Block Cipher 과제

[Code] : DES.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "BlockCipherMode.h"
#define BLOCK_MODE 1        /* 1: CBC, 2: CFB, 3: OFB, 4: CTR */
#include "DES.h"
int main()
{
    int i;
    BYTE p_text[128]={0,};
    BYTE key[9]={0,};
    BYTE IV[9]={0,};
    BYTE c_text[128]={0,};
    BYTE d_text[128]={0,};
    int msg_len;
    UINT64 ctr=0;
    /* 평문 입력 */
    printf("평문 입력: ");
    gets((char *)p_text);
    /* 비밀키 입력 */
    printf("비밀키 입력: ");
    scanf("%s", key);
    fflush(stdin);
    #if(BLOCK_MODE!=4)
        /* 초기화 벡터 입력 */
        printf("초기화 벡터 입력: ");
        scanf("%s", IV);
        fflush(stdin);
    #else
        /* 카운터 입력 */
        printf("ctr 입력: ");
        scanf("%u", &ctr);
        fflush(stdin);
    #endif
        /* 메시지 길이 계산 */
        msg_len=(strlen((char *)p_text) % BLOCK_SIZE) ?
                    ((strlen((char *)p_text) / BLOCK_SIZE +1)*8):
                    strlen((char *)p_text);
    printf("Cipher Block Codebook 암호화 실행\n");
    #if(BLOCK_MODE==1)
        DES_CBC_Enc(p_text, c_text, IV, key, msg_len);//DES-CBC 암호화
    #elif(BLOCK_MODE==2)
        DES_CFB_Enc(p_text, c_text, IV, key, msg_len);//DES-CFB 암호화
    #elif(BLOCK_MODE==3)
        DES_OFB_Enc(p_text, c_text, IV, key, msg_len);//DES-OFB 암호화
    #else
        DES_CTR_Enc(p_text, c_text, key, ctr, msg_len);//DES-CTR 암호화
    #endif
        /* 암호문 출력 */
        printf("\n암호문: ");
        for(i=0; i<msg_len; i++)
            printf("%x", c_text[i]);
        printf("\n");
    printf("Cipher Block Codebook 복호화 실행\n");
    #if(BLOCK_MODE==1)
        DES_CBC_Dec(c_text, d_text, IV, key, msg_len);//DES-CBC 복호화
    #elif(BLOCK_MODE==2)
        DES_CFB_Dec(c_text, d_text, IV, key, msg_len);//DES-CFB 복호화
```

```
    #elif(BLOCK_MODE==3)
        DES_OFB_Dec(c_text, d_text, IV, key, msg_len);//DES-CFB 복호화
    #else
        DES_CTR_Dec(c_text, d_text, key, ctr, msg_len);//DES-CTR 복호화
    #endif
        /* 복호문 출력 */
        printf("\n복호문: ");
        for(i=0; i<msg_len; i++)
            printf("%x", d_text[i]);
        printf("\n");
    return 0;
}
```

[Code] : BlockCipherMode.h

```
#include "DES.h"
void IntToByte(UINT64 int64, BYTE *text, int j){ // to byte
        for(int i = 0; i < 8; i++){
                text[j * BLOCK_SIZE + i] |= (int64 >> (56 - (i * 8)));
        }
}
void ByteToINT(BYTE *text, UINT64 *int64, int j){ // to bit
        for(int i = 0; i < 8; i++){
                *int64 |= (UINT64)text[j * BLOCK_SIZE + i] << (56 - (i * 8));
        }
}
// 암호화
void DES_CBC_Enc(BYTE* p_text, BYTE* c_text, BYTE* IV, BYTE* key, int msg_len){
        BYTE temp[BLOCK_SIZE] = { 0, };
        UINT64 iv_64 = 0;
        UINT64 iv_64_2 = 0;
        int block_count = (msg_len % BLOCK_SIZE) ? (msg_len / BLOCK_SIZE + 1) : (msg_len
/ BLOCK_SIZE);
        for(int i = 0; i < block_count; i++){
                for(int j = 0; j < BLOCK_SIZE; j++){
                        temp[j] = 0;
                }
                if(i == 0)
                        ByteToINT(IV, &iv_64, 0);
                else
                        ByteToINT(c_text, &iv_64, i-1);
                ByteToINT(p_text, &iv_64_2, i);
                iv_64 = iv_64_2 ^ iv_64;
                IntToByte(iv_64, temp, 0);
                DES_Encryption(&temp[0], &c_text[i * BLOCK_SIZE], key);
        }
}
void DES_CFB_Enc(BYTE* p_text, BYTE* c_text, BYTE* IV, BYTE* key, int msg_len) {
        BYTE temp[BLOCK_SIZE] = { 0, };
        UINT64 iv_64 = 0;
        UINT64 iv_64_2 = 0;
        int block_count = (msg_len % BLOCK_SIZE) ? (msg_len / BLOCK_SIZE + 1) : (msg_len
/ BLOCK_SIZE);
        for (int i = 0; i < block_count; i++)
                for (int j = 0; j < BLOCK_SIZE; j++)
                        temp[j] = 0;
                if(i == 0){
                        cout << "IV 사용" << endl;
                        DES_Encryption(IV, &temp[0], key);
                }else{
                        cout << "이전 블럭 사용" << endl;
                        DES_Encryption(&c_text[(i-1)*BLOCK_SIZE],&temp[0],key);
```

```
                }
        ByteToINT(temp, &iv_64, 0);
        ByteToINT(p_text, &iv_64, i);
        iv_64 = iv_64_2 ^ iv_64;
                IntToByte(iv_64, c_text, i);
          }
}
void DES_OFB_Enc(BYTE* p_text, BYTE* c_text, BYTE* IV, BYTE* key, int msg_len) {
          BYTE output[BLOCK_SIZE] = { 0, };
          BYTE output_pre[BLOCK_SIZE] = { 0, };
          UINT64 iv_64 = 0;
          UINT64 iv_64_2 = 0;
          int block_count = (msg_len % BLOCK_SIZE) ? (msg_len / BLOCK_SIZE + 1) : (msg_len
/ BLOCK_SIZE);
          for(int i = 0; i < block_count; i++){
                    for(int j = 0; j < BLOCK_SIZE; j++){
                              output[j] = 0;
                    }
                    if(i == 0){
                              cout << "IV 사용" << endl;
                              DES_Encryption(IV, &output[0], key);
                    }else{
                              cout << "이전 블럭 사용" << endl;
                              DES_Encryption(&output_pre[0], &output[0], key);
                    }
                    ByteToINT(output, &iv_64, 0);
                    ByteToINT(p_text, &iv_64, i);
                    iv_64 = iv_64_2 ^ iv_64;
                    IntToByte(iv_64, c_text, i);
                    for(int k = 0; k < BLOCK_SIZE; k++){
                              output_pre[k] = output[k];
                    }
          }
}
void DES_CTR_Enc(BYTE* p_text, BYTE* c_text, BYTE* key, UINT64 ctr, int msg_len) {
          BYTE output[BLOCK_SIZE] = { 0, };
          BYTE temp[BLOCK_SIZE] = { 0, };
          UINT64 iv_64 = 0;
          UINT64 iv_64_2 = 0;
          int block_count = (msg_len % BLOCK_SIZE) ? (msg_len / BLOCK_SIZE + 1) : (msg_len
/ BLOCK_SIZE);
          for (int i = 0; i < block_count; i++) {
                    for (int j = 0; j < BLOCK_SIZE; j++) {
                              temp[j] = 0;
                              output[j] = 0;
                    }
                    IntToByte(ctr + i, temp, 0);
                    DES_Encryption(&temp[0], &output[0], key);
                    ByteToINT(output, &iv_64, 0);
                    ByteToINT(p_text, &iv_64, i);
                    iv_64 = iv_64_2 ^ iv_64;
                    IntToByte(iv_64, c_text, i);
          }
}
// 복호화
void DES_CBC_Dec(BYTE* c_text, BYTE* d_text, BYTE* IV, BYTE* key, int msg_len) {
          BYTE temp[BLOCK_SIZE] = { 0, };
          UINT64 iv_64 = 0;
          UINT64 iv_64_2 = 0;
          int block_count = (msg_len % BLOCK_SIZE) ? (msg_len / BLOCK_SIZE + 1) : (msg_len
/ BLOCK_SIZE);
          // 0 : 거짓, 0 이외 : 참
          for (int i = 0; i < block_count; i++) {
                    for (int j = 0; j < BLOCK_SIZE; j++) {
```

```
                        temp[j] = 0;
                        d_text[i * BLOCK_SIZE + j] = 0;
                }
                DES_Decryption(&c_text[i * BLOCK_SIZE], &temp[0], key);
                if (i == 0) {
                        ByteToINT(IV, &iv_64, 0);
                }
                else {
                        ByteToINT(c_text, &iv_64, i - 1);
                }
                ByteToINT(temp, &iv_64_2, 0);
                iv_64 = iv_64_2 ^ iv_64;
                IntToByte(iv_64, d_text, i); //iv_64를 d_text
        }
}
void DES_CFB_Dec(BYTE* c_text, BYTE* d_text, BYTE* IV, BYTE* key, int msg_len) {
        BYTE temp[BLOCK_SIZE] = { 0, };
        UINT64 iv_64 = 0;
        UINT64 iv_64_2 = 0;
        int block_count = (msg_len % BLOCK_SIZE) ? (msg_len / BLOCK_SIZE + 1) : (msg_len
/ BLOCK_SIZE);
        // 0 : 거짓, 0이외 : 참
        for (int i = 0; i < block_count; i++) {
                for (int j = 0; j < BLOCK_SIZE; j++) {
                        temp[j] = 0;
                        d_text[i * BLOCK_SIZE + j] = 0;
                }
                if (i == 0) {
                        cout << "IV 사용" << endl;
                        DES_Encryption(IV, &temp[0], key);
                }
                else {
                        cout << "이전 블럭 사용" << endl;
                        DES_Decryption(&c_text[(i - 1) * BLOCK_SIZE], &temp[0], key);
                }
                ByteToINT(temp, &iv_64, 0);
                ByteToINT(c_text, &iv_64_2, i);
                iv_64 = iv_64_2 ^ iv_64;
                IntToByte(iv_64, d_text, i);
        }
}
void DES_OFB_Dec(BYTE* c_text, BYTE* d_text, BYTE* IV, BYTE* key, int msg_len) {
        BYTE output[BLOCK_SIZE] = { 0, };
        BYTE output_pre[BLOCK_SIZE] = { 0, };
        UINT64 iv_64 = 0;
        UINT64 iv_64_2 = 0;
        int block_count = (msg_len % BLOCK_SIZE) ? (msg_len / BLOCK_SIZE + 1) : (msg_len
/ BLOCK_SIZE);
        for (int i = 0; i < block_count; i++) {
                for (int j = 0; j < BLOCK_SIZE; j++) {
                        output[j] = 0;
                }
                if (i == 0) {
                        cout << "IV 사용" << endl;
                        DES_Encryption(IV, &output[0], key);
                }
                else {
                        cout << "이전 블럭 사용" << endl;
                        DES_Encryption(&output_pre[0], &output[0], key);
                }
                ByteToINT(output, &iv_64, 0);
                ByteToINT(c_text, &iv_64, i);
                iv_64 = iv_64_2 ^ iv_64;
                IntToByte(iv_64, d_text, i);
```

```
                    for (int k = 0; k < BLOCK_SIZE; k++) {
                            output_pre[k] = output[k];
                    }
            }
}
void DES_CTR_Dec(BYTE* c_text, BYTE* d_text, BYTE* key, UINT64 ctr, int msg_len){
            BYTE output[BLOCK_SIZE] = { 0, };
            BYTE temp[BLOCK_SIZE] = { 0, };
            UINT64 iv_64 = 0;
            UINT64 iv_64_2 = 0;
            int block_count = (msg_len % BLOCK_SIZE) ? (msg_len / BLOCK_SIZE + 1) : (msg_len
/ BLOCK_SIZE);
            for (int i = 0; i < block_count; i++) {
                    for (int j = 0; j < BLOCK_SIZE; j++) {
                            temp[j] = 0;
                            output[j] = 0;
                    }
                    IntToByte(ctr + i, temp, 0);
                    DES_Encryption(&temp[0], &output[0], key);
                    ByteToINT(output, &iv_64, 0);
                    ByteToINT(c_text, &iv_64, i);
                    iv_64 = iv_64_2 ^ iv_64;
                    IntToByte(iv_64, d_text, i);
            }
}
```

[Code] : DES.h

```
#pragma once
#include <stdint.h>
#include <cstdint>
#include <iostream>
#include <bitset>
using namespace std;
#define BLOCK_SIZE 8
#define DES_ROUND 16
typedef unsigned char BYTE;
typedef unsigned int UINT;
typedef unsigned long long UINT64;
int ip[64] = { 58, 50, 42, 34, 26, 18, 10, 2,
               60, 52, 44, 36, 28, 20, 12, 4,
               62, 54, 46, 38, 30, 22, 14, 6,
               64, 56, 48, 40, 32, 24, 16, 8,
               57, 49, 41, 33, 25, 17, 9, 1,
               59, 51, 43, 35, 27, 19, 11, 3,
               61, 53, 45, 37, 29, 21, 13, 5,
               63, 55, 47, 39, 31, 23, 15, 7 };
int iip[64] = { 40, 8, 48, 16, 56, 24, 64, 32,
                39, 7, 47, 15, 55, 23, 63, 31,
                38, 6, 46, 14, 54 ,22, 62, 30,
                37, 5, 45, 13, 53, 21, 61, 29,
                36, 4, 44, 12, 52, 20, 60, 28,
                35, 3, 43, 11, 51, 19, 59, 27,
                34, 2, 42, 10, 50, 18, 58, 26,
                33, 1, 41, 9, 49, 17, 57, 25 };
int E[48] = { 32, 1, 2, 3, 4, 5, 4, 5,
                              6, 7, 8, 9, 8, 9, 10, 11,
              12, 13, 12, 13, 14, 15, 16, 17,
              16, 17, 18, 19, 20, 21, 20, 21,
              22, 23, 24, 25, 24, 25, 26, 27,
              28, 29, 28, 29, 30, 31, 32, 1 };
/*
int E[48] = { 32, 6, 12, 16, 22, 28, 1, 7,
```

```
                                13, 17, 23, 29, 2, 8, 12, 18,
                24, 28, 3, 9, 13, 19, 25, 29,
                4, 8, 14, 20, 24, 30, 5, 9,
                15, 21, 25, 31, 4, 10, 16, 20,
                26, 32, 5, 11, 17, 21, 27, 1 };
*/
int s_box[8][4][16] = {
                        {
                                { 14,  4, 13,  1,  2, 15, 11,  8, 3, 10,  6, 12,  5,  9,  0,
7 },
                                { 0, 15,  7,  4, 14,  2, 13,  1, 10,  6, 12, 11,  9,  5,  3,
8 },
                                { 4,  1, 14,  8, 13,  6,  2, 11, 15, 12,  9,  7,  3, 10,  5,
0 },
                                { 15, 12,  8,  2,  4,  9,  1,  7, 5, 11,  3, 14, 10,  0,  6,
13 },
                        },
                {
                   { 15,  1,  8, 14,  6, 11,  3,  4,  9,  7,  2, 13, 12,  0,  5, 10 },
                                { 3, 13,  4,  7, 15,  2,  8, 14, 12,  0,  1, 10,  6,  9, 11,
5 },
                                { 0, 14,  7, 11, 10,  4, 13,  1,  5,  8, 12,  6,  9,  3,  2,
15 },
                                { 13,  8, 10,  1,  3, 15,  4,  2, 11,  6,  7, 12,  0,  5, 14,
9 },
                },
                {
                   { 10,  0,  9, 14,  6,  3, 15,  5,  1, 13, 12,  7, 11,  4,  2,  8 },
                                { 13,  7,  0,  9,  3,  4,  6, 10,  2,  8,  5, 14, 12, 11, 15,
1 },
                                { 13,  6,  4,  9,  8, 15,  3,  0, 11,  1,  2, 12,  5, 10, 14,
7 },
                                { 1, 10, 13,  0,  6,  9,  8,  7,  4, 15, 14,  3, 11,  5,  2,
12 },
                },
                {
                   { 7, 13, 14,  3,  0,  6,  9, 10,  1,  2,  8,  5, 11, 12,  4, 15 },
                                { 13,  8, 11,  5,  6, 15,  0,  3,  4,  7,  2, 12,  1, 10, 14,
9 },
                                { 10,  6,  9,  0, 12, 11,  7, 13, 15,  1,  3, 14,  5,  2,  8,
4 },
                                { 3, 15,  0,  6, 10,  1, 13,  8,  9,  4,  5, 11, 12,  7,  2,
14 },
                },
                {
                   { 2, 12,  4,  1,  7, 10, 11,  6,  8,  5,  3, 15, 13,  0, 14,  9 },
                                { 14, 11,  2, 12,  4,  7, 13,  1,  5,  0, 15, 10,  3,  9,  8,
6 },
                                { 4,  2,  1, 11, 10, 13,  7,  8, 15,  9, 12,  5,  6,  3,  0,
14 },
                                { 11,  8, 12,  7,  1, 14,  2, 13,  6, 15,  0,  9, 10,  4,  5,
3 },
                },
                {
                   { 12,  1, 10, 15,  9,  2,  6,  8,  0, 13,  3,  4, 14,  7,  5, 11 },
                                { 10, 15,  4,  2,  7, 12,  9,  5,  6,  1, 13, 14,  0, 11,  3,
8 },
                                { 9, 14, 15,  5,  2,  8, 12,  3,  7,  0,  4, 10,  1, 13, 11,
6 },
                                { 4,  3,  2, 12,  9,  5, 15, 10, 11, 14,  1,  7,  6,  0,  8,
13 },
                },
                {
                   { 4, 11,  2, 14, 15,  0,  8, 13,  3, 12,  9,  7,  5, 10,  6,  1 },
```

```
                        { 13,  0, 11,  7,  4,  9,  1, 10, 14,  3,  5, 12,  2, 15,  8,
6 },
                        { 1,  4, 11, 13, 12,  3,  7, 14, 10, 15,  6,  8,  0,  5,  9,
2 },
                        { 6, 11, 13,  8,  1,  4, 10,  7,  9,  5,  0, 15, 14,  2,  3,
12 },
            },
            {
                    { 13,  2,  8,  4,  6, 15, 11,  1, 10,  9,  3, 14,  5,  0, 12,  7 },
                        { 1, 15, 13,  8, 10,  3,  7,  4, 12,  5,  6, 11,  0, 14,  9,
2 },
                        { 7, 11,  4,  1,  9, 12, 14,  2,  0,  6, 10, 13, 15,  3,  5,
8 },
                        { 2,  1, 14,  7,  4, 10,  8, 13, 15, 12,  9,  0,  3,  5,  6,
11 },
            },
};
int P[32] = { 16, 7, 20, 21, 29, 12, 28, 17,
                          1, 15, 23, 26, 5, 18, 31, 10,
                          2, 8, 24, 14, 32, 27, 3, 9,
             19, 13, 30, 6, 22, 11, 4, 25 };
/*
int PC_1[56] = { 57, 49, 41, 33, 25, 17, 9,
                                    1, 58, 50, 42, 34, 26, 18,
                                    10, 2, 59, 51, 43, 35, 27,
                19, 11, 3, 60, 52, 44, 36,
                63, 55, 47, 39, 31, 23, 15,
                7, 62, 54, 46, 38, 30, 22,
                14, 6, 61, 53, 45, 37, 29,
                21, 13, 5, 28, 20, 12, 4 };
                                    */
int PC_1[56] = { 57, 1, 10, 19, 63, 7, 14, 21,
                                    49, 58, 2, 11, 55, 62, 6, 13,
                                    41, 50, 59, 3, 47, 54, 61, 5,
                33, 42, 51, 60, 39, 46, 53, 28,
                25, 34, 43, 52, 31, 38, 45, 20,
                17, 26, 35, 44, 23, 30, 37, 12,
                9, 18, 27, 36, 15, 22, 29 ,4};
int PC_2[48] = { 14, 17, 11, 24, 1, 5, 3, 28,
                                    15, 6, 21, 10, 23, 19, 12, 4,
                                    26, 8, 16, 7, 27, 20, 13, 2,
                                    41, 52, 31, 37, 47, 55, 30, 40,
                                    51, 45, 33, 48, 44, 49, 39, 56,
                                    34, 53, 46, 42, 50, 36, 29, 32 };
void PC2(UINT c, UINT d, BYTE *Key_Out);
UINT Cir_Shift(UINT n, int r);
void makeBit28(UINT *c, UINT *d, BYTE *Key_Out);
void PC1(BYTE *Key_In, BYTE *Key_Out);
void Key_Expansion(BYTE *key, BYTE round_key[16][6]);
void WtoB(UINT Left32, UINT Right32, BYTE *out);
void Swap(UINT *x, UINT *y);
UINT Permutation(UINT in);
UINT S_Box_Transfer(BYTE* in);
void EP(UINT Right32, BYTE* out);
UINT f(UINT Right32, BYTE* rKey);
void BtoW(BYTE *Plain64, UINT *Left32, UINT *Right32);
void IIP(BYTE *in, BYTE *out);
void IP(BYTE *in, BYTE *out);
void DES_Decryption(BYTE *c_text, BYTE *result, BYTE *key);
void DES_Encryption(BYTE *p_text, BYTE *result, BYTE *key);
void DES_Encryption(BYTE *p_text, BYTE *result, BYTE *key) {
        int i;
        BYTE data[BLOCK_SIZE] = { 0, };
        BYTE round_key[16][6] = { 0, };
```

```
	UINT L = 0, R = 0;
	/* Round Key 생성 */
	Key_Expansion(key, round_key);
	for (int j = 0; j < 16; j++) {
		bitset<6> x((intptr_t)round_key[j]);
		// cout << "ROUDN_KEY[" << j << "] :: \t" << x << endl;
	}
	/* 초기 순열 */
	IP(p_text, data);
	/* 64bit 블록을 32bit로 나눔 */
	BtoW(data, &L, &R);
	/* DES Round 1~16 */
	for (i = 0; i<DES_ROUND; i++) {
		L = L ^ f(R, round_key[i]);
		/* 마지막 라운드는 Swap을 하지 않는다. */
		if (i != DES_ROUND - 1) {
			Swap(&L, &R);
		}
		bitset<32> x((int)L);
		bitset<32> y((int)R);
		// cout << "L[" << i << "]" << x << "\t" << "R[" << i << "]" << y <<
endl;
	}
	for (int i = 0; i < 8; i++)
		data[i] = 0;
	/* 32bit로 나누어진 블록을 다시 64bit 블록으로 변환 */
	WtoB(L, R, data);
	/* 역 초기 순열 */
	IIP(data, result);
	// cout << endl << endl << "암호화 결과" << endl;
	for (int j = 0; j < 8; j++) {
		bitset<8> x((int)result[j]);
		// cout << j << "\t :: " << x << endl;
	}
}
void DES_Decryption(BYTE *c_text, BYTE *result, BYTE *key) {
	int i;
	BYTE data[BLOCK_SIZE] = { 0, };
	BYTE round_key[16][6] = { 0, };
	UINT L = 0, R = 0;
	/* 라운드 키 생성 */
	Key_Expansion(key, round_key);
	for (int j = 0; j < 16; j++) {
		bitset<6> x((intptr_t)round_key[j]);
		// cout << "ROUDN_KEY[" << j << "] :: \t" << x << endl;
	}
	// cout << endl;
	/* 초기 순열 */
	IP(c_text, data);
	/* 64bit 블록을 32bit로 나눔 */
	BtoW(data, &L, &R);
	/* DES Round 1~16 */
	for (i = 0; i<DES_ROUND; i++) {
		/* 암호화와 비교해서 라운드키를 역순으로 적용 */
		L = L ^ f(R, round_key[DES_ROUND - i - 1]);
		/* 마지막 라운드는 Swap을 하지 않는다. */
		if (i != DES_ROUND - 1) {
			Swap(&L, &R);
		}
		bitset<32> x((int)L);
		bitset<32> y((int)R);
		// cout << "L[" << i << "]" << x << "\t" << "R[" << i << "]" << y <<
endl;
```

```
        }
        for (int i = 0; i < 8; i++)
                data[i] = 0;
        /* 32bit로 나누어진 블록을 다시 64bit 블록으로 변환 */
        WtoB(L, R, data);
        /* 역 초기 순열 */
        IIP(data, result);
        // cout << endl << endl << "복호화 결과" << endl;
        for (int j = 0; j < 8; j++) {
                bitset<8> x((int)result[j]);
                // cout << j << "\t :: " << x << endl;
        }
}
void IP(BYTE *in, BYTE *out) {
        int i;
        BYTE index, bit, mask = 0x80;
        for (i = 0; i<64; i++) {
                index = (ip[i] - 1) / 8;
                bit = (ip[i] - 1) % 8;
                if (in[index] & (mask >> bit)) {
                        out[i / 8] |= mask >> (i % 8);
                }
        }
}
void IIP(BYTE *in, BYTE *out) {
        int i;
        BYTE index, bit, mask = 0x80;
        for (i = 0; i<64; i++) {
                index = (iip[i] - 1) / 8;
                bit = (iip[i] - 1) % 8;
                if (in[index] & (mask >> bit)) {
                        out[i / 8] |= mask >> (i % 8);
                }
        }
}
void BtoW(BYTE *Plain64, UINT *Left32, UINT *Right32) {
        int i;
        for (i = 0; i<8; i++) {
                if (i<4)
                        *Left32 |= (UINT)Plain64[i] << (24 - (i * 8));
                else
                        *Right32 |= (UINT)Plain64[i] << (56 - (i * 8));
        }
}
UINT f(UINT Right32, BYTE* rKey) {
        int i;
        BYTE data[6] = { 0, }; /* EP에 의한 48 bit output 저장 */
        UINT out;
        /* 1. Expansion Permutation: EP-box */
        EP(Right32, data);
        for (i = 0; i < 6; i++) {
                /* 2. 48 bit XOR between data and rKey: S-box */
                data[i] = data[i] ^ rKey[i];
        }
        /* 3 & 4. Straight permutation of 32-bit S-box output */
        out = Permutation(S_Box_Transfer(data));
        return out;
}
void EP(UINT Right32, BYTE* out) {
        int i;
        UINT bit8_Mask = 0x80, bit32_Mask = 0x80000000;
        for (i = 0; i<48; i++) {
                /* EP 테이블이 나타내는 위치의 비트값을 & 연산과 시프트 연산을 이용하여 추출 */
                if (Right32 & (bit32_Mask >> (E[i] - 1))) {
```

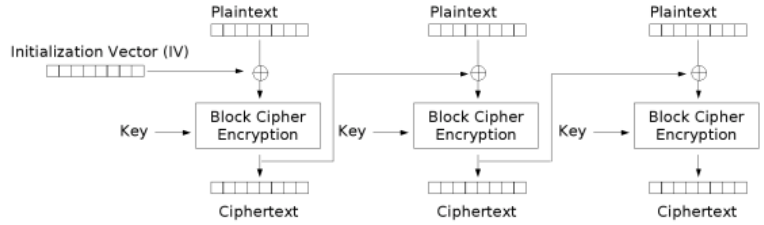
```
                                /* 추출한 값을 배열의 상위 비트부터 저장 */
                                out[i / 8] |= (BYTE)(bit8_Mask >> (i % 8));
                        }
                }
                return;
}
UINT S_Box_Transfer(BYTE* in) {
                int i, row, column, shift = 28;
                UINT temp = 0, result = 0, mask = 0x00000080;
                for (i = 0; i<48; i++) {
                        /* 입력값의 상위 비트부터 1 비트씩 차례로 추출하여 temp 에 저장 */
                        if (in[i / 8] & (BYTE)(mask >> (i % 8))) {
                                temp |= 0x20 >> (i % 6);
                        } else
                                ;
                        /* 추출한 비트가 6 비트가 되면 */
                        if ((i + 1) % 6 == 0) {
                                row = ((temp & 0x20) >> 4) + (temp & 0x01); /* 행의 값을 계산*/
                                column = (temp & 0x1E) >> 1; /* 열의 값을 계산*/

                         /* 4 비트의 결과 값을 result 에 상위 비트부터 4 비트씩 저장 */
                                result += ((UINT) s_box[i / 6][row][column] << shift);
                                shift -= 4;
                                temp = 0;
                        }
                }
                return result;
}
UINT Permutation(UINT in) {
                int i;
                UINT out = 0, mask = 0x80000000;
                for (i = 0; i < 32; i++) {
                        /* 순열 테이블이 나타내는 위치의 비트를 추출한 결과 값을 상위 비트부터 저장 */
                        if (in&(mask >> (P[i] - 1))) {
                                out |= (mask >> i);
                        } else
                                ; //do nothing
                }
                return out;
}
void Swap(UINT *x, UINT *y) {
                UINT temp;
                temp = *x;
                *x = *y;
                *y = temp;
}
void WtoB(UINT Left32, UINT Right32, BYTE *out) {
                int i;
                UINT mask = 0xff000000;
                for (i = 0; i<8; i++) {
                        if (i<4)
                                out[i] |= (Left32 & (mask >> (i * 8))) >> (24 - (i * 8));
                        else
                                out[i] |= (Right32 & (mask >> (i * 8))) >> (56 - (i * 8));
                }
}
void Key_Expansion(BYTE *key, BYTE round_key[16][6]) {
                int i;
                BYTE pc1_result[7] = { 0, };
                UINT c = 0, d = 0;
                /* 키를 순열 선택 1 테이블을 이용해서 재배치 */
                PC1(key, pc1_result);
                /* 56 비트의 데이터를 28 비트로 나누기 */
```

```
		makeBit28(&c, &d, pc1_result);
		/* 라운드키 생성 */
		for (i = 0; i < 16; i++) {
			c = Cir_Shift(c, i); /* 28 비트 데이터를 좌측으로 순환 이동 */
			d = Cir_Shift(d, i);
			PC2(c, d, round_key[i]); /* 순열 선택 2 테이블을 이용해서 재배치 */
		}
}
void PC1(BYTE *Key_In, BYTE *Key_Out) {
		int i, index, bit;
		UINT mask = 0x00000080;
		/* PC-1 이 나타내는 위치를 계산하여 입력값으로부터 해당 위치의 비트를 추출하고 결과값을
저장할 배열에 상위 비트부터 저장 */
		for (i = 0; i < 56; i++) {
			index = (PC_1[i] - 1) / 8;
			bit = (PC_1[i] - 1) % 8;
			if (Key_In[index] & (BYTE)(mask >> bit)) {
				// cout << "i / 8 :: " << i / 8 << "\t i % 8 :: " << i % 8 <<
endl;
				Key_Out[i / 8] |= (BYTE)(mask >> (i % 8));
			}
			//bitset<8> x((int)Key_Out[i % 8]);
			//cout << "Key_Out[ " << i % 8 << " ] ::: " << Key_Out[i % 8] << " ||| "
<< x << endl;
		}
}
void makeBit28(UINT *c, UINT *d, BYTE *Key_Out) {
		int i;
		BYTE mask = 0x80;
		for (i = 0; i < 56; i++) {
			if (i < 28) {
				if (Key_Out[i / 8] & (mask >> (i % 8))) {
					*c |= 0x08000000 >> i;
				} else
					; // do nothing
			} else {
				if (Key_Out[i / 8] & (mask >> (i % 8))) {
					*d |= 0x08000000 >> (i - 28);
				} else
					; // do nothing
			}
		}
}
UINT Cir_Shift(UINT n, int r) {
		int n_shift[16] = { 1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1 };
		if (n_shift[r] == 1) {
			n = (n << 1) + (n >> 27); /* 28bit 유효 자릿수에 기반한 circulation shift
*/
		} else {
			n = (n << 2) + (n >> 26);
		}
		n &= 0x0FFFFFFF;
		return n;
}
void PC2(UINT c, UINT d, BYTE *Key_Out) {
		int i;
		UINT mask = 0x08000000;
		/* PC-2 가 나타내는 위치를 계산하여 입력값으로부터 해당 위치의 비트를 추출하여 결과값을
저장할 배열에 상위 비트부터 저장 */
		for (i = 0; i<48; i++) {
			if (PC_2[i]<28) {
				if (c&(mask >> (PC_2[i] - 1))) {
					Key_Out[i / 8] |= 0x80 >> (i % 8);
				} else
```

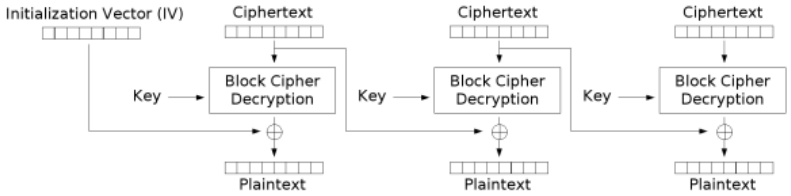
```
                                ; // do nothing
                        } else {
                                if (d&(mask >> (PC_2[i] - 1 - 28))) {
                                        Key_Out[i / 8] |= 0x80 >> (i % 8);
                                } else
                                        ; // do nothing
                        }
                }
}
```

[설명]

암호 블록 체인 방식(CBC)

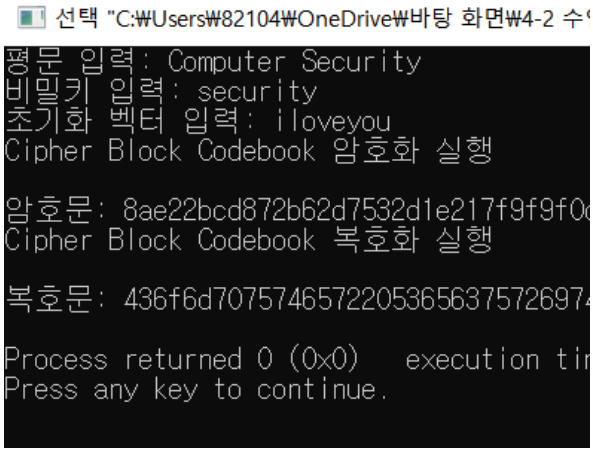


Cipher Block Chaining (CBC) mode encryption



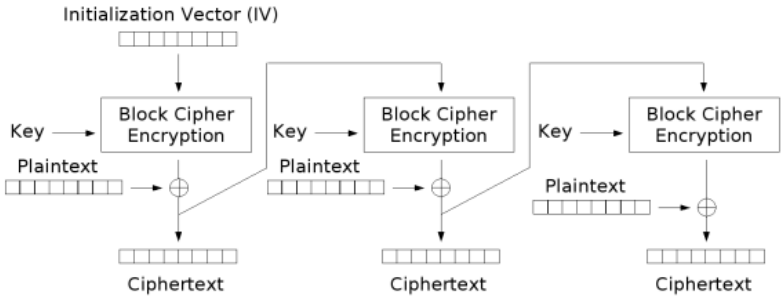
Cipher Block Chaining (CBC) mode decryption

: 각 블록이 암호화되기 전에 이전 블록의 암호화 결과와 XOR되며, 첫 블록의 경우에는 초기화 벡터가 사용된다. 초기화 벡터가 같은 경우 출력 결과가 항상 같기 때문에 매 암호화 마다 다른 초기화 벡터를 사용해야 한다. CBC는 암호화 입력 값이 이전 결과에 의존하기 때문에 병렬화가 불가능하지만, 복호화의 경우 각 블록을 복호화한 다음 이전 암호화 블록과 XOR하여 복구할 수 있기 때문에 병렬화가 가능하다.

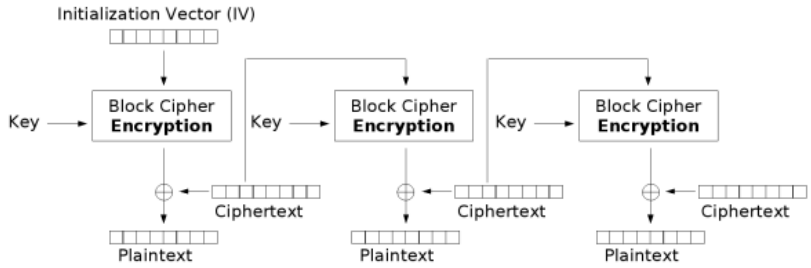


암호 피드백(CFB)

$$C_i = E_K(C_{i-1}) \oplus P_i$$
$$P_i = E_K(C_{i-1}) \oplus C_i$$
$$C_0 = \mathrm{IV}$$

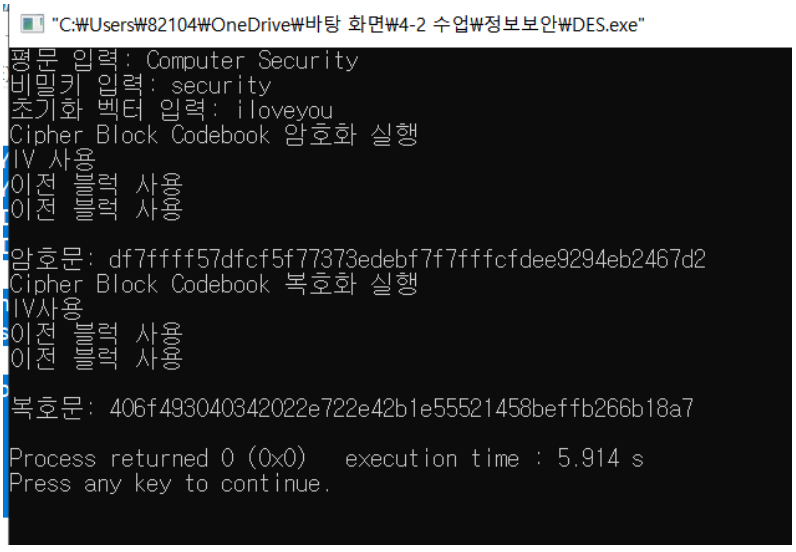


Cipher Feedback (CFB) mode encryption



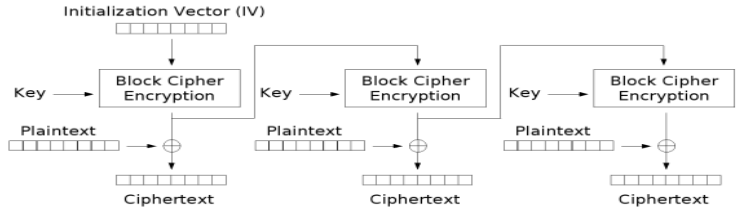
Cipher Feedback (CFB) mode decryption

블록 암호화 알고리즘에 따라 shift 연산을 사용하기도 하는데, 이런 방법을 shift되는 비트의 양에 따라 CFB-8혹은 CFB-1이라고 한다. 암호화, 복호화 연산의 각 Round마다 IV로부터 shift된 값을 사용한다.

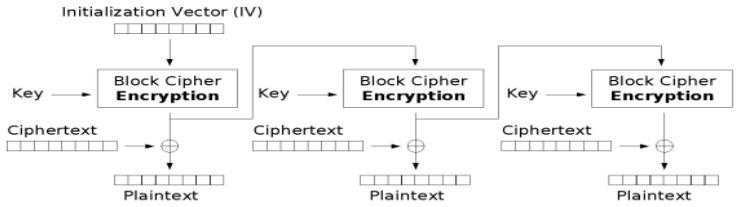


출력 피드백(OFB)

$$C_j = P_j \oplus O_j$$
$$P_j = C_j \oplus O_j$$
$$O_j = E_K(I_j)$$
$$I_j = O_{j-1}$$
$$I_0 = \mathrm{IV}$$



Output Feedback (OFB) mode encryption



Output Feedback (OFB) mode decryption

출력 피드백(output feedback, OFB)은 블록 암호를 동기식 스트림 암호로 변환한다.

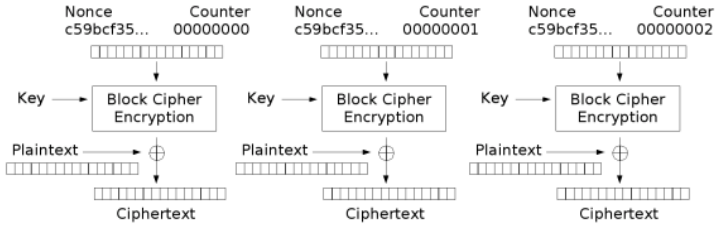
XOR 명령의 대칭 때문에 암호화와 암호 해제 방식은 완전히 동일하다:

```
"C:₩Users₩82104₩OneDrive₩바탕 화면₩4-2 수업₩정보보안₩DES.exe"
평문 입력: Computer Security
비밀키 입력: security
초기화 벡터 입력: iloveyou
Cipher Block Codebook 암호화 실행
IV사용
이전 블럭 사용
이전 블럭 사용

암호문: df7ffff57dfcf5f77457e5f3f7fbfdf7fb4eb27fb865b9e1
Cipher Block Codebook 복호화 실행
IV사용
이전 블럭 사용
이전 블럭 사용

복호문: df7ffff57dfcf5f77457e5f3f7fbfdf7fb4eb27fb865b9e1

Process returned 0 (0x0)   execution time : 5.965 s
Press any key to continue.
```

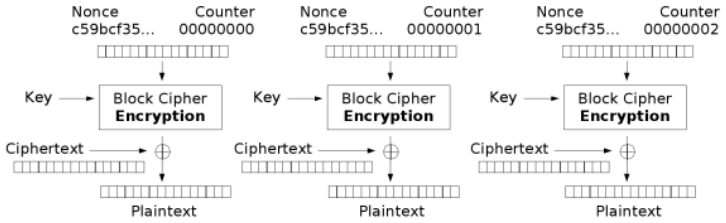
카운터(CTR)

카운터(Counter, CTR) 방식은 블록 암호를 스트림 암호로 바꾸는 구조를 가진다. 카운터 방식에서는 각 블록마다 현재 블록이 몇 번째인지 값을 얻어, 그 숫자와 nonce 를 결합하여 블록 암호의 입력으로 사용한다. 그렇게 각 블록 암호에서 연속적인 난수를 얻은 다음 암호화하려는 문자열과 XOR 한다.

카운터 모드는 각 블록의 암호화 및 복호화가 이전 블록에 의존하지 않으며, 따라서 병렬적으로 동작하는 것이 가능하다. 혹은 암호화된 문자열에서 원하는 부분만 복호화하는 것도 가능하다.



Counter (CTR) mode encryption



Counter (CTR) mode decryption

"C:₩Users₩82104₩OneDrive₩바탕 화면₩4-2 수업₩정보보안₩DES.exe"

```
평문 입력: Computer Security
비밀키 입력: security
ctr 입력: iloveyou
Cipher Block Codebook 암호화 실행

암호문: 6feffff7f7f4e5f724f365f7fd7ae9ffffcc8df2bd126bf
Cipher Block Codebook 복호화 실행

복호문: 6feffff7f7f4e5f724f365f7fd7ae9ffffcc8df2bd126bf

Process returned 0 (0x0)   execution time : 9.849 s
Press any key to continue.
```