```
#include <iostream>
using namespace std;
#define Nb 4

#if defined(AES256) && (AES256 == 1)
   #define Nk 8
   #define Nr 14
#elif defined(AES192) && (AES192 == 1)
   #define Nk 6
   #define Nr 12
#else
   #define Nk 4
 #define Nr 10
#endif
#ifndef MULTIPLY_AS_A_FUNCTION
  #define MULTIPLY_AS_A_FUNCTION 0
#endif

typedef uint8_t state_t[4][4];

static const uint8_t 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 };

#if (defined(CBC) && CBC == 1) || (defined(ECB) && ECB == 1)
static const uint8_t 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 };
#endif

static const uint8_t Rcon[11] = {
  0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36 };
```

```
static uint8_t getSBoxValue(uint8_t num)
{
  return sbox[num];
}

#define getSBoxValue(num) (sbox[(num)])

static void KeyExpansion(uint8_t* RoundKey, const uint8_t* Key)
{
  unsigned i, j, k;
  uint8_t tempa[4];
  for (i = 0; i < Nk; ++i)
  {
    RoundKey[(i * 4) + 0] = Key[(i * 4) + 0];
    RoundKey[(i * 4) + 1] = Key[(i * 4) + 1];
    RoundKey[(i * 4) + 2] = Key[(i * 4) + 2];
    RoundKey[(i * 4) + 3] = Key[(i * 4) + 3];
  }


  for (i = Nk; i < Nb * (Nr + 1); ++i)
  {
    {
      k = (i - 1) * 4;
      tempa[0]=RoundKey[k + 0];
      tempa[1]=RoundKey[k + 1];
      tempa[2]=RoundKey[k + 2];
      tempa[3]=RoundKey[k + 3];

    }
    if (i % Nk == 0)
    {

      {
        const uint8_t u8tmp = tempa[0];
        tempa[0] = tempa[1];
        tempa[1] = tempa[2];
        tempa[2] = tempa[3];
        tempa[3] = u8tmp;
      }

      // Function Subword()
      {
        tempa[0] = getSBoxValue(tempa[0]);
        tempa[1] = getSBoxValue(tempa[1]);
        tempa[2] = getSBoxValue(tempa[2]);
        tempa[3] = getSBoxValue(tempa[3]);
      }

      tempa[0] = tempa[0] ^ Rcon[i/Nk];
    }
#if defined(AES256) && (AES256 == 1)
    if (i % Nk == 4)
    {
      // Function Subword()
      {
        tempa[0] = getSBoxValue(tempa[0]);
        tempa[1] = getSBoxValue(tempa[1]);
        tempa[2] = getSBoxValue(tempa[2]);
        tempa[3] = getSBoxValue(tempa[3]);
      }
    }
#endif
    j = i * 4; k=(i - Nk) * 4;
    RoundKey[j + 0] = RoundKey[k + 0] ^ tempa[0];
    RoundKey[j + 1] = RoundKey[k + 1] ^ tempa[1];
    RoundKey[j + 2] = RoundKey[k + 2] ^ tempa[2];
    RoundKey[j + 3] = RoundKey[k + 3] ^ tempa[3];
  }
}

void AES_init_ctx(struct AES_ctx* ctx, const uint8_t* key)
{
  KeyExpansion(ctx->RoundKey, key);
}
#if (defined(CBC) && (CBC == 1)) || (defined(CTR) && (CTR == 1))
void AES_init_ctx_iv(struct AES_ctx* ctx, const uint8_t* key, const uint8_t* iv)
{
  KeyExpansion(ctx->RoundKey, key);
  memcpy (ctx->Iv, iv, AES_BLOCKLEN);
}
void AES_ctx_set_iv(struct AES_ctx* ctx, const uint8_t* iv)
{
  memcpy (ctx->Iv, iv, AES_BLOCKLEN);
}
#endif



static void AddRoundKey(uint8_t round, state_t* state, const uint8_t* RoundKey)
{
  uint8_t i,j;
  for (i = 0; i < 4; ++i)
  {
    for (j = 0; j < 4; ++j)
    {
      (*state)[i][j] ^= RoundKey[(round * Nb * 4) + (i * Nb) + j];
    }
  }
}

static void SubBytes(state_t* state)
{
  uint8_t i, j;
  for (i = 0; i < 4; ++i)
  {
    for (j = 0; j < 4; ++j)
    {
      (*state)[j][i] = getSBoxValue((*state)[j][i]);
    }
  }
}

static void ShiftRows(state_t* state)
{
  uint8_t temp;

  // Rotate first row 1 columns to left
  temp           = (*state)[0][1];
  (*state)[0][1] = (*state)[1][1];
  (*state)[1][1] = (*state)[2][1];
  (*state)[2][1] = (*state)[3][1];
  (*state)[3][1] = temp;
  // Rotate second row 2 columns to left
  temp           = (*state)[0][2];
  (*state)[0][2] = (*state)[2][2];
  (*state)[2][2] = temp;

  temp           = (*state)[1][2];
  (*state)[1][2] = (*state)[3][2];
  (*state)[3][2] = temp;

  // Rotate third row 3 columns to left
  temp           = (*state)[0][3];
  (*state)[0][3] = (*state)[3][3];
  (*state)[3][3] = (*state)[2][3];
  (*state)[2][3] = (*state)[1][3];
  (*state)[1][3] = temp;
}

static uint8_t xtime(uint8_t x)
{
  return ((x<<1) ^ (((x>>7) & 1) * 0x1b));
}


static void MixColumns(state_t* state)
{
  uint8_t i;
  uint8_t Tmp, Tm, t;
  for (i = 0; i < 4; ++i)
  {
    t   = (*state)[i][0];
    Tmp = (*state)[i][0] ^ (*state)[i][1] ^ (*state)[i][2] ^ (*state)[i][3] ;
    Tm  = (*state)[i][0] ^ (*state)[i][1] ; Tm = xtime(Tm);  (*state)[i][0] ^= Tm ^ Tmp ;
    Tm  = (*state)[i][1] ^ (*state)[i][2] ; Tm = xtime(Tm);  (*state)[i][1] ^= Tm ^ Tmp ;
    Tm  = (*state)[i][2] ^ (*state)[i][3] ; Tm = xtime(Tm);  (*state)[i][2] ^= Tm ^ Tmp ;
    Tm  = (*state)[i][3] ^ t ;              Tm = xtime(Tm);  (*state)[i][3] ^= Tm ^ Tmp ;
  }
}

#if MULTIPLY_AS_A_FUNCTION
static uint8_t Multiply(uint8_t x, uint8_t y)
{
  return (((y & 1) * x) ^
       ((y>>1 & 1) * xtime(x)) ^
       ((y>>2 & 1) * xtime(xtime(x))) ^
       ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^
       ((y>>4 & 1) * xtime(xtime(xtime(xtime(x)))))); /* this last call to xtime() can be omitted */
  }
#else
#define Multiply(x, y)                                \
      (  ((y & 1) * x) ^                              \
      ((y>>1 & 1) * xtime(x)) ^                       \
      ((y>>2 & 1) * xtime(xtime(x))) ^                \
      ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^         \
      ((y>>4 & 1) * xtime(xtime(xtime(xtime(x))))))   \

#endif

#if (defined(CBC) && CBC == 1) || (defined(ECB) && ECB == 1)
#define getSBoxInvert(num) (rsbox[(num)])
static void InvMixColumns(state_t* state)
{
  int i;
  uint8_t a, b, c, d;
  for (i = 0; i < 4; ++i)
  {
    a = (*state)[i][0];
    b = (*state)[i][1];
    c = (*state)[i][2];
    d = (*state)[i][3];

    (*state)[i][0] = Multiply(a, 0x0e) ^ Multiply(b, 0x0b) ^ Multiply(c, 0x0d) ^ Multiply(d, 0x09);
    (*state)[i][1] = Multiply(a, 0x09) ^ Multiply(b, 0x0e) ^ Multiply(c, 0x0b) ^ Multiply(d, 0x0d);
    (*state)[i][2] = Multiply(a, 0x0d) ^ Multiply(b, 0x09) ^ Multiply(c, 0x0e) ^ Multiply(d, 0x0b);
    (*state)[i][3] = Multiply(a, 0x0b) ^ Multiply(b, 0x0d) ^ Multiply(c, 0x09) ^ Multiply(d, 0x0e);
  }
}

static void InvSubBytes(state_t* state)
{
  uint8_t i, j;
  for (i = 0; i < 4; ++i)
  {
    for (j = 0; j < 4; ++j)
    {
      (*state)[j][i] = getSBoxInvert((*state)[j][i]);
    }
  }
}

static void InvShiftRows(state_t* state)
{
  uint8_t temp;
  // Rotate first row 1 columns to right
  temp = (*state)[3][1];
  (*state)[3][1] = (*state)[2][1];
  (*state)[2][1] = (*state)[1][1];
  (*state)[1][1] = (*state)[0][1];
  (*state)[0][1] = temp;

  // Rotate second row 2 columns to right
  temp = (*state)[0][2];
  (*state)[0][2] = (*state)[2][2];
  (*state)[2][2] = temp;

  temp = (*state)[1][2];
  (*state)[1][2] = (*state)[3][2];
  (*state)[3][2] = temp;

  // Rotate third row 3 columns to right
  temp = (*state)[0][3];
  (*state)[0][3] = (*state)[1][3];
  (*state)[1][3] = (*state)[2][3];
  (*state)[2][3] = (*state)[3][3];
  (*state)[3][3] = temp;
}
#endif
static void Cipher(state_t* state, const uint8_t* RoundKey)
{
  uint8_t round = 0;

  // Add the First round key to the state before starting the rounds.
  AddRoundKey(0, state, RoundKey);
  // There will be Nr rounds.
  // The first Nr-1 rounds are identical.
  // These Nr rounds are executed in the loop below.
  // Last one without MixColumns()
  for (round = 1; ; ++round)
  {
    SubBytes(state);
    ShiftRows(state);
    if (round == Nr) {
      break;
    }
    MixColumns(state);
    AddRoundKey(round, state, RoundKey);
  }
  // Add round key to last round
  AddRoundKey(Nr, state, RoundKey);
}

#if (defined(CBC) && CBC == 1) || (defined(ECB) && ECB == 1)
static void InvCipher(state_t* state, const uint8_t* RoundKey)
{
  uint8_t round = 0;

  // Add the First round key to the state before starting the rounds.
  AddRoundKey(Nr, state, RoundKey);

  // There will be Nr rounds.
  // The first Nr-1 rounds are identical.
  // These Nr rounds are executed in the loop below.
  // Last one without InvMixColumn()
  for (round = (Nr - 1); ; --round)
  {
    InvShiftRows(state);
    InvSubBytes(state);
    AddRoundKey(round, state, RoundKey);
    if (round == 0) {
      break;
    }
    InvMixColumns(state);
  }

}
#endif
#if defined(ECB) && (ECB == 1)
void AES_ECB_encrypt(const struct AES_ctx* ctx, uint8_t* buf)
{
  // The next function call encrypts the PlainText with the Key using AES algorithm.
  Cipher((state_t*)buf, ctx->RoundKey);
}

void AES_ECB_decrypt(const struct AES_ctx* ctx, uint8_t* buf)
{
  // The next function call decrypts the PlainText with the Key using AES algorithm.
  InvCipher((state_t*)buf, ctx->RoundKey);
}



#endif

#if defined(CBC) && (CBC == 1)
```

```
static void XorWithIv(uint8_t* buf, const uint8_t* Iv)
{
  uint8_t i;
  for (i = 0; i < AES_BLOCKLEN; ++i) // The block in AES is always 128bit no matter the key size
  {
    buf[i] ^= Iv[i];
  }
}

void AES_CBC_encrypt_buffer(struct AES_ctx *ctx, uint8_t* buf, size_t length)
{
  size_t i;
  uint8_t *Iv = ctx->Iv;
  for (i = 0; i < length; i += AES_BLOCKLEN)
  {
    XorWithIv(buf, Iv);
    Cipher((state_t*)buf, ctx->RoundKey);
    Iv = buf;
    buf += AES_BLOCKLEN;
  }
  memcpy(ctx->Iv, Iv, AES_BLOCKLEN);
}

void AES_CBC_decrypt_buffer(struct AES_ctx* ctx, uint8_t* buf, size_t length)
{
  size_t i;
  uint8_t storeNextIv[AES_BLOCKLEN];
  for (i = 0; i < length; i += AES_BLOCKLEN)
  {
    memcpy(storeNextIv, buf, AES_BLOCKLEN);
    InvCipher((state_t*)buf, ctx->RoundKey);
    XorWithIv(buf, ctx->Iv);
    memcpy(ctx->Iv, storeNextIv, AES_BLOCKLEN);
    buf += AES_BLOCKLEN;
  }

}

#endif



#if defined(CTR) && (CTR == 1)
void AES_CTR_xcrypt_buffer(struct AES_ctx* ctx, uint8_t* buf, size_t length)
{
  uint8_t buffer[AES_BLOCKLEN];

  size_t i;
  int bi;
  for (i = 0, bi = AES_BLOCKLEN; i < length; ++i, ++bi)
  {
    if (bi == AES_BLOCKLEN) /* we need to regen xor compliment in buffer */
    {

      memcpy(buffer, ctx->Iv, AES_BLOCKLEN);
      Cipher((state_t*)buffer,ctx->RoundKey);
      for (bi = (AES_BLOCKLEN - 1); bi >= 0; --bi)
      {
                /* inc will overflow */
        if (ctx->Iv[bi] == 255)
                {
```

```
          ctx->Iv[bi] = 0;

          continue;

        }

        ctx->Iv[bi] += 1;

        break;

      }

      bi = 0;

    }


    buf[i] = (buf[i] ^ buffer[bi]);

  }

}


#endif

#define CBC 1

#define CTR 1

#define ECB 1


static void phex(uint8_t* str);

static int test_encrypt_cbc(void);

static int test_decrypt_cbc(void);

static int test_encrypt_ctr(void);

static int test_decrypt_ctr(void);

static int test_encrypt_ecb(void);

static int test_decrypt_ecb(void);

static void test_encrypt_ecb_verbose(void);



int main(void)

{

    int exit;
```

```
#if defined(AES256)
   cout<<"\nTesting AES256\n\n";
#elif defined(AES192)
   cout<<"\nTesting AES192\n\n";
#elif defined(AES128)
   cout<<"\nTesting AES128\n\n";
#else
   cout<<"You need to specify a symbol between AES128, AES192 or AES256. Exiting";
   return 0;
#endif

   exit = test_encrypt_cbc() + test_decrypt_cbc() +
           test_encrypt_ctr() + test_decrypt_ctr() +
           test_decrypt_ecb() + test_encrypt_ecb();
   test_encrypt_ecb_verbose();

   return exit;
}


static void phex(uint8_t* str)
{

#if defined(AES256)
   uint8_t len = 32;
#elif defined(AES192)
   uint8_t len = 24;
#elif defined(AES128)
   uint8_t len = 16;
#endif
```

```
    unsigned char i;
    for (i = 0; i < len; ++i)
        printf("%.2x", str[i]);
    printf("\n");
}

static void test_encrypt_ecb_verbose(void)
{

    uint8_t i;
    uint8_t key[16]={ (uint8_t) 0x24, (uint8_t) 0x75, (uint8_t) 0xa2, (uint8_t) 0xb3,(uint8_t) 0x34,(uint8_t) 0x75,(uint8_t) 0x56, (uint8_t) 0x88, (uint8_t) 0x31, (uint8_t) 0xe2, (uint8_t) 0x12, (uint8_t) 0x00, (uint8_t) 0x13, (uint8_t) 0xaa, (uint8_t) 0x54, (uint8_t) 0x87 };
    uint8_t plain_text[16] = { (uint8_t) 0x00, (uint8_t) 0x04, (uint8_t) 0x12, (uint8_t) 0x14, (uint8_t) 0x12, (uint8_t) 0x04, (uint8_t) 0x12, (uint8_t) 0x00, (uint8_t) 0x0c, (uint8_t) 0x00, (uint8_t) 0x13, (uint8_t) 0x11, (uint8_t) 0x08, (uint8_t) 0x23, (uint8_t) 0x19, (uint8_t) 0x19};
                             //  (uint8_t) 0xae, (uint8_t) 0x2d, (uint8_t) 0x8a, (uint8_t) 0x57, (uint8_t) 0x1e, (uint8_t) 0x03, (uint8_t) 0xac, (uint8_t) 0x9c, (uint8_t) 0x9e, (uint8_t) 0xb7, (uint8_t) 0x6f, (uint8_t) 0xac, (uint8_t) 0x45, (uint8_t) 0xaf, (uint8_t) 0x8e, (uint8_t) 0x51,
                              // (uint8_t) 0x30, (uint8_t) 0xc8, (uint8_t) 0x1c, (uint8_t) 0x46, (uint8_t) 0xa3, (uint8_t) 0x5c, (uint8_t) 0xe4, (uint8_t) 0x11, (uint8_t) 0xe5, (uint8_t) 0xfb, (uint8_t) 0xc1, (uint8_t) 0x19, (uint8_t) 0x1a, (uint8_t) 0x0a, (uint8_t) 0x52, (uint8_t) 0xef,
                              // (uint8_t) 0xf6, (uint8_t) 0x9f, (uint8_t) 0x24, (uint8_t) 0x45, (uint8_t) 0xdf, (uint8_t) 0x4f, (uint8_t) 0x9b, (uint8_t) 0x17, (uint8_t) 0xad, (uint8_t) 0x2b, (uint8_t) 0x41, (uint8_t) 0x7b, (uint8_t) 0xe6, (uint8_t) 0x6c, (uint8_t) 0x37, (uint8_t) 0x10 };
}
    cout<<"ECB encrypt verbose:\n\n";
    cout<<"plain text:\n";
    for (i = (uint8_t) 0; i < (uint8_t) 1; ++i)
    {
        phex(plain_text + i * (uint8_t) 16);
    }
    cout<<"key:\n";
    phex(key);
    cout<<"\n";
    cout<<"ciphertext:\n";

    struct AES_ctx ctx;
    AES_init_ctx(&ctx, key);

    for (i = 0; i < 1; ++i)
    {
      AES_ECB_encrypt(&ctx, plain_text + (i * 16));
      phex(plain_text + (i * 16));
    }

}



static int test_encrypt_ecb(void)
{
#if defined(AES256)
    uint8_t key[] = { 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe, 0x2b, 0x73, 0xae, 0xf0, 0x85,
0x7d, 0x77, 0x81,
                      0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7, 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf,
0xf4 };
    uint8_t out[] = { 0xf3, 0xee, 0xd1, 0xbd, 0xb5, 0xd2, 0xa0, 0x3c, 0x06, 0x4b, 0x5a, 0x7e, 0x3d,
0xb1, 0x81, 0xf8 };
#elif defined(AES192)
    uint8_t key[] = { 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52, 0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90,
0x79, 0xe5,
                      0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b };
    uint8_t out[] = { 0xbd, 0x33, 0x4f, 0x1d, 0x6e, 0x45, 0xf2, 0x5f, 0xf7, 0x12, 0xa2, 0x14, 0x57, 0x1f,
0xa5, 0xcc };
#elif defined(AES128)
```

```
    uint8_t key[] = { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c };
    uint8_t out[] = { 0x3a, 0xd7, 0x7b, 0xb4, 0x0d, 0x7a, 0x36, 0x60, 0xa8, 0x9e, 0xca, 0xf3, 0x24, 0x66, 0xef, 0x97 };
#endif

    uint8_t in[]  = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a };
    struct AES_ctx ctx;

    AES_init_ctx(&ctx, key);
    AES_ECB_encrypt(&ctx, in);

    cout<<"ECB encrypt: ";

    if (0 == memcmp((char*) out, (char*) in, 16)) {
        cout<<"SUCCESS!\n";
              return(0);
    } else {
        cout<<"FAILURE!\n";
              return(1);
    }
}

static int test_decrypt_cbc(void)
{

#if defined(AES256)
    uint8_t key[] = { 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe, 0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
                      0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7, 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4 };
    uint8_t in[]  = { 0xf5, 0x8c, 0x4c, 0x04, 0xd6, 0xe5, 0xf1, 0xba, 0x77, 0x9e, 0xab, 0xfb, 0x5f, 0x7b, 0xfb, 0xd6,
                      0x9c, 0xfc, 0x4e, 0x96, 0x7e, 0xdb, 0x80, 0x8d, 0x67, 0x9f, 0x77, 0x7b, 0xc6, 0x70, 0x2c, 0x7d,
                      0x39, 0xf2, 0x33, 0x69, 0xa9, 0xd9, 0xba, 0xcf, 0xa5, 0x30, 0xe2, 0x63, 0x04, 0x23, 0x14, 0x61,
                      0xb2, 0xeb, 0x05, 0xe2, 0xc3, 0x9b, 0xe9, 0xfc, 0xda, 0x6c, 0x19, 0x07, 0x8c, 0x6a, 0x9d, 0x1b };
#elif defined(AES192)
    uint8_t key[] = { 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52, 0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5, 0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b };
    uint8_t in[]  = { 0x4f, 0x02, 0x1d, 0xb2, 0x43, 0xbc, 0x63, 0x3d, 0x71, 0x78, 0x18, 0x3a, 0x9f, 0xa0, 0x71, 0xe8,
                      0xb4, 0xd9, 0xad, 0xa9, 0xad, 0x7d, 0xed, 0xf4, 0xe5, 0xe7, 0x38, 0x76, 0x3f, 0x69, 0x14, 0x5a,
                      0x57, 0x1b, 0x24, 0x20, 0x12, 0xfb, 0x7a, 0xe0, 0x7f, 0xa9, 0xba, 0xac, 0x3d, 0xf1, 0x02, 0xe0,
                      0x08, 0xb0, 0xe2, 0x79, 0x88, 0x59, 0x88, 0x81, 0xd9, 0x20, 0xa9, 0xe6, 0x4f, 0x56, 0x15, 0xcd };
#elif defined(AES128)
    uint8_t key[] = { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c };
    uint8_t in[]  = { 0x76, 0x49, 0xab, 0xac, 0x81, 0x19, 0xb2, 0x46, 0xce, 0xe9, 0x8e, 0x9b, 0x12, 0xe9, 0x19, 0x7d,
                      0x50, 0x86, 0xcb, 0x9b, 0x50, 0x72, 0x19, 0xee, 0x95, 0xdb, 0x11, 0x3a, 0x91, 0x76, 0x78, 0xb2,
                      0x73, 0xbe, 0xd6, 0xb8, 0xe3, 0xc1, 0x74, 0x3b, 0x71, 0x16, 0xe6, 0x9e, 0x22, 0x22, 0x95, 0x16,
                      0x3f, 0xf1, 0xca, 0xa1, 0x68, 0x1f, 0xac, 0x09, 0x12, 0x0e, 0xca, 0x30, 0x75, 0x86, 0xe1, 0xa7 };
#endif
    uint8_t iv[]  = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f };
    uint8_t out[] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                      0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                    0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                    0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10 };
//  uint8_t buffer[64];
    struct AES_ctx ctx;

    AES_init_ctx_iv(&ctx, key, iv);
    AES_CBC_decrypt_buffer(&ctx, in, 64);



    if (0 == memcmp((char*) out, (char*) in, 64)) {
        cout<<"SUCCESS!\n";
              return(0);
    } else {
        cout<<"FAILURE!\n";
              return(1);
    }
}

static int test_encrypt_cbc(void)
{
#if defined(AES256)
    uint8_t key[] = { 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe, 0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
                    0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7, 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4 };
    uint8_t out[] = { 0xf5, 0x8c, 0x4c, 0x04, 0xd6, 0xe5, 0xf1, 0xba, 0x77, 0x9e, 0xab, 0xfb, 0x5f, 0x7b, 0xfb, 0xd6,
                    0x9c, 0xfc, 0x4e, 0x96, 0x7e, 0xdb, 0x80, 0x8d, 0x67, 0x9f, 0x77, 0x7b, 0xc6, 0x70, 0x2c, 0x7d,
                    0x39, 0xf2, 0x33, 0x69, 0xa9, 0xd9, 0xba, 0xcf, 0xa5, 0x30, 0xe2, 0x63, 0x04, 0x23, 0x14, 0x61,
                      0xb2, 0xeb, 0x05, 0xe2, 0xc3, 0x9b, 0xe9, 0xfc, 0xda, 0x6c, 0x19, 0x07, 0x8c, 0x6a, 0x9d, 0x1b };
#elif defined(AES192)
    uint8_t key[] = { 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52, 0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5, 0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b };
    uint8_t out[] = { 0x4f, 0x02, 0x1d, 0xb2, 0x43, 0xbc, 0x63, 0x3d, 0x71, 0x78, 0x18, 0x3a, 0x9f, 0xa0, 0x71, 0xe8,
                      0xb4, 0xd9, 0xad, 0xa9, 0xad, 0x7d, 0xed, 0xf4, 0xe5, 0xe7, 0x38, 0x76, 0x3f, 0x69, 0x14, 0x5a,
                      0x57, 0x1b, 0x24, 0x20, 0x12, 0xfb, 0x7a, 0xe0, 0x7f, 0xa9, 0xba, 0xac, 0x3d, 0xf1, 0x02, 0xe0,
                      0x08, 0xb0, 0xe2, 0x79, 0x88, 0x59, 0x88, 0x81, 0xd9, 0x20, 0xa9, 0xe6, 0x4f, 0x56, 0x15, 0xcd };
#elif defined(AES128)
    uint8_t key[] = { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c };
    uint8_t out[] = { 0x76, 0x49, 0xab, 0xac, 0x81, 0x19, 0xb2, 0x46, 0xce, 0xe9, 0x8e, 0x9b, 0x12, 0xe9, 0x19, 0x7d,
                      0x50, 0x86, 0xcb, 0x9b, 0x50, 0x72, 0x19, 0xee, 0x95, 0xdb, 0x11, 0x3a, 0x91, 0x76, 0x78, 0xb2,
                      0x73, 0xbe, 0xd6, 0xb8, 0xe3, 0xc1, 0x74, 0x3b, 0x71, 0x16, 0xe6, 0x9e, 0x22, 0x22, 0x95, 0x16,
                      0x3f, 0xf1, 0xca, 0xa1, 0x68, 0x1f, 0xac, 0x09, 0x12, 0x0e, 0xca, 0x30, 0x75, 0x86, 0xe1, 0xa7 };
#endif
    uint8_t iv[]  = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f };
    uint8_t in[]  = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                      0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                      0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                      0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10 };
    struct AES_ctx ctx;
    AES_init_ctx_iv(&ctx, key, iv);
    AES_CBC_encrypt_buffer(&ctx, in, 64);
    if (0 == memcmp((char*) out, (char*) in, 64)) {
        cout<<"SUCCESS!\n";
              return(0);
    } else {
        cout<<"FAILURE!\n";
              return(1);
    }
}

static int test_xcrypt_ctr(const char* xcrypt);
static int test_encrypt_ctr(void)
{
    return test_xcrypt_ctr("encrypt");
}

static int test_decrypt_ctr(void)
{
    return test_xcrypt_ctr("decrypt");
}

static int test_xcrypt_ctr(const char* xcrypt)
{
#if defined(AES256)
    uint8_t key[32] = { 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe, 0x2b, 0x73, 0xae, 0xf0, 0x85,
0x7d, 0x77, 0x81,
                        0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7, 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14,
0xdf, 0xf4 };
    uint8_t in[64]  = { 0x60, 0x1e, 0xc3, 0x13, 0x77, 0x57, 0x89, 0xa5, 0xb7, 0xa7, 0xf5, 0x04, 0xbb,
0xf3, 0xd2, 0x28,
                      0xf4, 0x43, 0xe3, 0xca, 0x4d, 0x62, 0xb5, 0x9a, 0xca, 0x84, 0xe9, 0x90, 0xca, 0xca, 0xf5,
0xc5,

                      0x2b, 0x09, 0x30, 0xda, 0xa2, 0x3d, 0xe9, 0x4c, 0xe8, 0x70, 0x17, 0xba, 0x2d, 0x84,
0x98, 0x8d,

                      0xdf, 0xc9, 0xc5, 0x8d, 0xb6, 0x7a, 0xad, 0xa6, 0x13, 0xc2, 0xdd, 0x08, 0x45, 0x79,
0x41, 0xa6 };

#elif defined(AES192)

    uint8_t key[24] = { 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52, 0xc8, 0x10, 0xf3, 0x2b, 0x80,
0x90, 0x79, 0xe5,

                      0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b };

    uint8_t in[64]  = { 0x1a, 0xbc, 0x93, 0x24, 0x17, 0x52, 0x1c, 0xa2, 0x4f, 0x2b, 0x04, 0x59, 0xfe,
0x7e, 0x6e, 0x0b,

                      0x09, 0x03, 0x39, 0xec, 0x0a, 0xa6, 0xfa, 0xef, 0xd5, 0xcc, 0xc2, 0xc6, 0xf4, 0xce, 0x8e,
0x94,

                      0x1e, 0x36, 0xb2, 0x6b, 0xd1, 0xeb, 0xc6, 0x70, 0xd1, 0xbd, 0x1d, 0x66, 0x56, 0x20,
0xab, 0xf7,

                      0x4f, 0x78, 0xa7, 0xf6, 0xd2, 0x98, 0x09, 0x58, 0x5a, 0x97, 0xda, 0xec, 0x58, 0xc6,
0xb0, 0x50 };

#elif defined(AES128)

    uint8_t key[16] = { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09,
0xcf, 0x4f, 0x3c };

    uint8_t in[64]  = { 0x87, 0x4d, 0x61, 0x91, 0xb6, 0x20, 0xe3, 0x26, 0x1b, 0xef, 0x68, 0x64, 0x99,
0x0d, 0xb6, 0xce,

                      0x98, 0x06, 0xf6, 0x6b, 0x79, 0x70, 0xfd, 0xff, 0x86, 0x17, 0x18, 0x7b, 0xb9, 0xff, 0xfd,
0xff,

                      0x5a, 0xe4, 0xdf, 0x3e, 0xdb, 0xd5, 0xd3, 0x5e, 0x5b, 0x4f, 0x09, 0x02, 0x0d, 0xb0,
0x3e, 0xab,

                      0x1e, 0x03, 0x1d, 0xda, 0x2f, 0xbe, 0x03, 0xd1, 0x79, 0x21, 0x70, 0xa0, 0xf3, 0x00,
0x9c, 0xee };

#endif

    uint8_t iv[16]  = { 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd,
0xfe, 0xff };

    uint8_t out[64] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e, 0x11, 0x73,
0x93, 0x17, 0x2a,

                      0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e,
0x51,
                    0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52,
0xef,
                    0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37,
0x10 };
    struct AES_ctx ctx;

    AES_init_ctx_iv(&ctx, key, iv);
    AES_CTR_xcrypt_buffer(&ctx, in, 64);



    if (0 == memcmp((char *) out, (char *) in, 64)) {
        cout<<"SUCCESS!\n";
              return(0);
    } else {
        cout<<"FAILURE!\n";
              return(1);
    }
}


static int test_decrypt_ecb(void)
{
#if defined(AES256)
    uint8_t key[] = { 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe, 0x2b, 0x73, 0xae, 0xf0, 0x85,
0x7d, 0x77, 0x81,
                    0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7, 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf,
0xf4 };
    uint8_t in[]  = { 0xf3, 0xee, 0xd1, 0xbd, 0xb5, 0xd2, 0xa0, 0x3c, 0x06, 0x4b, 0x5a, 0x7e, 0x3d, 0xb1,
0x81, 0xf8 };
#elif defined(AES192)
    uint8_t key[] = { 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52, 0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90,
0x79, 0xe5,
                    0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b };
    uint8_t in[]  = { 0xbd, 0x33, 0x4f, 0x1d, 0x6e, 0x45, 0xf2, 0x5f, 0xf7, 0x12, 0xa2, 0x14, 0x57, 0x1f,
0xa5, 0xcc };
#elif defined(AES128)
    uint8_t key[] = { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf,
0x4f, 0x3c };
    uint8_t in[]  = { 0x3a, 0xd7, 0x7b, 0xb4, 0x0d, 0x7a, 0x36, 0x60, 0xa8, 0x9e, 0xca, 0xf3, 0x24, 0x66,
0xef, 0x97 };
#endif

    uint8_t out[]   = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e, 0x11, 0x73,
0x93, 0x17, 0x2a };
    struct AES_ctx ctx;

    AES_init_ctx(&ctx, key);
    AES_ECB_decrypt(&ctx, in);

   cout<<"ECB decrypt: ";

    if (0 == memcmp((char*) out, (char*) in, 16)) {
        cout<<"SUCCESS!\n";
              return(0);
    } else {
        cout<<"FAILURE!\n";
              return(1);
    }
}
```