낸드플래시 구현 프로그램 계획서

과 목	소프트웨어공학
담 당 교 수	권 세 진
학 번	201720970
학 과	소프트웨어·미디어·산업공학부
이 름	권 대 한

목 차

1. 낸드 플래시 구현 프로그램이란?

Flash Memory는 HDD와 물리적인 구조가 완전히 다른 만큼 Flash Memory를 HDD 기반 소프트웨어에서 주 저장장치로 사용하기 위해서 Flash Memory에 적합한 데이터 액세스 방법이 필요하게 되었다. 기존의 HDD와 완전히 다른 물리적 구조, 데이터 저장 구조를 가짐에도, 기존 프로그램, HDD의 파일 시스템에서 Flash Memory는 어떻게 최적의 성능을 보여주는 것일까? 이는 Flash Memory를 관리하는 FTL 알고리즘, 컨트롤러로 인해 가능했다. 또 현재 Flash Memory의 상용화 이후 오랜 시간이 지났지만, 소프트웨어에서의 대응은 아주 미흡하다. 예를 들어 최신 버전의 운영체제인 Windows 10에서는 Trim(유휴 상태에서 필요 없는 블록을 지워주는 기술) 정도만 지원하며, SSD라고 인식을 하고 있음에도 불구하고, 유휴 상태에서 SSD에 Trim과 자동으로 디스크 조각 모음을 하도록 명령을 내려 SSD의 수명을 줄이고 있다. 심지어 어떤 운영체제는 인식조차 불가한 일도 있다.

NAND Flash의 셀당 저장 비트가 늘어날수록 수명이 수십, 수백 배 줄어들면서 데이터 액세스 시간, 속도 모두 상당히 떨어지게 되는데, 생산단가의 절약과 Flash Memory 진보를 위해서 Flash의 관리를 담당하는 FTL 알고리즘의 개선은 꾸준히 진행되어야 한다.
본 프로그램은 FTL 알고리즘을 제외한 NAND Flash의 저장공간, 읽기, 쓰기 과정을 시뮬레이션하면서 FTL 알고리즘, 그리고 스페어 공간(Over Provisioning)을 왜 할당해야 하는지 프로그램을 작성하면서 알아보겠다.

- 전체적인 프로그램 구성 계획

사용자에게 Text 기반의 명령을 입력받아, 각 기능이 작동하는 것이 프로그램의 요구점이므로, 콘솔의 입력 Stream에서 Token 단위로 명령 신호를 받아오면서 각 기능을 실행시킬 것이다.

또 이번 프로그램은 NAND Flash를 직접 수정하는 것이기에, 최소 데이터 읽기, 쓰기 단위인 Sector(=512 Byte)로 데이터를 구분할 것이며, 실제 데이터 인덱스인 PSN을 기반으로 데이터 액세스가 진행될 것이다. 그리고 Override(Update)가 발생한다면, 본래 FTL을 이용해 NAND Flash Erase를 최대한 방지할 것이지만, 이는 결국 비휘발성 메모리이므로, 언젠가는 데이터를 제거해야 할 상황이 올 것이다. 최소 지우기 단위인 Block을 이용한 함수까지 구현하면 추후 완벽한 NAND Flash 시뮬레이션이 가능하다고 생각이 들었다.

- 기능 구현에 대한 계획

(1) 프로그램 초기 실행 시

사용자에게 텍스트 입력과 처리하려고 하는 데이터를 동시에 입력받아서 각 기능을 실행해야 하기에, Ctrl + C 또는 exit 명령이 입력되기 전까지(개발자가 디버그를 마칠 때까지) 계속 명령을 받는 메인 기능 함수를 작성할 것이다. 그리고 메모리 해제 후 프로그램을 종료할 것이다.

(2) NAND Flash 구성 함수(Init)가 호출되었다면 //유휴 시간 Trim

XCode에서 가변길이배열을 허용하기에, 가변 길이 배열로 NAND Flash를 선언하려고 했다. 그러나 포인터 배열로 작성하기로 하였다. 과거 C99에서만 허용하는 내용이었기 때문이기도 하며, Visual Studio의 컴파일러는 이를 허용하지 않았기 때문이다.
그리고 포인터 배열을 사용한다면, 동적 메모리 할당, 해제를 꼭 해야 한다.

이 프로그램에서 수많은 STL Data Structure Type으로 Flash Memory를 구현하려고 하지 않는 이유도 포인터 배열을 사용하지 않으려는 이유와 유사하다. 일단 포인터 배열은 포인터 사용으로 인한 메모리 오버헤드가 발생하면서 사용하는 메모리 용량도 생각보다 커진다는 것으로 알고 있기에 임베디드 시스템에 부적합하다고 생각이 들었다. 마찬가지로 STL의 Data Structure Type 기반으로 작성한다면 메모리 오버헤드도 발생할 테지만, Link 되어있는 Lib들을 Flash Memory Controller에 담는다는 것은 더 말이 안 된다고 생각했다.

실제 프로그램 실행 후 구성 함수가 처음 호출되었을 때, 사용자가 입력한 크기를 기반으로, 데이터 공간을 할당하며, 만약 구성 함수가 두 번 이상 호출되었을 때, 사용자의 선택을 받아, 데이터가 존재하는지 확인하여 파일 스트림으로 백업을 진행하며, 기존의 저장소 공간 할당을 해제하고, NAND Flash를 재할당하면서 도출된 주솟값을 보관한다. 이후 사용자의 선택에 따라 데이터를 복원해주면 되겠다.

(3) NAND Flash 읽기 함수(Read)가 호출되었다면

Pointer 배열로 공간 할당이 되어있기에 Pointer를 통해 해당 Sector의 주솟값을 받아 모든 기능을 처리할 것이며. 데이터 읽기, 쓰기 최소 단위가 Sector이므로, 사용자에게 PSN(실제 섹터 넘버)를 입력받아 해당 인덱스의 데이터를 모두 출력시킬 것이다.
만약 NAND Flash의 크기를 넘는 데이터 액세스가 요청되었다면 이에 대한 예외 처리가 필요할 것이다.

(4) NAND Flash 쓰기 함수(Write)가 호출되었다면

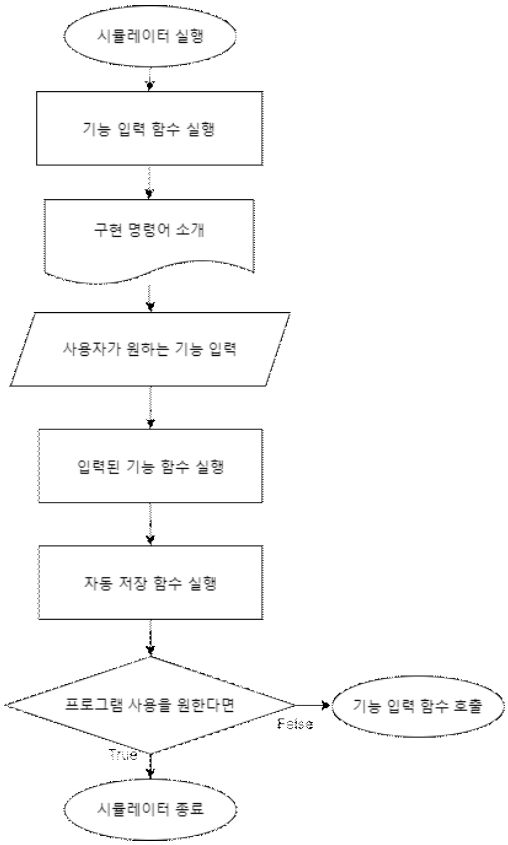
입력된 PSN을 기반으로 Sector 배열에 데이터를 입력시키고, 해당 PSN이 사용 중이라면, 이미 사용 중인 섹터라는 메시지와 함께, 한 번도 사용되지 않은 가장 빠른 번호의 PSN에 저장하고, 저장된 PSN, 미리 구현된 함수로부터 데이터를 출력해준다. 만약 Override를 원한다면 (E-B-W), 낸드 플래시만으로는 FTL의 꼼수를 사용해서 지워진 것처럼 보여줄 수가 없기에, 미리 구현된 NAND Flash 내 지우기 함수를 이용해 해당 섹터를 제외한 블록의 데이터를 RAM에 임시 저장하는 원시적인 방법으로 구현이 가능은 하겠지만, 이는 컴퓨터의 휘발성 메모리 RAM으로 구동했을 때 이며, 우리는 NAND Flash를 시뮬레이션하는 것이기에 해당 기능은 구현하지 않을 것이다.

(5) NAND Flash 지우기 함수(Erase)가 호출되었다면

NAND Flash의 지우기 연산을 실행 할 수 있는 가장 작은 단위가 블록 단위이기에, 시뮬레이터에서 지우기 함수가 호출된다면, 지울 블록 번호를 입력받아, 미리 구현한 지우기 함수(여기서 말하는 미리 구현한 함수는 재사용을 위해 범용적으로 만든 것을 말함)를 통해 단순히 해당 블록을 지우고 기능을 종료한다.

2. 논리 구성도

<시뮬레이터의 시작>

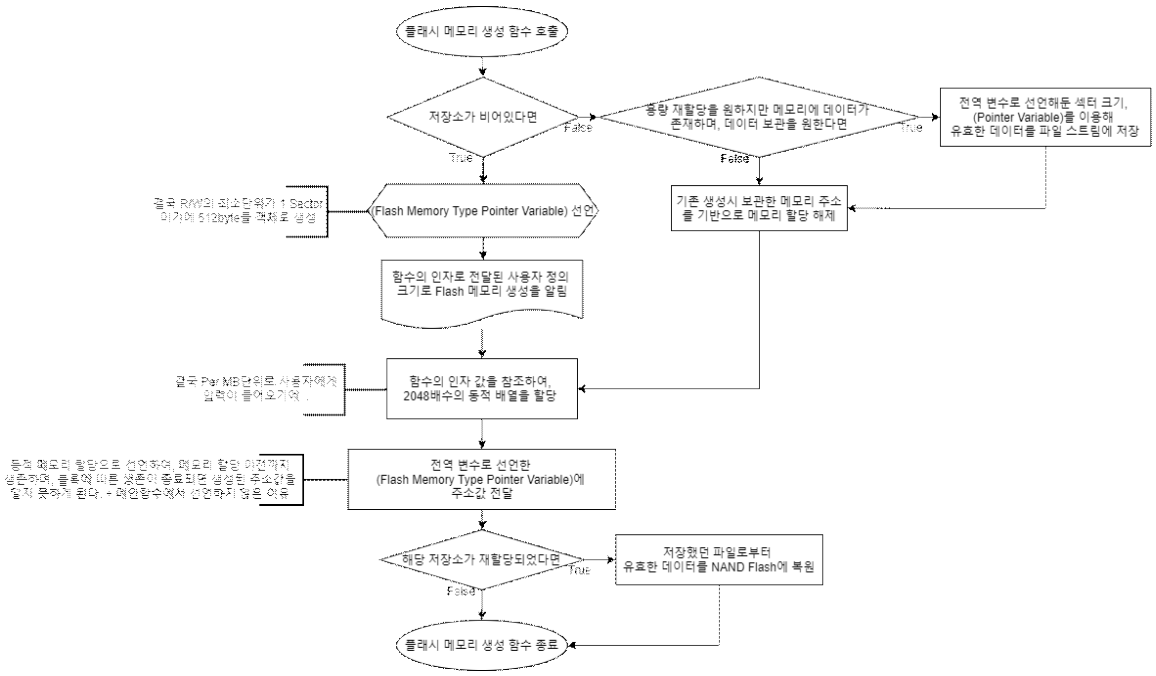


이번에 제작할 시뮬레이션 프로그램은 Flash Memory처럼 동작하는 프로그램을 만들어서 FTL 알고리즘을 이해하고 개선하는 것에 큰 의의가 생각이 들었다.

요구사항을 설계하면서 들었던 생각은 결국 데이터를 용량 맞춰 저장하면서 Flash Memory의 특성에 맞춰야 한다고 생각했으며, 결과적으로는 최대한 용량 낭비가 없게 저장하면서 추후 데이터를 읽고, 저장할 때도 효율적으로 사용할 수 있는 데이터 구조를 사용해야겠다고 생각했다. 그래서 내가 내린 결론은 STL의 Data Structure를 그대로 사용한다면, 메모리의 오버헤드가 발생할 것으로 생각되었기에, 기본적인 구조체 구조를 이용해 Flash Memory의 형태를 가지게 하며, 각 객체를 Sector로 구현하기로 하였다. 그리고 Struct 와 Class Type으로 구조체를 작성하였을 때 메모리를 차지하는 용량의 차이가 없었기에(XCode 기준), 이번 프로그램에서는 사용하지 않겠지만 Member Access Control이 가능한 Class를 이용한 배열 형태의 데이터 구조를 사용해 Flash Memory를 구현하도록 결정하였다. Flash Memory 특성상 액세스하고자 하는 데이터 인덱스를 알고 있으므로 연속된 메모리 주소로 저장하는 배열의 특성상 STL보다 빠른 액세스가 가능하며, 비교적 적은 메모리를 사용해 프로그램을 구현 할 수 있게 되었다.

프로그램 실행 시, Token 단위로 Stream의 입력을 구분해서, 각 기능의 실행을 구분하고, 실행 명령(데이터)을 가져옴으로, 각 기능에 인자를 전달해주는 함수에 실행 명령을 받아오며, 미리 구현된 각 기능 함수에 전달해줌으로 실질적인 기능을 실행한다.

<Flash Memory 생성 함수가 호출된다면>

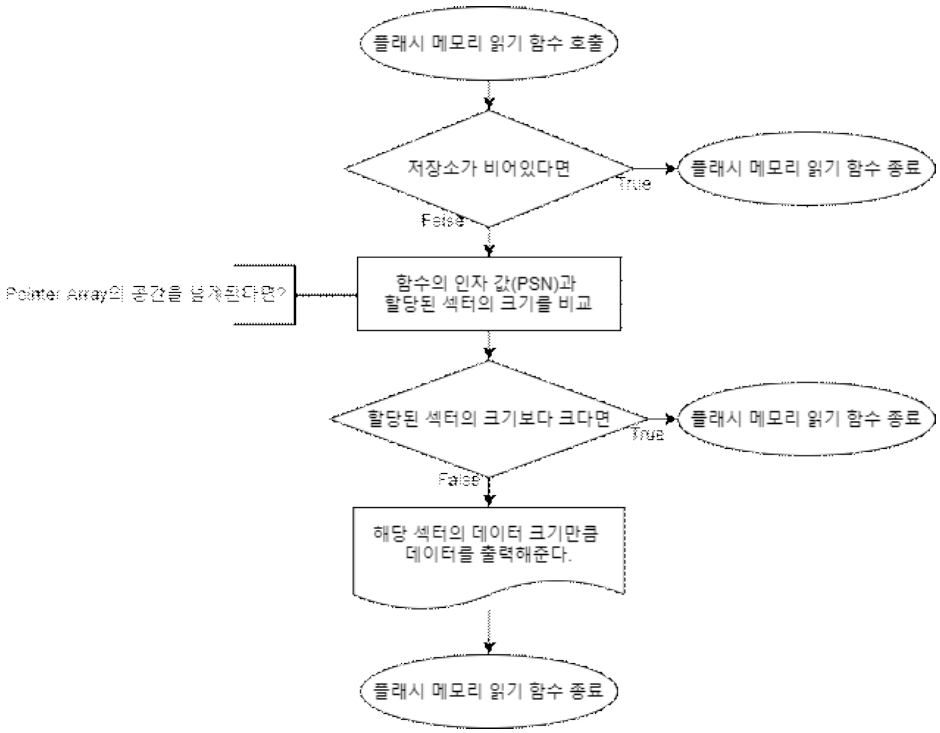


생성(할당) 함수가 실행되었을 때, 객체 배열을 할당하기 위해서, 포인터 배열을 선언하고, 사용자가 입력한 용량의 공간을 할당해줌으로 미리 구현한 생성 함수가 마무리된다.

프로그램을 설계하면서 이 부분이 중요하다고 생각했는데, 우리는 포인터 배열을 이용해 저장소 할당을 했기 때문에, 처음에는 인자로 포인터 배열의 주솟값을 받아오거나, Object의 포인터를 리턴하여 기능 구현을 하면 되겠다는 생각이 들었으며, 이는 Pointer Variable의 주솟값을 가져오는 방법으로 전체적인 기능을 구현하였다.

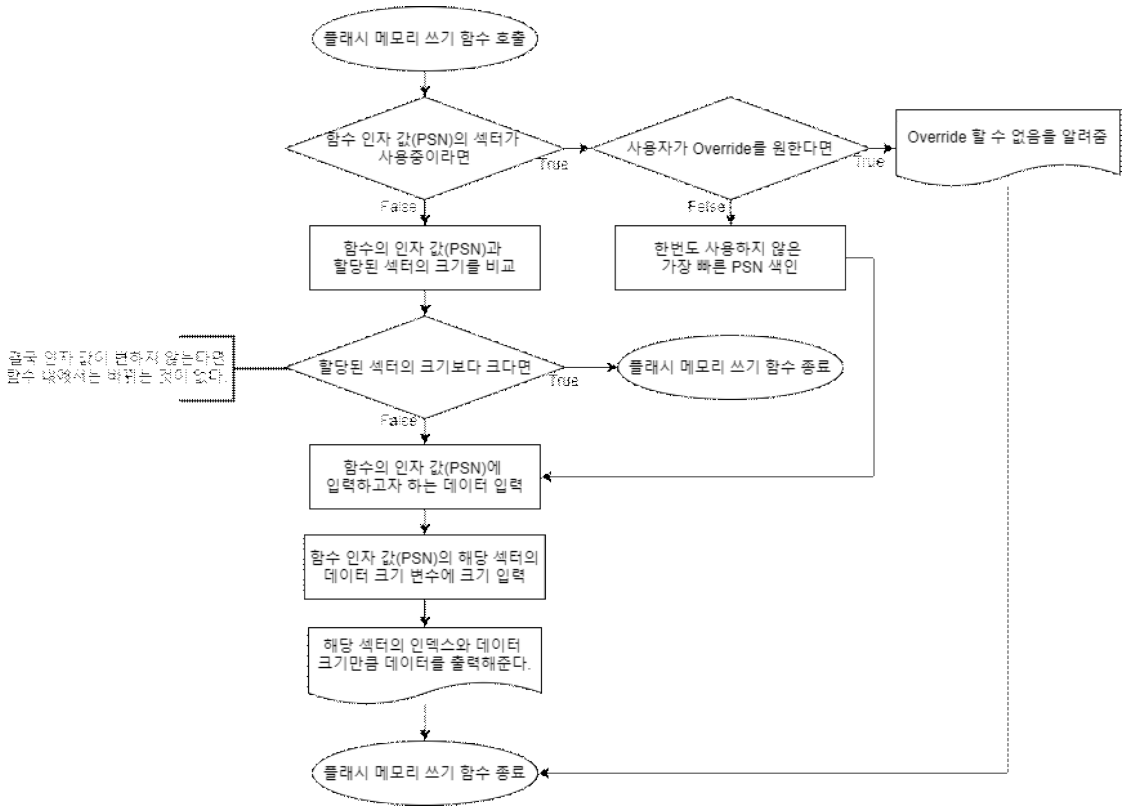
그리고 할당 함수가 두 번째 실행되었을 때 문제가 발생하는데, 이 프로그램은 결론적으로 시뮬레이션을 하면서 Flash Memory 컨트롤러 엔지니어가 원하는 결과가 나올 때까지 디버그하는 것이 사용 목적이라고도 생각이 들어 프로그램 사용자에게 저장소를 재할당할 것인지 물어보며, 만약 재할당하고자 한다면, Flash Memory 내에 데이터가 있는지 확인하며 만약 데이터가 존재한다면, 데이터의 손실이 있을 수 있음을 출력해준다. 이 과정에서 Sector를 백업할 때, Sector Write 시 미리 기록한 데이터 길이를 기반으로 파일 스트림을 이용해 임시 저장하여 데이터를 백업하며, 백업이 완료되면 기존의 포인터 배열의 메모리 해제 후 사용자가 원하는 크기의 저장소를 재할당을 마치면서, 미리 구현한 할당 함수가 종료되며, 할당 함수의 호출을 맡았던 함수에서 데이터가 재할당되었음을 감지해서, 파일 스트림으로부터 재할당 전 데이터 복원을 마치고 Flash Memory 할당 함수가 종료된다.

<Flash Memory 읽기 함수가 호출된다면>



필요 없는 데이터 인덱싱과 잘못된 메모리 주소의 접근을 막기 위해, Flash Memory가 할당되어 있지 않을 때 읽기 함수의 작동하지 못하도록 하였다. 반대로 저장소가 할당되어 있다면, Sector Read 함수가 작동되도록 설계하였으며, 사용자에게 선택된 Sector가 가진 데이터를 사용자에게 보여주기 위해서, 인덱스가 할당된 섹터 내에 속해있는지 확인하며(할당된 메모리를 체크 하는 것과는 다름), 인덱스에 속하는 경우, Sector에 기록된 데이터 크기를 받아와 데이터를 출력해주고, 동시에 해당 데이터의 길이와 Sector 내 저장한 데이터의 길이를 비교해서 데이터 무결성을 확인해주면서, 읽기 함수가 종료된다.

<Flash Memory 쓰기 함수가 호출된다면>

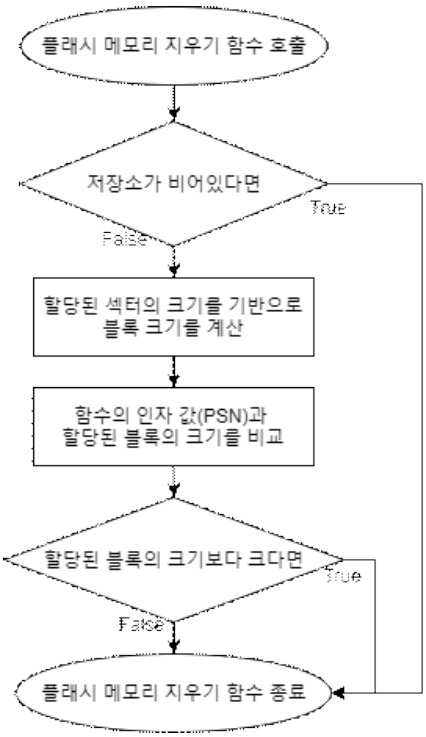


처음 쓰기 함수가 호출될 때 읽기 함수와 마찬가지로 잘못된 메모리 주소 접근을 방지하기 위해서 Flash Memory가 할당되었는지 확인하며, 입력된 PSN과 할당된 섹터 크기와 비교해 할당된 섹터에 속한다면, 입력된 데이터와 PSN을 기반으로 섹터에 데이터 저장 후, Size 변수에 데이터 크기를 기록해준다. 데이터의 길이를 측정하는 이유는 추후 데이터 무결성 확인에 필요하다고 생각해서 Sector마다(512Byte, 512개의 데이터) Size 변수를 포함하였다.

우리가 일반적으로 사용하는 Flash Memory와 다르게, Controller가 없기에, FTL 알고리즘과 같은 Flash Memory 고급 관리기능을 사용하지 못한다. 그러기에 데이터 작성을 위해 입력된 PSN(실제 섹터 인덱스)이 사용 중일 때, 사용자에게 이미 데이터가 존재함을 알려주며, Override 여부를 물어본다. 여기서 사용자가 Override를 원하지 않는다면, Sector Size와 해당 Sector 내 채워진 데이터를 기준으로 가장 빠른 번호의 PSN을 찾아 데이터를 입력시켜주며, 데이터 크기까지 포함해서 Sector에 넣어둔다.

만약 사용자가 Override를 원한다면, 현재 FTL 알고리즘 없이는 이를 구현하는 것이 의미가 없기에, Override 할 수 없음을 알려주고 쓰기 함수를 종료시킨다.

<Flash Memory 블록 지우기 함수가 호출된다면>



조금의 데이터 블록들을 관리하는 것이기에 데이터가 들어있는지, 정상적인 데이터가 들어있는 것인지, 직접 확인이 가능하지만, 만약 현재 상용화되어 있는 Flash Memory SSD처럼 1TB와 같은 용량을 가진 저장소를 일일이 확인하고 지우는 것을 불가능하다고 생각이 들었다. 그리고 추후 FTL 알고리즘을 이용해 Flash Memory를 관리하고자 할 때, 엔지니어가 정한 특정 조건에 따라, Gavage Collection과 같이 필요 없는 데이터를 Block 내에서 색인하고, 지워야 할 일이 생길 것이기에, 블록 단위로 제거하는 함수를 따로 만들어 확인 없이 입력된 블록 번호를 기반으로 실질적인 제거를 할 수 있어야 한다고 생각이 들었다. 블록 제거 함수에서는 기본적으로 할당된 공간 내에 입력된 블록이 속하는지 확인하며, 속한다면 해당 블록을 지우면서 함수가 종료된다.

3. 프로그램 실제 작성

Flash Memory를 구현하는 프로그램의 목적과 요구 점을 생각해봤을 때 엔지니어가 FTL 알고리즘을 만들거나, 리버스 엔지니어링 하기 위해 사용하는 프로그램이라 생각이 들었으며, 결국 Flash Memory는 비휘발성 메모리로 사용자가 모든 정보처리 장치에서 처리한 내용을 보관하는 곳이기에, 프로그램에 대한 데이터 무결성, 신뢰도가 최우선을 위해 실질적인 기능 구현보다는 예외적 처리에 더 신경을 썼다.

<프로그램 헤더>

```
1   #include <iostream>
2   #include <string>
3
4    //for debugging, reallocation
5   #include <fstream>
```

기본적인 C++의 헤더 iostream, 입력값을 관리하기 위한 string, 파일 입출력을 위한 fstream, 그리고 이 프로그램이 지금은 컴퓨터에서 작동하지만, 추후 메모리 테스트를 위해 사용용량이 제한되는 임베디드 시스템, Flash Memory에서 STL을 사용해서 제한된 저장공간, RAM을 낭비할 필요가 없다는 생각이 들었으므로, 기본적인 기능을 구현하기 위한 헤더 파일만 사용해서 NAND Flash의 특성을 구현하였다.

<Flash Memory 구조>

```
 9  class Flash
10  {
11  public:
12      char Sector[512]; // 512 + 1 byte, Pointer == (Variable *= 4)
13      int Valid_Check = 0; // length of data?
14      Flash()
15      {
16          for (int i = 0; i < 512; i++)
17          {
18              Sector[i] = 0; //init data allocation, 0 Fill
19          }
20      }
```

Class의 객체 1개를 1 섹터로 구현하도록 하였다.

현재 1개의 섹터는 Char Array 특성에 따라(512 + 1) + 4byte(Sector Length)의 용량으로 512byte의 데이터를 구현하며, 초기 선언 시 생성자를 통해 섹터 내 모든 데이터가 0으로 채워진다(0-Fill). 이 과정은 컴퓨터에서 포맷할 때의 과정과 같다. (빠른 포맷을 말함)

<프로그램 실행 부분, Pointer Variable>

```
int main()
{
    Flash* data;
    Exec_Func(data);
}
```

프로그램 실행 시, NAND Flash를 담당하는 Pointer를 선언하고, 각 기능을 실행하는 함수에 Pointer 변수의 주솟값을 전달해서 추후 실행될 기능 실행 함수를 통해 Flash Memory 관리기능을 작동시킬 때 데이터 무결성을 보장하였다.

<기능 실행 함수>

```
void Exec_Func(Flash*& o1)
{
while (true)
{
    cout << "시뮬레이션 하고 싶은 명령을 내려주세요. (init, read, write, erase, status, save, load, exit)" << endl;
    Input_Func(Input1, 6); //maximum length set
    Exit_Func(Input1); //Trusted Input...!
    if (Input1 == "init" || Input1 == "Init" || Input1 == "INIT") //if init + Eng Input?? => Error
    {
        Input_Func(Input2, 3); // xxxMB
        int temp = Conv_Func(Input2); //Mem Size
        if (temp == -1 || temp < 0)
        {
            continue;
        }
        init(o1, temp);
        pos = o1;
```

프로그램 실행 시 이 기능 함수는 프로그램이 가지고 있는 명령을 출력해준다. 이후 사용자에게 각 기능에 연결되어있는 명령을 입력받으며, 명령에 따라 Token을 입력받거나, 기능을 실행한다. 이와 같은 시나리오를 구현하기 위해서 기능 실행 함수 실행 시 초기 3개의 String 변수를 선언해서 기능에 따라 데이터를 다르게 입력받으면 된다.
예를 들어 write 0 XXXX와 같은 명령이 들어왔다고 가정해보자, 1번째 입력 Token에서 write가 입력되었기에, write를 위해 Sector Number인 2번째 Token을 입력받는다.
그러나 해당 과정에서 번호로 입력을 받아야 Sector Index로 사용할 수 있기에 Conv_Func를 통해 입력된 Token을 숫자로 변환하며, 쓰기 기능을 위해 3번째 데이터 Token을 입력받아 String 변수에 임시로 저장해둔다. 모든 입력과정이 끝났다면, Flash_Write 함수에 Flash Memory 주솟값과 Sector Number, 데이터를 전달해주며, 쓰기 기능을 시작한다.

<공간 할당 함수, 동적 배열 할당>

```
void init(Flash*& o1, int& v1)
{
    system("cls");
    bool temp = false;
    if (pos) //if allocated
    {
```

```
void Rinit(Flash*& o1, int& v1)
{
    Flash* temp = new Flash[2048 * v1]; //dynamic array allocation
    o1 = temp;
    secsize = v1 * 2048; //secsize / 32 == blksize
}
```

좌 : 공간 할당 명령 입력 후 실행되는 함수, 우 : 실제 공간 할당 함수

기능 작동 함수를 통해 공간 할당을 위한 init 명령이 내려졌다면, Flash Memory 주솟값과 숫자로 변환된 사용자가 할당하고자 하는 공간의 용량(MB 단위)을 공간 할당 함수인 init에 전달한다. 할당 함수가 처음 실행되었을 때는 실제 공간 할당 함수를 통해 원하는 용량만큼 Flash Memory를 할당한다면 아무 문제 없을 것이다. 그러나 할당 함수가 2번 실행되어 동적 할당이 두 번 발생한다면 어떻게 될까? 메모리 해제 없이 새로 동적 할당을 하게 되어, 프로그램이 차지하는 메모리 용량이 의미 없이 늘어날 뿐만 아니라, 기존에 데이터를 저장해둔 상태라면, 데이터 손실까지 일어날 것이다. 이를 방지하기 위해 전역 포인터 변수(8byte, XCode 기준)를 선언해서 메모리 할당을 확인하며, 할당되어 있다면(두 번째 이상 실행할 때), 데이터 손실이 발생 할 수도 있음을 설명하며, 재할당을 할 것인지 확인한다. 이후 전체 섹터를 색인하면서 데이터가 들어 있는 Sector의 정보(Index, 데이터, 데이터 길이)를 파일에 저장하고, 메모리 해제를 한다. 사용자가 원하는 크기로 Flash Memory를 재할당하며, 파일로부터 섹터 데이터를 불러와서 재할당을 구현하였다. 그러나 이 경우 용량 확장에는 데이터 손실이 없지만, 용량 축소 재할당의 경우, 할당된 공간을 넘어 잘못된 메모리 접근을 방지하기 위해, 데이터 손실이 발생시키면서, 강제 재할당을 실시한다.

<Sector 읽기 함수, 올바른 데이터 접근, 데이터 길이(무결성) 체크>

```
void Flash_read(Flash*& o1, int& v1)
{
    system("cls");
    if (!pos) //array isEmpty
    {
```

```
else if (pos && v1 < secsize)
{
    Sector_read(o1, v1); //real read func
```

```
void Sector_read(Flash*& o1, int& v1)
{
    Sector_Frame();
```

```
while (i < 512 && o1[v1].Sector[i] != 0)
{
    cout << o1[v1].Sector[i];
```

```
if (i == o1[v1].Valid_Check && o1[v1].Valid_Check)
```

그림 1 : read시 호출되는 함수, 그림 2 : 섹터 크기보다 작은 인덱스만 허용
그림 3 : 실질적으로 섹터를 읽는 함수, 그림 4 - 5 : 섹터의 내용 출력 + 데이터 길이 체크
그림 6 : 출력되는 데이터와 저장된 데이터 길이의 비교

읽기 함수가 호출되었다면, 호출된 Flash_read 함수를 통해 잘못된 메모리 접근을 막기 위해 Flash Memory가 할당되어 있는지, 읽고자 하는 섹터가 할당된 섹터에 속하는지 확인하며, 실질적으로 데이터를 읽는 Sector_read를 호출한다. 출력되는 데이터의 구분을 위해 데이터 틀을 출력하며, 기본적으로 섹터의 512개의 데이터를 가지기에, 512개의 데이터와 0이 아닐 때 섹터를 출력하도록 하였다. 저장된 데이터 길이와 출력된 데이터의 길이를 비교하며, 길이가 같다면 무결성이 확인되었다는 메시지를 출력 후 함수를 마친다.

<Sector 쓰기 함수, 올바른 데이터 접근, 데이터 길이 기록>

```
void Flash_write(Flash*& o1, int& v1, const string& o2)
{
    system("cls");
    if (!pos)
    {
```

```
void Sector_write(Flash*& o1, int& v1, const string& o2)
{
    const char* data = o2.c_str();
    unsigned int i = 0;
    while (i < o2.length()) //어차피 512byte를 넘는 데이터
    {
        o1[v1].Sector[i] = data[i]; //casting 불가하기에
        ++i;
    }
    o1[v1].Valid_Check = (int)o2.length(); //data size savi
```

좌 : 쓰기 명령으로 실행되는 함수, 우 : 실질적으로 섹터를 쓰는 함수

쓰기 명령으로 인해 초기 쓰기 함수 (Flash_write)가 호출될 때, Flash Memory가 할당되어 있는지, 쓰기 명령을 내리려는 섹터가 할당된 섹터 내에 속하는지 확인해야 해서 잘못된 메모리 접근을 방지해야 하며, 확인되었다면, 실제 섹터 쓰기 함수(Sector_write)를 호출해 실질적인 쓰기 작업을 시작한다. 콘솔의 Input Stream으로 데이터를 입력받을 때 String Type으로 입력을 받고, 섹터의 데이터 저장공간을 Char 배열 형식으로 할당해서, String Class에 의해 자동 Casting이 불가했기에, 입력된 데이터를 Const Char Type으로 Casting 해서 변환된 데이터를 Sector의 데이터 공간에 복사 작업을 실행하며, 데이터의 길이만큼 복사 작업을 마쳤다면, 마지막으로 추후 섹터 데이터 접근 시 데이터 무결성을 확인하기 위해서, 기록한 데이터의 길이를 Sector 내에 저장하고 쓰기 함수를 마친다.

<입력받은 블록 Index에 속하는 Sector 지우기>

```
void Flash_erase(Flash*& o1, int& v1)
{
    system("cls");
    if (!pos)
    {
else if (pos && v1 < (secsize / 32))
{
    Block_Erase(o1, v1);
    cout << "Block " << v1 << "의 삭
}
```

```
void Block_Erase(Flash*& o1, int& v1) //v1 is blk index 0-31,
{
    int start =  v1 * 32;
    int end = (v1 + 1) * 32; // end 전까지
    while (start < end)
    {
        memset(o1[start].Sector, '\0', 512); //0-511 Data Clear
```

좌 : erase 명령 호출 시 실행되는 함수, 우 : 블록 인덱스를 기반으로 섹터를 지우는 함수

Flash Memory의 최소 지우기 연산 단위가 Block이기에, Erase 명령과 같이 받은 변환된 숫자는 Block Index여야 한다. 그러나 데이터를 읽고 쓰는 최소 단위, 객체 내 저장된 단위는 Sector이므로, 입력된 Index를 기반으로 Block Index 내 속하는 32개의 Sector의 모든 데이터를 지워야 한다. 그러나 섹터를 Char Type Array로 구현했기 때문에 Sector의 데이터가 저장된 Array의 1번째 인덱스를 '\0'으로 채워 읽기 함수로 데이터 접근 시, 데이터가 비워진 것처럼 보이게 할 수는 있지만, 이는 실제 데이터가 저장되는 Flash Memory를 시뮬레이션하는 것이 이 프로그램의 목적이며 요구 점이므로, Char Array의 512개 데이터를 모두 '\0'으로 채워 실질적으로 모든 데이터를 비우고, 지우기 함수의 작동을 마친다.

<Flash 내 모든 섹터를 Indexing, 존재하는 데이터를 파일 출력>

```
void SecSave_Func(Flash*& o1)
{
    if (!pos)
    {
        cout << "Flash Memory 할당이 되어있지 않습니다. 확인 후 재실행해주세요. " << endl;
        return;
    }
for (int i = 0; i < secsize; i++)  //size != 0 saving!!, secsize exist
{
    if (o1[i].Valid_Check) //data is Full
    {
        s_ofs << i << " " << o1[i].Sector << " " << o1[i].Valid_Check << " " << "\n";
    }
```

그림 1 : Sector 내 데이터가 존재한다면, 파일로 저장하는 함수
그림 2 : SecSave_Func의 일부, Sector의 크기만큼 색인하여, 데이터가 존재한다면, File Stream을 통해 섹터의 인덱스, 섹터의 데이터, 데이터 길이를 파일에 저장함.

먼저 이 프로그램은 Sector 당 실용량 (512 + 1) + 4(Sector Data Length) + Pointer Overhead byte를 이용해 512byte의 데이터가 저장되도록 구현한 프로그램이기에, 모든 섹터의 데이터를 파일에 저장하게 된다면, 0-Fill(초기) 데이터까지 저장하는 것이므로, 섹터 내, 데이터 길이를 기록한 변수를 통해 실질적으로 데이터가 들어 있는 Sector를 구분하며, 데이터가 존재한다면 파일에 저장하고 모든 데이터를 색인했다면, 파일 출력 스트림을 닫고 저장함수를 마친다.

<저장된 파일로부터 섹터 인덱스, 섹터 데이터 불러오기>

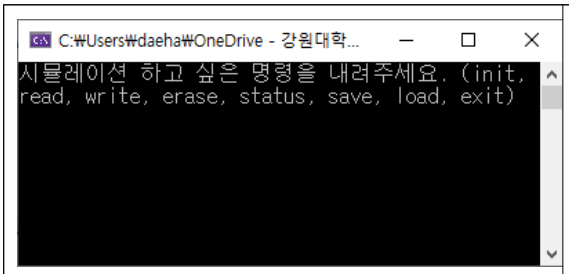
```
void SecLoad_Func(Flash*& o1)
{
    if (!pos)
    {
while (s_ifs >> index >> temp.Sector >> temp.Valid_Check && index < secsize)
{
    o1[index] = temp; //Object Copy
}
```

위 : Flash Memory의 할당 여부 확인,
아래 : 파일로부터 Sector 데이터를 가져오는 부분

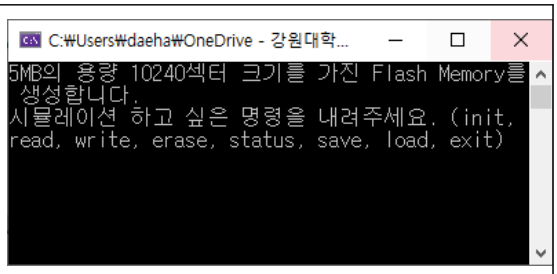
Sector 데이터를 로드하기 위해, Flash Memory의 공간 할당 여부를 확인하며, 파일 입력 스트림을 통해 로드하려는 파일이 존재할 때, 파일을 저장했을 때와 로드할 때의 Flash Memory의 할당 사이즈가 다를 수 있고, 할당된 섹터를 넘어 데이터를 저장하는 경우가 발생할 수 있는데, 이 때는 섹터 인덱스가 섹터 사이즈보다 작을 때만 파일 데이터를 임시 객체에 저장하고, 해당 섹터에 데이터를 복사해주며, 파일 스트림을 닫고 로드 함수를 마친다.

4. 프로그램 실행 화면

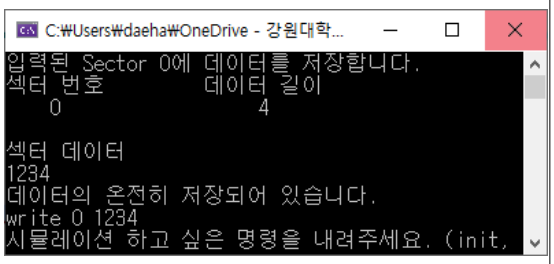
<테스트 결과>



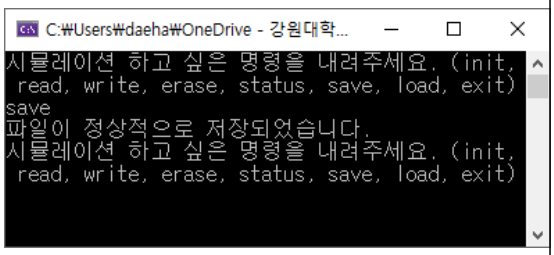
프로그램 실행 시, 구현 명령 출력



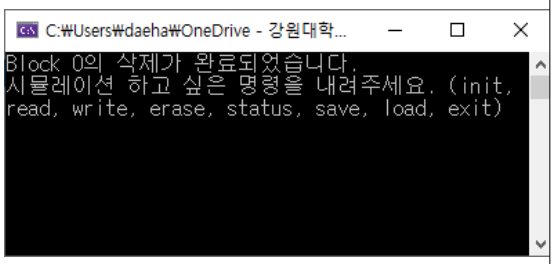
init 5를 입력해 공간 할당



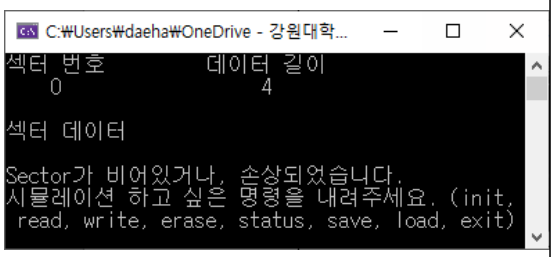
write 0 1234 입력으로 쓰기 명령



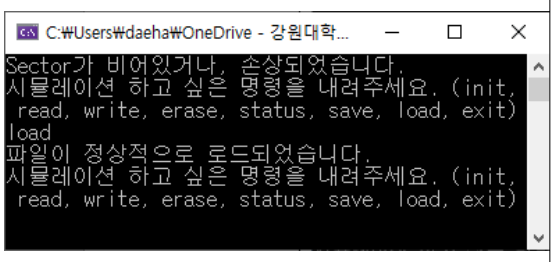
데이터 삭제 전 데이터 저장



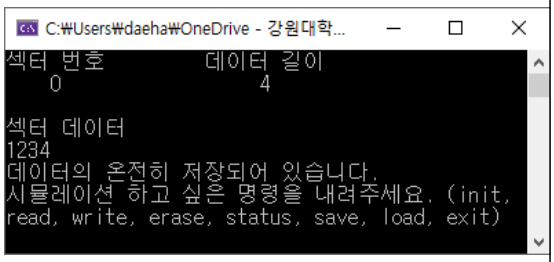
erase 0 입력해 블록 0 제거 명령



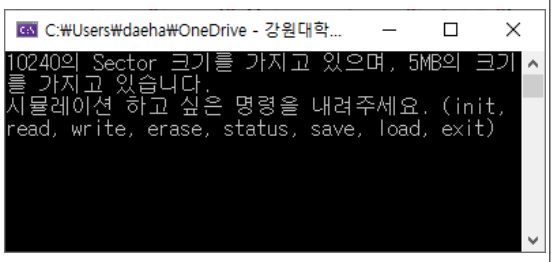
read 0 입력해 블록이 지워짐을 확인



load를 입력해 파일로부터 데이터 복원



read 0 입력 시 데이터 복원



status 입력 시 할당된 데이터 크기 출력

<프로그램 작성 후 결과 분석>

Flash Memory의 특성을 구현하는 프로그램을 작성하려고 할 때, 단지 하드웨어의 읽기, 쓰기, 지우기 특성을 고려하기만 하면 된다고 생각하여, 요구사항 분석과 프로그램 작성에 큰 어려움을 느끼지는 못했다. 그러나 Flash Memory에 가깝게 프로그램을 작성하였다고 생각이 들지는 않는다. 지금부터 내가 생각하는 이 프로그램의 문제점, 장점에 관해 설명하겠다.

문제 1. Flash Memory 수명 문제

이 프로그램의 작동 방식대로 실제 데이터 I/O가 발생하게 된다면, F2FS와 같은 Flash Memory에 최적화되어있는 File System, Application이라면 상관없겠지만, 일반적인 HDD 친화 Software, File System을 혼용해서 사용하므로, HDD File System에 대한 특성을 고려하지 않고 그대로 사용하면, Flash Memory의 수명을 급격하게 줄어들게 되며, 지금은 Wear-Leveling과 유휴 상태의 Trim과 같이 Flash의 성능, 수명 증대에 도움을 주는 고급 기능들이 없으므로, 같은 블록에서 계속 I/O가 발생하게 된다면, 결국 Data Access Time 저하, 블록 사용 불가 상태로 이어질 것이다. 이를 조금 구현해보기 위해, 지우기 연산이 일어날 때 0-Fill 대신 A-Fill, B-Fill로 섹터 데이터 수명 척도를 기록하려고 했지만, 이는 결국 하나의 Write 연산이 될 것이며, 실제 데이터 Write시 Erase 후 기록하려고 할 것이기에 매우 비효율적 방법이라고 생각되어 구현 기능을 제거하였다.

문제 2. String의 사용

STL을 최대한 적게 사용하려고 했는데, 문자열 처리에서 일일이 기능이 필요할 때마다 모두 구현을 해야 했기에, 어쩔 수 없이 String으로 문자열 처리를 진행하였다. 이로 인해 String Class 자체의 Overhead가 발생할 것이며, 추가로 STL의 Linker Lib도 필요할 것이기에, 최악의 구현 Case라고 볼 수 있다. 그러나 String을 사용한 장점도 있다고 생각한다. 문자열의 길이, Char, int로의 Casting 등 자체 구현된 기능들이 많아 덕분에 요구사항에 집중하며, 비교적 간단히 프로그램을 작성할 수 있었다.

문제 3. Sector, Char Type Variable 512개의 할당 문제

Sector 내에 Char[512]와 같이 1Byte의 데이터를 512개 선언하여, 실제 섹터의 크기 512Byte를 가지는 것처럼 보이게 구현은 하였지만, 실제 데이터가 가지는 Bit (HEX) 단위, 16진수 단위로의 데이터 저장이 불가능하다. 즉, 껍데기만 Sector의 형태를 가진 프로그램이라는 이야기이다.

문제 4. Sector Data Length 기록 시 int 사용 문제

정전이나, 데이터 I/O시 예상되지 못한 데이터 스트림 손실이 발생하여 데이터가 일부만 저장되었을 수도 있으므로, 데이터 길이를 기록하면서, 추후 액세스 시 데이터 무결성을 확인할 수 있는 척도지만, Integer로 데이터를 기록하면서, Sector 당 4Byte(x86으로 컴파일)의 용량을 차지한다는 문제이다. 더 좋은 방법이 있다고 생각하는데, 내 수준에서 현재 이것을 대체할 방법이 떠오르지 않아, Integer 형으로 데이터 길이를 저장하게 되었다.

장점 1. 최소한의 STL 사용 프로그램

추후 FTL을 연계한 프로그램에서 기능 구현은 어려울 것이며, 지금은 DRAM에 Flash의 공간을 할당해서 각 기능을 구현하기에, 속도, 처리 시간 차이가 안 느껴지겠지만, 추후 실제 임베디드 시스템에서 사용하고자 할 때, 프로그램 구동 속도, 용량에서 큰 차이가 날 것이라고 생각이 든다.