데이터통신 HW#4 <CRC-x Programming>

마감 기한: 2020년 6월 11일 23:59
작성 언어: C

> **!!! 교수님, 조교님께 드리는 말씀 !!!** 아래 **목차**에서 원하는 부분을 **클릭**하면 해당 부분으로 **바로가기**할 수 있고, **Ctrl+Home** 키로 맨 앞 **목차로 바로 돌아올 수** 있습니다. 코드 설명과 실행 사진을 넣다 보니 내용에 비해 스크롤이 길어졌는데, 이 기능으로라도 좀더 편하게 보셨으면 좋겠습니다.

작성자 정보

- 과목명: 데이터통신(002분반)
- 학과: 전자정보통신공학과
- 학번: 18010697

- 이름: 김해리

수행 목적

CRC-8, 16, 32 에러 검출을 손수 구현하여 원리를 학습하고, 반복 수행을 통해 성능을 체감하도록 한다.

코드 설명

전처리기를 이용하여 기본형과 확장형을 한 소스파일에서 관라하도록 만들었습니다.

```
/* 기본형과 확장형을 한 소스 코드에 작성하기 위한 전처리*/
#define EXTEND
// #undef EXTEND        // (확장형) 이용시 이 line 주석처리할 것.
```

문제에서 제시한 CRC 다항식은 이진수표현으로 바꾸어 전역변수로 선언해주었습니다.

```
/* CRC Code */
const uint8_t CRC_8[9] = { 1, 0,0,0,0, 0,1,1,1 };
const uint8_t CRC_16[17] = { 1, 0,0,0,1, 0,0,0,0, 0,0,1,0, 0,0,0,1 };
const uint8_t CRC_32[33] = { 1, 0,0,0,0, 0,1,0,0, 1,1,0,0, 0,0,0,1, 0,0,0,1,
1,1,0,1, 1,0,1,1, 0,1,1,1 };
```

자료형

Data와 Message는 0 또는 1을 담는 uint8_t 배열로 담기로 했습니다. 처음엔 하나의 변수에 하나의 데이터를 다 담아 비트 단위 연산자를 이용해 과제를 수행할 생각이었으나, 제시문에 의하면 최대 72bit(=9Byte)까지의 수를 저장해야 했는데 아무리 큰 정수 자료형이라도 최대 64Bit까지만 다룰 수 있었기 때문에, 하드웨어스러운 구현을 하기보다는 시뮬레이션 기능에 집중하기로 하였습니다.

Data의 길이를 유동적으로 정하여 다뤄야 하기 때문에 데이터 배열은 **동적 할당**을 했고, 하드웨어적 구현과는 거리가 있지만 연산의 편의를 위해 **구조체를 따로 만들어** 데이터의 비트 수와 Hex 표현을 갖고 다니도록 하였습니다.

```
typedef struct BinaryData
{
    int length;                 // # of bits
    uint8_t* data;              // data(uint8_t) array header
    char hex[15];               // HEX expression of data.
maximum(string_length)=length(AA:BB:CC:DD:EE)= 15
} BD;
```

원본 메시지와 CRC를 덧붙인 전송 메시지를 *BinaryData 형*(이하 *BD 형*이라고 부름)으로 선언할 것입니다

- BinaryData형의 초기화

```
void initBD(BD* B, int length)
{
    B->length = length;
    B->data = (uint8_t*)malloc(sizeof(uint8_t) * (B->length));
}
```

BD형의 pointer와 전체 길이를 받아 동적 할당을 해주는 함수입니다.

Main 함수

변수 초기화

```
    BD Data, Message;
    int size_of_data = 4, CRC_type = 8;
    int num_of_errorbit;
```

이전에 만들어준 구조체, BinaryData형으로 전송하고자 하는 원본 데이터(*Data*)와 CRC로 생성한 코드를 덧붙인 데이터(*Message*)를 선언해줍니다. *size_of_data*는 원본 데이터(*Data*)에 들어갈 데이터의 Byte 수입니다. 기본형에서는 4Byte 데이터를 대상으로 CRC를 수행하므로 4로 초기화 해주었습니다. *CRC_type*은 CRC-x에서 x에 들어갈 값을 담습니다(8, 16, 32). *num_of_errorbit*은 실제로 flip된 비트를 기록하는 변수입니다.

데이터 입력 받기

```
// [기본형 1. 숫자 입력받기] & [확장형 1. 숫자 생성하기]
#ifdef EXTEND       // [확장형]

    printf("[SYSTEM] 전송할 Data의 크기(Byte 수)를 입력하세요 : ");
    scanf("%d", &size_of_data);
    printf("\n");
    createRandomData(size_of_data, &Data); // 랜덤으로 데이터 생성
    printf("[SYSTEM] Data가 랜덤으로 생성되었습니다.\n");
#endif
#ifndef EXTEND      // [기본형]
    printf("[SYSTEM] 전송할 Data를 입력하세요(Hex) : ");
    scanf("%s", (Data.hex)); printf("\n");
    getBinaryfromHex(&Data);
#endif

    printf("[SYSTEM] 전송할 Data는 \n"); // 출력부
    printf("         length : %d\n", Data.length);
    printf("         (HEX) %s\n", Data.hex);
    printf("         (BINARY) "); printBinary(&Data); printf("\n\n");
```

데이터를 받는 부분은 기본형과 확장형에서 다르게 컴파일 되도록 전처리를 했습니다. 입력을 받거나 랜덤으로 생성한 데이터의 정보(bit수, 16진수 표현, 이진수 표현) 마지막에 출력합니다.

- 기본형
    1. 4Byte 데이터를 16진수 형태(AA:BB:CC:DD)로 입력받아 문자열로 저장합니다.
    2. 문자열로 나타낸 데이터를 *getBinaryfromHex()* 함수를 통해 uint8_t 배열에 binary 형으로 저장합니다.
- 확장형
    1. 전송할 원본 데이터의 크기(Byte 수)를 입력 받습니다.

2. *createRandomData()* 함수에 크기와 Data의 주소를 전달해, 랜덤한 16진수 숫자열을 생성합니다.

나머지 연산으로 데이터 전송열 생성하기

```
    // [기본형 2. 나머지(Remainder)와 데이터 전송열(Message)]

#ifdef EXTEND       // [확장형 - CRC 선택하기]
    printf("[SYSTEM] 적용할 CRC-x의 종류를 입력하세요. (8, 16, 32 중 입력): ");
    scanf("%d", &CRC_type);
#endif
    // 데이터 전송열 선언 & 초기화하기
    initBD(&Message, Data.length + CRC_type);
    for (int i = 0; i < Data.length; i++)       Message.data[i] = Data.data[i];
    for (int i = Data.length; i < Message.length; i++) Message.data[i] = 0;

    // 데이터 전송열을 CRC code로 나누기
    getRemainder(&Message, CRC_type); // Message를 shift & XOR
    for (int i = 0; i < Data.length; i++)   Message.data[i] = Data.data[i]; //
나누기 작업 하느라 훼손된 Data 부분 복구

    printf("[SYSTEM] 송신 메시지:  "); printBinary(&Message); printf("\n\n");

    printf("********* 전송 이후 ********* \n\n");
```

- 공통
    1. 데이터 전송열의 크기(원본 데이터 길이+CRC-x의 차수)를 참고하여 배열을 선언해줍니다. 앞부터 원본 데이터와 동일한 정보로 채워 주고, 뒤는 0으로 채웁니다.
    2. *getRemainder()* 함수에 전송열을 보내 나눗셈 연산을 처리합니다.
    3. 나머지 앞 부분을 다시 원본 데이터로 채워 줍니다.
- 확장형
    - CRC-x의 x를 콘솔 입력으로 받습니다.
- 기본형
    - *CRC_type*은 8로 초기화 되어 있습니다.

노이즈 생성하기

```
    // [기본형 3. & 확장형 3.  Noise 만들기]
    num_of_errorbit = makeNoiseOnData(&Message);
    printf("[SYSTEM] 수신 메시지:  "); printBinary(&Message); printf("\n\n");
```

데이터 전송열을 *makeNoiseOnData()*로 넘깁니다. 에러 비트 수를 이용자에게서 입력받는 것은 함수 내부에서 진행합니다.

CRC 기능 검증하기

```
    // [기본형 4. CRC 기능 검사하기]
    getRemainder(&Message, CRC_type);
    printf("[SYSTEM] CRC로 나눈 나머지:  "); printBinary(&Message);
printf("\n\n");

    if(!checkError(&Message, CRC_type))
    {
        printf("[SYSTEM] 전송 중 오류가 없는 것으로 판단됩니다.\n");
```

```
        if (num_of_errorbit == 0)
        {
            printf("[SYSTEM] CRC-x check를 성공적으로 수행했습니다.\n");
        }
        else
        {
            printf("[SYSTEM] CRC-x check에 실패했습니다.\n");

        }
    }
    else
    {
        printf("[SYSTEM] 오류를 감지했습니다.\n");
        if (num_of_errorbit != 0)
        {
            printf("[SYSTEM] CRC-x check를 성공적으로 수행했습니다.\n");
        }
        else
        {
            printf("[SYSTEM] CRC-x check에 실패했습니다.\n");

        }
    }
```

getRemainder() 함수에 메시지 전송열을 넣어 CRC 다항식으로 나눈 나머지를 구합니다.

checkError() 함수에는 uint8_t 배열 중 1이 존재한다면 1을 반환하고, 아니라면 0을 반환합니다.

사용자 정의 함수

void initBD(BD*, int length)

```
void initBD(BD* B, int length)
{
    B->length = length;
    B->data = (uint8_t*)malloc(sizeof(uint8_t) * (B->length));
}
```

- intput: BD 포인터와 동적할당 받을 비트 수
- 기능: BD형 변수의 uint8_t 포인터에 입력받은 *length*만큼 메모리를 할당합니다.

void getBinaryfromHex(BD*)

```
void getBinaryfromHex(BD* B)
{
    int i=0, cnt = 0, size;
    int index=0;

    while (B->hex[index] != '\0')
    {
        if (B->hex[index] == ':') cnt++;
        index++;
    }
    size = cnt + 1;
```

```
    initBD(B, size * 8);

    index = 0;
    while (B->hex[i] != '\0')
    {
        // 0~f를 0000~1111까지 매칭해서 B->data에 4bit씩 채워 넣는 부분.
        // 그냥 scanf에서 char로 받지 말고 Hex 데이터로서 읽도록 구현했으면 나았겠다는
아쉬움이 있음.
        switch (B->hex[i])
        {/* 생략. 0~9, A~F의 문자를 매칭해 B->data를 4개씩 채움. */}

        i++;
    }
}
```

- input: BD형 포인터
- 기능: BD의 *hex* 에 저장된 데이터의 문자열을 참고하여 data 배열의 값을 채웁니다.

void printBinary(BD*)

```
void printBinary(BD* B)
{
    int i = 0;
    for (i = 0; i < (B->length); i++)
    {
        if ((i % 8 == 0) && (i != 0)) printf("_");
        printf("%u", B->data[i]);
    }
}
```

- input: BD형 포인터
- 기능: BD에 있는 data값을 출력합니다. 보기 좋도록 8자리마다 _로 구분합니다.

void getRemainder(BD* Message,int CRC_type)

```
void getRemainder(BD* Message, int CRC_type){
    // Message의 끝 부분을 나머지로 고침
    int i;
    int data_length = Message->length - CRC_type;

    if (CRC_type == 8)
    {
        for (i = 0; i < data_length; i++)
        {
            if (Message->data[i] == 1)
            {
                // shift and XOR
                for (int j = 0; j <= 8; j++)
                {
                    Message->data[i + j] = Message->data[i + j] ^ CRC_8[j];
                }
            }
        }
    }
    else if (CRC_type == 16)
    {
```

```
        for (i = 0; i < data_length; i++)
        {
            if (Message->data[i] == 1)
            {
                // shift and XOR
                for (int j = 0; j <= 16; j++)
                {
                    Message->data[i + j] = Message->data[i + j] ^ CRC_16[j];
                }
            }
        }
    }
    else if (CRC_type == 32)
    {
        for (i = 0; i < (Message->length - CRC_type); i++)
        {
            if (Message->data[i] == 1)
            {
                // shift and XOR
                for (int j = 0; j <= 32; j++)
                {
                    Message->data[i + j] = Message->data[i + j] ^ CRC_32[j];
                }
            }
        }
    }
    else printf("<ERROR> getRemainder에 잘못된 CRC_type 입력됨\n");

    // Remainder 부분만 출력
    printf("[SYSTEM] Data를 CRC Code로 나눈 나머지는 ");
    for (i = data_length; i < Message->length; i++) printf("%u", Message-
>data[i]);
    printf("\n");
}
```

- intput: CRC type과 연산 대상인 BD의 포인터
- 기능: CRC type에 따라 다른 다항식으로 나누기 연산을 합니다. 입력받은 BD의 데이터를 연산 결과로 덮어 씁니다.

int makeNoiseOnData(BD*)

```
int makeNoiseOnData(BD* Message)
{
    int num_of_error;
    int position_array[100];

    printf("오류 비트 수를 입력하세요. ");
    printf("(범위 = 0 ~ %2d)\n", Message->length);

#ifndef EXTEND // [기본형] console input
    printf("(99 입력시 burst error) : ");
    scanf("%d", &num_of_error);
    printf("\n");
    if(num_of_error == 99)
    {
        printf("burst가 시작되는 지점과 오류 비트 수를 입력하세요 :");
        int start, error_seq;
```

```
        scanf("%d%d", &start, &error_seq);
        for(int i=0; i<error_seq; i++)
        {
            Message->data[(start+i)%(Message->length)] = !Message->data[(start +
i) % (Message->length)];
        }
        return error_seq;
    }
    else
    {
        printf("%d 개의 오류 위치를 각각 입력하세요: ", num_of_error);
        for(int i=0; i < num_of_error; i++)
        {
            // 큰 수를 넣었을 시에 ERROR 메세지 띄우는 안전장치를 해야 하나..?
            int position;
            scanf("%d", &position);
            Message->data[position] = ! Message->data[position];
        }
        return num_of_error;
    }
#endif
#ifdef EXTEND       // [확장형 - 랜덤 생성]
    scanf("%d", &num_of_error); printf("\n");

    printf("flip이 일어난 자리: ");
    for(int i=0; i<num_of_error; i++)
    {
        int position = rand() % (Message->length);
        Message->data[position] = ! Message->data[position];
        if (i % 5 == 0) printf("\n");
        printf("<%2d> %d ", i+1, position);
        // 문제점 - 중복해서 같은 자리가 나올 가능성 있음. 이걸 검출을 시켜야 할지, 중복
없이 해야 하는 건지 아닌지 고민 됨...
    }
    printf("\n\n");
    return num_of_error;
#endif
}
```

- input: 데이터전송열 BD의 포인터
- 기능: 콘솔로부터 noise에 대한 정보를 입력 받고 데이터를 변형 시킵니다.

int checkError(BD* Message, int CRC_type)

```
int  checkError(BD* Message, int CRC_type)
{
    for (int i = (Message->length - 1); i >= (Message->length - CRC_type); i--)
    {
        if(Message->data[i] == 1) return 1;
    }
    return 0;
}
```

- input: 데이터 전송열 BD의 포인터와 CRC-x의 종류
- 기능: 메시지에 1이 있다면 즉시 1을 반환하고, 나머지 부분을 다 검사해도 1이 없다면 0을 반환합니다.

void createRandomData(int size, BD*)

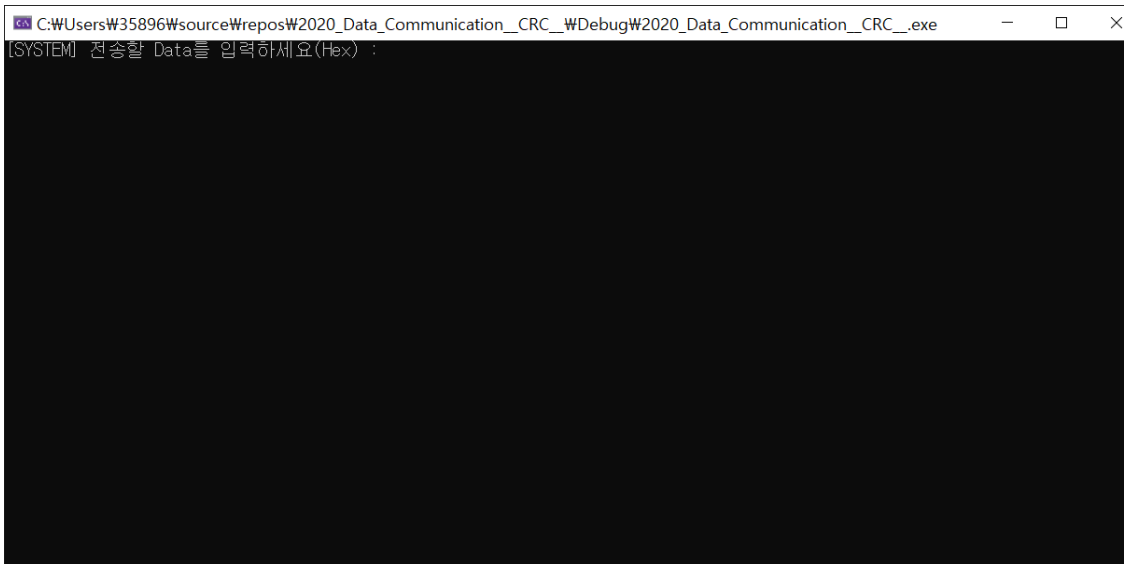
```
void createRandomData(int size, BD* B)
{
    int hexnum, i=0;
    B->length = size * 8;
    size *= 2;
    while(size>0)
    {
        if (i % 3 == 2)
        {
            B->hex[i++] = ':'; continue;
        }
        hexnum = rand() % 16;
        if (hexnum < 10) B->hex[i++] = '0' + hexnum;
        else B->hex[i++] = (hexnum - 10) + 'A';
        size--;
    }
    B->hex[i] = '\0';
    getBinaryfromHex(B);
}
```

- input: 생성하고자 하는 데이터의 바이트 수와 BD형 포인터
- 기능: 입력 받은 *size*에 맞는 랜덤한16진수 숫자열을 먼저 생성하고, 이를 바탕으로 *getBinaryfromHex()* 함수를 호출해 이진수 숫자열도 생성합니다.

기본형 결과

에러 입력 1 (몇 비트 에러를 줄지 정한 후 flip 시킬 자리 입력하는 방식)

1. 데이터 입력하기



2. 데이터 전송열 만들기

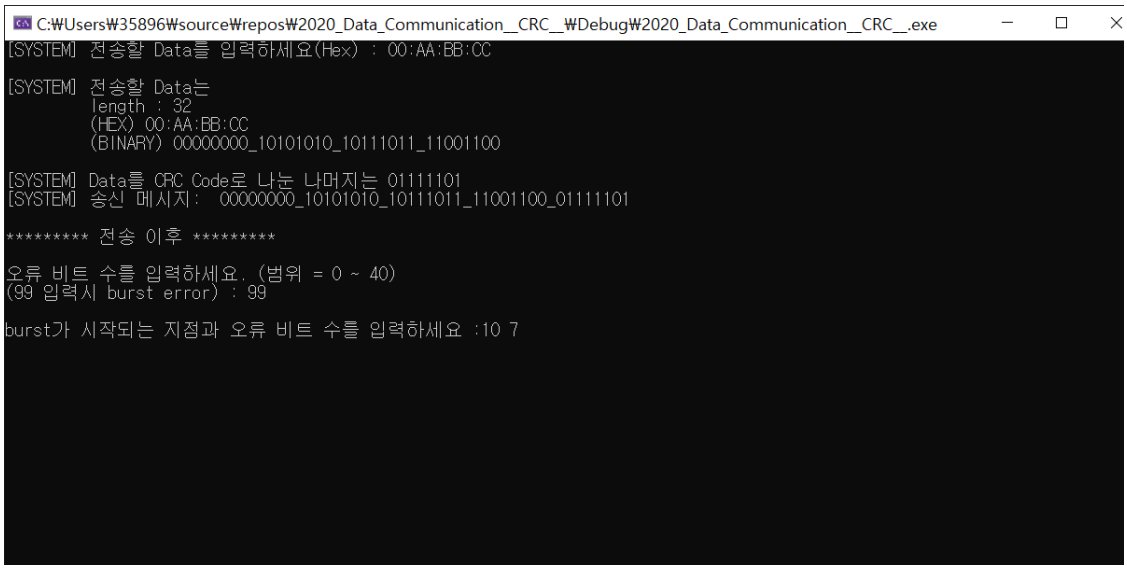
3. 임의의 noise 생성하기 & CRC 검출 효과 검증하기

노이즈를 발생시키자 나머지가 0이 아닌 수가 나왔고, 성공적으로 노이즈가 발생했음을 검출해냈습니다.

에러 입력 2 (burst error)

1. 데이터 입력 및 전송열 만들기

2. 발생시킬 noise 설정 입력(burst error로 입력하기 위해 99라고 써 주었음.)

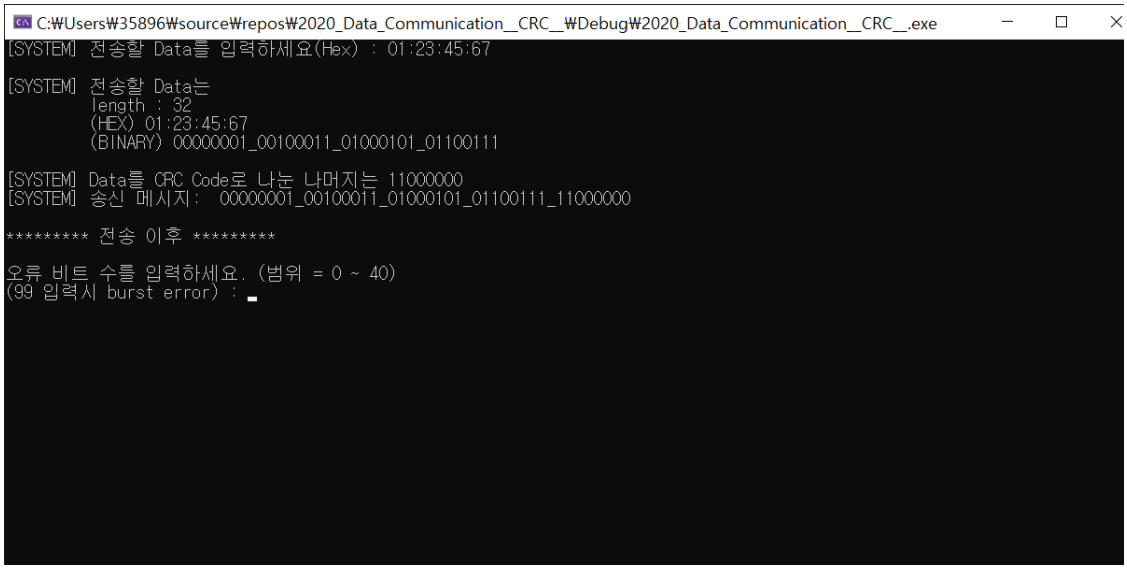


3. Error 검출

역시 나머지가 0이 아닌 수가 나왔고, burst error도 감지하는 것을 확인하였습니다.

에러가 0일 때

1. 데이터입력

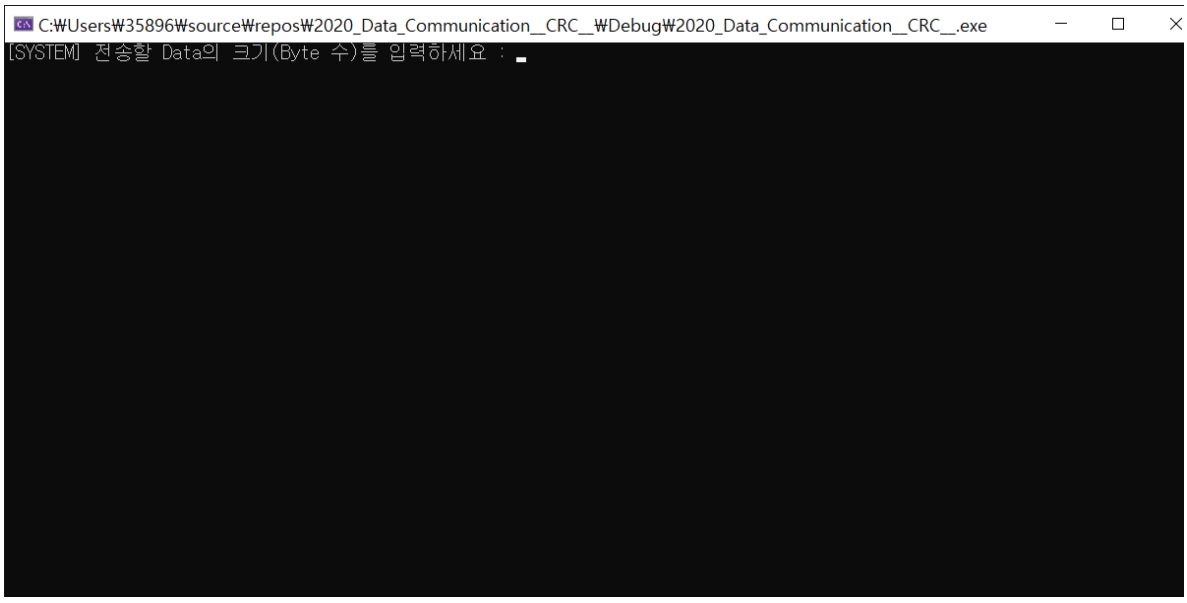


2. Error 검출

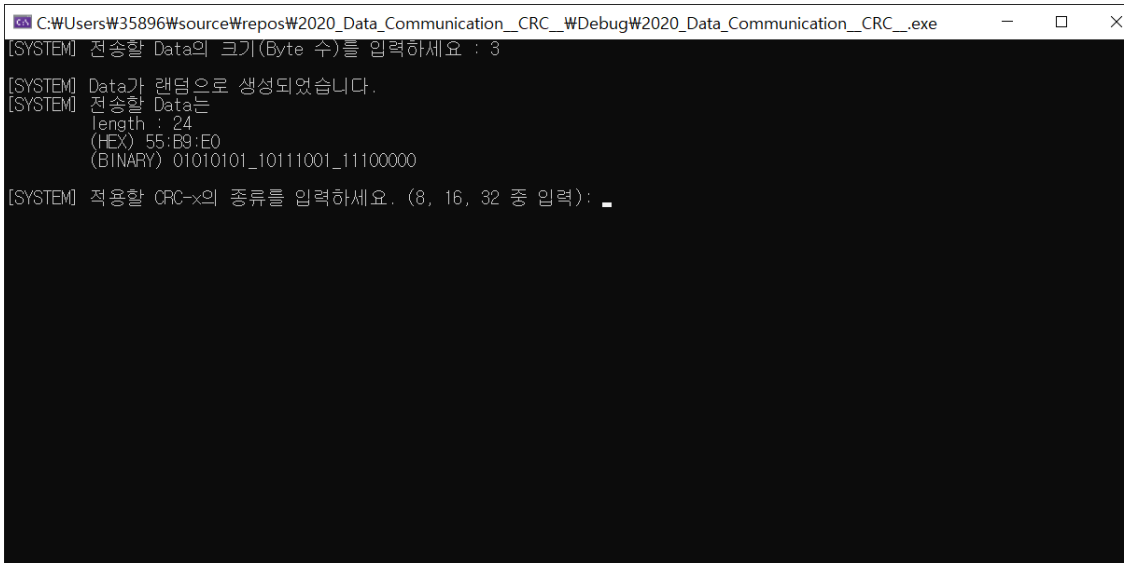
나머지가 0이 나오며, Error가 없을 시에는 Error가 없다고 판단하는 것을 확인했습니다.

확장형 결과

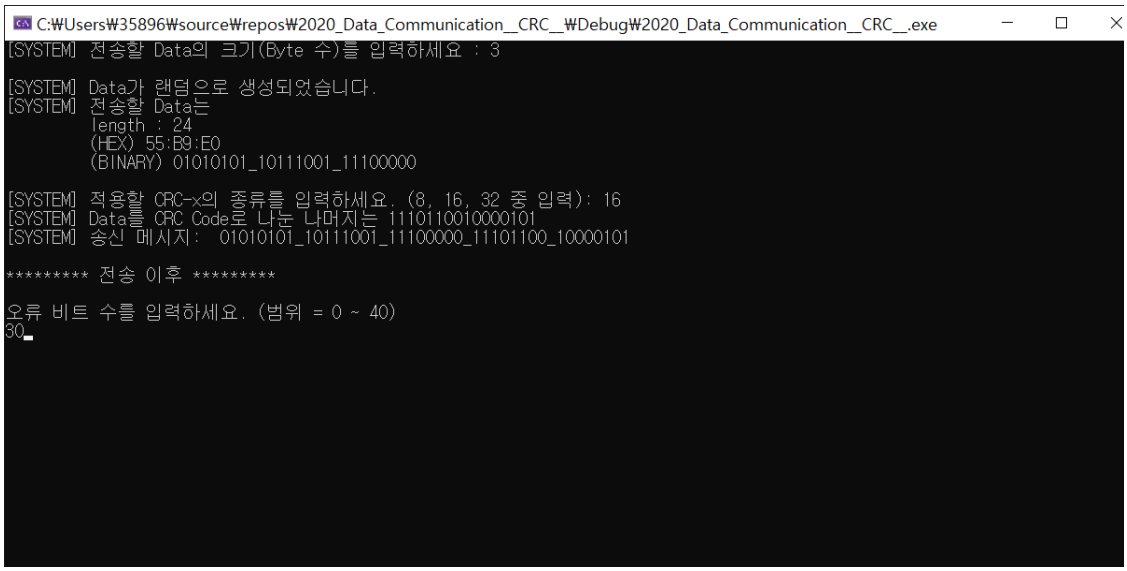
확장형 기능 소개



1. 데이터 크기를 입력 받아 랜덤한 데이터 생성



2. 사용할 CRC-x를 선택 후 메시지 전송열 생성



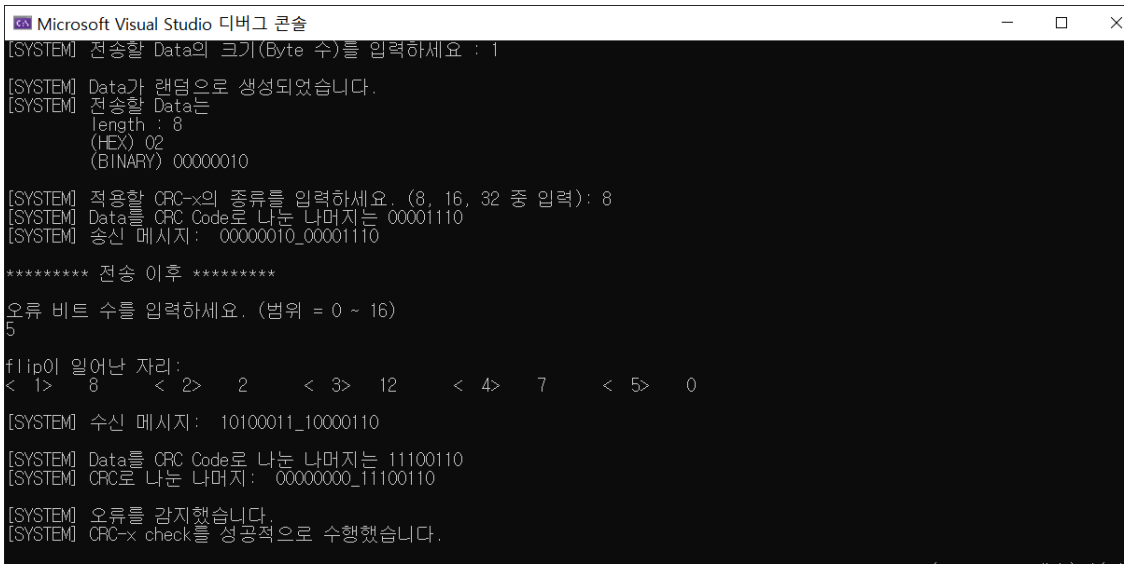
3. noise로 만들 flip bit의 개수를 입력 받아 에러 생성 & 검출

flip bit은 중복(한 번 뒤집힌 비트가 다시 뒤집힘)이 있을 수 있도록 했습니다. 중복을 피하도록 프로그램을 짤까 생각하기도 했지만, 과제 제시문의 추가확장형에서 error bit을 1000개 까지도 입력할 수 있게 한 점 때문에 그냥 완전한 랜덤으로 구현하였습니다.

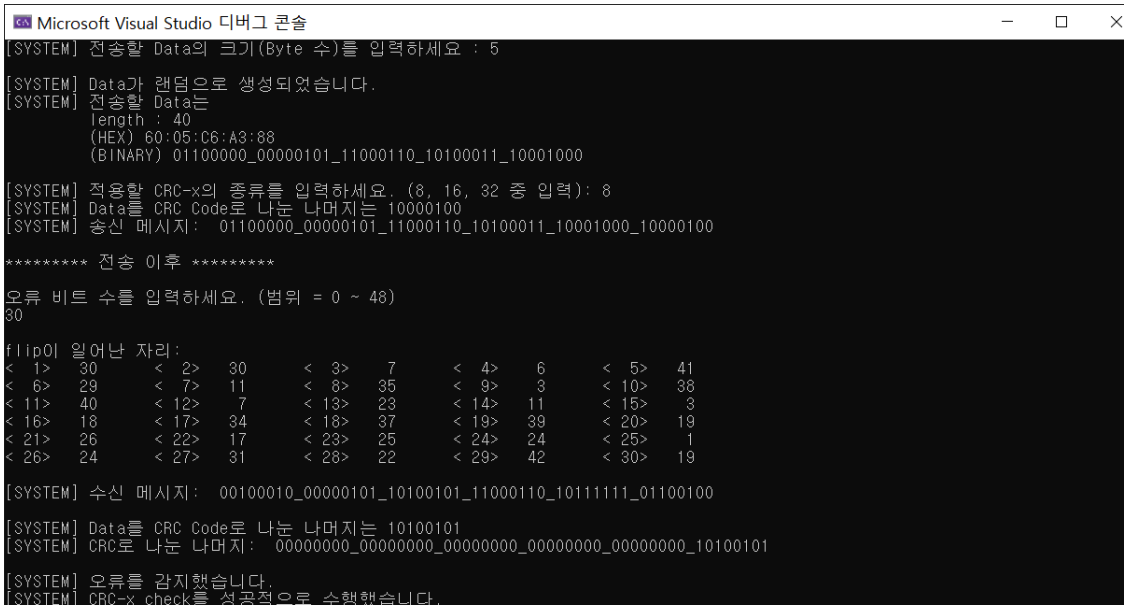
CRC-8

각 CRC를 실행한 부분은 대표적으로 1 Byte 데이터에 대해 실행한 것과 5 Byte 데이터에 대해 실행한 것을 보여드리도록 하겠습니다.

- 1 Byte

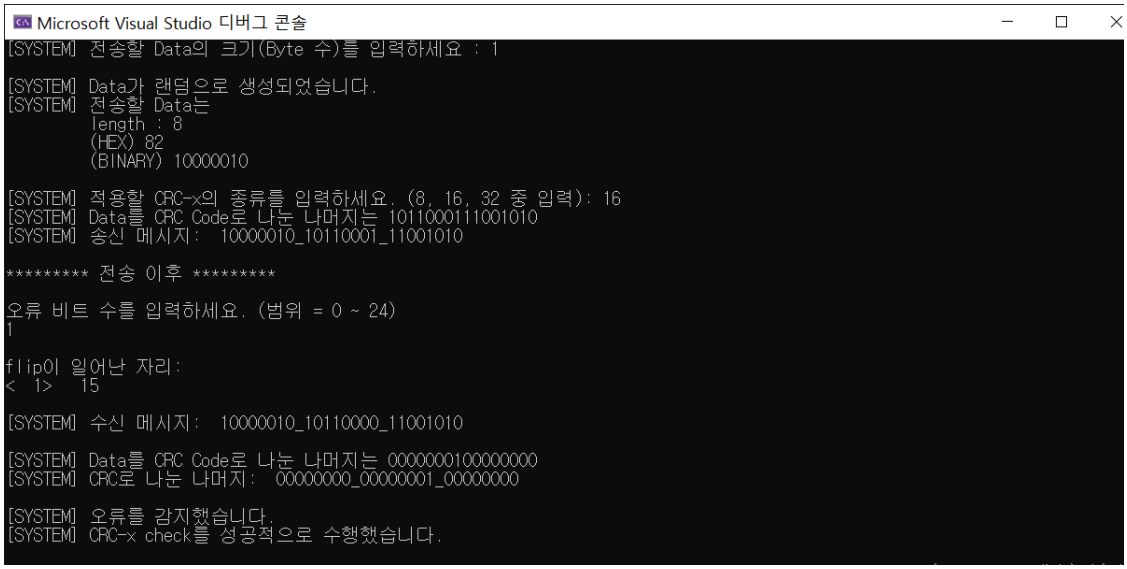


- 5 Byte



CRC-16

- 1Byte



- 5 Byte

CRC-32

- 1Byte

- 5 Byte

추가확장 결과

앞서 짰던 코드가 입출력 부분을 크게 구분하지 않고 마구잡이로 작성한 코드라서 동일한 소스 파일에 담는 것은 무리라고 생각했습니다. 그래서 추가 확장은 반복 수행, 파일입출력과 최소한의 모니터링만 할 수 있도록 핵심 알고리즘을 제외한 부분을 전부 고쳤습니다.

Main

```
int main() {
    BD Data, Message;
    int size_of_data = 4, CRC_type = 8;
    int num_of_error;
    FILE* fp = fopen("crc_simulate.txt", "w");
    // num of error, accuracy 순서로 출력할 것임.

    srand(time(NULL));  // 난수 시드 설정

    for (int i = 0; i <= 1000; i++) {
        int wrong_cnt = 0;
        fprintf(fp, "%d,", i);
        num_of_error = i;
        // [기본형 1. 숫자 입력받기] & [확장형 1. 숫자 생성하기]
        for (int j = 0; j < 1000; j++) {

            // size_of_data = rand() % 4 + 1;
            createRandomData(size_of_data, &Data); // 랜덤으로 데이터 생성

            // [기본형 2. 나머지(Remainder)와 데이터 전송열(Message)]

            // 데이터 전송열 선언 & 초기화하기
            initBD(&Message, Data.length + CRC_type);
            for (int i = 0; i < Data.length; i++)                Message.data[i]
= Data.data[i];
            for (int i = Data.length; i < Message.length; i++)  Message.data[i]
= 0;

            // 데이터 전송열을 CRC code로 나누기
            getRemainder(&Message, CRC_type); // Message를 shift & XOR
```

```
            for (int i = 0; i < Data.length; i++)   Message.data[i] =
Data.data[i]; // 나누기 작업 하느라 훼손된 Data 부분 복구

            // [기본형 3. & 확장형 3.  Noise 만들기]
            makeNoiseOnData(&Message, num_of_error);

            // [기본형 4. CRC 기능 검사하기]
            getRemainder(&Message, CRC_type);

            if (!checkError(&Message, CRC_type))
            {
                if (num_of_error != 0)
                {
                    wrong_cnt += 1;
                }
            }
            else
            {
                if (num_of_error == 0)
                {
                    wrong_cnt += 1;
                }
            }

            // memory deallocation
            free(Data.data);
            free(Message.data);
        }
        printf("error_bit = %d \twrong_cnt = %d\n", num_of_error, wrong_cnt);
        fprintf(fp, "%d\n", 1000 - wrong_cnt);
    }
    fclose(fp);
    return 0;
}
```

사용자 정의 함수는 printf를 지운 것 외에 큰 변화는 없습니다.

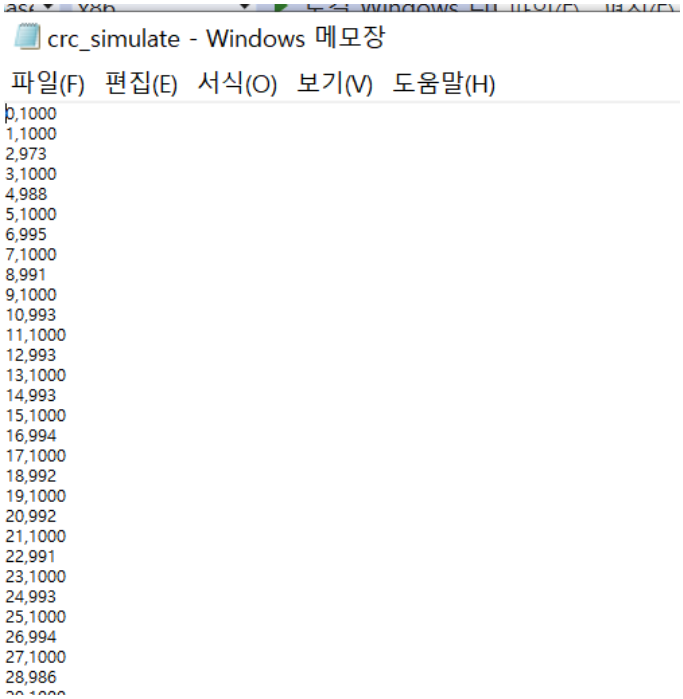
실험 조건

- Data의 크기는 4Byte로 고정한다.
  (1~5 Byte 사이 랜덤 값으로 한 번 실험해봤는데, 변인을 늘리는 것은 error bit에 따른 변화를 관찰하기에는 용이하지 않을 것 같아 방침을 수정했습니다.)
- Data의 내용은 랜덤 값으로 설정한다.
- 각 CRC-x에 대해 error bit 개수가 0개~1000개일 때를 실험한다. 하나의 error bit 실험에는 랜덤으로 생성한 4Byte 데이터 1000개를 표본으로 삼아 CRC code를 적용해본다.

실험 결과

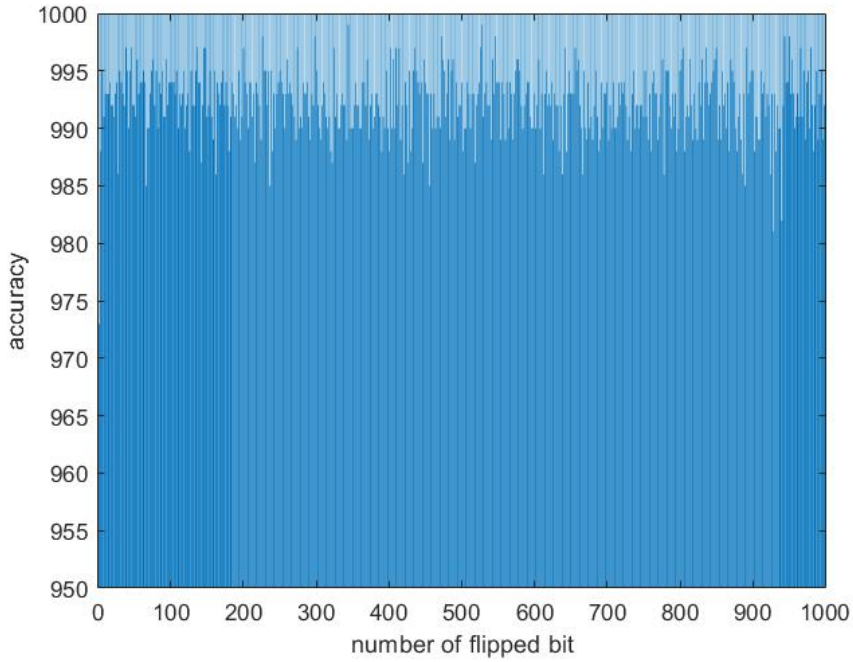
위 코드를 이용해 생성한 txt 파일을 MATLAB으로 옮겨 bar 그래프를 그렸습니다.

CRC-8

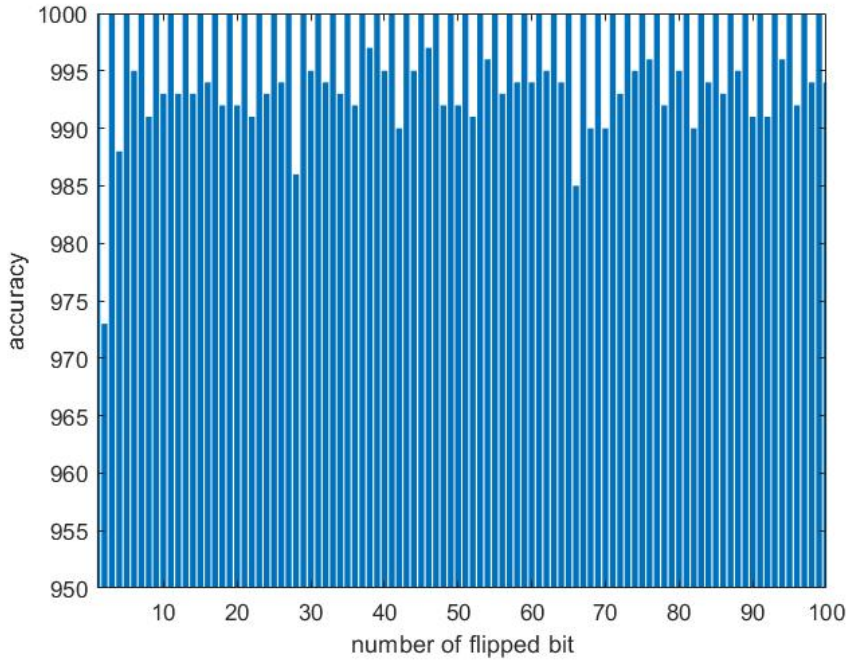


메모장 데이터는 이런 식으로 나왔습니다. **뒤집힌 비트 수가 홀수 개일 때는 100%의 정확성**을 보이는 것이 특이한 점입니다.

다음은 이 데이터를 이용해 그린 그래프입니다. x축은 error bit 수, y축은 CRC가 제대로 오류를 검출한 횟수를 의미합니다. 모든 값이 970 위쪽에 있어서 관측을 위해 y축의 범위는 950~1000으로 제한했습니다.



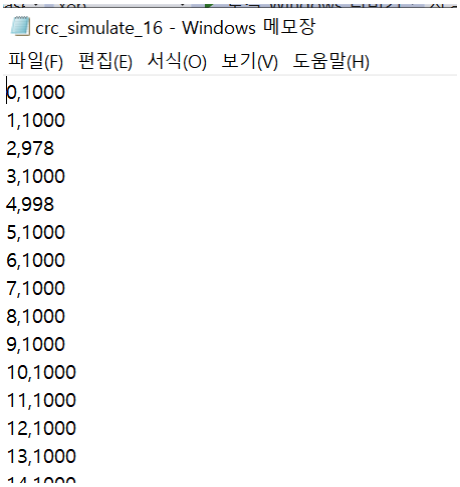
전체 평균은 996.1259입니다. 대략 **평균적으로는 99.612%의 정확도**를 갖는다는 결과입니다.



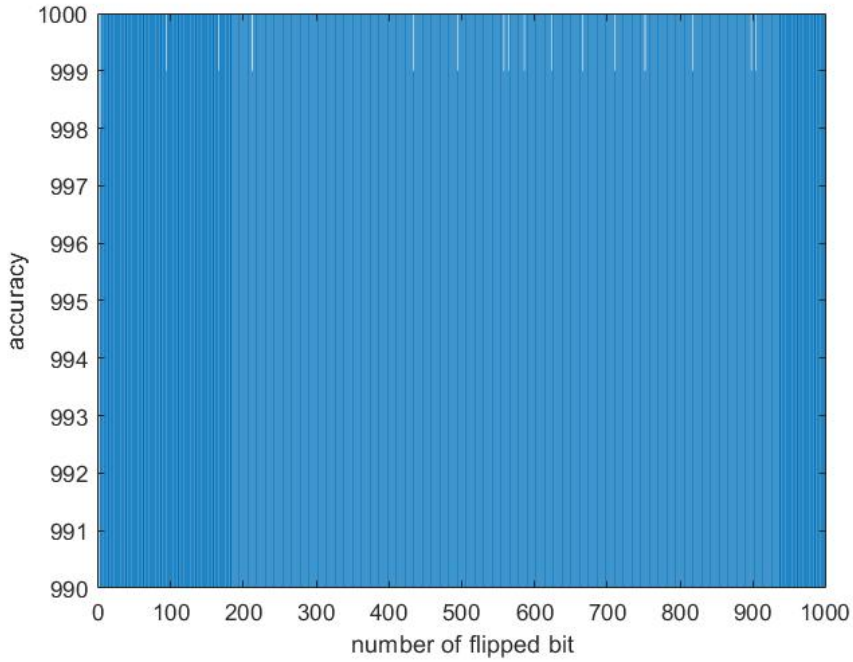
그래프를 x축 방향으로 확대해보기도 했지만, flipped bit의 개수와 정확도의 연관성은 홀수개일 때 100%라는 점 외에는 크게 느끼지 못 했습니다.

CRC-16

CRC-16의 결과입니다. 한 눈에 보아도 CRC-8보다 정확도가 높습니다.



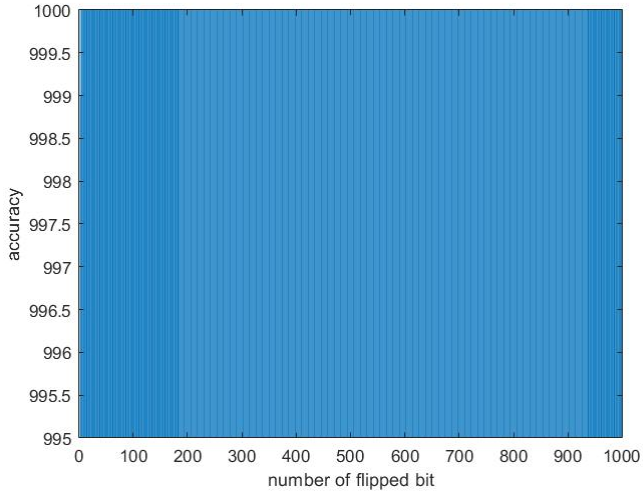
MATLAB으로 그림을 그리면 대략 이런 결과가 나옵니다.



틀리게 판정할 때가 있더라도 1개에서 그치는 것을 관찰할 수 있었습니다. **평균 정답률은 99.9961%**입니다.

CRC -32

CRC-32를 4바이트 데이터에 적용했을 땐 정확도가 100%가 아닌 경우를 찾기가 힘들었습니다.



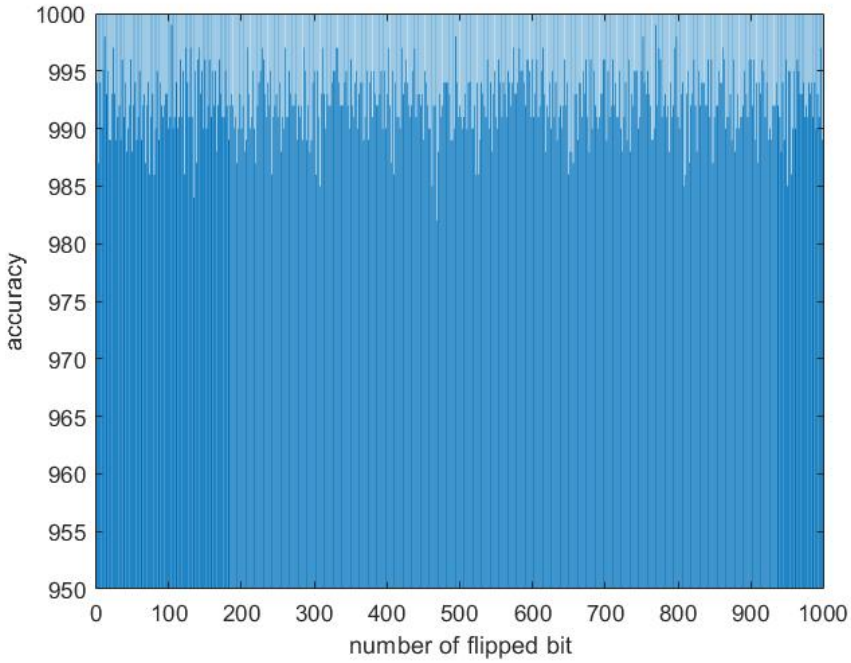

```
>> find(M(:, 2) ~= 1000)

ans =

     3
     5
```

매트랩의 find 기능을 이용해 정답률이 100%가 아닌 때를 찾으려 하니, error bit이 2개와 3개일 때만 한 번 씩 틀리고, 나머지는 정확히 맞췄다는 결과가 나왔습니다. **평균 정답률은 99.99840%** 입니다.

추가 실험 - Data의 Size가 얼마나 커야 CRC의 정확성이 떨어질까?

CRC의 정확성이 Data size에 의해서도 영향을 받을 것 같다는 생각에, 데이터 크기를 늘려서 한 번 실험해 보았습니다. CRC-16과 CRC-32는 아무래도 영향을 주려면 아주 큰 변화를 줘야 할 것 같아서 CRC-8에서만 **150Byte**의 데이터를 처리하도록 프로그램을 돌려 보았습니다.

```
/* Main */
int main() {
    BD Data, Message;
    int size_of_data = 150, CRC_type = 8;
    int num_of_error;
    FILE* fp = fopen("crc_simulate_sizetest.txt", "w");
    // num of error, accuracy 순서로 출력할 것임.
```



아까 4Byte로 실험해보았을 때와 크게 다르지 않은 결과가 나왔습니다. **평균 정확도는 99.61459%** 입니다. 300Byte로 늘려 실험해 보았을 땐 평균 정확도가 99.608% 가 되었습니다. 기대했던 결과는 나오지 않았습니다.