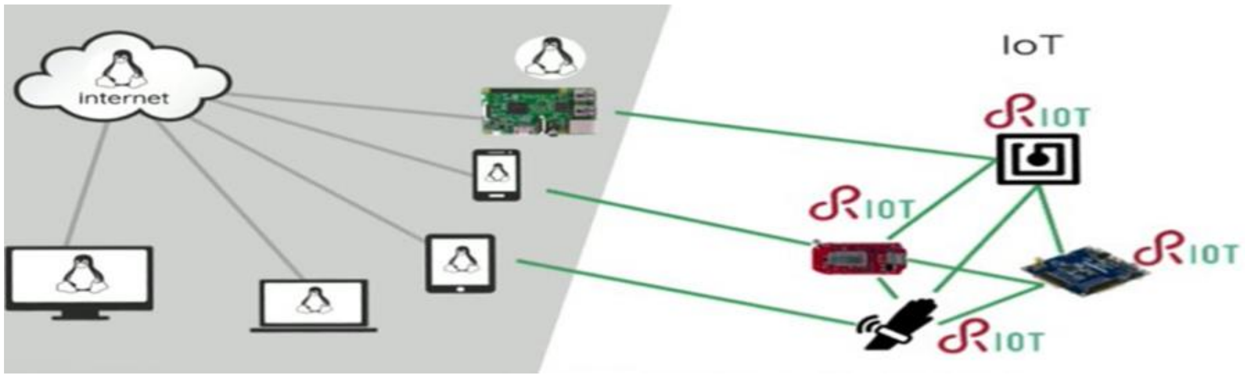
RIOT is based on a microkernel architecture, and provides features including, but not limited to:

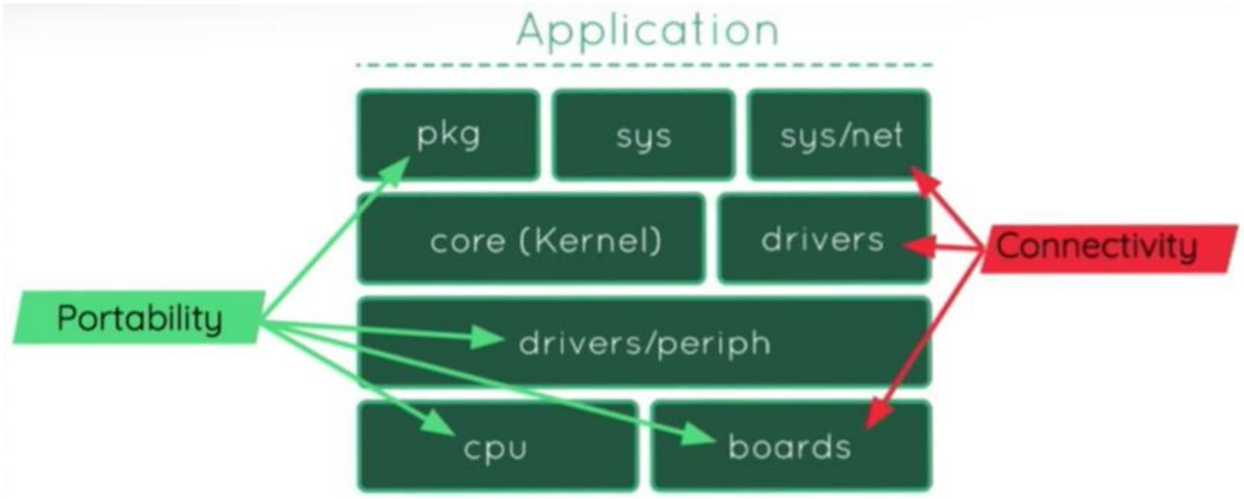
- a preemptive, tickless scheduler with priorities (energy performance)
- flexible memory management
- high resolution, long-term timers
- support 100+ boards
- the native port allows to run RIOT as-is on Linux, BSD, and MacOS. Multiple instances of RIOT running on a single machine can also be interconnected via a simple virtual Ethernet bridge
- Low latency interrupt handling (for reactivity)
- Modular structure
- Preeptive multi-threading & powerful Inter-Process Comunication (IPC)
- Portability – code your application once & run everywhere
- Posix sockets, pthreads
- Shell -  interact via shell
- Crypto & hashes (aes, 3des, …)

"If you can deploy linux on your device use RIOT" . Low-end IoT device resource constraints:

- Kernel performance
- System-level interoperability
- Network-level interoperability
- Trust

RIOT: OS that fits IoT devices





- Easy porting of RIOT to new hardware
  - `periph` interfaces
    - Porting is a matter of hours or days
    - E.g. support for new ARM Cortex-M boards is 'trivial'
  - Reusable `*_common` modules
    - Reduce code duplication

## Links

**Forum -** https://forum.riot-os.org/

**Quickstart Guide –** https://doc.riot-os.org/index.html#the-quickest-start

**Tutorial -** https://github.com/RIOT-OS/Tutorials/blob/master/README.md

**Specific Toolchain Installation -**https://doc.riot-os.org/getting-started.html

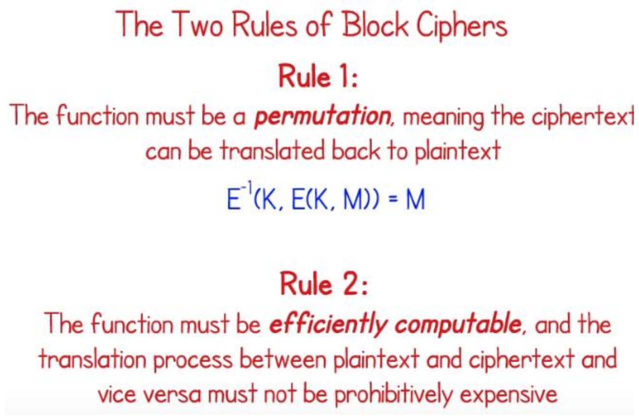
**Riot Documentation -** https://riot-os.org/api/

**Riot Intro Video -** https://www.youtube.com/watch?v=TOdPmiU-cXA

# Block Cipher Ciphers

*https://www.wolfssl.com/what-is-a-block-cipher/*

“ A **block cipher** is an **encryption method** that applies a deterministic algorithm along with a symmetric key to encrypt a block of text, rather than encrypting one bit at a time as in stream ciphers. For example, a common block cipher, AES, encrypts 128 bit blocks with a key of predetermined length: 128, 192, or 256 bits. Block ciphers are pseudorandom permutation (PRP) families that operate on **the fixed size block of bits**. PRPs are functions that cannot be differentiated from completely random permutations and thus, are considered reliable, until proven unreliable.”



The Two Rules of Block Ciphers

**Rule 1:**

The function must be a ***permutation***, meaning the ciphertext can be translated back to plaintext

$E^{-1}(K, E(K, M)) = M$

**Rule 2:**

The function must be ***efficiently computable***, and the translation process between plaintext and ciphertext and vice versa must not be prohibitively expensive

# AES - Advanced Encryption Standard

AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting the data.
Gets a 128 bits message → 128 cipher text with some Key
Key can be: 128, 192, 256 bits
AES operates on a 4 × 4 column-major order array of bytes, for instance, 16 bytes, b0 , b1 , . . . , b15 are represented as this two-dimensional array:

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

Processed N rounds
In 128 bits key, is completed 10 Runds
In 192 bits key, is completed 12 Runds
In 128 bits key, is completed 14 Runds

## 1. KeyExpansion

Round keys are derived from the cipher key using the AES key schedule. AES requires a separate 128-bit round key block for each round plus one more.

### Round Constants

The round constant Rcon*i* for round *i* of the key expansion is the 32-bit word:

$$rcon_i = [rc_i \quad 00_{16} \quad 00_{16} \quad 00_{16}]$$

**Values of $rc_i$ in hexadecimal**

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A |
|---|---|---|---|---|---|---|---|---|---|---|
| $rc_i$ | 01 | 02 | 04 | 08 | 10 | 20 | 40 | 80 | 1B | 36 |

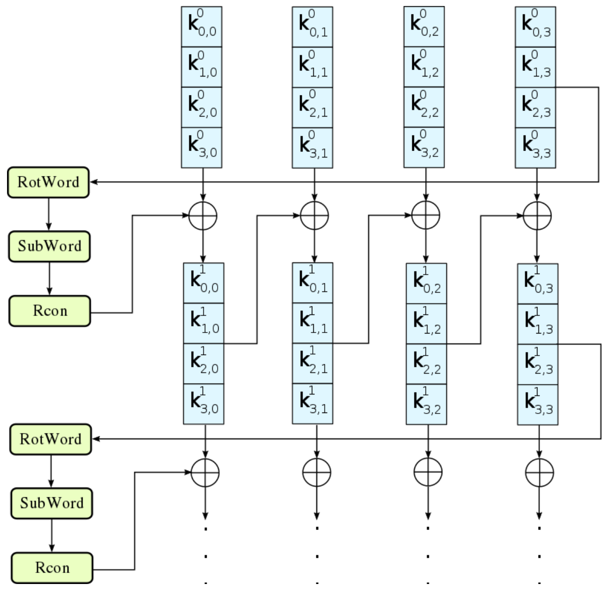
```
static const u32 rcon[] = {

    0x01000000, 0x02000000, 0x04000000, 0x08000000,
    0x10000000, 0x20000000, 0x40000000, 0x80000000,
    0x1B000000, 0x36000000,
};
```

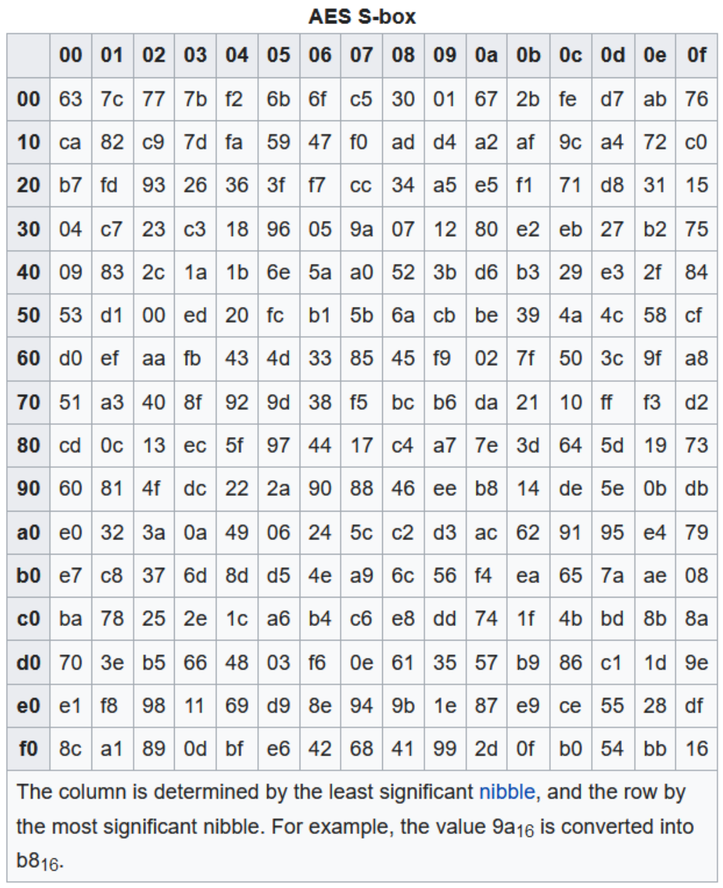
AES uses up to rcon10 for AES-128 (as 11 round keys are needed), up to rcon8 for AES-192, and up to rcon7 for AES-256.

### Key Schedule

1. The 1º round key is equal to the key (16 bytes);
2. For the next round keys:
    a. get the last 4 Bytes of the previous key and execute the Expansion core:
        i. Rotate left 8bits ( *q = ( *q >> 8) | ((*q & 0xff) << 24);
        ii. S-box substitution;
        iii. Rconst XOR;

b. The first 4 bytes of the new key = Expansion Core ^ The first 4 bytes of the previous key (^ = XOR);
c. The second 4 bytes of the key = The first 4 bytes of the new key ^ The second 4 bytes of the previous key;
d. The third 4 bytes of the key = The second 4 bytes of the new key ^ The third 4 bytes of the previous key;
e. The fourth 4 bytes of the key = The third 4 bytes of the new key ^ The fourth 4 bytes of the previous key;
f. With the 16 bytes, it represents the new Round Key;


$k^0_{0,0}$ $k^0_{1,0}$ $k^0_{2,0}$ $k^0_{3,0}$
$k^0_{0,1}$ $k^0_{1,1}$ $k^0_{2,1}$ $k^0_{3,1}$
$k^0_{0,2}$ $k^0_{1,2}$ $k^0_{2,2}$ $k^0_{3,2}$
$k^0_{0,3}$ $k^0_{1,3}$ $k^0_{2,3}$ $k^0_{3,3}$
RotWord
SubWord
Rcon
$k^1_{0,0}$ $k^1_{1,0}$ $k^1_{2,0}$ $k^1_{3,0}$
$k^1_{0,1}$ $k^1_{1,1}$ $k^1_{2,1}$ $k^1_{3,1}$
$k^1_{0,2}$ $k^1_{1,2}$ $k^1_{2,2}$ $k^1_{3,2}$
$k^1_{0,3}$ $k^1_{1,3}$ $k^1_{2,3}$ $k^1_{3,3}$
RotWord
SubWord
Rcon


**AES S-box**

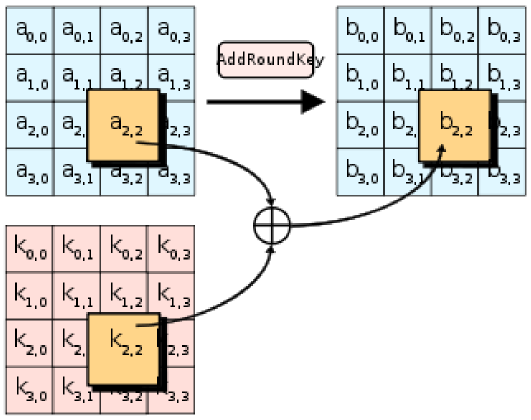
| | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0a | 0b | 0c | 0d | 0e | 0f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **00** | 63 | 7c | 77 | 7b | f2 | 6b | 6f | c5 | 30 | 01 | 67 | 2b | fe | d7 | ab | 76 |
| **10** | ca | 82 | c9 | 7d | fa | 59 | 47 | f0 | ad | 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 |

The column is determined by the least significant nibble, and the row by the most significant nibble. For example, the value $9a_{16}$ is converted into $b8_{16}$.

## 2. Initial round key addition:

**a. AddRoundKey**

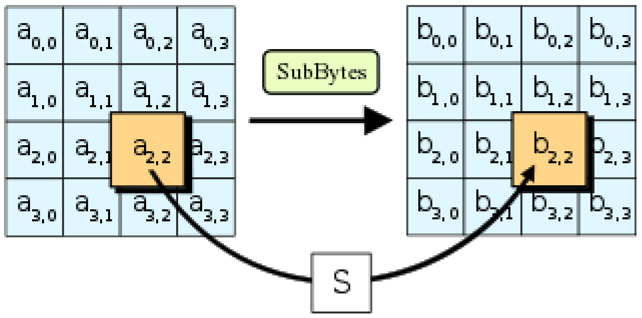
In the AddRoundKey step, the subkey is combined with the state. For each round, a subkey is derived from the main key. The subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR.



## 3. Rounds (until #9, #11 or #13)

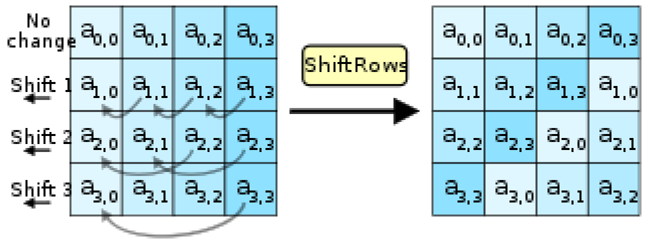
### a. SubBytes

In the SubBytes step, each byte Aij in the state array is replaced with a SubByte S(Aij) using an 8-bit substitution box.



### b. ShiftRows

The ShiftRows step operates on the rows of the state. It cyclically shifts the bytes in each row by a certain offset. For AES, the first row is left unchanged. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted by offsets of two and three respectively.



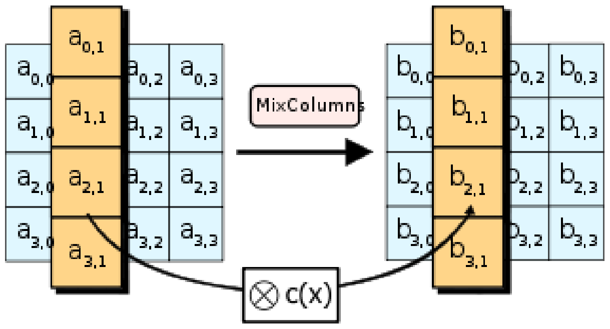
### c. MixColumns

In the MixColumns step, the four bytes of each column of the state are combined using an invertible linear transformation.

During this operation, each column is transformed using a fixed matrix:

$$\begin{bmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{bmatrix} \qquad 0 \le j \le 3$$

**B**0,j = [2 3 1 1]* [**A**0,j **A**1,j **A**2,j **A**3,j]
**B**1,j = [1 2 3 1]* [**A**0,j **A**1,j **A**2,j **A**3,j]
**B**2,j = [1 1 2 3]* [**A**0,j **A**1,j **A**2,j **A**3,j]
**B**3,j = [3 1 1 2]* [**A**0,j **A**1,j **A**2,j **A**3,j]



If value is bigger than 1Byte, necessary to make a polynomial reduction MOD
XOR with: $x^8 * x^4 + x^3 + x^1 + x$^0 or 100011011

d. **AddRoundKey**
(explained before)

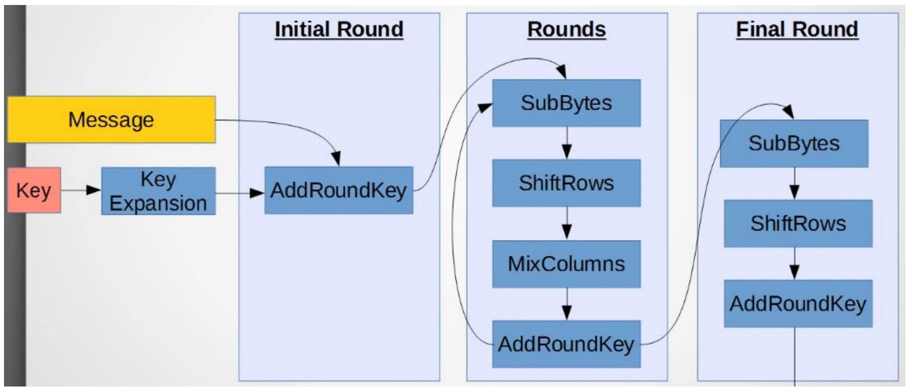
## 4. Final round (#10, or #12 or #14)

a. **SubBytes**
(explained before)
b. **ShiftRows**
(explained before)
c. **AddRoundKey**
(explained before)

## Optimization of the cipher

On systems with 32-bit or larger words, it is possible to speed up execution of this cipher by combining the SubBytes and ShiftRows steps with the MixColumns step by transforming them into a sequence of table lookups.

Overview of the AES process described before:



# RIOT/sys/crypto

RIOT supports the following block ciphers:

- AES-128
- 3DES (deprecated)
- NULL

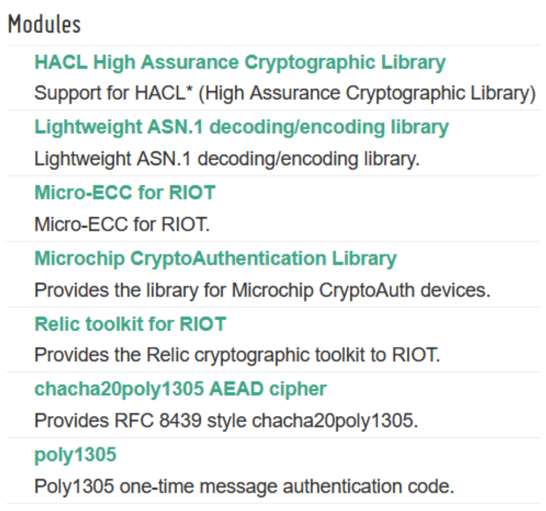
Depending on the selected block ciphers, a sufficient large buffer size of the cipher_context_t is used for en-/de-cryption operations.
Example:

```
#include "crypto/ciphers.h"

cipher_t cipher;
uint8_t key[AES_KEY_SIZE] = {0},
        plain_text[AES_BLOCK_SIZE] = {0},
        cipher_text[AES_BLOCK_SIZE] = {0};

if (cipher_init(&cipher, CIPHER_AES_128, key, AES_KEY_SIZE) < 0)
    printf("Cipher init failed!\n");

if (cipher_encrypt(&cipher, plain_text, cipher_text) < 0)
    printf("Cipher encryption failed!\n");
else
    od_hex_dump(cipher_text, AES_BLOCK_SIZE, 0);
```

**Object dump** - Allows to print out data dumps of memory regions in hexadecimal or/and ASCII representation.
Some aspects of the AES implementation can be fine-tuned by pseudo-modules:

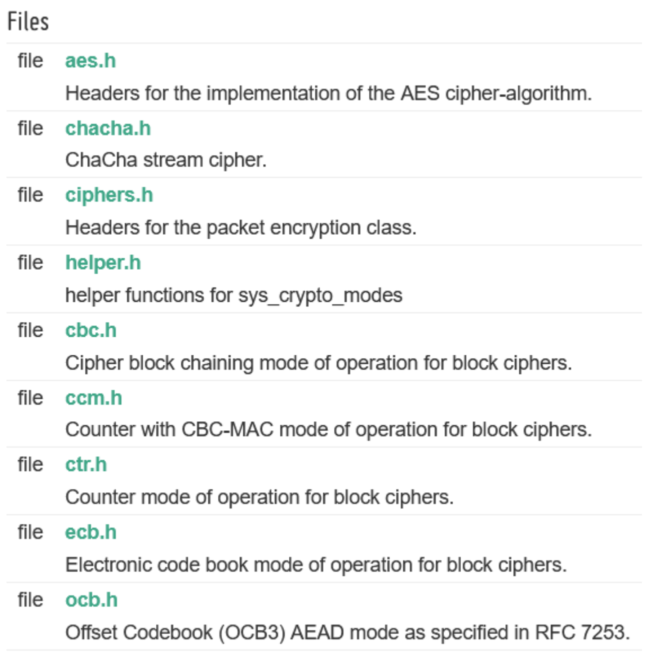
- **crypto_aes_precalculated**: Use pre-calculated T-tables. This improved speed at the expense of increased program size. The default is to calculate most tables on the fly.
- **crypto_aes_unroll**: enable manually-unrolled loops. The default is to not have them unrolled.

To encrypt data of arbitrary size there are different operation modes like: CBC, CTR or CCM.



Modules

| | |
|---|---|
| HACL High Assurance Cryptographic Library | Support for HACL* (High Assurance Cryptographic Library) |
| Lightweight ASN.1 decoding/encoding library | Lightweight ASN.1 decoding/encoding library. |
| Micro-ECC for RIOT | Micro-ECC for RIOT. |
| Microchip CryptoAuthentication Library | Provides the library for Microchip CryptoAuth devices. |
| Relic toolkit for RIOT | Provides the Relic cryptographic toolkit to RIOT. |
| chacha20poly1305 AEAD cipher | Provides RFC 8439 style chacha20poly1305. |
| poly1305 | Poly1305 one-time message authentication code. |



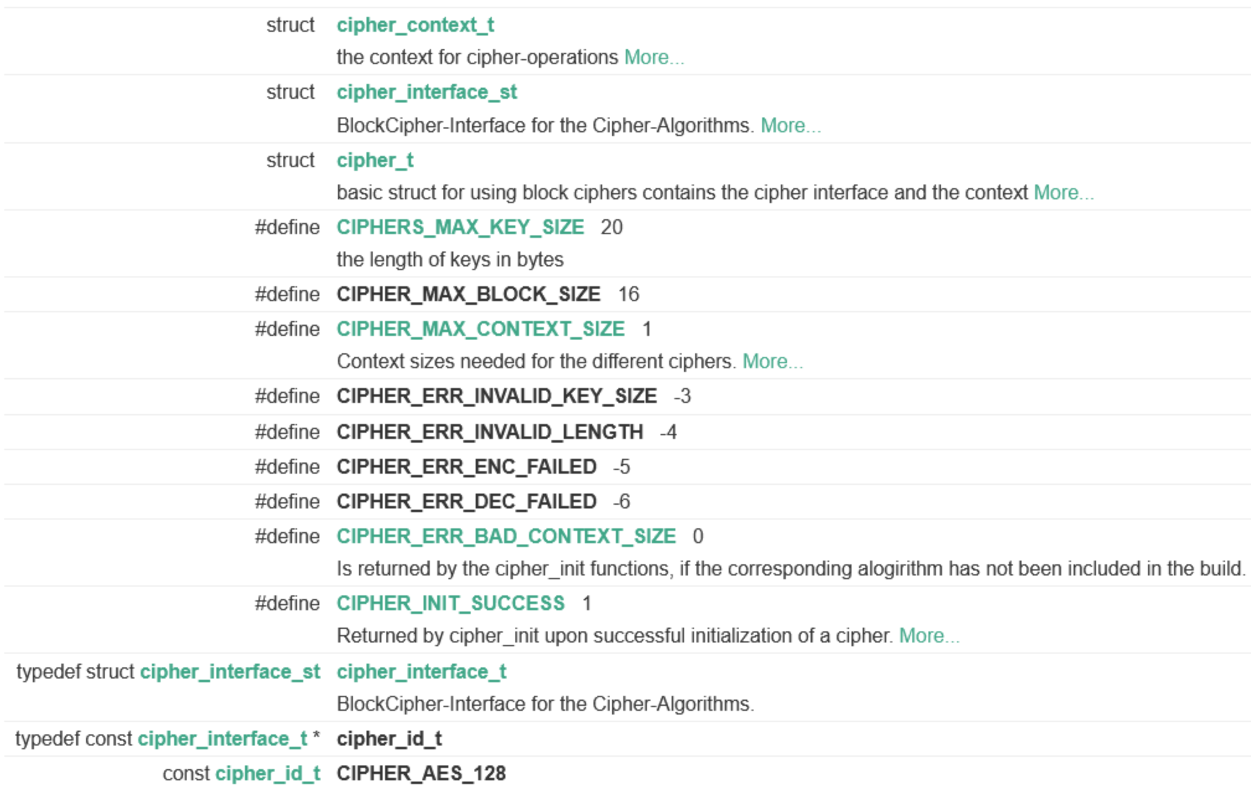
Files

| | | |
|---|---|---|
| file | aes.h | Headers for the implementation of the AES cipher-algorithm. |
| file | chacha.h | ChaCha stream cipher. |
| file | ciphers.h | Headers for the packet encryption class. |
| file | helper.h | helper functions for sys_crypto_modes |
| file | cbc.h | Cipher block chaining mode of operation for block ciphers. |
| file | ccm.h | Counter with CBC-MAC mode of operation for block ciphers. |
| file | ctr.h | Counter mode of operation for block ciphers. |
| file | ecb.h | Electronic code book mode of operation for block ciphers. |
| file | ocb.h | Offset Codebook (OCB3) AEAD mode as specified in RFC 7253. |

## ciphers.h

Headers for the packet encryption class. They are used to encrypt single packets.

Data Structures:



| | |
|---|---|
| struct | **cipher_context_t**<br>the context for cipher-operations More... |
| struct | **cipher_interface_st**<br>BlockCipher-Interface for the Cipher-Algorithms. More... |
| struct | **cipher_t**<br>basic struct for using block ciphers contains the cipher interface and the context More... |
| #define | **CIPHERS_MAX_KEY_SIZE** 20<br>the length of keys in bytes |
| #define | **CIPHER_MAX_BLOCK_SIZE** 16 |
| #define | **CIPHER_MAX_CONTEXT_SIZE** 1<br>Context sizes needed for the different ciphers. More... |
| #define | **CIPHER_ERR_INVALID_KEY_SIZE** -3 |
| #define | **CIPHER_ERR_INVALID_LENGTH** -4 |
| #define | **CIPHER_ERR_ENC_FAILED** -5 |
| #define | **CIPHER_ERR_DEC_FAILED** -6 |
| #define | **CIPHER_ERR_BAD_CONTEXT_SIZE** 0<br>Is returned by the cipher_init functions, if the corresponding alogirithm has not been included in the build. |
| #define | **CIPHER_INIT_SUCCESS** 1<br>Returned by cipher_init upon successful initialization of a cipher. More... |
| typedef struct **cipher_interface_st** | **cipher_interface_t**<br>BlockCipher-Interface for the Cipher-Algorithms. |
| typedef const **cipher_interface_t** * | **cipher_id_t** |
| const **cipher_id_t** | **CIPHER_AES_128** |

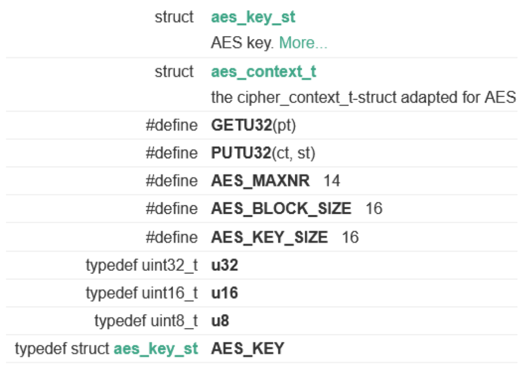
Funtions:

- cipher_decrypt() - Decrypt data of BLOCK_SIZE length.
- cipher_encrypt() - Encrypt data of BLOCK_SIZE length.
- cipher_get_block_size() - Get block size of cipher
- cipher_init() - Initialize new cipher state.

## aes.h

Headers for the implementation of the AES cipher-algorithm
#include "crypto/ciphers.h"

Data Structures:



| | |
|---|---|
| struct | **aes_key_st**<br>AES key. More... |
| struct | **aes_context_t**<br>the cipher_context_t-struct adapted for AES |
| #define | **GETU32**(pt) |
| #define | **PUTU32**(ct, st) |
| #define | **AES_MAXNR** 14 |
| #define | **AES_BLOCK_SIZE** 16 |
| #define | **AES_KEY_SIZE** 16 |
| typedef uint32_t | **u32** |
| typedef uint16_t | **u16** |
| typedef uint8_t | **u8** |
| typedef struct **aes_key_st** | **AES_KEY** |

Funtions:

- **aes_init()** - initializes the AES Cipher-algorithm with the passed parameters
- **aes_encrypt()** - encrypts one plainBlock-block and saves the result in cipherblock. More...
- **aes_decrypt()** - decrypts one cipher-block and saves the plain-block in plainBlock

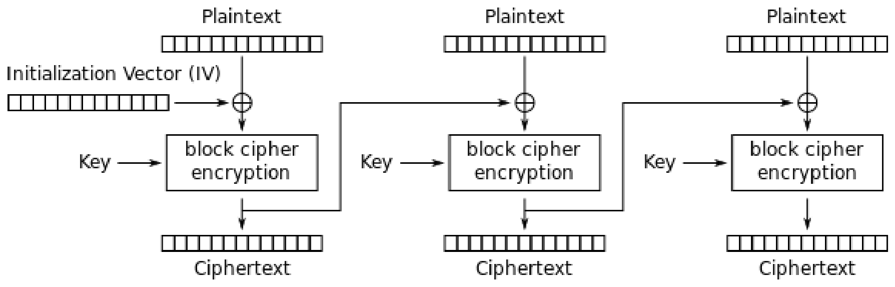
## AES Modes and Testing:

```
   [MODES]      Dworkin, M., "Recommendation for Block Cipher Modes of
                Operation: Methods and Techniques", NIST Special
                Publication 800-38A, December 2001.

NIST has defined five modes of operation for AES and other FIPS-
   approved block ciphers [MODES].  Each of these modes has different
   characteristics.  The five modes are: ECB (Electronic Code Book), CBC
   (Cipher Block Chaining), CFB (Cipher FeedBack), OFB (Output
   FeedBack), and CTR (Counter).
```

**CIPHER BLOCK CHAINING (CBC)**

“Block cipher modes of operation have been developed **to eliminate the chance of encrypting identical blocks of text the same way**, the ciphertext formed from the previous encrypted block is applied to the next block. A block of bits called an **initialization vector** (IV) is also used by modes of operation to ensure ciphertexts remain **distinct even when the same plaintext message is encrypted** a number of times. (…)

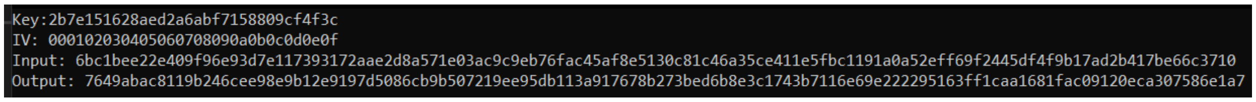
Where an IV is crossed with the initial plaintext block and the encryption algorithm is completed with a given key and the **ciphertext is then outputted**. This **resultant cipher text is then used in place of the IV** in subsequent plaintext blocks.”



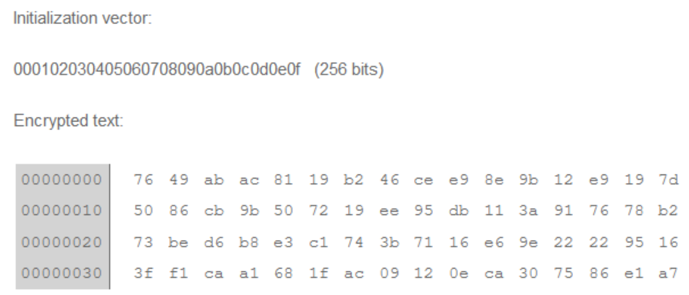
Cipher Block Chaining (CBC) mode encryption

CBC allows for plain text with dinamic size. Must be multiple of 16. Receives has parameters the IV, the key, the input (text) and input length. Returns the encrypted result (output)

```
cipher_init(&cipher, CIPHER_AES_128, key, key_len);
cipher_encrypt_cbc(&cipher, iv, input, input_len, data);
```



```
Key:2b7e151628aed2a6abf7158809cf4f3c
IV: 000102030405060708090a0b0c0d0e0f
Input: 6bc1bee22e409f96e93d7e117393172aae2d8a571e03ac9c9eb76fac45af8e5130c81c46a35ce411e5fbc1191a0a52eff69f2445df4f9b17ad2b417be66c3710
Output: 7649abac8119b246cee98e9b12e9197d5086cb9b507219ee95db113a917678b273bed6b8e3c1743b7116e69e222295163ff1caa1681fac09120eca307586e1a7
```



Initialization vector:

000102030405060708090a0b0c0d0e0f (256 bits)

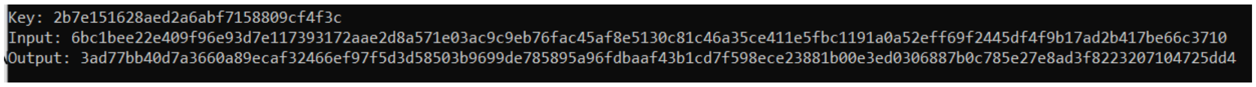
Encrypted text:

```
00000000   76 49 ab ac 81 19 b2 46 ce e9 8e 9b 12 e9 19 7d
00000010   50 86 cb 9b 50 72 19 ee 95 db 11 3a 91 76 78 b2
00000020   73 be d6 b8 e3 c1 74 3b 71 16 e6 9e 22 22 95 16
00000030   3f f1 ca a1 68 1f ac 09 12 0e ca 30 75 86 e1 a7
```

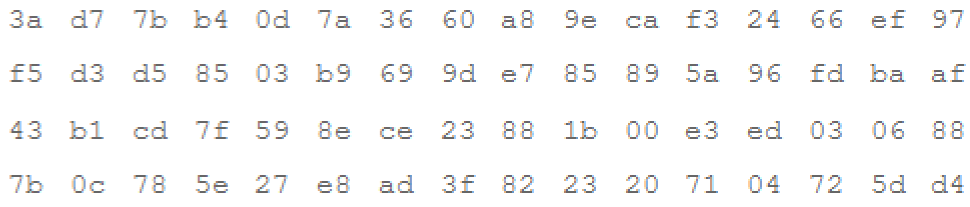
**ELETRONIC CODE BOOK (ECB)**

Same as CBC, ECB allows for plain text with dinamic size (must be multiple of 16). Receives has parameters teh key, the input (text) and input length. Returns the encrypted result (output).

```
cipher_init(&cipher, CIPHER_AES_128, key, key_len);
cipher_encrypt_ecb(&cipher, input, input_len, data);
```



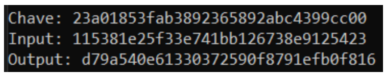
```
Key: 2b7e151628aed2a6abf7158809cf4f3c
Input: 6bc1bee22e409f96e93d7e117393172aae2d8a571e03ac9c9eb76fac45af8e5130c81c46a35ce411e5fbc1191a0a52eff69f2445df4f9b17ad2b417be66c3710
Output: 3ad77bb40d7a3660a89ecaf32466ef97f5d3d58503b9699de785895a96fdbaaf43b1cd7f598ece23881b00e3ed0306887b0c785e27e8ad3f8223207104725dd4
```



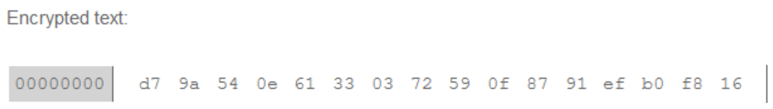
```
3a d7 7b b4 0d 7a 36 60 a8 9e ca f3 24 66 ef 97
f5 d3 d5 85 03 b9 69 9d e7 85 89 5a 96 fd ba af
43 b1 cd 7f 59 8e ce 23 88 1b 00 e3 ed 03 06 88
7b 0c 78 5e 27 e8 ad 3f 82 23 20 71 04 72 5d d4
```

The standard AES is ECB, but for a plaintext of 16 bytes

```
aes_init(&ctx, TEST_1_KEY, sizeof(TEST_1_KEY));
aes_encrypt(&ctx, TEST_1_INP, data);
```



```
Chave: 23a01853fab3892365892abc4399cc00
Input: 115381e25f33e741bb126738e9125423
Output: d79a540e61330372590f8791efb0f816
```



Encrypted text:

```
00000000   d7 9a 54 0e 61 33 03 72 59 0f 87 91 ef b0 f8 16
```

**COUNTER (CTR)**

https://tools.ietf.org/html/rfc3686

```
        ES-CTR requires the encryptor to generate a unique per-packet value,
and communicate this value to the decryptor. This calls this per-packet value
an initialization vector (IV).  The same IV and key combination MUST NOT be
```

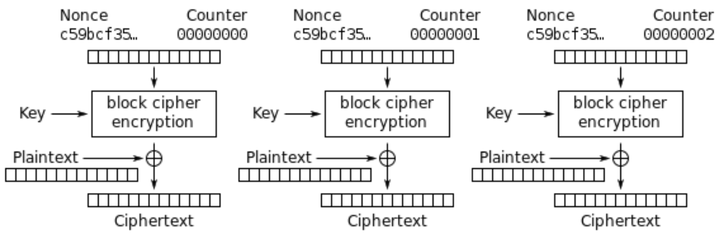
used more than once. The encryptor can generate the IV in any manner that ensures uniqueness. Common approaches to IV generation include incrementing a counter for each packet and linear feedback shift registers (LFSRs).
Nonce applies additional protection against precomputation attacks. The nonce value need not be secret. However, the nonce MUST be unpredictable prior to the establishment of the IPsec security association that is making use of AES-CTR.

AES-CTR has many properties that make it an attractive encryption algorithm for in high-speed networking. AES-CTR uses the AES block cipher to create a stream cipher. Data is encrypted and decrypted by XORing with the key stream produced by AES encrypting sequential counter block values. AES-CTR is easy to implement, and AES-CTR can be pipelined and parallelized. AES-CTR also supports key stream precomputation.

A hardware implementation (and some software implementations) can create a pipeline by unwinding the loop implied by this round structure. For example, after a 16-octet block has been input, one round later another 16-octet block can be input, and so on. In AES-CTR, these inputs are the sequential counter block values used to generate the key stream.

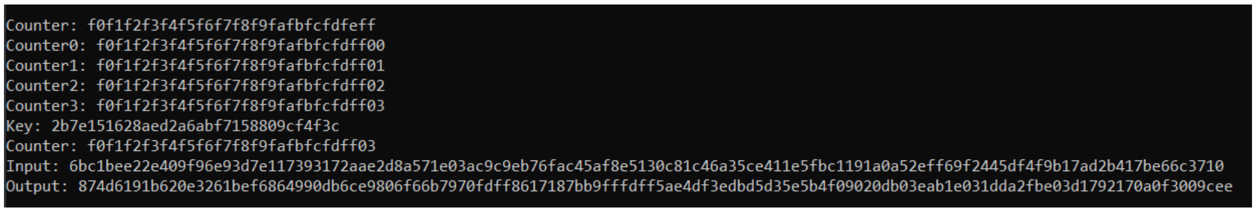
The sender can precompute the key stream. Since the key stream does not depend on any data in the packet, the key stream can be precomputed once the nonce and IV are assigned. This precomputation can reduce packet latency. The receiver cannot perform similar precomputation because the IV will not be known before the packet arrives.
AES-CTR uses the only AES encrypt operation (for both encryption and decryption), making AES-CTR implementations smaller than implementations of many other AES modes.

When used correctly, AES-CTR provides a high level of confidentiality. Unfortunately, AES-CTR is easy to use incorrectly. For this reason, it is inappropriate to use this mode of operation with static keys.

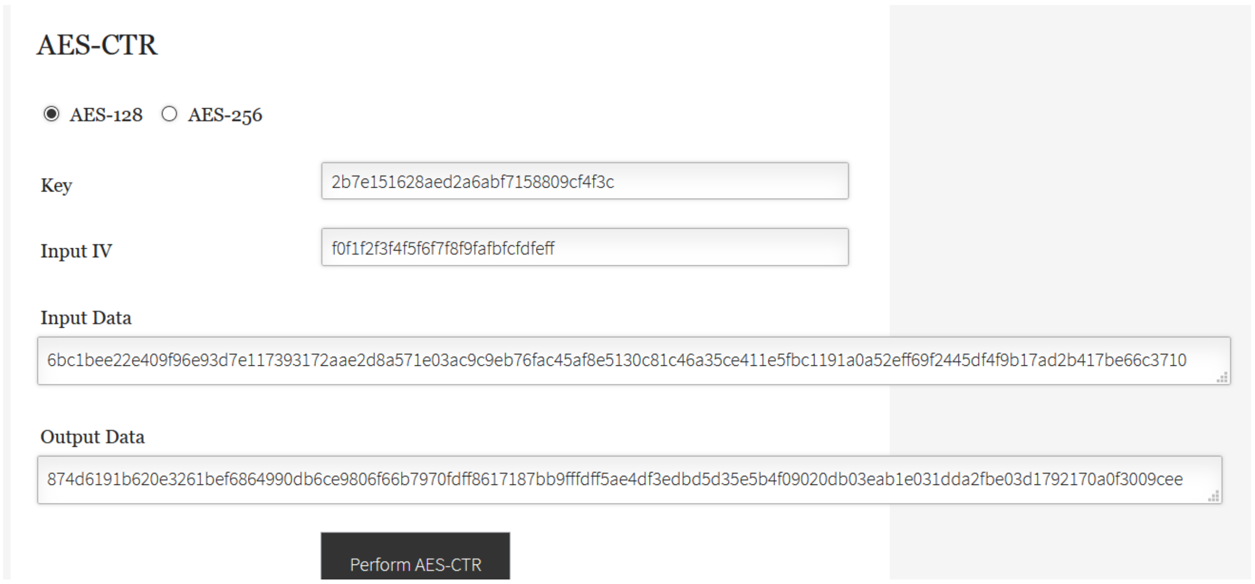


Counter (CTR) mode encryption

```
CTRBLK := NONCE || IV || ONE
FOR i := 1 to n-1 DO
  CipherText[i] := PlainText[i] XOR AES(CTRBLK)
  CTRBLK := CTRBLK + 1
END
CT[n] := PT[n] XOR TRUNC(AES(CTRBLK))
```



```
Counter: f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff
Counter0: f0f1f2f3f4f5f6f7f8f9fafbfcfdff00
Counter1: f0f1f2f3f4f5f6f7f8f9fafbfcfdff01
Counter2: f0f1f2f3f4f5f6f7f8f9fafbfcfdff02
Counter3: f0f1f2f3f4f5f6f7f8f9fafbfcfdff03
Key: 2b7e151628aed2a6abf7158809cf4f3c
Counter: f0f1f2f3f4f5f6f7f8f9fafbfcfdff03
Input: 6bc1bee22e409f96e93d7e117393172aae2d8a571e03ac9c9eb76fac45af8e5130c81c46a35ce411e5fbc1191a0a52eff69f2445df4f9b17ad2b417be66c3710
Output: 874d6191b620e3261bef6864990db6ce9806f66b7970fdff8617187bb9fffdff5ae4df3edbd5d35e5b4f09020db03eab1e031dda2fbe03d1792170a0f3009cee
```



[● AES-128] ○ AES-256

| | |
|---|---|
| Key | 2b7e151628aed2a6abf7158809cf4f3c |
| Input IV | f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff |

Input Data

```
6bc1bee22e409f96e93d7e117393172aae2d8a571e03ac9c9eb76fac45af8e5130c81c46a35ce411e5fbc1191a0a52eff69f2445df4f9b17ad2b417be66c3710
```

Output Data

```
874d6191b620e3261bef6864990db6ce9806f66b7970fdff8617187bb9fffdff5ae4df3edbd5d35e5b4f09020db03eab1e031dda2fbe03d1792170a0f3009cee
```

## COUNTER WITH CBC-MAC (CCM)

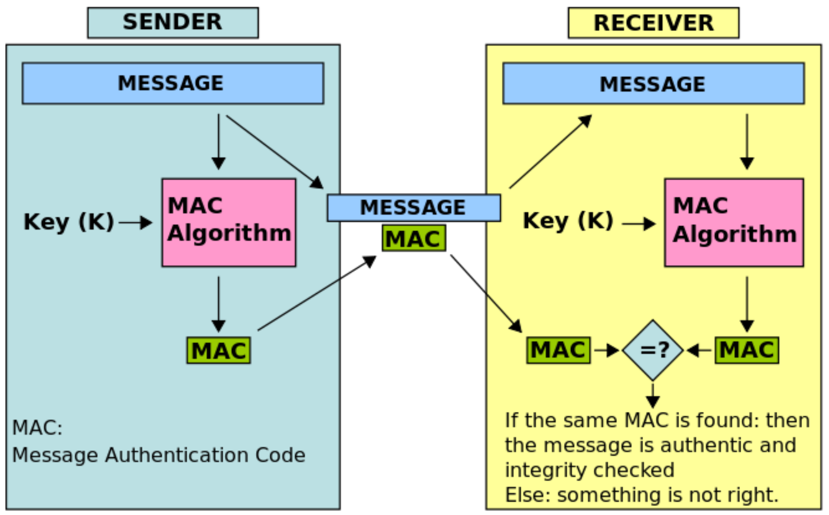
Counter with CBC-MAC (CCM) is a generic authenticated encryption block cipher mode. CCM is defined for use with 128-bit block ciphers, such as the Advanced Encryption Standard (AES).

https://tools.ietf.org/html/rfc3610#ref-STDWORDS

CCM mode (counter with cipher block chaining message authentication code; counter with CBC-MAC) is a mode of operation for cryptographic block ciphers. It is an authenticated encryption algorithm designed to provide both authentication and confidentiality. CCM mode is only defined for block ciphers with a block length of 128 bits.

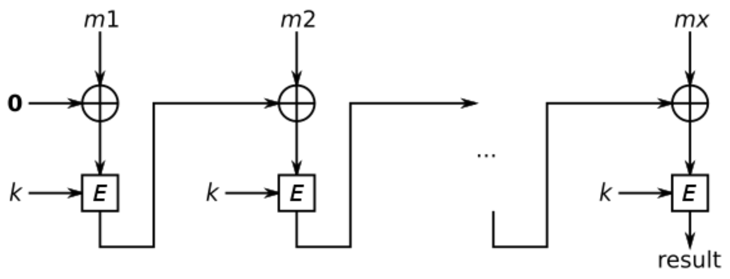
The nonce of CCM must be carefully chosen to never be used more than once for a given key. This is because CCM is a derivation of CTR mode and the latter is effectively a stream cipher.

**Message Authentication Code (MAC)** - known as a tag, is a short piece of information used to authenticate a message—in other words, to confirm that the message came from the stated sender (its authenticity) and has not been changed. The MAC value protects a message's data integrity, as well as its authenticity, by allowing verifiers (who also possess the secret key) to detect any changes to the message content.


SENDER
MESSAGE
Key (K)
MAC Algorithm
MAC
MAC:
Message Authentication Code
MESSAGE
MAC
RECEIVER
MESSAGE
Key (K)
MAC Algorithm
MAC
=?
MAC
If the same MAC is found: then the message is authentic and integrity checked
Else: something is not right.

The sender of a message runs it through a MAC algorithm to produce a MAC data tag. The message and the MAC tag are then sent to the receiver. The receiver in turn runs the message portion of the transmission through the same MAC algorithm using the same key, producing a second MAC data tag. The receiver then compares the first MAC tag received in the transmission to the second generated MAC tag. If they are identical, the receiver can safely assume that the message was not altered or tampered with during transmission (data integrity).

However, to allow the receiver to be able to detect replay attacks, the message itself must contain data that assures that this same message can only be sent once (e.g. time stamp, sequence number counter , etc). Otherwise an attacker could record this message and play it back at a later time, producing the same result as the original sender.



**CBC-MAC Vs CBC**

- CBC-MAC is deterministic (no IV, IV = 0);
- In CBC-MAC only the final value is output (to verify the tag recompute the result)

For CBC-MAC, the sender and receiver must agree on the length parameter L in Advance for the tag to be secure.

To produce a secure tag from an arbitrary length message, encode L (number of message blocks) in the first single Block and use the result in the following cipher block chain (padding L with zeros if necessary).

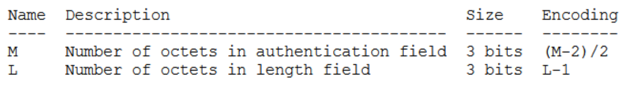
**Encryption and authentication -** CCM mode combines the well-known CBC-MAC with the well-known counter mode of encryption. These two primitives are applied in an "authenticate-then-encrypt" manner, that is, CBC-MAC is first computed on the message to obtain a tag t; the message and the tag are then encrypted using counter mode. The same encryption key can be used for both, provided that the counter values used in the encryption do not collide with the (pre-)initialization vector used in the authentication.

**RIOT** `cipher_encrypt_ccm()`

```
Encrypt and authenticate data of arbitrary length in ccm mode.

cipher           Already initialized cipher struct
auth_data        Additional data to authenticate in MAC
auth_data_len    Length of additional data, max (2^16 - 2^8)
mac_length       length of the appended MAC (between 4 and 16 - only
                 even values)
length_encoding  maximal supported length of plaintext
                 (2^(8*length_enc)).
nonce            Nounce for ctr mode encryption
nonce_len        Length of the nonce in octets
                 (maximum: 15-length_encoding)
input            pointer to input data to encrypt
input_len        length of the input data, [0, 2^32]
output           pointer to allocated memory for encrypted data. It
                 has to be of size data_len + mac_length.

return                 Length of encrypted data on a successful encryption,
                       can be 0 if input_len=0 (no plaintext)
```

M, the size of the authentication field, valid values are 4, 6, 8, 10, 12, 14, and 16 octets.

L, the size of the length field. This value requires a trade-off between the maximum message size and the size of the Nonce. Valid values of L range between 2 octets and 8 octets.



| Name | Description | Size | Encoding |
|---|---|---|---|
| M | Number of octets in authentication field | 3 bits | (M-2)/2 |
| L | Number of octets in length field | 3 bits | L-1 |

To authenticate and encrypt a message the following information is required:

1. An encryption key K suitable for the block cipher.

2. A nonce N of 15-L octets. Within the scope of any encryption key K, the nonce value MUST be unique. That is, the set of nonce values used with any given key MUST NOT contain any duplicate values. Using the same nonce for two different messages encrypted with the same key destroys the security properties of this mode.

3. The message m, consisting of a string of l(m) octets where 0 <= l(m) < 2^(8L). The length restriction ensures that l(m) can be encoded in a field of L octets.

4. Additional authenticated data a, consisting of a string of l(a) octets where 0 <= l(a) < 2^64. This additional data is authenticated but not encrypted and is not included in the output of this mode. It can be used to authenticate plaintext packet headers, or contextual information that affects the interpretation of the message. Users who do not wish to authenticate additional data can provide a string of length zero.

| Name | Description | Size |
|---|---|---|
| K | Block cipher key | Depends on block cipher |
| N | Nonce | 15-L octets |
| m | Message to authenticate and encrypt | l(m) octets |
| a | Additional authenticated data | l(a) octets |

## Authentication

We first define a sequence of blocks B_0, B_1, ..., B_n and then apply CBC-MAC to these blocks.

The first block B_0 is formatted as follows, where l(m) is encoded in most-significant-byte first order:

| Octet Number | Contents |
|---|---|
| 0 | Flags |
| 1 ... 15-L | Nonce N |
| 16-L ... 15 | l(m) |

Within the first block B_0, the Flags field is formatted as follows:

Flags = 64*Adata + 8*M' + L'.

| Bit Number | Contents |
|---|---|
| 7 | Reserved (always zero) |
| 6 | Adata |
| 5 ... 3 | M' |
| 2 ... 0 | L' |

If l(A)>0 (as indicated by the Adata field), then one or more blocks of authentication data are added. These blocks contain l(A) and A encoded in a reversible manner. We first construct a string that encodes l(A). The length encoding conventions are summarized in the following table.

| First two octets | Followed by | Comment |
|---|---|---|
| 0x0000 | Nothing | Reserved |
| 0x0001 ... 0xFEFF | Nothing | For 0 < l(a) < (2^16 - 2^8) |
| 0xFF00 ... 0xFFFD | Nothing | Reserved |
| 0xFFFE | 4 octets of l(a) | For (2^16 - 2^8) <= l(a) < 2^32 |
| 0xFFFF | 8 octets of l(a) | For 2^32 <= l(a) < 2^64 |

The blocks encoding A are formed by concatenating this string that encodes l(A) with a itself, and splitting the result into 16-octet blocks, and then padding the last block with zeroes if necessary. These blocks are appended to the first block B0. After the (optional) additional authentication blocks have been added, we add the message blocks. The message blocks are formed by splitting the message m into 16-octet blocks, and then padding the last block with zeroes if necessary. If the message m consists of the empty string, then no blocks are added in this step.

The result is a sequence of blocks B0, B1, ..., Bn. The CBC-MAC is computed by:

```
X_1 := E( K, B_0 )
X_i+1 := E( K, X_i XOR B_i )  for i=1, ..., n
T := first-M-bytes( X_n+1 )
```

Where E() is the block cipher encryption function, and T is the MAC value. CCM was designed with AES in mind for the E() function, but any 128-bit block cipher can be used. Note that the last block B_n is XORed with X_n, and the result is encrypted with the block cipher. If needed, the ciphertext is truncated to give T.

## Encryption

To encrypt the message data we use Counter (CTR) mode. We first define the key stream blocks by:

```
S_i := E( K, A_i )   for i=0, 1, 2, ...
```

The values A_i are formatted as follows, where the Counter field i is encoded in most-significant-byte first order. Flags = L'.

```
Octet Number   Contents
------------   ---------
0              Flags
1 ... 15-L     Nonce N
16-L ... 15    Counter i
```

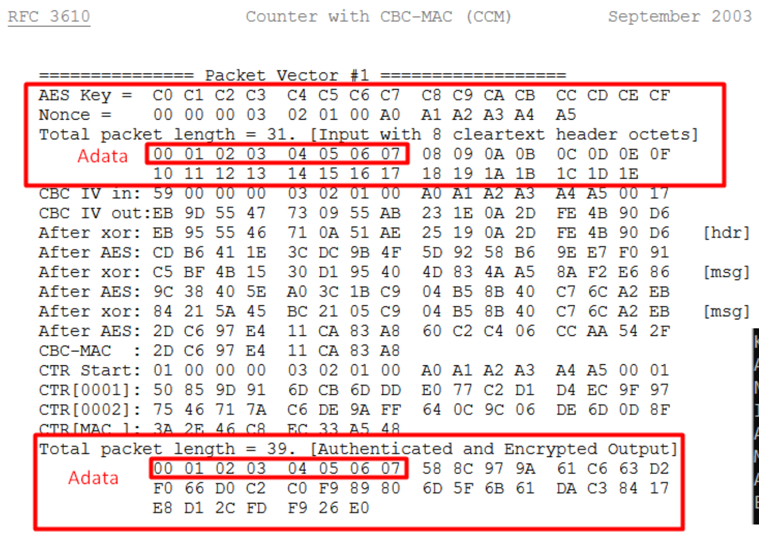
```
The Flags field is formatted as follows:

   Bit Number   Contents
   ----------   ----------------------
   7            Reserved (always zero)
   6            Reserved (always zero)
   5 ... 3      Zero
   2 ... 0      L'
```

The message is encrypted by XORing the octets of message m with the first l(m) octets of the concatenation of S_1, S_2, S_3, ... . Note that S_0 is not used to encrypt the message. The authentication value U is computed by encrypting T with the key stream block S_0 and truncating it to the desired length.

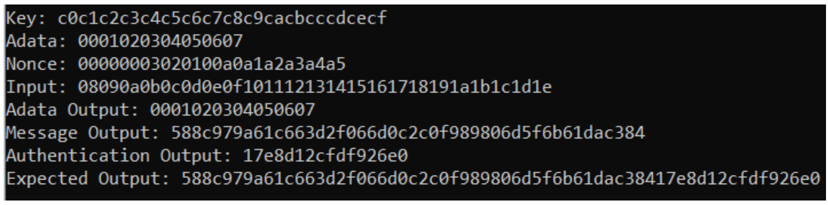
```
U := T XOR first-M-bytes( S_0 )
```

## Output

The result C consists of the encrypted message followed by the encrypted authentication value U.



```
RFC 3610              Counter with CBC-MAC (CCM)          September 2003


   =============== Packet Vector #1 ==================
   AES Key =  C0 C1 C2 C3  C4 C5 C6 C7  C8 C9 CA CB  CC CD CE CF
   Nonce =    00 00 00 03  02 01 00 A0  A1 A2 A3 A4  A5
   Total packet length = 31. [Input with 8 cleartext header octets]
              00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
              10 11 12 13  14 15 16 17  18 19 1A 1B  1C 1D 1E
   CBC IV in: 59 00 00 00  03 02 01 00  A0 A1 A2 A3  A4 A5 00 17
   CBC IV out:EB 9D 55 47  73 09 55 AB  23 1E 0A 2D  FE 4B 90 D6
   After xor: EB 95 55 46  71 0A 51 AE  25 19 0A 2D  FE 4B 90 D6   [hdr]
   After AES: CD B6 41 1E  3C DC 9B 4F  5D 92 58 B6  9E E7 F0 91
   After xor: C5 BF 4B 15  30 D1 95 40  4D 83 4A A5  8A F2 E6 86   [msg]
   After AES: 9C 38 40 5E  A0 3C 1B C9  04 B5 8B 40  C7 6C A2 EB
   After xor: 84 21 5A 45  BC 21 05 C9  04 B5 8B 40  C7 6C A2 EB   [msg]
   After AES: 2D C6 97 E4  11 CA 83 A8  60 C2 C4 06  CC AA 54 2F
   CBC-MAC  : 2D C6 97 E4  11 CA 83 A8
   CTR Start: 01 00 00 00  03 02 01 00  A0 A1 A2 A3  A4 A5 00 01
   CTR[0001]: 50 85 9D 91  6D CB 6D DD  E0 77 C2 D1  D4 EC 9F 97
   CTR[0002]: 75 46 71 7A  C6 DE 9A FF  64 0C 9C 06  DE 6D 0D 8F
   CTR[MAC ]: 3A 2E 46 C8  EC 33 A5 48
   Total packet length = 39. [Authenticated and Encrypted Output]
              00 01 02 03  04 05 06 07  58 8C 97 9A  61 C6 63 D2
              F0 66 D0 C2  C0 F9 89 80  6D 5F 6B 61  DA C3 84 17
              E8 D1 2C FD  F9 26 E0
```

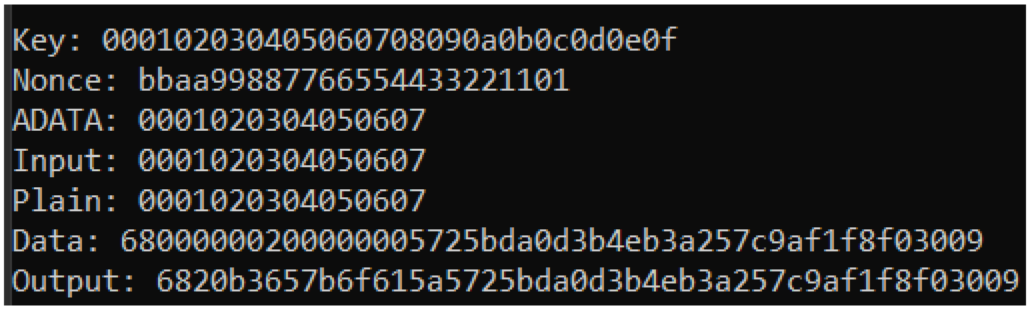


```
Key: c0c1c2c3c4c5c6c7c8c9cacbcccdcecf
Adata: 0001020304050607
Nonce: 00000003020100a0a1a2a3a4a5
Input: 08090a0b0c0d0e0f101112131415161718191a1b1c1d1e
Adata Output: 0001020304050607
Message Output: 588c979a61c663d2f066d0c2c0f989806d5f6b61dac384
Authentication Output: 17e8d12cfdf926e0
Expected Output: 588c979a61c663d2f066d0c2c0f989806d5f6b61dac38417e8d12cfdf926e0
```

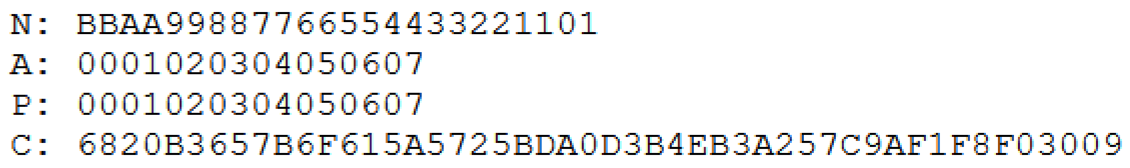
**OFFSET CODE BOOK (OCB)**

https://tools.ietf.org/html/rfc7253

OCB - a shared-key blockcipher-based encryption scheme that provides confidentiality and authenticity for plaintexts and authenticity for associated data.



```
Key: 000102030405060708090a0b0c0d0e0f
Nonce: bbaa99887766554433221101
ADATA: 0001020304050607
Input: 0001020304050607
Plain: 0001020304050607
Data: 680000000200000005725bda0d3b4eb3a257c9af1f8f03009
Output: 6820b3657b6f615a5725bda0d3b4eb3a257c9af1f8f03009
```

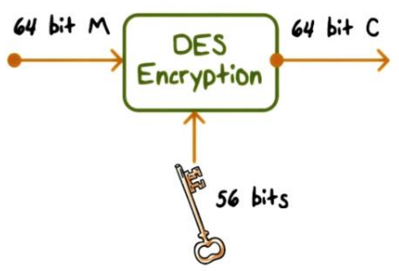


```
N: BBAA99887766554433221101
A: 0001020304050607
P: 0001020304050607
C: 6820B3657B6F615A5725BDA0D3B4EB3A257C9AF1F8F03009
```

## #DES - Data Encryption Standard

Symmetric – one/same key

Encryption on blocks of 64 bit and key of 56 bit



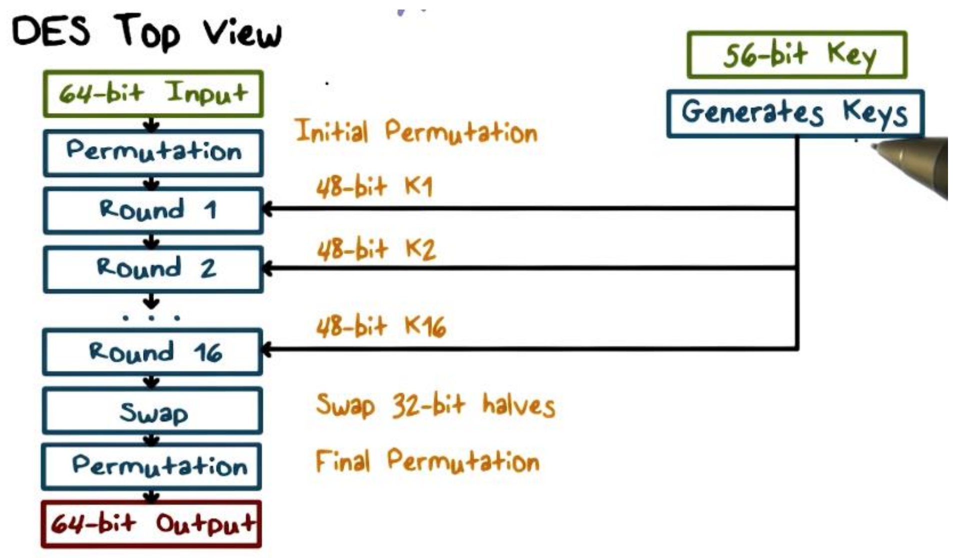
# represents number of rounds of encryption

**EEE2:** Encrypt with Key → change Key and Encrypt → Encrypt with the first key

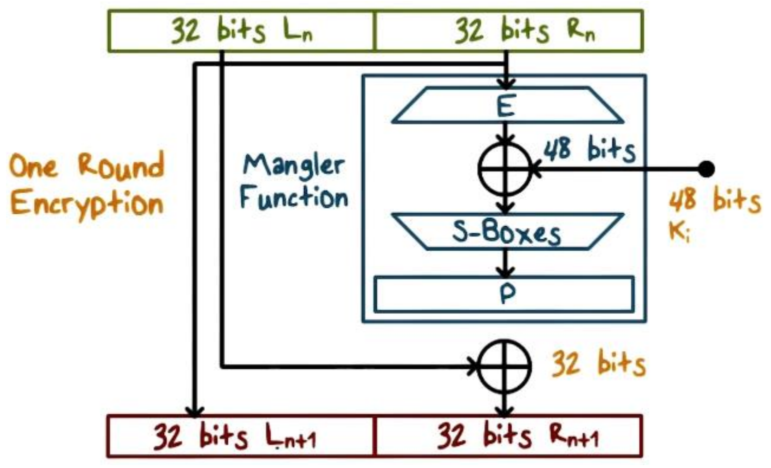
**EDE3:** Encrypt with Key → change Key and Decrypt → change Key and Encrypt

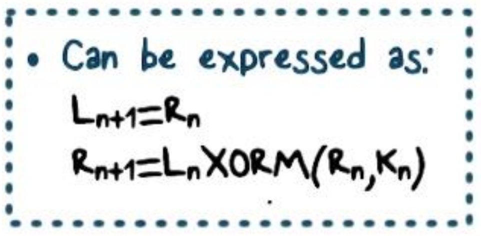
**EDE3:** Encrypt with Key → change Key and Encrypt → change Key and Encrypt

Key is 64 bit. However for each byte there is a parity bit, making it **8 bit parity + 56 key**



Bits Permutation




• Can be expressed as:

$L_{n+1}=R_n$

$R_{n+1}=L_n \text{XOR} M(R_n,K_n)$