論文2001-38CI-1-4

플래쉬 메모리 관리 알고리즘 개발
(Development of Flash Memory Management Algorithm)

朴 寅 圭*
(In Gyu Park)

요 약

플래쉬 메모리는 데이터의 저장과 변경이 가능하고 전원이 차단되어도 저장된 데이터를 보존할 수 있는 RAM과 ROM의 장점을 모두 가지고 있는 메모리로서, 고성능의 전기 특성을 가지고 있어 이동 환경에서의 저장매체로 매우 적합하다. 향후 많은 정보 단말기에 사용하게 될 플래쉬 메모리 및 스마트미디어 카드에 파일을 저장, 삭제, 재생하는 효과적인 알고리즘이 필요하다. 플래쉬 메모리를 일정한 크기의 세그먼트로 할당하고, 파일을 여러 개의 세그먼트로 분산시켜 관리한다. 본 논문에서는 특정 파일이 저장되어 있는 위치를 저장하는 테이블 및 보조 테이블을 이용하여 효과적인 파일 관리 알고리즘을 제시한다. 기존의 알고리즘은 비교적 작은 용량의 플래쉬 메모리를 관리하고 비교적 파일을 빈번한 횟수의 고속 저장, 삭제 등의 작동이 요구되는 응용에는 적합치 않는다. 본 논문에서 제안된 알고리즘은 몇 개의 플래쉬 메모리 칩 및 스마트미디어 카드로 구성된 비교적 큰 용량의 메모리 관리에 적합하다.

Abstract

The Flash memory market is an exciting market that has quickly over the last 10 years. Recently Flash memory provides a high-density, truly non-volatile, high performance read-write memory solutions, also is characterized by low power consumption, extreme ruggedness and high reliability. Flash memory is an optimum solution for large nonvolitile storage operations such as solid file storage, digital video recorder, digital still camera, The MP3 player and other portable multimedia communication applications requiring non-volatility. Regardless of the type of Flash memory, Flash media management software is always required to manage the larger Flash memory block partitions. This is true, since Flash memory cannot be erased on the byte level common to memory, but must be erased on a block granularity. The management of a Flash memory manager requires a keen understanding of a Flash technology and data management methods. Though Flash memory's write performance is relatively slow, the suggested algorithm offers a higher maximum write performance. Algorithms so far developed is not suitable for applications which is requiring more fast and frequent accesses. But, the proposed algorithm is focused on the justifiable operation even in the circumstance of fast and frequent accesses.

Ⅰ. 서 론

플래쉬 메모리는 전원이 없어도 저장되어 있는 데이터를 그대로 보존할 수 있고, 더욱이 최근에는 높은 집적도(128Mb), 소형 패키지, 낮은 전력 소모, 짧은 접근 시간, 그리고 충 격 및 노이즈에 대한 내성 등을 갖추고 있고, 일본 도시바, 한국 삼성, 기타 미국 회사에서 적극적으로 생산하고 있는 실정이다. 이렇게 되면, 기존의 여러 ROM 및 RAM을 대체할 수 있게 되며, 더욱이 이러한 여러 개의 플래쉬 칩을 모아 하나의 작고 얇은 플라스틱 패키지 카드 형태로서, MMC (Mulitimedia card), 또는 Smartmedia card 등의 상품으로 출시되고 있고, 향후 폭팔적인 수요가 예상된

* 正會員, 弘益大學校 電子電氣工學部
(Department of Electronics and Electrical Engineering)
接受日字:2000年5月22日, 수정완료일:2000年12月20日

다. 따라서, 이동 환경에서 저장매체로서 플래쉬 메모리, MMC, Smartmedia card 등이 매우 적합하여 개인 휴대용 정보단말기의 시장 규모를 크게 성장시킬 것으로 예상한다. 예를 들어 PDA, MP3 player, HPC, 디지털 카메라, 전자 수첩, 이동 전화 단말기 등에서는 주 기억 장치로서 사용될 것이다.

플래쉬 메모리가 여러 가지 장점을 가지고 있지만, 반면에 RAM이나 ROM과는 다른 전기적인 특성을 가지고 있다. 즉 읽고 쓰는 방법이 어드레스를 주면 바로 데이터를 읽거나 쓰는 방식이 아니라 우선 해당 명령어를 주고, 어드레스도 몇 번에 나눠주어야 한다. 또한 데이터를 갱신하려면 먼저 갱신하려는 메모리 공간을 지워야 한다. 플래쉬 메모리 공간을 삭제하려면, 하나의 셀 단위로 삭제하는 것이 아니라, 블록 단위로 삭제하여야 한다. 만약 하나의 블록 안에 두 개의 파일이 존재하면, 두 개의 파일을 삭제하는 결과를 초래하게 되므로, 결코 하나의 블록 안에는 두 개의 파일이 존재하여서는 안 된다. 이와 같이 플래쉬 메모리를 저장 매체로 사용하기 위하여 플래쉬 메모리의 특성에 적합한 메모리 공간의 할당과 일반 메모리와 동일하게 사용할 수 있도록 특별한 파일 관리가 필요하게 된다.

본 논문에서는 MP3 player에서 사용하는 MP3 오디오 파일, 음성 압축 파일, USB 제 어 프로그램의 데이터를 PCB에 장착된 플래쉬 메모리 및 별도 옵션 장착이 가능한 SmartMedia card까지 데이터를 관리하고, 혹은 SmartMedia card의 데이터만을 관리하는 알고리즘을 제안하고, 이러한 알고리즘을 구현하는 과정에서의 문제점 및 그 해결하는 방법을 제시하고자 한다. SmartMedia card에만 파일을 생성, 삭제, 수정 등의 기능이 가능하게 되면, SmartMedia card를 초소형, 고밀도, 이동성 저장매체로 사용할 수 있게 되어 궁극적으로 현재의 FDD를 대체하는 효과가 있으므로 정보단말기의 활성을 크게 기여할 것이다.

플래쉬 메모리 파일 시스템은 데이터를 효율적으로 관리하기 위한 각종 관리 데이터와 실제로 저장해야 할 대상인 순수 데이터로 이루어져 있다. 파일의 생성, 삭제, 수정 등의 플래쉬 메모리의 데이터를 변화시키는 과정은 플래쉬 메모리에 직접 순수 데이터를 변화하기 전에 먼저 각종 관리 파일의 관리 데이터를 어떻게 효과적으로 다루느냐에 좌우될 것이다.

본 논문에서 다루게 될 주요 내용을 소개하면, 제 2장에서 그 동안 여러 기관에서 개발해 온 플래쉬 메모리를 이용하는 4가지의 파일 관리 알고리즘에 대하여 소개하고 상호 장단점을 간단히 비교한다. 제 3장에서 본 논문에서 제시할 알고리즘이 작동될 메모리 파일 시스템의 환경을 소개한다. 내용을 살펴보면, 세 종류의 파일 관리용 테이블을 제시하고 구조를 소개한다. 또한 플래쉬 메모리 공간 할당에서의 단위 메모리 크기를 최적화 하여 세그먼트 단위로 분리하는 구조를 설명한다. 제 4장에서는 이러한 세 종류의 파일 관리 테이블로 어떻게 가변 길이의 파일을 용이하게 저장하고 관리하는 과정을 자세히 설명한다. 제 5장에서는 플래쉬 메모리에 파일을 USB controller로부터 download, upload, refresh, initialize 시켜주는 알고리즘 및 실험 결과를 고찰한다.

Ⅱ. 기존의 알고리즘

먼저 일부 정보 단말기 개발 업체에서 이미 연구된 플래쉬 메모리에 레코드를 관리하는 알고리즘에 대하여 간단히 분석한다. 물론 이러한 알고리즘은 256K 플래쉬 메모리와 같은 작은 용량의 제한된 환경 하에서 메모리 절약을 우선하는 알고리즘이다. 그렇지만 본 논문에서는 128M-비트 플래쉬 메모리가 6개, 대용량의 스마트 미디어 카드 등의 비교적 용량이 큰 환경에 맞는 알고리즘을 개발하는 기본 자료로 사용하기에 충분하다.

1. '2-블록' 알고리즘

가장 간단한 알고리즘으로 메모리의 구성을 크게 2개의 구역으로 나누고, 하나의 구역은 하나의 세그먼트를 이룬다. 구역-0은 파일에 관한 정보를 저장하는 카달로그 구역이고, 구역-1은 데이터 구역으로 데이터를 저장한다. 데이터 구역에 차례대로 데이터를 순차적으로 저장하면 되므로, 내부 단편화가 없어 당연히 알고리즘이 간단하고 이해하기 쉬운 장점이 있고, 더욱이 메모리 극대화와 소용량의 메모리에서는 어느 정도 장점이 있겠지만, 메모리가 어느 정도 차 있게 되면 데이터의 저장이나 삭제가 오랜 시간이 걸릴 것이다. 또한 현재의 하나의 플래쉬 메모리 칩이 어느 정도의 대용량인 된 현재로서는 적당하지 않다.

2. '객체 연결' 알고리즘

하나의 정보는 여러 레코드들로 이루어진 한 덩어리의 레코드 집합으로 구성되어 있는 경우에 해당되는

알고리즘으로, 레코드들이 메모리에서 위치적으로 흩어져서 저장되기 때문에 한 레코드 집합의 모든 레코드들을 순차적으로 접근할 때 많은 시간을 소모한다. 또한 레코드를 연결하여 주기 위해 포인터를 사용하는데, 포인터의 사용은 메모리를 낭비하고 알고리즘을 복잡하게 만든다. 이외에도 메모리를 일정한 크기(세그먼트)보다 작은 레코드들이 저장될 때 레코드가 저장되고 남은 공간은 다른 레코드들이 사용할 수 없게 된다. 즉 내부 단편화가 생긴다. 이러한 알고리즘에서는 전체 메모리를 크게 2 구역으로 나누어, 첫 번째 구역에 여러 파일의 레코드 집합에 대한 정보를 저장한다. 다른 구역은 데이터를 저장한다. 만약 모든 블록의 세그먼트를 같게 하면 그 크기보다 작은 객체는 내부 단편화를 만든다. 또한 세그먼트 크기보다 큰 파일은 여러 개의 세그먼트를 할당받음으로서 그 세그먼트들을 연결하는데 쓰이는 포인터가 과부하되어 전체 성능이 떨어지게 된다.

3. '블록 할당' 알고리즘

메모리 전체를 서로 다른 일정한 크기의 여러 블록으로 나눈다. 블록은 레코드 집합들의 정보를 저장하는 카탈로그 블록, 각 레코드를 저장하는 데이터 블록, 그리고 오바플로우에 대비한 오바플로우 블록 등, 3 부분으로 나누어진다.

레코드들은 연속적으로 저장되고 '레코드 끝 표시'에 의해 구분되고, 레코드 집합의 끝은 '파일 끝 표시'로 표시하게 된다. 만약 '파일 끝 표시'가 이러한 데이터 블록에 있지 않으면, 이 레코드 집합에 레코드를 삽입하면서 오바플로우가 발생한 것을 표시한다. 이러한 오바플로우 문제 해결을 위하여 오바플로우 블록에 여러 레코드 집합의 데이터들이 섞여서 저장될 수 있는 공간으로 한 레코드 집합의 데이터를 연속으로 저장하고, 각 레코드 집합의 마지막에는 항상 '파일 끝 표시'라 하여 레코드 집합의 끝임을 나타낸다. 그러나 이러한 알고리즘은 레코드를 삭제하고 다시 그 장소에 보다 큰 용량의 레코드를 저장할 수가 없어 대단히 비효율적이다.

Ⅲ. 플래쉬 메모리 파일 시스템 환경

현재 응용시스템에 가장 많이 사용하는 플래쉬 메모리로 삼성 KM29U128T 칩으로 용량은 128MB(16M ×8)이다. 하나의 응용 예로 MP3 player에서 많은 MP3 파일을 저장할 수 있는 환경인, 최대 6개의 플래쉬 칩을 PCB에 장착 가능하고, 옵션으로 256MB 스마트미디어 카드도 장착 가능한 경우를 연구 대상으로 삼는다. 이러한 환경에서 전체 용량은 1024MB이다.

KM29U128T 플래쉬 메모리칩의 구조를 간단히 설명하면, 한 페이지는 512 바이트이고 하나의 블록은 32 페이지로 이루어져 있다. 따라서 16M 바이트 사이즈의 KM29U128T 메모리는 1024 블록으로 구성된다. 이러한 플래쉬 메모리의 특성의 하나로, 삭제하는 단위는 한 바이트가 아니라 하나의 블록 단위로만 삭제가 가능하다. 이와 같은 물리적인 메모리 특성에 맞추어, 파일 시스템의 구조를 설계한다.

메모리 관리용 논리적 용어인 하나의 뱅크는 1개의 플래쉬 메모리칩에 해당되는 사이즈이고, 한 개의 뱅크는 1024개의 세그먼트로 구성된다. 이러한 기본 삭제 단위인 16K-바이트를 하나의 세그먼트로 설정하고 이러한 세그먼트를 기본 단위로 데이터를 저장, 삭제한다. 플래쉬 메모리칩에 따라 액세스 메모리 번지를 하드웨어적으로 독립적으로 정의하여야 하기 때문에, 플래쉬 메모리를 삭제(erase)시킬 수 있는 최소 단위를 세그먼트로 적용시킨 것이다.

(1) Flash Memory MAP

실제로 본 시스템 메모리 구성은 PCB에 장착할 수 있는 플래쉬 메모리칩을 최대 6개 및 하나의 스마트미디어 카드를 옵션으로 구성할 수 있다. 플래쉬 메모리칩을 논리적인 최소 단위로 16K-바이트의 세그먼트들로 분할시킨다. 이것은 플래쉬 메모리의 전기적 특성으로 삭제시킬 수 있는 최소 단위가 16K-바이트이기 때문에 파일 관리를 위해 이러한 16K-K바이트를 세그먼트 단위로 설정한다.

6개의 플래쉬 메모리칩을 최대한으로 장착한 경우, 총 6개의 뱅크를 가질 수 있으며, 하나의 뱅크는 1024개의 세그먼트로 구성되므로, 총 6144개의 세그먼트로 구성할 수 있다. 6개의 플래쉬 메모리칩을 최대한으로 장착하고 동시에 스마트 미디어 카드를 장착한 경우에는, 총 7개의 뱅크를 가질 수 있으며, 플래쉬 메모리로만 6144개의 세그먼트와 스마트 미디어 카드의 용량에 따라 최소 2048개에서 최대 10240개의 세그먼트를 추가할 수 있다.

첫 번째 뱅크에 속하는 1024개의 세그먼트에서 앞의 6개의 세그먼트에 순수 데이터가 아닌 메모리 관리 및

특수 데이터 저장 용도로 사용한다. 표1에 6개의 플래쉬 메모리 칩을 논리적인 분할시킨 뱅크, 세그먼트의 ID를 보여준다. 특별히 관리용 FAT가 앞의 3개의 세그먼트에 저장된다.

① FLASH_FAT 영역

-데이터가 저장되어 있는 블록(연속된 세그먼트 집합)들의 위치 정보를 저장한다.

-한 블록은 2-바이트 크기의 한 쌍의 FIE 정보로 표시한다.

-FLASH_FAT의 크기는 16K-바이트로 4096개의 블록 정보를 기록할 수 있다.

② FLASH_BFAT

FLASH_FAT와 FLASH_VFAT의 FIE들을 수정, 정리하기 위해 임시 FLASH_FAT 혹은 FLASH_VFAT의 내용을 일부 복사하거나 또는 새로운 데이터를 추가한 후에 다시 FLASH_FAT 혹은 FLASH_VFAT에 그대로 복사한다. FLASH_BFAT를 초기화시킨다. 즉 일종의 임시 backup 용 저장 장소이다.

③ FLASH_VFAT

FLASH_VFAT에는 비어있는 블록(연속된 세그먼트 그룹)의 위치 정보를 저장한다.

④ LCD FONT 영역

LCD FONT DATA 영역은 bitmap LCD 화면에 표시될 문자, 숫자, 그림 등의 데이터를 보관한다.

⑤ ENCODER 영역

Encoder 칩에 다운로드될 프로그램을 저장한다. Encoder 칩 내부에는 DSP가 내장되어 있어, 운용 프로그램을 다운로드시켜 작동시킨다.

⑥ DECODER 영역

Decoder 칩에 다운로드될 프로그램을 저장한다. Decoder 칩 내부에는 DSP가 내장되어 있어, 운용 프로그램을 다운로드시켜 작동시킨다.

⑦ FLASH_DATA 영역

FLASH_DATA 영역은 BANK-0의 SEGMENT-6에서부터 BANK-5의 SEGMENT- 6142 까지 위치한다. FLASH_DATA 영역은 실제적인 데이타가 저장되고 할당 단위는 세그먼트 크기가 된다.

(2) 스마트미디어 카드 Memory MAP

스마트미디어 카드에만 데이터를 저장 및 관리가 가능하도록 설계한다. 이러한 설계 개념은 스마트미디어 카드를 이동성이 뛰어난 저장할 수 있어 보다 넓은 응용에 활용 가능하게 될 것이다. 예를 들어 MP3 오디오 파일을 스마트미디어 카드에만 저장시키게 되면 추후에 스마트미디어 카드를 다시 장착시켜 언제든

표 1. 6개의 플래쉬 메모리 칩 구조에 대응되는 세그먼트 배당
Table 1. Segment Assignment in 6 Flash memory bank.

<table>
<tr><th colspan="6">뱅크</th><th>구 분</th><th>영 역</th><th>크 기</th></tr>
<tr><td rowspan="6">B0</td><td rowspan="5">B1</td><td rowspan="4">B2</td><td rowspan="3">B3</td><td rowspan="2">B4</td><td rowspan="7">B5</td><td>SEG 0</td><td>FLASH FAT</td><td>16K</td></tr>
<tr><td>SEG 1</td><td>FLASH BFAT 영역</td><td>16K</td></tr>
<tr><td>SEG 2</td><td>FLASH VFAT영역</td><td>16K</td></tr>
<tr><td>SEG 3</td><td>LCD FONT 영역</td><td>16K</td></tr>
<tr><td>SEG 4</td><td>ENCODER 영역</td><td>16K</td></tr>
<tr><td>SEG 5</td><td>DECODER 영역</td><td>16K</td></tr>
<tr><td>SEG 6 ∫ SEG 1023</td><td rowspan="6">Data 저장 영역</td><td>1018*16k</td></tr>
<tr><td rowspan="2">B4</td><td>SEG 1024 ∫ SEG 2047</td><td>16M</td></tr>
<tr><td rowspan="2">B3</td><td>SEG 2048 ∫ SEG 3071</td><td>16M</td></tr>
<tr><td rowspan="2">B2</td><td>SEG 3072 ∫ SEG 4095</td><td>16M</td></tr>
<tr><td rowspan="2">B1</td><td>SEG 4096 ∫ SEG 5119</td><td>16M</td></tr>
<tr><td rowspan="2">B0</td><td>SEG 5120 ∫ SEG 6143</td><td>16M</td></tr>
</table>

지 재생시킬 수 있게 된다.

마찬가지로 맨 앞 3개의 세그먼트를 스마트미디어 카드 관리용으로 배정한다. 이러한 스마트미디어 카드는 현재 128M-비트, 256M-비트 등의 제한적인 제품만이 출시되고 있지만 조만간 출시될 예정인 1G-비트 대용량 제품도 사용 가능하도록 표 2와 같이 다양한 스마트미디어 카드들에 대응되는 세그먼트를 설정한다.

(3) FAT(File Allocation Table) 종류 및 저장 위치

텅 비어있는 플래쉬 메모리에 데이터를 기록시키면, 당연히 처음부터 차례로 세그먼트에 기록되기 시작한다. 그러나 반복적인 기록과 삭제 작업으로 인해 어느 특정 음악 파일 및 음성 파일은 연속적으로 저장되지 않고 여러 세그먼트 영역에 걸쳐서 저장될 것이다.

그렇다면 분산 저장된 파일을 효과적으로 관리하는 방법을 모색하여야 한다. 이를 위하여 파일의 저장 위치 정보를 일괄적으로 모아 놓은 테이블인 FAT(File Allocation Table) 을 도입한다. 이렇게 분산되어 저장되어진 파일을 일괄적으로 관리하기 위한 시스템이 필요하다. FAT(File Allocation Table)는 이러한 분산 저장을 만족시킬 수 있는 방향으로 관리할 수 있도록 하여야 하고, 동시에 플래쉬 메모리 구조에 적합하도록 구성하여야 한다.

본 플래쉬 관리시스템에서는 세 종류의 FAT를 도입하여 서로 다른 용도이면서도 상호 작동하면서 메모리를 관리하도록 한다. 첫 번째 FAT(File Allocation Table)는 세그먼트-0에 위치하며 저장된 파일의 위치 정보를 저장하고 있다. 두 번째 테이블인 BFAT (Backup File Allocation Table)에는 임시로 FAT 혹은 VFAT의 일부 내용을 저장시켜두는 일종의 버퍼로 이용된다. 이것은 플래쉬 메모리의 특성을 감안하여, FAT 혹은 VFAT의 내용을 갱신하려는 경우, 임시로 필요한 데이터를 BFAT에 저장해 놓거나 첨가시킨다. 이러한 수정, 갱신 작업이 끝나면, BFAT를 완전히 삭제시켜 작업을 종료한다. 세 번째 테이블인 VFAT는 플래쉬 메모리의 빈 영역의 위치 정보를 갖는다. 이미 저장되어 있는 파일을 삭제하는 경우, 삭제된 파일의 위치를 VFAT에 추가시켜 놓는다. 그러면, VFAT는 어떤 용도로 사용하는가에 대하여 설명하여 보면, 메모리에 새로운 파일을 저장하려면, 먼저 빈 공간이 어디에 위치하여 있는지를 파악하여야 한다. 따라서 빈 공간의 위치를 저장시켜 놓으면 보다 빠르게 위치 정보를 얻을 수 있게 될 것이다.

표 2. 여러 스마트미디어 카드에 대응되는 세그먼트 배당

Table 2. Segment Assignment in Several kinds of Smart-Media Card.

Smartmedia Card 종류	구 분	영 역	크 기
16M, 32M, 64M, 128M, 256M, 512M, 1G	SEG_0	SMART_FAT	16K
	SEG_1	SMART_BFAT 영역	16K
	SEG_2	SMART_VFAT영역	16K
	EG 3 ∫S SIEG_1023	SMART Data 저장 영역	1021*16K
	SIEG 1024 ∫ SIEG_2047		16M
	SIEG 2048 ∫ SIEG_4095		32M
	SIEG 4096 ∫ SIEG_6043		64M
	SIEG 6044 ∫ SIEG_12087		128M
	SIEG 12088 ∫ SIEG_24175		256M
	SIEG 24176 ∫ SIEG_48352		512M

4. FAT 구조

하나의 파일을 플래쉬 메모리에 할당하는 방법에 대하여 설명한다. 하나의 파일은 여러 개의 블록으로 분산 저장되고, 하나의 블록은 연속된 몇 개의 세그먼트로 구성된다. 따라서 하나의 파일이 분산 저장된 블록끼리는 서로 인접할 수가 없다. 어느 특정 파일의 위치 정보를 해당 블록별로 일정한 포맵으로 FAT에 저장시킨다.

하나의 블록은 인접하고 있는 세그먼트들의 그룹을 의미하므로, 모든 세그먼트에 대한 위치를 표현하기보다는 시작_세그먼트 위치와 마지막_세그먼트 위치를 표시하는 것이 보다 효과적이다. 이러한 블록 단위의 위치 정보를 FIE(File Information Element)라 부른다. 따라서 하나의 파일에 관한 FAT에는 여러 FIE들의 집합으로 표현한다.

FAT에는 여러 파일의 위치 정보의 그룹으로 구성하고 있다.

FAT = {파일 #1의 위치 정보, 파일 #2의 위치 정보, ... , 파일 #n의 위치 정보}

어느 특정 파일은 여러 블록들의 집합이므로 어느 특정 파일의 위치 정보는 해당 블록의 FIE들의 집합으로 구성한다.

특정 파일의 위치 정보 = {블록 #1의 FIE, ... , 블록 #n의 FIE}

특정 블록에 대한 FIE 정보는 블록을 구성하고 있는 세그먼트들의 FIE로 구성할 수 있다. 블록을 이루고 있는 연속된 모든 세그먼트의 FIE를 모두 표현하는 것이 아니라, 처음 세그먼트의 FIE와 마지막 세그먼트의 FIE 등의 한 쌍으로 표현한다.

특정 블록의 FIE = {시작_세그먼트 FIE, 끝_세그먼트 FIE}

정리하면 FAT는 가장 작은 정보 단위인 세그먼트의 FIE들의 집합으로 구성하게 된다.

이러한 가장 작은 단위의 세그먼트 FIE는 1-바이트로 구성하고 있으며, 어떠한 정보를 포함
시킬 것인가에 대하여 다음 소절에서 논의한다.

다음에는 FAT에 내장되어 있는 정보를 이용하여 어떠한 정보를 유도할 수 있는 것에 대하여 논의한다. 먼저 이러한 정보를 이용하여 첫 번째로 저장되어 있는 파일의 개수를파악할 수 있으며, 두 번째로 메모리에 저장되어 있는 순서로 파일의 번호를 매긴다면 프로그램으로 특정 파일에게 파일 번호를 부여할 수 있다. 즉 모든 파일은 고유의 번호를 가지도록 하여 파일의 수정, 삭제, 생성 등의 관리를 용이하도록 한다.

5. FIE 구조

하나의 블록마다 한 쌍의 FIE로 구성한다. 다시 말하여 2개의 FIE가 하나의 블록에 관한 정보를 제공하고 있다. 다시 말하여 시작_세그먼트의 위치를 나타내는 FIE와 마지막_세그먼트의 위치를 나타내는 FIE 등의 한 쌍의 FIE로 표현한다.

본 논문에서 이러한 FIE를 여덟 가지의 종류로 구분하고 이에 대하여 논의한다. 앞에서 언급한 바와 같이 첫 번째 구분 요소로 시작_세그먼트의 FIE 혹은 마지막_세그먼트의 FIE로 구분하고, 두 번째 구분 요소로 처음 블록의 FIE 혹은 연결 블록의 FIE 등으로 나누고, 세 번째 구분 요소로 파일이 MP3 파일인지 혹은 voice 파일 등으로 구분한다.

이러한 여덟 가지의 FIE를 구분하기 위하여 3-비트가 필요한데 실제로는 2-비트로 구분한다. 먼저 하나의 블록은 한 쌍의 FIE로 구성되기 때문에 당연히 첫 번째 FIE는 블록의 첫 번째 세그먼트의 FIE이고 두 번째 FIE는 마지막 세그먼트의 FIE이기 때문에 자동적으로 구분할 수 있기 때문에 4종류의 FIE로 구분하면 충분하기 때문에 2-바이트로 FIE의 종류를 구분할 수 있다.

표 3 및 표 4에서 첫 번째 필드에 2-비트를 할당하여 어떠한 종류의 세그먼트인지를 구분한다. 위에서 논의된 3가지 구분 요소를 융합하여 표 5, 표 6과 같이 구분한다. 자세히 설명하여 보면, 특정 시작 세그먼트

표 3. 시작 세그먼트의 FIE format
Table 3. FIE Format of Start Segment.

← High 8 bit						→		← Low 8 bit							→
B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
Start ID		Bank/high address				High address							Mid address		

표 4. 마지막 세그먼트의 FIE format
Table 4. FIE Format of Last Segment.

← High 8 bit →								← Low 8 bit →							
B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
Final ID		Bank/high address				High address							Mid address		

표 5. Start 필드
Table 5. Start Field.

Start ID	시작_세그먼트의 FIE 종류
00	S_M : MP3 파일으로서 시작 블록의 시작_세그먼트
01	SC_M : MP3 파일으로서 중간 블록의 시작_세그먼트
10	S_V : Voice 파일으로서 시작 블록의 시작_세그먼트
11	SC_V : Voice 파일으로서 중간 블록의 시작_세그먼트

표 6. Final 필드
Table 6. Final Field.

Final ID	마지막_세그먼트의 FIE 종류
00	E_M : MP3 파일으로서 시작 블록의 마지막_세그먼트
01	EC_M : MP3 파일으로서 중간 블록의 마지막_세그먼트
10	E_V : Voice 파일으로서 마지막 블록의 마지막_세그먼트
11	EC_V : Voice 파일으로서 마지막 블록의 마지막_세그먼트

표 7(a). Address 필드(1)
Table 7(a). Address Field(1).

B13~B9	0000	0001	0010	0011	0100	0101	하위 1-bit
역활	bank-0	bank-1	bank-2	bank-3	bank-4	bank-5	high address
종류	Flash memory						32Mbit Smart Media Card

표 7(b). Address 필드(2)
Table 7(b). Address Field(2).

B8~B0	하위 2-bit	하위 3-bit	4-bit
역활	high address	high address	high address
종류	64-Mbit Smart Media Card	96-Mbit Smart Media Card	128-Mbit Smart Media Card

가 MP3 파일로 마지막 블록에 속함을 의미하는 S_M(Start MP3 세그먼트), voice 파일로 마지막 블록에 속함을 의미하는 S_V(Start Voice 세그먼트), 마지막 블록의 마지막 세그먼트를 나타내는 의미하는 E_M(End MP3 세그먼트)과 E_V(End Voice 세그먼트), 현재의 블록이 마지막 블록이 아닌 중간 블록내의 마지막 세그먼트, 즉 잠시 끊임을 의미하는 EC_M (End Continue MP3 세그먼트)와 EC_V(End Continue Voice 세그먼트), 처음 블록이 아닌 바로 앞의 블록에 연속되는 다음 블록의 시작 세그먼트, 즉 계속을 의미하는 SC_M(Start Continue MP3 파일) 및 SC_V(Start Continue Voice 파일) 등의 여덟 가지의 세그먼트로 정의한다.

두 번째 필드(B13~B10)로 세그먼트가 차지하고 있는 위치를 나타내기 위하여 4-비트를 할당한다. 실제로 플래쉬 메모리 뱅크-0에서 뱅크-5를 나타내는 필드 비트(0000~0101)에 의하여 선택된다. 그러나 그 밖의 필드 비트에서는 플래쉬 메모리를 선택하는 것이 아니라 향후 용량에 따라 많은 종류의 스마트미디어 카드가 출시될 것을 예상하여 표 7과 같이 특정 스마트미디어 카드의 address로 정의한다.

스마트미디어 카드가 장착될 경우에는 스마트미디어

카드의 용량에 따라 high address의 상위 비트로 일부 혹은 전부를 표 7과 같이 사용한다. 32M-비트 스마트 미디어 카드가 사용된다면, 하위 1-비트를 high address로 사용한다. 또한 64M-비트 스마트미디어 카드가 사용된다면, 하위 2-비트를 high address로, 96M-비트 스마트미디어 카드가 사용된다면, 하위 3-비트를 high address로, 128M-비트 스마트미디어 카드가 사용된다면, 모든 4-비트를 high address로 사용한다.

세 번째 필드[B9-B3]로 high address를 나타내는 7-비트를 할당한다. 하나의 플래쉬 메모리 칩인, KM29U128T은 1024개의 세그먼트로 구성되어 있어 10-비트로 충분하다. 따라서 high address로 7-비트, mid-address로 3-비트로 구성한다. 그러나 현재 시판중인 스마트 미디어 카드는 2048개의 세그먼트로 구성된 것도 있고, 차후에는 더 많은 세그먼트로 구성될 것이다. 여기서는 2048개의 세그먼트로 구성된 스마트 미디어 카드만을 다루기로 한다.

FAT에는 모든 파일의 저장 위치만을 가지게 되지만, 파일 번호에 대한 정보는 프로그램으로 계산하여 얻게 된다. FAT 정보는 항상 저장된 순서로 정렬되어 있으므로 저장된 순서가 파일 번호가 된다. 이렇게 관리 프로그램에 의하여 추출되고, 추출된 파일 번호는 PC와 USB 제어기를 통한 PC에게 특정 파일을 올려 보내거나 PC로부터 파일을 내리 받을 때 사용된다. 이러한 파일 번호는 파일의 삭제, 추가에 따라 변화를 반드시 수정되어야 한다. 다음은 이러한 FAT에 저장되어 있는 특정 파일의 정보 맨 앞에는 데이터가 아닌 파일의 관한 여러 정보, 파일 이름, 파일의 종류, 파일의 특성, LCD 화면 하단에 표시할 가수 명, 노래 가사 등의 데이터를 1K-바이트에 할당한다. 나머지에는 순수 데이터를 저장한다.

Ⅳ. 플래쉬 메모리 관리

6개의 플래쉬 메모리칩 및 스마트미디어 카드를 포함한 메모리를 통합 관리하고 동시에 스마트미디어 카드에만 파일을 하는 저장하거나 갱신할 수 있는 방법을 고안한다. 스마트 미디어 카드에만 파일을 저장시킬 수 있게 되면, 향후 스마트 미디어 카드를 이동성이 좋은 저장 매체로 사용할 수 있게 되어 포터블 정보 단말기의 활성에 크게 기여할 것으로 믿는다. 따라서 스마트 미디어 카드에도 앞부분에 세 가지의 FAT를 별도로 할당시켜, 스마트 미디어 카드만을 관리할 수 있도록 한다.

우선 FLASH_FAT에는 플래쉬 메모리에 저장되어 있는 모든 파일의 위치 정보를 갖고 있다. 하나의 파일은 몇 개의 세그먼트에 나누어 저장될 수도 있다. 이러한 몇 개의 세그먼트도 반드시 인접한 세그먼트들로만 구성되어 있는 것이 아니라 몇 개의 세그먼트의 조합으로 분산시켜 저장될 수 있다. 연속으로 저장된 세그먼트들의 그룹이 하나의 블록인데 이러한 블록의 정보는 첫 번째 바이트에 블록의 시작_세그먼트의 위치 주소가 두 번째 바이트에 블록의 마지막_세그먼트의 위치 주소를 저장한다.

실제 예로 6개의 플래쉬 메모리칩으로 구성된 시스템에서 초기화된 경우, 즉 완전히 텅 비어져 있는 경우의 FLASH_FAT의 초기값은 표 8과 같고, FLASH_BFAT의 초기값은 표 9와 같고, 그렇지만 FLASH_VFAT의 초기값은 모든 영역이 텅 비어 있는 경우이므로 맨 앞부터 6개의 세그먼트를 제외한 전 영역을 하나의 비어 있는 블록으로 표 9와 같다.

1M-바이트의 파일이 하나의 블록으로 64개의 연속된 세그먼트에 저장되어 있는 경우에 해당되는 테이블의 내용은 표 11 및 표 12와 같다. 또한 1M-바이트의

표 8. 초기화된 FLASH_FAT
Table 8. Initialied FLASH_FAT.

위치	0	1	2	3	4	5	6	7	8	9	10	11	...	5120
값	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×	0×FF	0×FF	0×FF

표 9. 초기화된 FLASH_VFAT
Table 9. Initialied FLASH_VFAT.

위치	0	1	2	3	4	5	6	7	8	9	10	11	...	5120
값	0×00	0×0C	0×02	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 10. 초기화된 FLASH_BFAT
Table 10. Initialied FLASH_BAT.

위치	0	1	2	3	4	5	6	7	8	9	10	11	...	5120
값	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 11. 1-MB 크기의 파일이 연속으로 저장되어 경우의 FLASH_FAT
Table 11. FLASH_FAT in case 1-Mb file is stored in sequence.

위치	0	1	2	3	4	5	6	7	8	9	10	11	...	5120
값	0×00	0×0C	0×00	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 12. 1-MB 크기의 파일이 연속으로 저장되어 경우의 FLASH_VFAT
Table 12. FLASH_VFAT in case 1-Mb file is stored in sequence.

위치	0	1	2	3	4	5	6	7	8	9	10	11	...	5120
값	0×00	0×8C	0×02	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 13. 1-MB 크기의 파일을 하나 더 추가한 경우의 FLASH_FAT
Table 13. FLASH_FAT in case 1-Mb file is added.

위치	0	1	2	3	4	5	6	7	8	9	10	11	...	5120
값	0×00	0×0C	0×00	0×8A	0×00	0×8C	0×01	0×0A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 14. 1-MB 크기의 파일이 하나더 추가된 경우의 FLASH_VFAT
Table 14. FLASH_VFAT in case 1-Mb file is added.

위치	0	1	2	3	4	5	6	7	8	9	10	11	...	5120
값	0×10	0×8C	0×02	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

파일 2개가 두개의 블록으로 각각 64개의 연속된 세그먼트에 저장되어 있는 경우에 해당되는 각 테이블의 내용은 표 11 및 표 12와 같다.

- FLASH_BFAT 혹은 Smart_BFAT

FAT에는 데이터가 이미 저장되어 있는 세그먼트에 관한 정보를 가지고 있고 VFAT에는 빈 세그먼트에 관한 정보를 가지고 있다. 만약 특정 파일을 저장하거나 혹은 특정 파일을 삭제하려는 경우, FAT 혹은 VFAT에 등록되어 있는 세그먼트 정보를 수정하여야 하는 데, 생성/삭제 등의 명령어로 해당되는 데이터만을 바로 수정할 수 없다. 이미 언급하였지만, 플래쉬 메모리의 특성상 일단 세그먼트 단위로 완전히 삭제한 후에 다시 저장시켜야 되므로, 실제로 BFAT를 임시 저장 장치로 이용하여 FAT 혹은 VFAT 데이터를 수정한다. 일단 다른 메모리 영역으로 변치 않을 데이터는 BFAT로 그대로 옮기고, 변할 부분은 추가로 삽입시키거나 삭제한 다음, 이렇게 임시적으로 수정시킨 BFAT 데이터를 다시 원 위치시킨다. 이러한 BFAT를 중간 저장장치로 사용하는 FAT 혹은 VFAT 영역의 데이터를 수정하는 알고리즘을 정리하면 다음과 같다.

① BFAT가 완전히 비어있는 초기 조건에서, FAT 혹은 VFAT 자체의 데이터 중에서 변하지 않는 데이터는 그대로 읽어 BFAT에 그대로 저장시키고, 만약 새로 추가되는 데이터는 바로 BFAT의 해당 위치에 삽입한다. 삭제시킬 데이터는 BFAT에 저장시키지 말고 그대로 버린다.

② BFAT 영역에 저장이 완료되면, FAT 혹은 VFAT 영역을 완전히 삭제시킨다. 왜냐하면 BFAT 영역의 모든 데이터를 다시 FAT 혹은 BFAT에 옮기려면 먼저 FAT 혹은 VFAT를 완전히 비워야 하기 때문이다.

③ BFAT 영역의 모든 데이터를 FAT 혹은 VFAT 영역에 복사하는 형식으로 이루어진다.

④ BFAT를 다시 완전히 삭제시켜 놓아야 한다.

다음은 1-MB의 파일 2개가 앞 부분에 각각 64개의 세그먼트씩 저장되어 잇는 상태에서 두 번째 파일을 삭제시키는 경우의 과정을 위에서 언급한 법칙에 따라 표 15에서 표 41까지의 과정을 설명하고 있다.

표 15. 2개의 1-MB 파일이 저장된 경우의 FLASH_FAT
Table 15. FLASH_FAT in case two 1-Mb files are stored.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×00	0×0C	0×00	0×8A	0×00	0×8C	0×01	0×0A	0×FF	0×FF	0×FF	0×FF	0×FF

표 16. 2개의 1-MB 파일이 저장된 경우의 FLASH_VFAT
Table 16. FLASH_VFAT in case two 1-Mb files are stored.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×10	0×0C	0×02	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 17. 초기화되어 있는 FLASH_BFAT
Table 17. Initialized FLASH_BFAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

⇩

표 18. 첫번째 1-MB 파일을 삭제하려는 경우의 FLASH_FAT
Table 18. FLASH_FAT in case the first 1-Mb file is deleted.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×00	0×0C	0×00	0×8A	0×00	0×8C	0×01	0×0A	0×FF	0×FF	0×FF	0×FF	0×FF

표 19. 첫번째 1-MB 파일을 삭제하려는 경우의 FLASH_VFAT
Table 19. FLASH_VFAT in case the first 1-Mb file is deleted.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×10	0×0C	0×02	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 20. 첫 번째 1MB 파일을 삭제하려는 경우의 FLASH_BFAT
Table 20. FLASH_BFAT in case the first 1-Mb file is deleted.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×00	0×8C	0×01	0×0A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

* 삭제하려는 FIE의 나머지 FIE을 복사

⇩

표 21. 다시 초기화된 FLASH_FAT
Table 21. Re-initialized FLASH_FAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 22. FLASH_VFAT
Table 22. FLASH_VFAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×10	0×0C	0×02	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 23. FLASH_BFAT
Table 23. FLASH_BFAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×00	0×8C	0×01	0×0A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

⇩

표 24. FLASH_BFAT의 내용을 다시 복사해온 FLASH_FAT
Table 24. FLASH_VFAT which is copied from FLASH_BFAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×00	0×8C	0×01	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 25. FLASH_VFAT
Table 25. FLASH_VFAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×10	0×0C	0×02	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 26. FLASH_BFAT
Table 26. FLASH_BFAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×00	0×8C	0×01	0×0A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

⇩

표 27. FLASH_FAT
Table 27. FLASH_FAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×00	0×8C	0×01	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0xFF	0xFF

표 28. FLASH_VFAT
Table 28. FLASH_VFAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×10	0×8C	0×02	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 29. 다시 초기화된 FLASH_BFAT
Table 29. Re-initialized FLASH_BFAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF
값	0×10	0×8C	0×02	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 30. FLASH_FAT
Table 30. FLASH_FAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×00	0×8C	0×01	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 31. FLASH_VFAT
Table 31. FLASH_VFAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×10	0×8C	0×02	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 32. 새로운 데이터가 추가된 FLASH_BFAT
Table 32. FLASH_BFAT which is added with new Data.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×00	0×0C	0×00	0×8A	0×01	0×8C	0×02	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF

⇩

표 33. FLASH_FAT
Table 33. FLASH_FAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×00	0×8C	0×01	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 34. 다시 초기화된 FLASH_VFAT
Table 34. Re-initialized FLASH_VFAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 35. 새로운 데이터가 추가된 FLASH_BFAT
Table 35. FLASH_BFAT which id added with new Data.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×00	0×0C	0×00	0×8A	0×01	0×8C	0×02	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF

⇩

표 36. FLASH_FAT
Table 36. FLASH_FAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×00	0×8C	0×01	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF	0×FF

표 37. FLASH_BFAT를 그대로 복사한 FLASH_VFAT
Table 37. FLASH_VFAT which is copied from FLSH_VFAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×00	0×0C	0×00	0×8A	0×01	0×8C	0×02	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF

표 38. FLASH_BFAT
Table 38. FLASH_BFAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0×00	0×0C	0×00	0×8A	0×01	0×8C	0×02	0×8A	0×FF	0×FF	0×FF	0×FF	0×FF

⇩

표 39. 최종 FLASH_FAT
Table 39. the final FLASH_FAT.

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0x00	0+xC	0x01	0x8A	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF

표 40. 최종 FLASH_VFAT
Table 40. The final FLASH_VFAT.

위치	0	1x	2	3	4	5	6	7	8	9	10	...	5120
값	000	0x0C	0x00	0x8A	0x01	0x8C	0x02	0x8A	0xFF	0xFF	0xFF	0xFF	0xFF

표 41. 초기화 시킨 최종 FLASH_BFAT
Table 41. The final FLASH_BFAT .

위치	0	1	2	3	4	5	6	7	8	9	10	...	5120
값	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF

V 알고리즘 구현 및 고찰

본 절에서는 제안된 FAT, BFAT, VFAT 등을 이용한 파일 관리 알고리즘을 상세히 구현하고 이와 관련된 생성/삭제/탐색 등의 기능의 보장성, 메모리 효율 등의 고려 사항을 논의한다.

하나의 파일을 플래쉬 메모리의 여러 개의 비어있는 세그먼트의 집합인 하나의 빈 블록에 저장시키기에는 너무 사이즈가 큰 경우에는 다른 여러 빈 블록에 나누어 저장시켜야 한다. 이와 같이 하나의 파일을 빈 블록에 모두 저장시킬 수 없는 경우를 대비하여, 언제든지 FAT, BFAT, VFAT 등을 참조하여 저장하거나 읽도록 설계될 수밖에 없는 것은 이미 앞에서 언급하였다. 많은 시행 착오를 거치면서 이러한 FAT을 관리하고, 운용하는 기본 알고리즘의 원칙을 나열하고, 알고리즘을 제시한다.

첫째, 프로그램에서 어느 특정 파일을 접근하는 파라

미터로서 파일 번호를 사용할 수밖에 없다. 그렇다고 파일마다 파일의 번호를 불변 상수로 미리 정하여 놓기보다는, 상대적인 가변의 숫자로 지정하여 사용하는 것이 보다 효과적이다. 따라서 파일의 번호는 FAT에 저장되어 있는 FIE의 순서대로 프로그램으로 필요한 때마다 계산하여 사용한다.

둘째, 특정 블록에 있는 데이터를 읽거나 쓰거나 할 때, 한 단위의 데이터마다 번지를 지정하는 방식이 아니고 시작 번지를 지정해주고 연속해서 읽어오는 방식, 소위 번지 자동 증가 방식을 사용하기 때문에 쓰거나 읽는 횟수를 카운트하지 않으면 현재 가리키고 있는 주소를 알 수 없다. 또한 이와 같은 경우 현재 카운트되는 주소를 세그먼트 끝 번지와 비교하여 종료 여부를 파악하여야 한다. 이렇게 되면 프로세서에 부하가 상당히 많이 걸리게 된다. 이러한 점을 극복하는 방법으로 FAT를 참조하여 파일을 읽는 함수 및 현재 사용하고 있는 파일의 정보를 갱신하는 함수 등을 정의하여 사용하도록 한다.

셋째, 한 쌍의 FIE중에서 첫 번째 FIE로부터 블록의 시작_세그먼트 위치 정보를 얻어야 한다. 우선 최상위 2-비트로 현재 블록의 종류를 파악한다. 즉 현재의 블록이 파일의 시작 블록인지 혹은 연결된 중간 블록인지를 인지한다. 다음에는 현재 블록의 시작_세그먼트의 위치 파악을 위하여 3-비트의 뱅크 어드레스, 7-비트의 high address와 3-비트의 mid address로서 현재 가리키고 있는 세그먼트의 처음 위치를 알아내도록 한다. 당연히 low address는 0×00이 된다.

넷째, 한 쌍의 FIE중에서 두 번째 FIE로부터 블록의 마지막_세그먼트 위치 정보를 얻어야 한다. 마찬가지로 최상위 2-비트로 현재 블록이 마지막 블록인지 혹은 중간 블록인지를 파악한다.

다섯 번째, 그렇다면 실제로 FAT에 그대로 두고 파일에 대한 위치 정보를 획득할 수 있는지에 대하여 논의하여 보면, 만약 파일을 지우거나 추가하는 경우에는 실시간으로 수행 할 필요는 없기 때문에 중간 데이터 버퍼없이 플래쉬 메모리에서 직접 쓰고 읽는 과정이 수행이 될 수 있지만 MP3 음악 파일을 재생하는 경우에는 실시간으로 데이터를 계속 읽어야 하고, 현 블록에서 다른 블록 영역에도 접근을 해야할 경우에는 작업이 복잡해진다. 그래서 하나의 파일에 대한 정보만을 버퍼로 잡힌 데이터 메모리에 올려놓고 이를 참조하며 작업을 할 수 있다.

여섯 번째 플래쉬 메모리를 하드웨어 특성상 이미 저장되어 있는 장소에 덮어쓰기가 불가능하기 때문에 다른 메모리에 비해 특별한 빈 메모리 공간의 관리 정책이 요구된다. 이를 위하여 우선 고려할 사항은 새로운 데이터를 저장하기 위한 빈 공간을 확보하고 있어야 하는데, 이를 위하여 반드시 초기화시 또한 파일을 저장, 출력시 반드시 빈 공간 만들기를 수행하여야 한다. 물론 이 동작은 큰 부하가 걸리는 부분이다. 두 번째 고려할 사항은 전체 세그먼트가 불균등하게 사용되는 문제를 해결하여야 한다. 시간이 지날수록 세그먼트 사용 횟수의 불균등으로 인해 제한된 빈 공간 만들기 횟수를 넘어선 세그먼트는 더 이상 재활용을 할 수 없기 때문에 전체 저장 공간을 급속도로 감축하게 된다. 따라서 신뢰성 있는 시스템 구축을 위하여 반드시 세그먼트 사용 평준화가 성취되어야 한다.

다음은 위에서 언급한 여러 사항을 고려하여 파일을 읽거나, 특히 멀티미디어 데이터는 특성상 연속하여 읽어야 함으로, 다시 말하여 재생하거나, 연속해서 데이터를 저장시키는 알고리즘에 대하여 상세히 설명한다.

1. READ/WRITE/ERASE 루틴 특성

플래쉬 메모리에 read/write/erase 작동을 시켜주는 루틴의 함수가 비교적 복잡한데, 이러한 복잡성을 피할 수 없는 근본적인 요인을 먼저 살펴본다. 먼저 단순한 데이터를 read 하는 작동은 별로 어려움이 없지만, 문제는 연속으로 읽는 작동, 즉 오디오 데이터를 USB controller가 PC로 데이터를 올리거나 혹은 오디오 디코더가 연속으로 읽어 압축된 오디오 데이터를 복원시켜야 하는 경우를 반드시 고려하여야 한다. USB를 통과하여 PC로 올리는 경우에는 반드시 실시간으로 작동할 필요는 없다. 그러나 오디오 디코더로 데이터를 옮길 경우에는 실시간으로 작동하여야 하므로, 더욱이 디코더가 일반적으로 DSP에 의한 소프트에어 디코딩이기 때문에 실제는 대단히 빠르게 작동하도록 설계하여야 한다.

먼저 USB controller가 데이터를 64-바이트의 Packet 단위로 이동시킬 수 있는 것에 비하여 플래쉬 메모리에서는 512-바이트의 버퍼가 내재해 있어, 512-바이트 단위로 데이터가 이동되기 때문이다. 또한, USB에서 프로그램을 작성하는데 있어서는 데이터의 이동에 따라서 발생하는 Event에 준하여 작성을 해야 한다. 즉, 매번 64-바이트의 입출력이 발생할 때마다 루틴을 불러 데이터를 처리해야 하기 때문에, 대용량의

데이터를 처리할 경우, 각 함수에서 데이터를 카운트하여 매 상황에 맞는 루틴을 실행하도록 프로그램을 작성해야 한다.

다음에는 오디오 디코딩으로 데이터를 옮기는 방법으로 RISC CPU에서 먼저 읽어 자체내의 버퍼에 일단 옮겼다가 다시 오디오 디코더로 옮기는 방법으로 설계하여야 한다. RISC 칩의 내부 메모리 사이즈는 비교적 크지 않은 것이 일반적이다. 이동성이 우수한 소형 멀티미디어 정보 단말기에 DRAM, EEPROM 등을 외부에 장착시키는 하드웨어는 맞지 않으므로 버퍼 사이즈를 작게 설정된 조건하에서 알고리즘을 개발하여야 한다. 따라서 알고리즘의 단순화를 위하여 USB의 버퍼 사이즈인 64-바이트씩 옮기는 루틴으로 제한한다.

2. READ 알고리즘

파일을 재생할 때는 플래쉬 메모리로부터 데이터를 연속적으로 읽어 그 데이터 수를 카운트한다. 세그먼트 구간이 끝나기 전에 데이터 개수를 체크하면서 읽는다.

이러한 루틴은 64-바이트 단위로 처리가 완료된다. 따라서 64-바이트마다 read 루틴을 시작한다. 루틴 작동을 설명하여 보면, 처음에 FIE를 새로 읽어야 하는지를 판단한다. 만약 새로 읽을 필요가 있다면, FIE 정보를 읽어 온다. FIE 정보 중에서 세그먼트의 시작 주소를 함수 pSEGMENT에 저장한다. 다음에는 512-바이트씩 나누어 읽어야 하므로 512-바이트의 시작이 확인되면 read 작동을 시작한다. 64-바이트를 전부 읽으면 종료한다. 그 후에 매번 루틴이 돌 때마다 64-바이트씩 플래쉬 메모리를 읽어서 USB를 거쳐 데이터를 PC로 보낸다. 만약 중간에 pSGEMENT가 FIE End를 넘어설 경우 pFAT를 증가시키고, 다음의 FIE를 읽어온다. 또한, 읽어온 데이터가 File End를 나타낼 경우, 루틴을 종료한다.

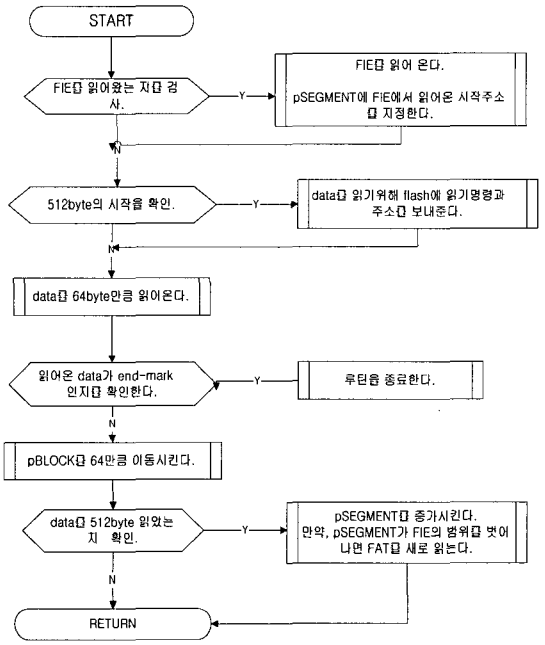

그림 1. READ 루틴 Flow Chart
Fig. 1. READ Routine Flow Chart.

3. WRITE 알고리즘

기본적인 동작은 READ 루틴과 동일하다. 단지 데이터를 읽어오는 부분이 쓰는 부분으로 바뀌어진 것이므로, READ 루틴과 거의 유사하다. 다른 점은 먼저 빈 공간의 정보를 갖고 있는 VFAT를 참조하여 빈 공간에 데이터를 저장하여야 한다. 따라서 VFAT로부터 FIE를 읽어와서 해당 세그먼트에 데이터를 저장한다.

표 42. Read 루틴 주요 변수
Table 42. Functions in Read Routine.

변 수	설 명
pFAT	FAT 세그먼트 내에서 바이트 단위로 현재의 위치 정보를 가지고 있다.
segFAT	FAT가 위치하고 있는 세그먼트 주소를 나타낸다. 0x0000이면 플래쉬 메모리, 0x0200이면 Smart Card를 지정한다.
pSEGMENT	데이터 정보가 위치하고 있는 세그먼트 주소를 나타낸다.
pBLOCK	데이터 세그먼트 내에서 바이트 단위의 현재의 위치를 나타낸다.
START	현재 USB에서 들어오는 값이 데이터인지 명령어인지를 구분한다. False이면 명령어, True라면 데이터를 나타낸다.
FIRST	FIE를 새로 읽어야 하는지를 나타낸다.
sFIE0	한 쌍의 FIE 중에서 첫 번째 FIE를 저장한다.
sFIE1	한 쌍의 FIE 중에서 두 번째 FIE를 저장한다.

- READ 루틴

```
void R_FLASH_UPLOAD_FILE ( void )
{
        if (FIRST == TRUE )
        {
                R_FLASH_READ_FIE (pFAT , segFAT );
                pSEGMENT = (sFIE0 & 0x3fff );
                FIRST = FALSE;
        } FAT의 FIE를 새로 읽어오는 부분.

        if ((pBLOCK&0x01ff) == 0 )
        {
                R_FLASH_SET_READ_STATUS ( pBLOCK , pSEGMENT );
        } 512byte의 시작일 경우 READ Command를 실행
        AUTOPTRH = 0x7e;
        AUTOPTRL = 0x00;
        for ( temp= 0; temp < 64 ; temp ++ )
        {
                AUTODATA = FLASH_INTERN ; 플래쉬에서 data를 읽어오는 부분.
        }
if (( IN2BUF[0]==0x66 )&&( IN2BUF[1]==0x66 )&&( IN2BUF[2]==0xBB )&&( IN2BUF[3]==0xBB ))
         {
                INOUT = OUT;
                START = FALSE;
        } 현재 들어온 data가 file END를 표현하는 지를 비교하는 부분
        EPIO[IN2BUF_ID].bytes = 64;
        pBLOCK = R_FLASH_SET_BLOCK_POINTER ( FOR64 , pBLOCK );
                if ( pBLOCK == 0 )
        {
                pSEGMENT++;
                if ( pSEGMENT > (sFIE1  & 0x3fff ) )
                {       FIRST = TRUE;    pFAT += 4; }
                현재 읽고있는 DATA가 FIE내에서 유효한지를 검사.
        } DATA를 16kB만큼 읽었다면 pSEGMENT를 증가시킨다.
}
```

이렇게 데이터를 저장한 후, 당연히 FAT 및 VFAT를 갱신하여야 한다. VFAT에서는 정보를 삭제하게 되고, FAT에는 정보를 추가하게 된다. VFAT에서 몇 바이트 데이터를 삭제하려고 하여도 세그먼트를 모두 삭제해야 한다는 것이다. 따라서, VFAT 갱신시 BFAT에 임시로 저장한 다음에 VFAT에 옮기는 방법을 사용한다.

파일을 플래쉬 메모리에 저장하는 기본 알고리즘을 설명하면, 먼저 플래쉬 메모리에 저장되어 있는 마지막 블록의 마지막_세그먼트의 바로 다음 세그먼트에서부터 시작된다. 만약 마지막 블록의 마지막_세그먼트에 이미 저장되어 있으면, 비어있는 세그먼트를 찾아서 저장한다.

① 새로운 파일의 저장 요청이 들어오게 되면 파일 관리 시스템은 빈 공간의 첫 번째 세그먼트를 찾기 위하여 우선 VFAT 영역에서 Vacant-Segment-Group에 관한 정보를 갖고 있는 첫 번째 FIE를 읽어온다.

② 읽어온 FIE에서 사용 가능한 세그먼트 영역을 찾고 그러한 세그먼트 영역에 데이터를 저장한다.

③ 새로 저장된 세그먼트에 관한 정보가 FAT 영역에 추가된다. 동시에 해당되는 세그먼트에 관한 정보를 VFAT로부터 삭제시킨다.

이런 순서를 거쳐 FAT 혹은 VFAT를 저장하게 되면 항상 저장된 순서대로 FIE가 정렬된다. 그리고 항상 하나의 파일은 시작부터 끝까지 몇 개의 블록 정보로 이루어지게 된다.

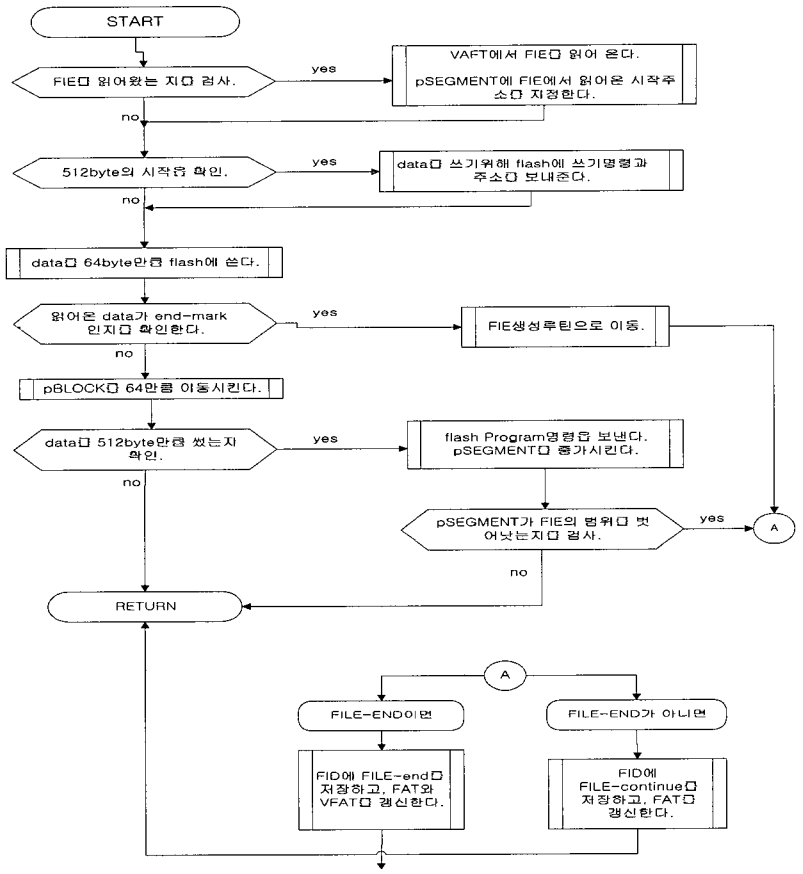

그림 2. WRITE 루틴 Flow Chart
Fig. 2. WRITE Routine Flow Chart.

표 43. Write 루틴 주요 변수
Table 43. Functions in WRITE Routine.

변 수	설 명
pFAT	FAT 세그먼트 내에서 현재 읽고 있는 FAT의 위치를 지정한다.
segFAT	FAT가 DATA의 몇 번째 세그먼트에 위치하는지를 나타낸다. 0×0000이면 Flash Memory, 0×0200이면 Smart Card를 지정한다.
pVFAT	VFAT 세그먼트 내에서 현재 읽고 있는 VFAT의 위치를 지정한다.
segVFAT	VFAT가 DATA의 몇 번째 세그먼트에 위치하는지를 나타낸다. 0×0000이면 Flash Memory, 0×0200이면 Smart Card를 지정한다.
pSEGMENT	현재 쓰고 있는 DATA의 세그먼트 위치를 나타낸다.
pBLOCK	현재 쓰고 있는 DATA의 세그먼트 내에서 Byte 단위의 위치를 나타낸다.
START	현재 USB에서 들어오는 값이 DATA인지 Command인지를 구분하는 인자. False라면 Command, True라면 DATA를 나타낸다.
sFIE0	현재 읽어온 첫 번째 FIE를 저장한다
sFIE1	현재 읽어온 두 번째 FIE를 저장한다.
wFIE0	현재 쓰기 위한 첫 번째 FIE를 저장한다.
wFIE1	현재 쓰기 위한 두 번째 FIE를 저장한다.
FID	현재 쓰고 있는 DATA의 타입을 결정한다.(00:Music,11:Voice)

- **WRITE 루틴**

```
void R_FLASH_DOWNLOAD_FILE ( void )
{
        if ( FIRST == TRUE )
        {
                R_FLASH_READ_FIE (pVFAT , segVFAT );
                pSEGMENT = (sFIE0 & 0x3fff );
                FIRST = FALSE;
        } VFAT를 읽어 온다.
        if ((pBLOCK & 0x01ff ) == 0)
        {
                R_FLASH_SET_WRITE_STATUS ( pBLOCK , pSEGMENT );
        } 512byte의 시작이면, write를 위해 address와 command를 보낸다.
        count = EPIO[OUT2BUF_ID].bytes;
        AUTOPTRH = 0x7d;
        AUTOPTRL = 0xc0;
        for ( temp = 0; temp < count ; temp ++ )
        {
                FLASH_INTERN = OUT2BUF[temp];
        }
        pBLOCK = R_FLASH_SET_BLOCK_POINTER ( FOR64 , pBLOCK );
        START = R_CHECK_END1 ( );
        Flie의 끝을 검사하고,FIE를 생성한다.
        EPIO[OUT2BUF_ID].bytes = 0;
        if ((pBLOCK & 0x01ff ) == 0 )
        {
HighC(CLE);
        FLASH_SendCommand (  FC_PROGRAM );
        FLASH_CheckBusy();
Delay (0xff);
        Delay (0xff);
        Delay (0xff);
        Delay (0xff);
        FLASH_SendCommand(FC_STATUS);
        LowC(CLE);
        temp = FLASH_INTERN;
        temp = 0;
        FLASH_ChipEnable ( 7 );
        } 512byte를 받으면, Write를 하는 부분. (CheckBusy문제로 인해 dealy를 추가.)
        if (START == FALSE )
        {
                R_FLASH_ERASE_BLOCK ( segVFAT );
                Delay ( 0xff);
                R_FLASH_COPY_BLOCK ( segBFAT , segVFAT );
                Delay ( 0xff);
                R_FLASH_ERASE_BLOCK ( segBFAT );
                return;
        } VFAT를 갱신한다.
        if ( pBLOCK == 0 )
        {
```

```
pSEGMENT++;
                if ( pSEGMENT > (sFIE1  & 0x3fff ) )
                {
                        wFIE1 = ( sFIE1 & 0x3fff );
                        wFIE0 = ( sFIE0 & 0x3fff ) + (FID << 14 );
                        if (FID > 1 ) {      wFIE1 += ( 1 << 14 ); FID = C_V; }
                        else            {    wFIE1 += ( 2 << 14 ); FID = C_M; }
                        R_FLASH_WRITE_FIE ( pFAT , segFAT );
                        pFAT += 4;
                        pVFAT += 4;
                        FIRST = TRUE;
                }
        }
```

현재 쓰고 있는 data가 VFAT의 허용범위를 넘는지를 판단.
넘는 다면 다음 VFAT의 FIE를 읽어온다.

4. ERASE 알고리즘

Erase 루틴은 Read/Write 루틴에 비하여 상대적으로 간단하다. 데이터를 삭제하게 되면, 데이터가 저장되어 있는 테이블인 FAT와 빈 공간에 대한 정보를 가지고 있는 BFAT를 갱신하고, 당연히 해당 세그먼트에서 데이터를 삭제하면 되는 기본 원리를 이용한다.

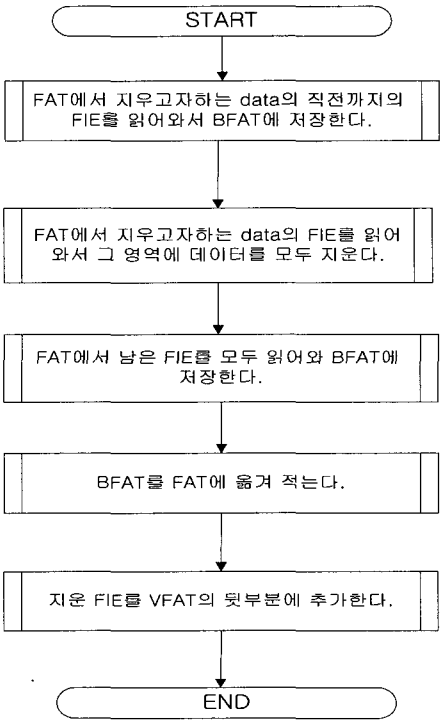

그림 3. ERASE 루틴 Flow Chart
Fig. 3. ERASE Routine Flow Chart.

FAT의 갱신은 FAT의 내용을 처음부터 차례로 BFAT에 옮기다가 삭제하고자 하는 부분의 값만을 건너뛰고, 마지막까지 BFAT에 그대로 옮겨 놓는 것이다. 결과적으로 BFAT는 FAT에서 지우고자 하는 부분을 제외한 모든 FAT값을 가지게 된다. 이것을 다시 FAT로 옮기는 것이다. VFAT의 경우는 간단하게 FAT에서의 삭제된 부분의 FIE를 VFAT의 뒷부분에 추가한다.

VI. 결 론

플래쉬 메모리가 갖고 있는 구조적, 전기적 특성으로 인하여 한 바이트 단위로 삭제시킬 수 없고, 블록 단위로만 삭제가 가능하고, 또한 포터블 소형 정보 단말기의 메임 프로세서는 별로 크지 않은 메모리를 갖게 될 수밖에 없는 환경에서 플래쉬 메모리로부터 파일을 생성, 재생, 관리를 위하여 세 종류의 FAT로 신뢰성 있고, 충분히 실시간으로 재생 가능한 결과를 얻어 플래쉬 메모리의 한계를 극복할 수 있었다. 더욱이 6개의 128M-비트 메모리칩 및 다양한 종류의 스마트미디어 카드로 구성된 비교적 큰 용량의 시스템에서도 잘 작동하였다. 스마트미디어 카드에만 파일을 저장 관리할 수 있도록 구현하였기 때문에 향후 FDD 대체용으로 미래의 소형 저장매체를 기대할 수 있게 되는 놀라운 효과가 기대된다. 3M-바이트 정도의 MP3 파일을 PC에서 RS-232C로 다운로드 하는 경우 20분 가량 걸리는데 반하여, USB로 12Mbps 속도로 다운로드 하는 경우에는 약 15초 가량 걸리며 이러한 고속으로 플래쉬 메모리에 입출력 작동함을 확인하였다..

- ERASE 루틴

```
void R_FLASH_ERASE_FILE  ( void )
{
        pVFAT = R_FLASH_SET_FIE_END ( pVFAT , segVFAT );
        temp = 0;
        while ( temp < pFAT )
        {
                R_FLASH_READ_4BYTE ( temp, segFAT );
                R_FLASH_WRITE_4BYTE ( temp , segBFAT );
                if ((FIELD[0] == 0xff)&&(FIELD[1] == 0xff)) break;
                temp += 4;
        } FAT에서 지우고자하는 file의 FIE의 직전까지의 값을 BFAT에 저장.
        while ( START == TRUE )
        {
                R_FLASH_READ_FIE (pFAT , segFAT );
                pSEGMENT = (sFIE0 & 0x3fff );
                while ( pSEGMENT <= (sFIE1 & 0x3fff ) )
                {
                        R_FLASH_ERASE_BLOCK ( pSEGMENT );
                        pSEGMENT++;
                }
                pFAT += 4;
                START = R_CHECK_END2 ();
                wFIE0 = sFIE0;
                wFIE1 = sFIE1;
                R_FLASH_WRITE_FIE ( pVFAT , segVFAT );
        }
FAT에서 FIE를 읽어서 그영역의 DATA를 삭제한다. 또, VFAT의 마지막에 현재의 FIE를 추가한다.
        while ( TRUE )
        {
                R_FLASH_READ_4BYTE ( pFAT , segFAT );
                R_FLASH_WRITE_4BYTE ( temp , segBFAT );
                if ((FIELD[0] == 0xff)&&(FIELD[1] == 0xff)) break;
                temp += 4;
                pFAT += 4;
        } FAT에서 지우고자하는 file의 FIE의 다음부터 마지막까지의 값을 BFAT에 저장.

        R_FLASH_ERASE_BLOCK ( segFAT ); // ERASE FAT !!
        R_FLASH_COPY_BLOCK ( segBFAT , segFAT ); // BFAT --> FAT !!
        R_FLASH_ERASE_BLOCK ( segBFAT ); // ERASE BFAT !!
} FAT를 갱신한다.
```

실제로 USB controller를 통하여 PC로부터 MP3 압축 파일을 다운로드 하는 실험하는 과정에서 많은 시행 착오를 겪어야만 하였다. PC에서 USB 인터훼이스를 거쳐 플래쉬 메모리로 신호가 통과하는 과정에서 작은 노이즈에도 신호가 불안하게 작동하였다. USB 인터훼이스 라인의 임피던스를 정밀하게 맞추고, 노이즈 제거 전용칩을 첨가하고, USB controller의 reset 회로를 안정화시키고, 플래쉬 메모리칩의 입출력 신호의 노이즈를 제거함으로서 제대로 알고리즘 소프트웨어를 개발 할 수 있었다.

향후 보다 전기적으로 안정된 플래쉬 메모리가 출현할 것으로 예상하고, 또한 한번에 삭제 가능한 블록 사이즈가 작아지게 되면, 보다 간단한 알고리즘으로 변형이 용이할 것이다.

참 고 문 헌

[1] Data Sheet, 16M×8 nit NAND Flash memory

Docu,ent, KM29U128T, Samsung Electronics, July, 1998.
[2] Data book, Flash mmeory I, Volume 1, Intel, march 1999.
[3] Data book, Flash mmeory II, Volume 2, Intel, march 1999
[4] 산업자원부, 최종보고서, 휴대용 정보 단말기(PDA) 개발에 관한 연구, 1998, 10,31
[5] 산업자원부, 1차년도 중간보고서, 휴대용 정보 단말기(PDA) 개발에 관한 연구, 1995, 10,31

저 자 소 개

朴 寅 圭(正會員)
1948년생, 1972년 서울대학교 전자공학과 졸업, 1982년 The Ohio State University EE 석사 졸업, 1986년 Purdue University EE PhD. 1986~1987 University of Houston 조교수, 1987~1988 삼성전자 종합 연구소 제 5연구실장, 1988년~현재 홍익대학교 전자전기공학부 주관심 분야: 컴퓨터 구조, 멀티미디어 시스템 등임