CHAPTER 3

FAT 파일시스템

01. FAT 파일시스템 소개
02. FAT 파일시스템 구조
03. 부트 레코드
04. 부트 레코드 실습
05. 예약된 영역
06. FAT 영역
07. 데이터 영역
08. Long File Names
09. 데이터 영역 실습
10. FAT 파일시스템 실전
11. FAT 파일시스템 최종 실습

개인용 컴퓨터에서 가장 많이 쓰는 파일시스템을 무엇일까? 또한 가장 많은 운영체제에서 지원하는 파일시스템은 무엇일까? 정답은 FAT 파일시스템일 것이다. FAT 파일시스템은 개인용 컴퓨터에서 가장 많이 쓰이는 파일시스템이며, Microsoft의 MS-DOS와 Windows 전 계열은 물론이고 Linux, 다수의 Unix, Mac OS에서도 지원하고 있다. 이만하면 개인용 컴퓨터 환경에서 가장 유명한 파일시스템이라고 부를 만하다. 하지만 많은 사용자가 쓰고 있다고 해서 FAT 파일시스템이 가장 좋은 파일시스템이라고 보기는 어렵다. FAT 파일시스템은 세상에 나온 지 벌써 20년도 넘은 구형 파일시스템이며 최신의 다른 파일시스템들이 가지고 있는 파일 보안이나 저널링 기법과 같은 고급 기능이 포함되어 있지 않다. 그럼에도 불구하고 임베디드 개발자들이 실무에서 구현하기에는 FAT 파일시스템이 가장 매력적일 것이다. 왜냐하면 FAT 파일시스템은 다른 파일시스템에 비해 구조가 비교적 단순하기 때문이다.

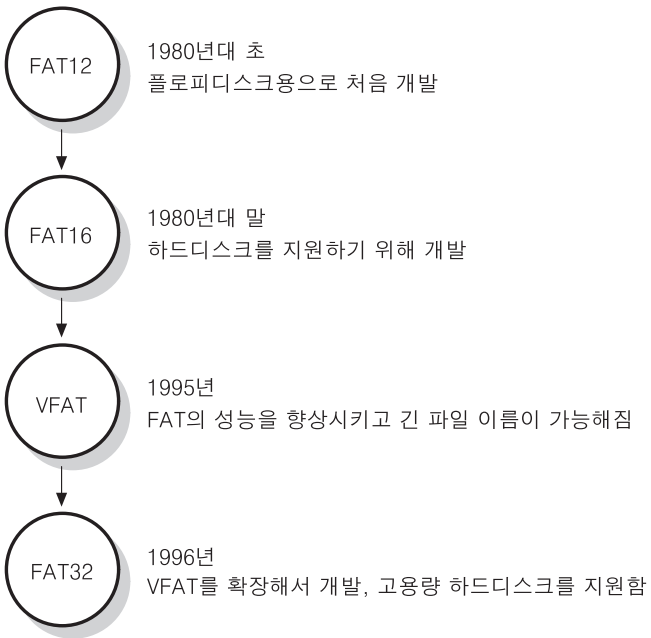
FAT 파일시스템으로 개발을 하게 되면 쉽게 Windows 운영체제와 호환이 되는 제품을 만들 수 있다는 이점이 있다. 반면, 단순한 구조를 가지고 있는 만큼 배드 섹터 처리나 파일 보안 기능이 다른 파일시스템에 비해 취약하다. 이동을 하면서 HDD에 읽기/쓰기를 해야 하는 장비에 FAT 파일시스템을 구현하려는 개발자들은 이런 점을 유의해야 한다.

이번 장에서는 FAT 파일시스템의 구조와 FAT16과 FAT32의 차이점, 임베디드 장치에 FAT 파일시스템을 포팅하기 위한 관련 기술을 익힐 것이다. 이 책에서도 이번 장이 가장 활용도가 높으니 꼭 익히고 넘어가기 바란다.

01 FAT 파일시스템 소개

본격적으로 FAT 파일시스템의 구조를 분석하기 전에 FAT 파일시스템을 전반적으로 살펴보자. 이 절에서는 FAT 파일시스템의 역사와 특징을 살펴보고 윈도우는 FAT 파일시스템을 어떻게 이용하고 있는지 살펴볼 것이다.

FAT 파일시스템은 그 역사가 길며, 긴 역사 동안 여러 번의 변화가 있었다. 이번 절에서는 FAT 파일시스템의 변천사에 대해 살펴보자. 이런 변화의 과정에 대해 살펴보면 FAT 파일시스템을 이해하는 데 좀 더 도움이 될 것이다.



[그림 3-1] FAT 파일시스템의 변화 과정

FAT 파일시스템의 탄생

FAT 파일시스템은 1976년에 Microsoft의 빌 게이츠에 의해 최초로 구현되었다. 빌 게이츠가 FAT 파일시스템을 구현하게 된 목적은 자신의 회사 제품인 BASIC에서 플로피디스크를 관리하는 데 이용하기 위해서였다. 이것이 FAT 파일시스템의 최초 버전인 FAT12의 시작이

다. 하지만 Microsoft의 BASIC은 운영체제가 아닌 프로그래밍 언어였기 때문에 FAT12는 널리 쓰이지 않았고 단지 BASIC의 한 기능이었을 뿐이었다.

이런 FAT 파일시스템이 정작 PC 환경에서 널리 쓰이게 된 계기는 Tim Paterson이라는 사람이 QDOS라는 운영체제를 개발하면서 QDOS가 이용할 파일시스템으로 FAT 파일시스템을 선택했기 때문이다. 그는 당시 존재하던 여러 파일시스템을 비교한 끝에 FAT 파일시스템을 선택했는데 그 이유는 FAT 파일시스템이 매우 단순한 구조를 가지고 있었기 때문이었다. Tim Paterson이 보기에 FAT 파일시스템은 플로피디스크를 관리하기에 매우 적절해 보였고 조금만 수정한다면 32MB 이상도 관리할 수 있을 것이라고 생각했다. 그는 32MB 정도면 PC 환경에서는 절대로 충분한 용량이라고 믿었다.

여기서 잠깐

컴퓨터 역사상 이런 착각을 하는 일화가 몇 가지가 더 있다. IBM은 개인용 컴퓨터의 성능은 16bit CPU 컴퓨터면 충분하다고 판단하는 바람에 최초 32bit CPU 컴퓨터 출시를 컴팩(Compaq)에게 빼앗겼고, Microsoft는 개인용 컴퓨터의 메모리 크기는 640KB면 충분하다고 판단하는 바람에 MS-DOS를 사용했던 많은 유저들을 고통으로 몰아넣었다.

1980년 QDOS가 세상에 발표됨과 동시에 FAT 파일시스템도 같이 소개된다. 이것이 FAT 파일시스템의 공식적인 시작이다. 이후 QDOS는 여러 이유에 의해 창조자의 손을 떠나 IBM과 Microsoft로 넘어가게 된다. IBM과 Microsoft는 QDOS의 이름을 각각 PC-DOS와 MS-DOS로 바꾸고 x86 PC의 운영체제로 탑재시켰다. IBM x86 PC가 엄청난 성공을 거두며 PC 시장의 표준처럼 자리 잡게 되고 다른 PC 메이커들도 IBM 호환 기종을 만들어내자 x86 PC의 운영체제인 MS-DOS도 대중적인 운영체제가 된다. 이렇게 IBM 호환 PC의 성공을 등에 업고 FAT 파일시스템은 자연스럽게 PC에서 가장 많이 쓰이는 파일시스템으로 자리를 잡게 된다.

FAT12

1980년에 나온 FAT 파일시스템의 최초 버전은 다른 버전들과 구분하기 위해 FAT12라고 불린다. FAT12는 플로피디스크에 파일을 저장하기 위해 개발되었다. 당시의 PC는 HDD가 없고 5.25인치 플로피디스크가 대세였던 시기였으므로 당연한 것이었다. 최초로 발표된 FAT12는 구조가 매우 단순해서 디렉토리(directory)라는 개념도 없었다. 1983년 MS-DOS 2.0이 나오고 나서 FAT12에 계층형 디렉토리가 지원되게 되고, 현재 FAT 파일시스템의 기

본 구조를 거의 갖추게 된다. 1984년에 발표된 MS-DOS 3.0에서는 FAT12를 이용해서 IBM PC-AT에서 HDD를 사용할 수 있게 했고, 5.25인치 1.2M 양면 플로피디스크를 사용할 수 있도록 했다.

FAT16

HDD 기술의 발달로 개인용 컴퓨터에 HDD를 장착하는 경우가 점점 많아지자, HDD에서 사용할 파일시스템의 필요성이 증가하게 된다. Microsoft는 시장의 요구에 따라 1988년에 MS-DOS 4.0을 통해서 FAT12의 다음 버전인 FAT16을 발표한다. FAT 파일시스템은 FAT16으로 확장되어서야 비로소 HDD에 사용할만한 파일시스템이 된다. 구조적으로 FAT16은 FAT12와 거의 동일하며 단지 FAT16에서는 클러스터(Cluster)를 표현하는 비트 수가 12개에서 16개로 늘어난 덕에 이론적으로 최대 2^{16}인 65,535개의 클러스터를 표현할 수 있게 되었다는 정도가 차이점이다. 이는 클러스터 크기를 32KB로 할 경우 2GB까지 표현할 수 있으며 이 정도면 그 당시로서는 평생 써도 남을 정도로 충분한 용량 표현이었다. Microsoft는 다른 여러 FAT 문제점(예를 들어 파일명 길이 제한 문제)은 그렇다 치더라도 용량 표현 한계 때문에 더 이상 고민하지 않아도 된다는 기쁨에 사로잡혔을 것이다.

VFAT(Virtual FAT)

1995년 Microsoft는 자사의 차세대 운영체제인 Windows 95에 FAT 파일시스템을 탑재하면서 그 성능이나 기능을 향상시켰다. 정확하게 따지자면 이런 FAT의 향상 버전을 VFAT 또는 FAST FAT라고 불러야 맞지만 많은 사람들은 똑같이 FAT 파일시스템이라고 부른다. VFAT 파일시스템으로 향상되면서 변경된 내용을 살펴보면 우선 32bit 보호 모드(Protected Mode)에 적합하게 코드를 재작성하여 성능을 향상시켰고, 독점 모드(Exclusive Mode)를 추가하여 동시에 여러 프로그램이 같은 파일을 접근할 경우에 대비하였다. 하지만 이런 것들은 파일시스템의 구조 변화라기보다는 운영체제의 파일시스템 처리 성능이 개선된 것이다. 이런 것보다 우리가 주목해야 할 가장 중요한 변화는 FAT가 VFAT로 확장되면서 LFNs(Long File Names)를 지원한다는 것이다. 기존의 FAT 파일시스템은 파일명이 최대 8Byte, 확장명은 최대 3Byte가 고작이었다(이것을 8.3 File Naming이라고 한다). 게다가 파일명과 확장명 모두 대문자만 가능하도록 설계되어 있었다. Microsoft는 이러한 단점을 개선하기 위해

서 LFNs를 고안했다. LFNs 방식은 최대 255자까지 파일명을 적을 수 있으며 LFNs를 지원하지 않는 이전 버전의 DOS와도 하위 호환성을 가진다. 이런 LFNs 기능이 추가된 VFAT 파일시스템이 최초로 Windows 95에 적용된 덕분에 Windows 95는 파일명의 길이 제한으로부터 자유로울 수 있게 되었다.

LFNs 덕분에 파일명의 길이가 255자까지 늘어났다고는 하지만 VFAT로 향상된 FAT16 파일시스템의 최대 용량은 여전히 2GB 정도밖에는 되지 않았다. 이 시기에는 이미 용량이 2GB가 넘는 HDD들이 출시되고 있었으므로 Microsoft는 다시 한 번 FAT 파일시스템의 용량 표현 문제 때문에 고민에 빠지게 된다.

FAT32

Windows 95 사용자가 2GB가 넘는 하드디스크를 사용하고자 한다면 FAT16의 용량 표현 한계 때문에 어쩔 수 없이 파티션을 나눠서 사용하는 불편을 감수해야만 했다. 때문에 Microsoft는 1996년에 Windows 95 OSR2를 발표하면서 FAT32를 선보인다. FAT32는 VFAT를 기반으로 수정하였으며 클러스터를 표현하는 bit를 32개로 늘렸다. 하지만 32bit 중 최상위 4bit는 예약 영역으로 사용되지 않으므로 총 28bit를 이용해서 클러스터를 표현한다. 이는 클러스터 크기를 16KB로 한다면 이론상 최대 4TB까지 가능한 용량이다. 실제로도 Windows는 FAT32를 2TB까지 인식할 수 있지만 여러 이유 때문에 FAT32의 최대 용량을 32GB로 제한하고 있다. 다만 약간의 제약 아래 다른 장치가 포맷을 한 32GB 이상의 FAT32 볼륨을 인식한다. 만약 Windows에서 용량이 32GB 이상 되는 저장장치를 포맷한 후 사용하고 싶다면 파티션을 분할하든지 FAT32를 포기하고 NTFS로 사용해야 한다.

FAT32는 FAT 파일시스템의 마지막 버전이 될 것이다. Microsoft는 FAT32를 마지막으로 더 이상 FAT 파일시스템의 다음 버전은 없을 거라고 말한다. Microsoft는 낡고, 그다지 성능이 좋지 않은 FAT 파일시스템 대신 NTFS에 주력하고 있다. FAT 파일시스템은 오랫동안 차지했던 영광의 자리를 NTFS에게 넘겨주고 있는 것이다.

FAT 파일시스템들의 비교

앞에서 살펴보았듯이 FAT 파일시스템은 20년 넘게 사용되어오면서 FAT12, FAT16, FAT32

순으로 계속 버전이 향상되어 왔다. 각각의 FAT 파일시스템들은 그 구조가 매우 비슷하지만, 엄연히 다른 파일시스템이다. 각각의 FAT 파일시스템을 비교해 보면서 특징을 살펴보자. 이런 특징들을 살펴봄으로써 위에서 배운 내용들을 정리할 수 있고 각각의 파일시스템의 차이점에 대해 좀 더 명확히 할 수 있다.

[표 3-1] FAT 파일시스템 비교

구분	FAT12	FAT16	FAT32
사용 용도	플로피디스크용	저용량 하드디스크	고용량 하드디스크
클러스터 표현 비트 수	12bit	16bit	32bit(28bit만 사용)
최대 클러스터 개수	4,084개	65,524개	약 2^{28}개
최대 볼륨 크기	16MB	2GB	2TB
파일의 최대 크기	볼륨 크기만큼	볼륨 크기만큼	4GB
디렉토리당 최대 파일 개수	X	65,535개	65,535개
루트 디렉토리의 파일 개수 제한	있음	있음	없음

여기서 잠깐

위 표 중에서 각각의 FAT 파일시스템의 최대 클러스터 개수가 논란이 될 수 있다. 이론상 FAT12의 최대 클러스터 개수는 4,096개이며, FAT16은 65,535개이다. 많은 문서들이 최대 클러스터 개수에 대해 이론상의 개수로 표시하고 있어서 더욱 혼란스러운데, Microsoft에서 발표한 정식 문서에는 최대 클러스터 개수를 위 표와 같이 표시하고 있다.

FAT12, FAT16, FAT32 파일시스템간의 가장 큰 차이점은 클러스터 표현 비트 수의 차이에 따른 최대 클러스터 개수가 다르다는 점이다. 표현 가능한 클러스터의 개수가 많아질수록 더 많은 용량을 표현할 수 있으므로 FAT32가 가장 큰 용량을 표현할 수 있다. 그 외에 살펴볼 특징으로는 루트 디렉토리의 파일 개수 제한이다. FAT12와 FAT16의 경우 루트 디렉토리의 파일 개수에 제한을 두고 있는 반면 FAT32는 루트 디렉토리의 경우라도 일반 디렉토리와 동일하게 디렉토리 내 최대 파일 개수까지 저장이 가능하다.

FAT 파일시스템의 호환성

각각의 파일시스템들의 호환성에 대해 살펴보자. 어떤 파일시스템을 구현할 것인지를 선택하는 데에는 성능만큼이나 호환성도 생각해 봐야 한다. 그래야 좀 더 많은 사용자들을 확보할 수 있기 때문이다.

[표 3-2] 파일시스템과 Windows와의 호환 여부

운영체제	FAT12	FAT16	FAT32	NTFS
MS-DOS	●	●		
Windows 95	●	●		
Windows 95 OSR2	●	●	●	
Windows 98	●	●	●	
Windows Me	●	●	●	
Windows NT 4.0	●	●		●
Windows 2000	●	●	●	●
Windows XP	●	●	●	●

위 표는 Windows가 지원하는 파일시스템을 나타낸 것이다. 가장 호환성이 좋은 파일시스템은 DOS를 포함한 모든 윈도우가 지원하는 FAT12와 FAT16 파일시스템이다. FAT12는 플로피디스크용이므로 고려대상에서 제외한다면 FAT16이 가장 호환성이 좋은 파일시스템이 된다. FAT16 파일시스템을 구현하면 운영체제에 구애받지 않아도 되는 장점이 있다. 하지만 FAT16은 용량 표현이 작다는 단점이 있다. 용량이 2GB를 넘는 파티션을 FAT16으로 포맷해서 사용한다는 것은 조금 무리가 있다. NTFS는 성능이나 여러 면에서 가장 좋은 파일시스템이긴 하지만 호환성이 낮다.

그러므로 지금까지 살펴본 내용을 정리해서 생각해 봤을 때 FAT 파일시스템 중에서는 FAT32가 가장 무난한 선택일 듯하다. FAT16에 비해서 최대 표현 용량도 큰 편이며, 루트 디렉토리의 파일 개수 제한도 없다. 하지만 FAT32도 몇 가지 제약이 있다. FAT32는 MS-DOS, Windows 95와는 호환되지 않는다. 또한 나중에 살펴보겠지만 FAT32를 사용하려면 저장장치의 용량이 최소한 32MB 이상은 되어야 한다. 용량이 32MB 이하인 CF 카드나 USB Memory Stick에 FAT 파일시스템을 사용하려면 FAT16 외에는 방법이 없다.

클러스터 크기와 슬랙(Slack) 문제

여기에서는 클러스터 크기와 볼륨 크기와는 어떤 관계가 있는지, 그리고 클러스터 크기에 따른 장단점은 무엇인지에 대해서 살펴보자. 이런 내용을 살펴봄으로써 좀 더 효율적인 파일시스템을 구현하는 데 도움이 된다.

[표 3-3] Windows 기준의 클러스터 크기

볼륨 크기	FAT16 클러스터 크기	FAT32 클러스터 크기	NTFS 클러스터 크기
16MB ~ 32MB	512Byte	지원 안 함	512Byte
32MB ~ 64MB	1KB	512Byte	512Byte
64MB ~ 128MB	2KB	1KB	512Byte
128MB ~ 256MB	4KB	2KB	512Byte
256MB ~ 512MB	8KB	4KB	512Byte
512MB ~ 1GB	16KB	4KB	1KB
1GB ~ 2GB	32KB	4KB	2KB
2GB ~ 4GB	64KB(1)	4KB	4KB
4GB ~ 8GB	지원 안 함	4KB	4KB
8GB ~ 16GB	지원 안 함	8KB	4KB
16GB ~ 32GB	지원 안 함	16KB	4KB
32GB ~ 2TB	지원 안 함	인식 가능(2)	4KB

(1) NT, 2000, XP에서 클러스터 크기를 64KB로 해서 2~4GB의 볼륨을 FAT16으로 포맷할 수 있다. 하지만 일부 응용 프로그램에서는 64KB 클러스터 볼륨을 인식하지 못하는 경우가 있으므로 호환성을 생각한다면 2GB가 넘는 볼륨은 FAT16이 아닌 다른 파일시스템을 사용하기를 권장한다.

(2) Windows는 32GB가 넘는 볼륨에 FAT32를 이용해서 포맷하는 것을 지원하지 않는다. 하지만 다른 장치가 포맷한 32GB가 넘는 FAT32 볼륨은 인식할 수 있다.

여기서 잠깐

위 표에 나온 볼륨 용량에 따른 클러스터 크기는 Windows 2000을 기준으로 작성되었다. 다른 버전의 Windows에서는 위의 기준과 다를 수 있다.

위의 표는 Windows가 볼륨 용량에 따라 할당하는 클러스터의 크기 변화를 보여준다. 여러분들이 FAT 파일시스템을 구현할 때에는 위의 규칙에 따라 클러스터 크기를 Windows와 똑같이 맞추지 않아도 되지만, Windows의 규칙대로 구현하는 것이 호환성을 높이는 데 도움이 된다. 만약 여러분이 위 표에서 '지원 안 함'이라고 써있는 부분에 해당하는 영역을 설계했다면(예를 들어 FAT16을 이용해서 클러스터의 크기를 128KB로 잡아서 4~8GB의 볼륨을 사용하도록 설계했다면) Windows가 친절하게 여러분이 만든 FAT 파일시스템을 인식해 주길 바라는 건 곤란하다.

클러스터 크기가 크거나 작음에 따라서 장단점이 있다. 우선 클러스터의 크기가 작은 경우의 장점은 클러스터 할당 때문에 버려지는 용량이 적다는 것이고, 단점은 FAT 영역의 크기가 커진다는 것이다.

반면에 클러스터의 크기가 큰 경우의 장점은 FAT 영역의 크기가 작다는 점과 클러스터의 크기가 작은 경우에 비해 파일당 할당하는 클러스터 수가 적기 때문에 그에 따른 작업의 오버헤드(overhead)가 적다는 점이다. 단점은 크기가 작은 파일이 많은 경우 버려지는 용량이 많아진다는 점이다. 이렇게 버려지는 부분을 슬랙(slack)이라고 한다.

아래 표는 위에서 설명한 클러스터 크기에 대한 장단점을 요약한 것이다.

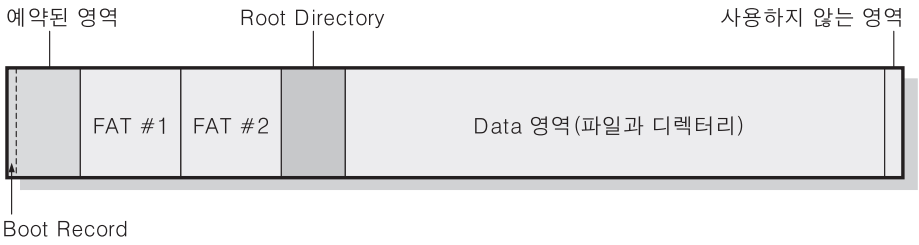
[표 3-4] 클러스터 크기에 따른 장단점

구분	장점	단점
클러스터 크기가 작을 때	버려지는 용량이 적다.	FAT 영역이 커진다.
클러스터 크기가 클 때	FAT 영역이 작다. Cluster 처리 부담이 적다.	버려지는 용량(Slack)이 많다.

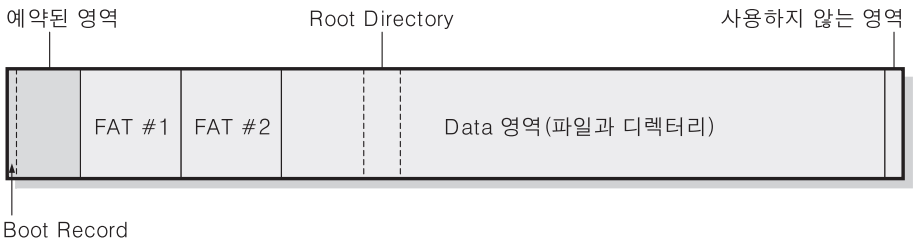
그래도 굳이 클러스터 크기는 작은 것이 좋은지 큰 것이 좋은지를 따져보자면, 클러스터 크기는 작은 것이 낫다. 클러스터를 크게 사용함으로써 버려지게 되는 용량이 꽤나 많기 때문이다. 예를 들어 4GB 용량의 파티션을 FAT16으로 포맷했다고 한다면 Windows는 클러스터 크기를 무려 64KB로 할당할 것이다. 이 파티션에 사용자가 1Byte 용량밖에 안 되는 파일을 10개 저장했다고 가정한다면 무조건 한 클러스터에는 한 파일만을 담아야 하므로 10개의 클러스터를 사용해야 한다. 64KB X 10개 = 640KB가 되는데, 총 용량이 겨우 10Byte밖에 안 되는 파일 10개를 저장하기 위해서 639KB를 버리는 것은 너무나도 아까운 일이다. 때문에 클러스터 크기가 작은 편이 용량을 효율적으로 사용하는 데 더 좋은 방법이며, NTFS 파일시스템에서는 이러한 사실 때문에 클러스터 할당 크기를 작게 해놓았다. 단, 영상 데이터만을 저장한다거나 대용량 파일만을 저장한다면 굳이 클러스터를 작게 할 필요는 없다.

02 FAT 파일시스템 구조

FAT 파일시스템은 데이터를 관리하기 위해서 하드디스크를 여러 영역으로 나누어서 사용하고 있다. 이 절에서는 FAT 파일시스템이 하드디스크를 어떻게 나누어서 사용하고 있는가와 각각의 영역에 대한 기본적인 내용을 살펴본다.



[그림 3-2] FAT16 파일시스템의 구조



[그림 3-3] FAT32 파일시스템의 구조

FAT 파일시스템의 구조

FAT16과 FAT32는 기본적인 구조에서 거의 비슷하나 다른 점이 몇 가지 있다. 위 그림에서 보듯이 가장 큰 차이점은 FAT16은 루트 디렉토리 영역이 따로 존재한다는 것이다. FAT16에서 루트 디렉토리 영역은 하위 디렉토리들과 달리 반드시 FAT #2 영역 뒤에 와야 하지만 FAT32는 루트 디렉토리 역시 일반 디렉토리 중의 하나로 간주하기 때문에 데이터 영역 어디에 오든지 상관이 없게 되었다.

참고로 위의 그림에서는 편의상 부트 레코드 영역이나 예약된 영역이 실제 비율보다 크게 그려져 있는데, 실제로 각 영역별 할당 비율을 보면 부트 레코드나 예약된 영역은 매우 작기 때문에 정확히 비율대로 그림을 그리면 거의 안 보이게 된다. 때문에 설명에 도움이 되도록 크게

그려놓았다. 각 영역에 대해서는 다음 절부터 하나하나 자세하게 살펴볼 것이다. 일단 여기에서는 전체적인 내용을 간단히 살펴보자.

부트 레코드(Boot Record)

이 영역은 볼륨의 첫 번째 섹터를 의미한다. 또한 예약된(Reserved) 영역의 첫 번째 섹터를 의미하기도 한다. 이 영역에는 Windows를 부팅시키기 위한 기계어 코드와 FAT 파일시스템의 여러 설정 값들이 담겨 있다. 이 영역은 BIOS Parameter Block(BPB)이라고도 한다. 이 영역은 크기가 1섹터밖에 안 되는 작은 영역이지만 Windows가 FAT 파일시스템을 인식하는데 가장 중요한 역할을 담당한다. Windows는 볼륨을 인식하려고 할 때 우선 부트 레코드를 읽어서 분석을 하기 때문이다. 만약 볼륨의 내용이 완벽히 살아 있어도 부트 레코드 섹터를 실수로 지워버린다면 Windows는 해당 볼륨을 인식할 수 없다. 이 영역에 대해 한 가지 주목해야 할 점은 Boot Record 형태가 FAT16과 FAT32가 약간 다르다는 점이다. 이것에 대해서는 나중에 자세히 살펴볼 것이다.

예약된(Reserved) 영역

이 영역은 미래를 위해 예약해 놓은 영역이다. 보통 FAT16인 경우에는 1섹터를, FAT32인 경우에는 32섹터를 할당한다. FAT16의 경우 예약된 영역의 크기가 1섹터라면 부트 레코드가 1섹터를 차지하므로 부트 레코드 다음에 바로 FAT 영역이 오게 된다. 이 영역은 미래를 위해 예약되어 있는 영역으로 사용되지 않는 영역이다. 만약 여러분이 자신만의 특별한 정보를 기록하고 싶다면 이 영역이 적절할 것이다. 단 FAT32의 경우 예약된 영역 안에 몇 개의 섹터를 사용하여 정보를 저장하고 있으므로 그것과 겹치지 않게 유의해야 한다.

FAT #1 영역과 FAT #2 영역

FAT 영역은 클러스터들을 관리하는 테이블이 모여 있는 공간이다. FAT 영역을 통해서 어떤 클러스터가 비어 있는지, 어떤 파일에 어떤 클러스터가 연결되어 있는지를 알 수 있게 된다. 만약 이 영역에 조금이라도 손상이 가게 된다면 돌이킬 수 없는 큰 재앙이 온다. Microsoft

엔지니어들도 이러한 사실을 잘 알고 있었는지, FAT 영역의 백업본을 한 개 이상 만들어 두게 해놓았다. FAT #1 영역과 FAT #2 영역은 동일한 내용을 담고 있다. FAT #2 영역은 일종의 백업본으로 FAT #1에 손상이 갔을 경우를 대비해서 만들어 놓은 것이다. FAT 영역은 최소 1개 이상 되어야 하며, 3개를 만들어도 된다. 전 세계의 많은 FAT 코드는 보통 2개의 영역을 만든다.

루트 디렉토리 영역(FAT16에만 존재)

FAT16에서는 루트 디렉토리의 위치가 무조건 FAT #2 영역 뒤쪽으로 고정되어 있다(위치는 고정되어 있지만 크기는 가변적일 수 있다). 이 방식은 루트 디렉토리가 어디에 있는지를 조사할 필요가 없어서 개발의 편의성이 증가하게 되는 장점이 있지만, 루트 디렉토리의 파일 개수 제한 등 단점이 있어서 FAT32에서는 구조가 바뀌었다. FAT32에서는 루트 디렉토리가 데이터 영역 어디에 와도 상관이 없으며, 단지 그 위치를 부트 레코드에 기록하고 있을 뿐이다. 이 방법의 유일한 단점은 부트 레코드가 망가지면 루트 디렉토리를 찾아내기가 매우 힘들다는 것이다. 이러한 문제에 대한 대책으로 대부분의 FAT 코드들이 FAT32인 경우라도 FAT #2 영역 바로 뒤에 루트 디렉토리를 위치시킨다. 그렇기 때문에 부트 레코드가 망가져서 루트 디렉토리의 위치를 알 수 없게 되어버린 경우에도 간단히 찾을 수 있게 된다.

데이터 영역(파일과 디렉토리)

이 영역에는 파일 또는 디렉토리가 저장되어 있다. 이 영역은 지금까지 봐왔던 영역들이 전부 섹터(Sector) 단위로 읽기/쓰기를 했던 것과는 다르게 클러스터(Cluster)라고 불리는 논리적인 단위로 읽기/쓰기가 된다.

사용하지 않는(Unused) 영역

이 영역은 물리적으로는 사용해도 상관없지만, FAT 파일시스템이 볼륨을 구조화시키는 과정에서 잉여분이 조금 남는 영역이다. 이 영역은 버려진다는 생각에 조금 아까울 수도 있지만, 실제로 보면 매우 적은 양이므로 무시할 수 있을 정도이다.

영역별 정리

[표 3-5] FAT 파일시스템의 영역별 크기와 접근 단위

구분	부트 레코드	예약된 영역	FAT 영역	데이터 영역
크기	1섹터	1~32섹터	파티션의 1~0.5%	파티션의 99%이상
접근 단위	섹터	섹터	섹터	클러스터

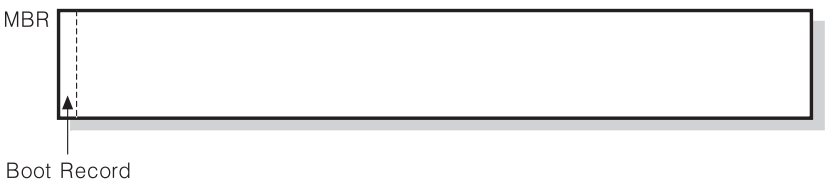
위 표는 지금까지 설명한 각 영역의 크기와 접근 단위를 나열한 것이다. 데이터 영역을 제외한 다른 영역들은 데이터 영역을 관리할 목적으로 사용되므로 작으면 작을수록 좋다. 데이터 영역을 제외한 다른 영역들은 모두 섹터 단위로 읽기/쓰기가 이루어진다.

03 부트 레코드

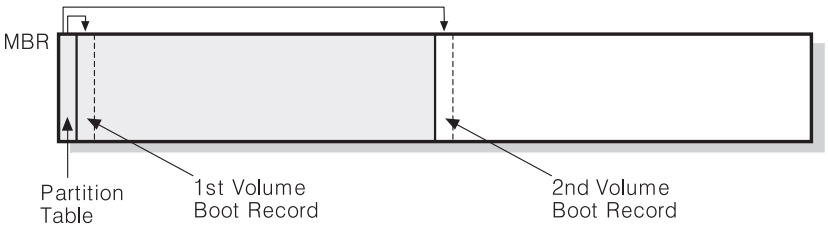
볼륨의 여러 설정 값들은 전부 부트 레코드에 저장되기 때문에 이 영역을 잘 분석하는 것이 Windows와 호환되는 FAT를 구현하는 첫걸음이라고 할 수 있다. 부트 레코드는 섹터 1개(512Byte)로 이루어져 있는 작은 영역이지만 중요한 내용이 많으니 유심히 살펴보자.

부트 레코드 소개

BIOS Parameter Block(BPB)이라고도 불리는 이 영역은 해당 볼륨의 여러 가지 설정 값들을 저장하고 있으며, 해당 볼륨이 부팅될 수 있도록 부팅에 필요한 실행 코드도 포함하고 있다. Boot Record가 저장되는 위치는 해당 볼륨의 첫 번째 섹터이며 FAT 파일시스템을 구성하는 영역 중 맨 앞에 위치하는 영역인 예약된(Reserved) 영역의 첫 번째 섹터이기도 하다. 만약 하드디스크에 볼륨이 여러 개 있다면 각각의 볼륨에 부트 레코드도 하나씩 있게 된다. 만약 하드디스크에 파티셔닝이 되어 있지 않아서 볼륨이 하나라면 부트 레코드는 MBR (Master Boot Record) 영역에 존재하게 된다.

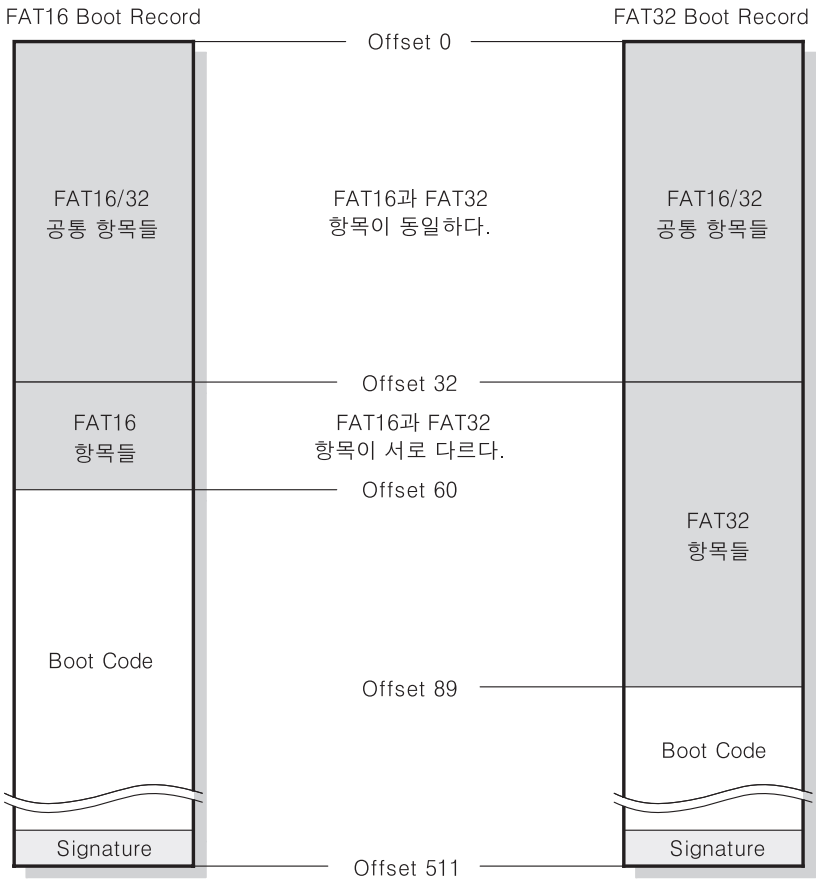


[그림 3-4] 파티션 없이 단일 볼륨인 하드디스크의 부트 레코드 위치



[그림 3-5] 파티션이 존재하고 볼륨이 2개인 하드디스크의 부트 레코드 위치

부트 레코드에서 FAT16과 FAT32의 차이



[그림 3-6] FAT16과 FAT32의 부트 레코드 레이아웃

부트 레코드의 0번 오프셋에서 32번 오프셋까지는 FAT16과 FAT32가 공통된 항목을 가진다. 32번 오프셋 이후는 FAT16과 FAT32의 항목이 서로 달라서 호환이 안 되는데, 이 말은 여러분이 FAT 파일시스템을 구현할 때 FAT16용 부트 레코드 구조체와 FAT32용 부트 레코드 구조체를 따로 선언해야 한다는 말이다. 볼륨이 어떤 파일시스템으로 이루어져 있는지를 조사해볼 때에는 0번 오프셋에서 32번 오프셋까지의 공통 항목의 내용을 조사하면 알 수 있게 된다.

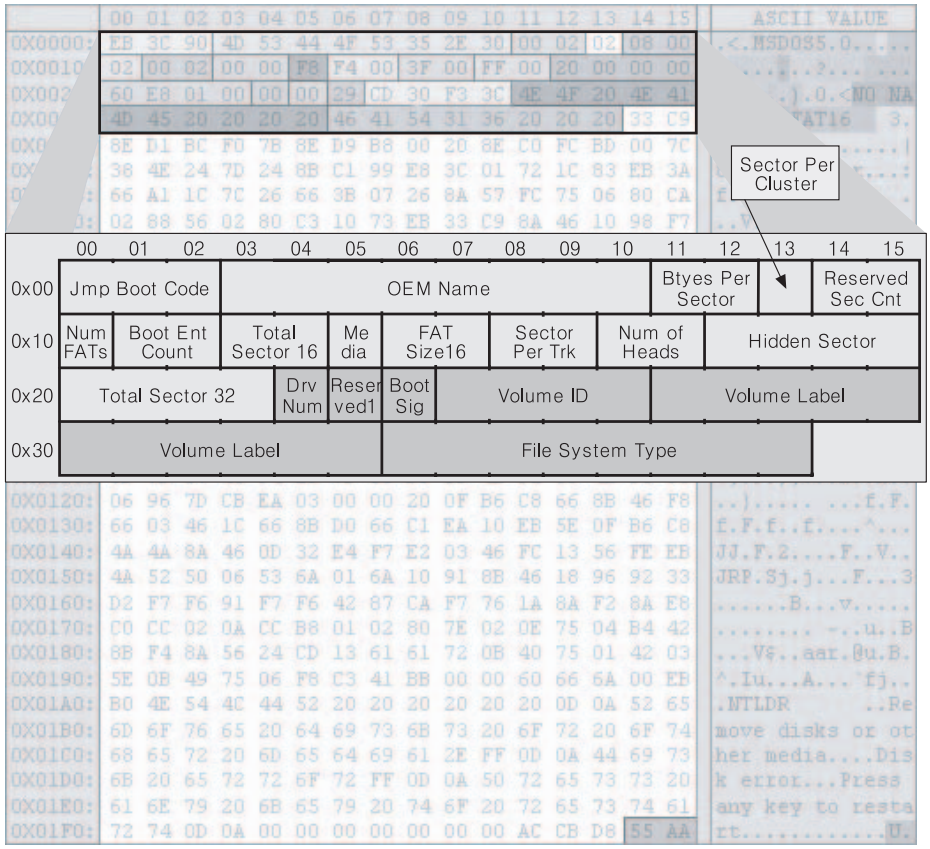
FAT16 부트 레코드

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	ASCII VALUE
0X0000:	EB	3E	90	50	72	6F	6C	69	66	69	63	00	02	20	01	00	.>.Prolific.. ..
0X0010:	02	10	02	00	00	F8	1F	00	20	00	40	00	20	00	00	00	..... .@. ...
0X0020:	E0	DF	03	00	80	01	29	00	00	00	00	4E	4F	20	4E	41	.... .)....NO NA
0X0030:	4D	45	20	20	20	20	46	41	54	31	36	20	20	20	F1	7D	ME FAT16 .}
0X0040:	FA	33	C9	8E	D1	BC	FC	7B	16	07	BD	78	00	C5	76	00	.3.....{...x..v.
0X0050:	1E	56	16	55	BF	22	05	89	7E	00	89	4E	02	B1	0B	FC	.V.U."..~..N....
0X0060:	F3	A4	06	1F	BD	00	7C	C6	45	FE	0F	8B	46	18	88	45	......|.E...F..E
0X0070:	F9	FB	38	66	24	7C	04	CD	13	72	3C	8A	46	10	98	F7	..8f$|...r<.F...
0X0080:	66	16	03	46	1C	13	56	1E	03	46	0E	13	D1	50	52	89	f..F..V..F...PR.
0X0090:	46	FC	89	56	FE	B8	20	00	8B	76	11	F7	E6	8B	5E	0B	F..V.. ..v....^.
0X00A0:	03	C3	48	F7	F3	01	46	FC	11	4E	FE	5A	58	BB	00	07	..H...F..N.ZX...
0X00B0:	8B	FB	B1	01	E8	94	00	72	47	38	2D	74	19	B1	0B	56	.......rG8-t...V
0X00C0:	8B	76	3E	F3	A6	5E	74	4A	4E	74	0B	03	F9	83	C7	15	.v>..^tJNt......
0X00D0:	3B	FB	72	E5	EB	D7	2B	C9	B8	D8	7D	87	46	3E	3C	D8	;.r...+...}.F><.
0X00E0:	75	99	BE	80	7D	AC	98	03	F0	AC	84	C0	74	17	3C	FF	u.. }.......t.<.
0X00F0:	74	09	B4	0E	BB	07	00	CD	10	EB	EE	BE	83	7D	EB	E5	t...........}..
0X0100:	BE	81	7D	EB	E0	33	C0	CD	16	5E	1F	8F	04	8F	44	02	..}..3...^....D.
0X0110:	CD	19	BE	82	7D	8B	7D	0F	83	FF	02	72	C8	8B	C7	48	....}.}....r...H
0X0120:	48	8A	4E	0D	F7	E1	03	46	FC	13	56	FE	BB	00	07	53	H.N....F..V....S
0X0130:	B1	04	E8	16	00	5B	72	C8	81	3F	4D	5A	75	A7	81	BF	.....[r..?MZu...
0X0140:	00	02	42	4A	75	9F	EA	00	02	70	00	50	52	51	91	92	..BJu....p.PRQ..
0X0150:	33	D2	F7	76	18	91	F7	76	18	42	87	CA	F7	76	1A	8A	3..v...v.B...v..
0X0160:	F2	8A	56	24	8A	E8	D0	CC	D0	CC	0A	CC	B8	01	02	CD	..V$............
0X0170:	13	59	5A	58	72	09	40	75	01	42	03	5E	0B	E2	CC	C3	.YZXr.@u.B.^....
0X0180:	03	18	01	27	0D	0A	49	6E	76	61	6C	69	64	20	73	79	...'..Invalid sy
0X0190:	73	74	65	6D	20	64	69	73	6B	FF	0D	0A	44	69	73	6B	stem disk...Disk
0X01A0:	20	49	2F	4F	20	65	72	72	6F	72	FF	0D	0A	52	65	70	I/O error...Rep
0X01B0:	6C	61	63	65	20	74	68	65	20	64	69	73	6B	2C	20	61	lace the disk, a
0X01C0:	6E	64	20	74	68	65	6E	20	70	72	65	73	73	20	61	6E	nd then press an
0X01D0:	79	20	6B	65	79	0D	0A	00	49	4F	20	20	20	20	20	20	y key...IO
0X01E0:	53	59	53	4D	53	44	4F	53	20	20	20	53	59	53	80	01	SYSMSDOS SYS .
0X01F0:	00	57	49	4E	42	4F	4F	54	20	53	59	53	00	00	55	AA	.WINBOOT SYS..U.

[그림 3-7] FAT16 부트 레코드 영역을 출력한 값

위 그림은 실제로 FAT16의 부트 레코드를 출력해 놓은 것이다. 그림에서 보이는 512Byte의 16진수 내용이 FAT16 부트 레코드의 모든 것이다. 위의 값들 중에서 바탕에 색깔을 칠해놓은 값들은 각각 의미가 있는 설정 값들인데, Windows가 FAT 파티션을 인식하는 데 필수적인 사항들이므로 꼭 적어주어야 하는 중요한 내용들이다. 그 외에 바탕에 색이 없는 영역들은 Windows가 해당 볼륨을 부팅할 때 사용하는 부트 코드 영역이다. 이 파티션을 부팅용으로 이용하지 않을 것이라면 전부 '0' 으로 채워 넣어도 상관없는 영역이다. 이제부터 위 그림에서 색이 칠해져 있는 부분에 대해 하나씩 자세히 살펴보자.

이번 절을 읽으면서 위의 그림과 비교해 보는 것도 이해를 돕는 데 있어 좋은 방법 중 하나이다. 한 가지 유의해야 할 점은 FAT 파일시스템은 리틀 엔디언(Little Endian)으로 설계되었기 때문에 위의 그림에서 출력된 값들은 실제 저장한 값과 반대로 뒤집혀 있다는 것이다.



[그림 3-8] FAT16 부트 레코드 항목들

위 그림은 [그림 3-7]에서 보여준 데이터들이 어떤 항목인지를 보여주는 그림이다. 이제 위 그림에 있는 각각의 항목들을 자세하게 살펴볼 것이다. 항목 중에 흐리게 칠해진 항목들은 FAT16과 FAT32 둘 다 가지는 공통 항목을 의미하고, 진하게 칠해진 항목들은 FAT16 전용 항목을 의미한다.

FAT16과 FAT32 부트 레코드 공통 항목 설명

이 름	Jump Boot Code				
위 치 (Offset)	0~2	크 기 (Size)	3 Byte	일반적인 값 (Value)	EB 3C 90 (기계어임)
설 명	부트 코드로 점프하라는 명령어. x86 계열 PC는 부팅 시 이 항목을 읽어서 부트 코드가 있는 영역으로 점프하게 된다. 해당 볼륨을 이용해서 부팅할 일이 없다면 이 부분은 어떤 값이 와도 무방하다.				

이 름	OEM Name				
위 치 (Offset)	3~10	크 기 (Size)	8 Byte	일반적인 값 (Value)	ASCII Code 'MSDOS5.0'
설 명	이 항목은 단지 OEM 회사를 나타내기 위한 문자열일 뿐이다. Windows는 이 항목에 대해서 어떠한 참조도 하지 않는다. Windows는 보통 'MSWIN4.1'이라고 적거나 'MSDOS5.0'이라고 적는다. 이 항목에 여러분이 어떤 값을 적든지 상관없다.				

이 름	Bytes Per Sector				
위 치 (Offset)	11~12	크 기 (Size)	2 Byte	일반적인 값 (Value)	512
설 명	섹터당 바이트 수를 적는다. 즉, 하드디스크의 한 섹터가 몇 바이트를 담고 있는지에 대한 항목이다. 이 항목에는 필수적으로 다음 값만 적어주어야 한다. **512, 1024, 2048, 4096 (이 값 외에 다른 값은 적지 않는다.)** Windows나 다른 FAT를 인식하는 장치들에서 여러분이 기록한 FAT 하드디스크가 잘 인식되길 바란다면 반드시 이 항목에 512라고 적어주어야 한다. 섹터 크기가 512Byte가 아닌 하드디스크는 거의 없기 때문에 여러분들도 512라고 적는 것이 좋다.				

이 름	Sector Per Cluster				
위 치 (Offset)	13	크 기 (Size)	1 Byte	일반적인 값 (Value)	32
설 명	클러스터당 섹터 수. 한 클러스터가 몇 개의 섹터로 이루어지는가를 적는 사항이다. 이 값은 반드시 '0'보다 큰 값이어야 하며 2의 배수이어야 한다.				

이쯤 되면 클러스터의 크기를 알 수 있게 된다.

Sector Per Cluster X Bytes Per Sector = 클러스터 크기

예를 들어보자. 클러스터당 섹터 수가 2이고, 섹터당 512Byte라면 2 X 512 = 1024Byte, 즉 클러스터의 크기는 1KB가 되는 것이다.

이 름	Reserved Sector Count				
위 치 (Offset)	14~15	크 기 (Size)	2 Byte	일반적인 값 (Value)	FAT16 은 '1' FAT32 는 '32'
설 명	예약된(Reserved) 영역의 섹터 수를 저장한다. 예약된 영역의 크기는 최소한 1개 섹터 이상 되어야 하는데, 그것은 부트 레코드가 예약된 영역의 첫 번째 섹터에 위치하기 때문이다. 때문에 이 항목의 값이 '0'이 되어서는 안 된다. FAT16에서는 이 항목에 보통 '1'이라는 값을 가지며, FAT32는 보통 '32'라는 값을 가진다. 예약된 영역에서 부트 레코드나 특별히 다른 정보를 담지 않고 있는 섹터들은 전부 '0'으로 초기화되어 있어야 한다.				

이 름	Number of FATs				
위 치 (Offset)	16	크 기 (Size)	1 Byte	일반적인 값 (Value)	2
설 명	볼륨에 있는 FAT 영역의 수를 담고 있다. 이 항목은 최소한 '1' 이상의 값을 가져야 하며, '2'라고 적는 것을 추천한다. 주의해야 할 점은 이 항목에다 '2'라고 적어놓고 실제로는 FAT 영역을 3개 만든다든지 1개를 만드는 행동은 좋지 않다. CF Card와 같은 저용량 매체에서 Data 영역을 조금이라도 더 확보하기 위해 이 항목을 '1'로 지정하는 개발자들도 있다. Windows는 이 항목의 값이 '1'이더라도 친절하게 호환성을 보장하지만, 세상의 수많은 FAT 코드들도 다 그럴 것이라고는 장담하지 못한다. 왜냐하면 세상에는 Number of FATs의 값이 무조건 '2'일 것이라는 가정 하에 코드를 짜는 게으른 개발자들도 있기 때문이다.				

이 름	Root Directory Entry Count				
위 치 (Offset)	17~18	크 기 (Size)	2 Byte	일반적인 값 (Value)	FAT16은 '512' FAT32는 '0'
설 명	FAT16 경우에 한해 이 항목은 루트 디렉토리에 몇 개의 엔트리(파일과 디렉토리 모두 동일하게 1개의 엔트리로 본다)를 수용할 것인지를 나타낸다. 이 항목의 값은 항상 Bytes Per Sector의 짝수이어야 한다. 즉, Bytes Per Sector가 512라면 이 항목에는 512, 1024, 2048과 같은 값이 들어갈 수 있다. 최대 호환성을 위해서 이 항목의 값은 '512'로 적는 게 좋다. **중요한 점은 FAT32에서는 이 항목의 값이 반드시 '0'이어야만 한다는 것이다.**				

Root Directory Entry Count 항목이 존재하는 이유는 FAT16의 경우 루트 디렉토리 영역이 따로 분리되어 있기 때문이다. 이 영역의 크기를 지정하기 위해서 위 항목이 존재하게 된다. FAT16에서는 루트 디렉토리에 넣을 수 있는 디렉토리나 파일의 수에 제한이 있는데 보통은 512개이며, 더 넣고 싶다면 Root Directory Entry Count 항목의 값을 변경하면 된다. 단, 루트 디렉토리의 크기가 부족하다고 FAT16 파일시스템 사용 도중 마음대로 늘릴 수 없으니, 포맷을 할 때 신중히 결정해야 한다.

이 름	Total Sector 16				
위 치 (Offset)	19~20	크 기 (Size)	2 Byte	일반적인 값 (Value)	FAT16은 가변적임. FAT32는 반드시 '0'
설 명	이 항목은 볼륨상에 있는 총 섹터 수를 2Byte로 나타낸다. 중요한 것은 저장장치의 총 섹터 수가 아닌 볼륨의 총 섹터 수라는 점이다. 볼륨에 존재하는 모든 영역을 합친 섹터 수를 기록해야 한다. 이 항목의 값에 '0'이 올 수 있다. 만약 이 항목의 값이 '0'이라면 아래에서 설명할 Total Sector 32 항목의 값은 절대 '0'이 되어서는 안 된다. FAT16의 경우라도 총 섹터 수가 '0xFFFF'보다 크다면 Total Sector 32의 항목을 이용한다. **중요한 점은 FAT32에서는 이 항목의 값이 반드시 '0'이어야 한다는 것이다.**				

이 름	Media				
위 치 (Offset)	21	크 기 (Size)	1 Byte	일반적인 값 (Value)	0xF8
설 명	이 항목은 이 볼륨이 어떤 미디어에 저장되어 있는지를 나타낸다. 보통은 고정식 디스크 값인 '0xF8'이 쓰인다. CF Card나 USB Memory는 이동식이므로 '0xF8'이 아닐 것이라고 생각하기 쉽다. 하지만 실제로 Windows가 CF Card를 포맷하면 이 항목에 값을 '0xF8'이라고 적는다. 그러니 여러분들도 플로피디스크 외의 저장장치라면 이 항목에 '0xF8'을 적는 것이 좋다. 이 항목에 저장된 값은 나중에 알아보게 될 FAT Entry 0번에도 저장되어 있다. 이것에 대해서는 나중에 자세히 알아볼 것이다.				

[표 3-6] Media 항목에 들어갈 수 있는 값들

값	용량	저장장치 형태
0xF0	2.88MB	3.5inch, 양면, 36sector 플로피디스크
0xF0	1.44MB	3.5inch, 양면, 18sector 플로피디스크
0xF9	720KB	3.5inch, 양면, 9sector 플로피디스크
0xF9	1.2MB	5.25inch, 양면, 15sector 플로피디스크
0xFD	360KB	5.25inch, 양면, 9sector 플로피디스크
0xFF	320KB	5.25inch, 양면, 8sector 플로피디스크
0xFC	180KB	5.25inch, 단면, 9sector 플로피디스크
0xFE	160KB	5.25inch, 단면, 8sector 플로피디스크
0xF8	X	고정식 디스크(플로피디스크를 제외한 모든 장치)

이 름	FAT Size 16				
위 치 (Offset)	22~23	크 기 (Size)	2 Byte	일반적인 값 (Value)	FAT16은 가변적임 FAT32는 반드시 '0'
설 명	이 항목은 FAT 영역의 섹터 수를 저장하는 항목이다. FAT #1 영역과 FAT #2 영역을 합친 값이 아닌 1개의 FAT 영역의 섹터 수를 적어야 한다. 이 항목은 FAT12와 FAT16만 사용한다. FAT32는 FAT Size 32라는 항목이 따로 있다. **중요한 점은 FAT32에서는 이 항목의 값이 반드시 '0'이어야 한다는 것이다.**				

이 름	Sector Per Track				
위 치 (Offset)	24~25	크 기 (Size)	2 Byte	일반적인 값 (Value)	63
설 명	이 항목은 x86 프로세서 계열에서 발생하는 인터럽트(INT) 0x13을 위해 존재하는 항목이다. 저장장치의 트랙당 섹터 수를 저장한다. Windows 계열에서는 더 이상 이 항목을 참조하지 않는다.				

이 름	Number Of Heads				
위 치 (Offset)	**26~27**	크 기 (Size)	**2 Byte**	일반적인 값 (Value)	255
설 명	이 항목은 x86 프로세서 계열에서 발생하는 인터럽트(INT) 0x13을 위해 존재하는 항목이다. 저장장치의 헤더 수를 저장한다. Windows 계열에서는 더 이상 이 항목을 참조하지 않는다.				

이 름	Hidden Sector				
위 치 (Offset)	**28~31**	크 기 (Size)	**4 Byte**	일반적인 값 (Value)	32
설 명	이 항목은 x86 프로세서 계열에서 발생하는 인터럽트(INT) 0x13을 위해 존재하는 항목이다. 해당 볼륨 앞에 존재하는 (숨겨진) 섹터 수를 저장한다. 파티션되지 않은 볼륨에서는 반드시 '0'이어야 한다. Windows 계열에서는 더 이상 이 항목을 참조하지 않는다.				

이 름	Total Sector 32				
위 치 (Offset)	**32~35**	크 기 (Size)	**4 Byte**	일반적인 값 (Value)	가변적임
설 명	이 항목은 볼륨상에 있는 총 섹터 수를 4Byte로 나타낸다. 중요한 것은 저장장치의 총 섹터 수가 아닌 볼륨의 총 섹터 수라는 점이다. 볼륨에 존재하는 모든 영역을 전부 합친 섹터 수를 적어야 한다. FAT16의 경우 이 항목의 값에 '0'이 올 수 있다. 만약 이 항목의 값이 '0'이라면 위에서 설명한 Total Sector 16의 항목의 값은 절대 '0'이 되어서는 안 된다. **FAT32에서는 이 항목의 값이 반드시 '0'이 아니어야 한다.** 즉, FAT32는 Total Sector 16의 항목은 이용하지 않고, Total Sector 32만을 이용한다.				

지금까지 Offset 0~35에 해당하는 항목들을 살펴보았다. 여기까지는 FAT16과 FAT32가 동일하게 가지는 항목들이다. Offset 36부터는 FAT16과 FAT32가 각각 다른 항목들을 가지게 되므로 따로 설명하겠다.

FAT16 부트 레코드 항목 설명

이제부터 FAT16에 해당하는 항목들을 살펴보자. 주의해야 할 점은 위치(Offset) 값의 시작이 36부터라는 점이다.

이 름	Drive Number				
위 치 (Offset)	**36**	크 기 (Size)	**1 Byte**	일반적인 값 (Value)	0x80
설 명	이 항목은 x86 프로세서 계열에서 발생하는 인터럽트(INT) 0x13을 위해 존재하는 항목이다. INT 0x13에서 정의한 드라이브 값을 설정한다. 플로피디스크는 '0x0', 그 외의 대부분의 저장장치는 '0x80'의 값은 가진다. Windows 계열에서는 더 이상 이 항목을 참조하지 않는다. **이 항목은 FAT16에만 해당한다.**				

이 름	Reserved1				
위 치 (Offset)	37	크 기 (Size)	1 Byte	일반적인 값 (Value)	0
설 명	Windows NT 계열에서 사용하려고 만든 항목이다. 이 항목은 항상 '0'으로 해야 한다. **이 항목은 FAT16에만 해당한다.**				

이 름	Boot Signature				
위 치 (Offset)	38	크 기 (Size)	1 Byte	일반적인 값 (Value)	0x29
설 명	확장 부트 서명. 이 항목의 값은 항상 '0x29'여야 한다. 이 항목이 의미하는 것은 여기가 끝이 아니라 이 항목 아래쪽에 세 가지 항목이 더 추가되었다는 것을 말한다. **이 항목은 FAT16에만 해당한다.**				

이 름	Volume ID				
위 치 (Offset)	39~42	크 기 (Size)	4 Byte	일반적인 값 (Value)	가변적임
설 명	볼륨 시리얼 번호. 고유의 임의의 시리얼 번호를 생성해서 기록한다. **이 항목은 FAT16에만 해당한다.**				

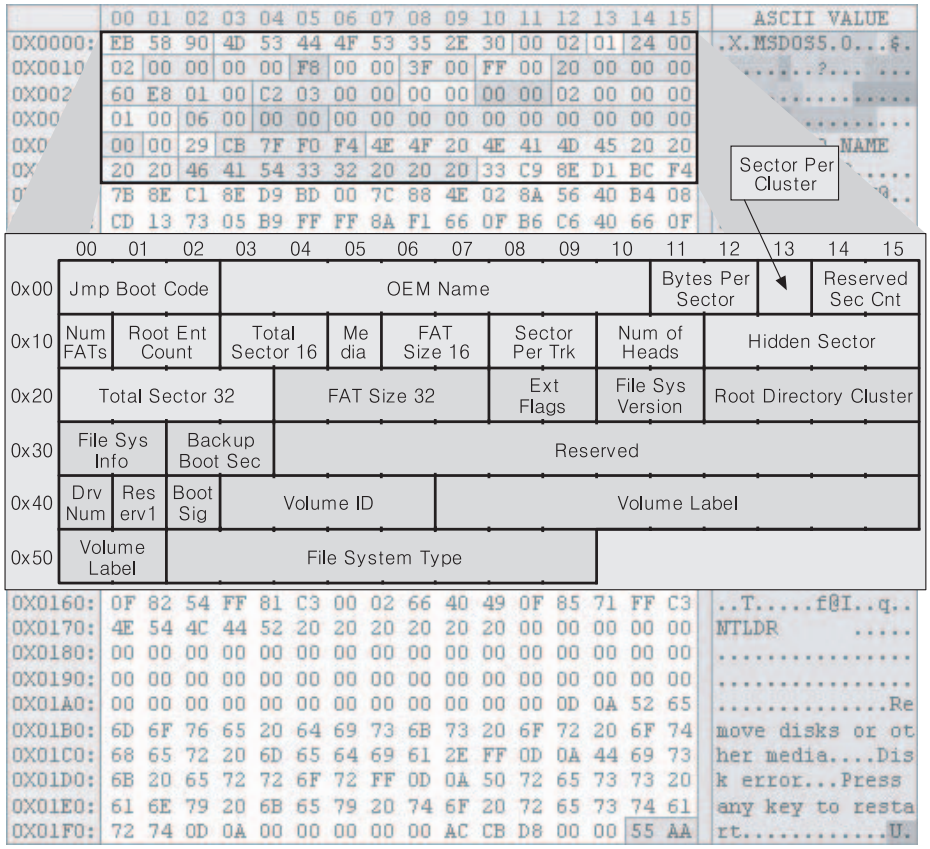
이 름	Volume Label				
위 치 (Offset)	43~53	크 기 (Size)	11 Byte	일반적인 값 (Value)	가변적임
설 명	볼륨 레이블. 해당 볼륨의 레이블을 적는다. 여기에 적은 문자열은 Windows에서 드라이브 이름으로 쓰이게 된다. 사실 볼륨 레이블을 적는 부분은 2군데가 있는데, 그 중 한곳은 여기이고, 다른 한곳은 루트 디렉토리에 존재하는 레이블 엔트리이다. 볼륨 레이블이 변경될 경우 2군데 모두 값을 바꿔 주어야 한다. 루트 디렉토리에 레이블 엔트리를 생성하지 않았다면 이 항목은 "NO NAME "이라고 적어 주어야 한다. **이 항목은 FAT16에만 해당한다.**				

이 름	File System Type				
위 치 (Offset)	54~61	크 기 (Size)	8 Byte	일반적인 값 (Value)	ASCII Code "FAT16 "
설 명	이 항목에는 "FAT12 ", "FAT16 ", "FAT " 중 적절한 문자열을 적어주면 된다. 주의할 점은 여러분이 어떤 하드디스크가 FAT16인지 FAT32인지를 조사할 때 이 영역에 적힌 값을 이용해서 판단을 내리면 안 된다는 점이다. 이 영역을 읽어봤더니 "FAT16 "이라고 적혀 있어도 곧바로 해당 볼륨이 FAT16일 것이라고 결론을 내리면 안 된다. Windows 역시 이 항목에 적은 값을 참조하지 않는다. 이 항목은 단순한 문자열이며 그 외에 어떤 의미도 없다. **이 항목은 FAT16에만 해당한다.**				

지금까지 FAT16에 해당하는 항목에 대해 모두 알아보았다.

FAT32 부트 레코드 항목 설명

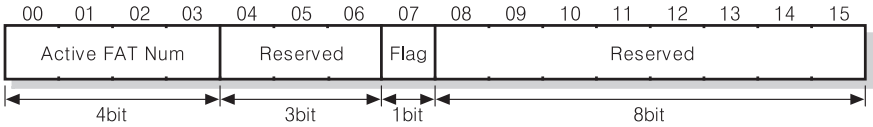
이제부터 FAT32에 해당하는 항목들을 살펴보자. 주의할 점은 위치(Offset) 값의 시작이 36 부터라는 점이다. 그 이전에 해당하는 값들은 FAT 16/32의 공통 항목이다.



[그림 3-9] FAT32 부트 레코드 항목

이 름	FAT Size 32				
위 치 (Offset)	**36~39**	크 기 (Size)	**4 Byte**	일반적인 값 (Value)	가변적임
설 명	이 항목은 FAT 영역의 섹터 수를 저장하는 항목이다. FAT #1 영역과 FAT #2 영역을 합친 값이 아닌 1개의 FAT 영역의 섹터 수를 적어야 한다. **이 항목은 FAT32에만 해당한다.**				

이 름	Ext Flags				
위 치 (Offset)	**40~41**	크 기 (Size)	**2 Byte**	일반적인 값 (Value)	0x00
설 명	FAT 테이블을 조작하는 것에 대하여 여러 설정 값을 가진다. 설정 값에 대한 사항은 아래 표를 참고한다. **이 항목은 FAT32에만 해당한다.**				



[그림 3-10] Ext Flags 항목

[표 3-7] Ext Flags 항목 설명

속성 이름	크기	설명
Active FAT Number	4Bit	활동시킬 FAT의 번호. 0부터 시작한다. Flag의 값이 '1'인 경우에만 의미가 있다.
Reserved	3Bit	사용되지 않음. 미래를 위해 예약된 공간.
Flag	1Bit	0 : 변경 내용을 모든 FAT 영역에 반영하라. 1 : 변경 내용을 Active Fat Number에 적힌 FAT 영역에만 반영하고, 다른 영역에는 손대지 마라.
Reserved	8Bit	사용되지 않음. 미래를 위해 예약된 공간.

이 름	File System Version				
위 치 (Offset)	42~43	크 기 (Size)	2 Byte	일반적인 값 (Value)	0x00
설 명	이 항목은 FAT32의 버전 정보를 나타낸다. 상위 바이트는 주 버전, 하위 바이트는 부 버전을 나타낸다. Windows는 이 항목을 조사하여 만약 적혀 있는 버전이 자신이 인식할 수 있는 버전보다 높다면 파티션 인식 작업을 포기하게 된다. 지금까지 나온 FAT 버전은 전부 0x00으로 설정되어 있고, 아직까지 이 버전 정보가 올라간 적이 없다. 그러므로 여러분들도 반드시 0x00으로 적어주길 바란다. **이 항목은 FAT32에만 해당한다.**				

이 름	Root Directory Cluster				
위 치 (Offset)	44~47	크 기 (Size)	4 Byte	일반적인 값 (Value)	2
설 명	이 항목은 루트 디렉토리의 클러스터 값을 담고 있다. FAT16과는 다르게 FAT32는 루트 디렉토리가 데이터 영역 어디에 와도 상관없다. 그러므로 이곳에 루트 디렉토리의 시작 위치를 적어놓는 것이다. 보통의 경우 이 항목의 값은 2가 들어가는데 여러분들도 이 값을 따르는 게 좋다. 만약 실수로 이 항목의 값이 엉망이 되어 버렸다면 루트 디렉토리의 위치를 알 수 없게 되는데 항상 루트 디렉토리를 클러스터 2번에 위치시켜 놓는다면 쉽게 루트 디렉토리의 위치를 복구할 수 있다. **이 항목은 FAT32에만 해당한다.**				

이 름	File System Information				
위 치 (Offset)	48~49	크 기 (Size)	2 Byte	일반적인 값 (Value)	1
설 명	이 항목은 FSInfo 구조체가 어디에 저장되어 있는지를 가리킨다. 보통의 경우 볼륨의 1번 섹터에 저장된다. FSInfo 구조체에 대해서는 나중에 다시 설명하겠다. **이 항목은 FAT32에만 해당한다.**				

이 름	Boot Record Backup Sector				
위 치 (Offset)	50~51	크 기 (Size)	2 Byte	일반적인 값 (Value)	6
설 명	부트 레코드는 매우 중요하기 때문에 백업을 해두는 게 현명한데, 그 위치를 여기에 적어둔다. 사본의 위치는 예약된(Reserved) 영역 어디에 오든 상관없지만, 일반적으로 6번 섹터를 이용한다. 만약 이 항목의 값이 '0'이라면 백업을 하지 않았다는 의미이다. **이 항목은 FAT32에만 해당한다.**				

이 름	Reserved				
위 치 (Offset)	52~63	크 기 (Size)	12 Byte	일반적인 값 (Value)	0
설 명	미래를 위해 예약된 영역이다. 이 영역은 항상 '0'으로 채워져 있어야 한다. **이 항목은 FAT32에만 해당한다.**				

이 름	Drive Number				
위 치 (Offset)	64	크 기 (Size)	1 Byte	일반적인 값 (Value)	1
설 명	이 항목은 인터럽트(INT) 0x13을 위해 존재하는 항목이다. Windows 계열에서는 더 이상 이 항목을 참조하지 않으므로 어떤 값을 써도 무방하다. **이 항목은 FAT32에만 해당한다.**				

이 름	Reserved1				
위 치 (Offset)	65	크 기 (Size)	1 Byte	일반적인 값 (Value)	0
설 명	Windows NT 계열에서 사용하려고 만든 항목이다. 이 항목은 항상 0으로 해야 한다. **이 항목은 FAT32에만 해당한다.**				

이 름	Boot Signature				
위 치 (Offset)	66	크 기 (Size)	1 Byte	일반적인 값 (Value)	0x29
설 명	확장 부트 서명이다. 이 항목의 값은 항상 '0x29'여야 한다. 이 항목이 의미하는 것은 여기가 끝이 아니라 이 항목 아래쪽에 세 가지 항목이 더 추가되었다는 것을 말한다. **이 항목은 FAT32에만 해당한다.**				

이 름	Volume ID				
위 치 (Offset)	67~70	크 기 (Size)	4 Byte	일반적인 값 (Value)	가변적임
설 명	볼륨 시리얼 번호이다. 고유의 임의의 시리얼 번호를 생성해서 기록한다. **이 항목은 FAT32에만 해당한다.**				

이 름	Volume Label				
위 치 (Offset)	71~81	크 기 (Size)	11 Byte	일반적인 값 (Value)	가변적임
설 명	볼륨 레이블이다. 해당 파티션의 볼륨 레이블을 적는다. 여기에 적은 글자는 Windows에서 디스크 이름으로 쓰이게 된다. 사실 볼륨 레이블을 적는 부분은 2군데가 있는데, 그 중 한곳은 여기이고, 다른 한곳은 루트 디렉토리이다. 볼륨 레이블이 변경될 경우 2군데 모두 값을 바꿔주어야 한다. 볼륨 레이블이 없을 경우에는 "NO NAME "이라고 적어준다. **이 항목은 FAT32에만 해당한다.**				

이 름	File System Type				
위 치 (Offset)	82~89	크 기 (Size)	8 Byte	일반적인 값 (Value)	ASCII Code "FAT32 "
설 명	Windows는 이 항목에 항상 "FAT32 "라고 적는다. 자세한 설명은 FAT16에서 이 항목에 대한 설명을 참조한다. 중요한 점은 이 항목이 FAT의 형태 결정에 아무 일도 하지 않는다는 것이다. **이 항목은 FAT32에만 해당한다.**				

여기서 잠깐

부트 레코드 항목 중에는 문자열을 저장하는 항목이 몇 군데 있다. OEM Name, Volume Label, File System Type 항목은 문자열을 저장한다. 이들 항목에 문자열을 저장하는 데 있어서 주의해야 할 점은 빈 공간은 반드시 Space(0x20)로 처리해야 한다는 것이다. 예를 들어 C 언어로 File System Type에 'FAT16'이라는 문자열을 넣을 때 다음과 같이 코드를 작성했다고 하자.

```
strcpy( FileSystemType, "FAT16" );
```

그러면 저장되는 값은 다음과 같다.

F	A	T	1	6	NULL	X	X
0x46	0x41	0x54	0x31	0x36	0x0	알 수 없음	알 수 없음

하지만 FAT 파일시스템에서 문자열의 모든 빈 공간에는 Space가 들어가야 한다. 그러므로 다음과 같은 저장 형태를 가져야 한다.

F	A	T	1	6	Space	Space	Space
0x46	0x41	0x54	0x31	0x36	0x20	0x20	0x20

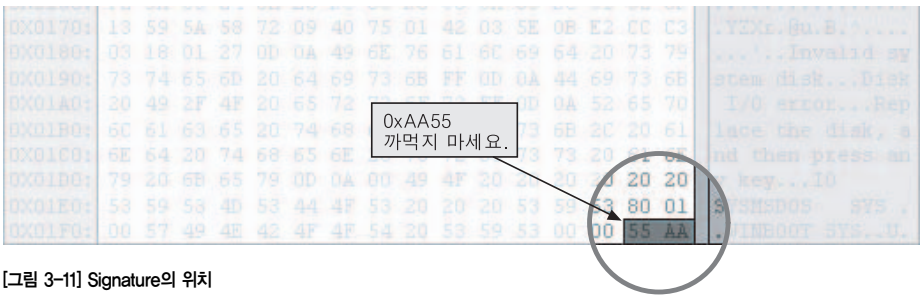
위와 같은 형태로 들어가게 하려면 일반적으로 C 언어에서 제공하는 문자열 제어 함수를 사용하면 안 된다. 직접 특별한 문자열 제어 함수를 구현하든지 소스 작성하기가 귀찮다면 일일이 배열에 값을 넣어주면 된다.

부트 레코드의 끝부분

부트 레코드를 구현하는 데 있어 한 가지 주의 사항이 있다면 FAT16/32 모두 부트 레코드 영역의 Offset 511~512에는 반드시 0xAA55라는 서명이 들어가야 한다는 것이다.

이 름	Signature				
위 치 (Offset)	510~511	크 기 (Size)	2 Byte	일반적인 값 (Value)	반드시 0xAA55
설 명	이 항목은 부트 레코드가 손상되었는지 아닌지를 알기 위한 용도로 사용한다. 이 항목에 '0xAA55' 라는 값이 적혀 있지 않다면 Windows는 부트 레코드가 망가졌다고 판단하고 인식을 못하게 된다. **이 항목은 FAT16과 FAT32 모두 해당된다.**				

많은 FAT 관련 문서에서는 "Signature의 위치는 부트 레코드 영역의 가장 끝부분이다."라고 설명하고 있지만, 이것은 섹터의 크기가 512Byte일 경우에만 유효한 내용이다. Microsoft에서 제공하는 공식 문서에 따르면 섹터의 크기가 512Byte보다 큰 경우라도 Signature는 Offset 510~511에 와야 한다.



[그림 3-11] Signature의 위치

04 부트 레코드 실습

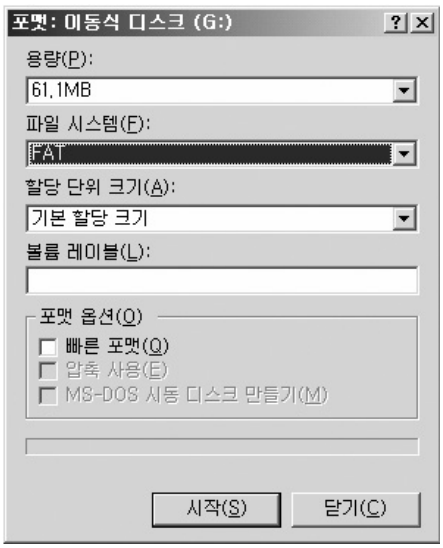
지금까지 부트 레코드가 어떻게 구성되어 있는지에 대해 배웠으니, 이제 실습을 해보자. 이번에 해볼 내용은 부트 레코드 영역을 읽어서 각 영역의 크기와 위치를 알아내는 프로그램을 구현하는 것이다.

실습할 내용

이번 실습에서는 부트 레코드 영역에 있는 항목들의 값을 이용해서 볼륨의 여러 영역의 크기와 위치를 알아내는 방법을 익혀볼 것이다. 이 실습을 통해 여러분들은 부트 레코드를 활용하는 방법을 이해할 수 있으며, FAT 파일시스템의 구조에 대해 좀 더 확실하게 알 수 있을 것이다. 또한 각각의 영역에 대한 크기를 구하는 것은 FAT 파일시스템을 사용하기 위한 기본 작업이기 때문에 이번 실습에서 얻은 정보들은 앞으로도 계속 활용하게 된다.

실습 전 준비 사항

실습용으로 사용할 저장장치, 예를 들어 USB Memory Stick이나 하드디스크를 준비한다. 실습용으로 사용할 저장장치를 마련했다면 FAT16 파일시스템으로 미리 포맷을 해놓자.



[그림 3-12] 저장장치 포맷 화면

이제 마지막으로 소스를 컴파일해줄 Microsoft의 Visual C++ 컴파일러를 준비하면 모든 준비는 끝나게 된다.

아래 소스는 FAT16 파일시스템의 부트 레코드를 읽어서 화면에 그 정보를 보여주는 코드이다. 만일 FAT32 파일시스템도 인식되게 하려면 여러분들이 아래 코드를 적절히 수정해 주어야 한다.

아래 소스 코드는 지금까지 소스에서 구현했던 함수나 구조체, 전역 변수들을 그대로 이용하고 있다. 하지만 기존에 구현했던 함수나 구조체들을 다시 소스에 보여주는 것은 내용이 중복되므로 생략했다. 이점 유의하면서 코드를 살펴보기 바란다.

소스 코드

예제

FAT_exam1.c

```
001  #include <stdio.h>
002  #include <windows.h>
003  #include <stdlib.h>
004
005  #pragma pack(1)
006  typedef struct _FAT16_BPB_struct{
007    U8            JmpBoot[3];
008    U8            OEMName[8];
009    U16           BytsPerSec;
010    U8            SecPerClus;
011    U16           RsvdSecCnt;
012    U8            NumFATs;
013    U16           RootEntCnt;
014    U16           TotSec16;
015    U8            Media;
016    U16           FATs16;
017    U16           SecPerTrk;
018    U16           NumHeads;
019    U32           HiddSec;
020    U32           TotSec32;
021    //----- 공통 영역 ------
022    U8            DriveNumber;
```

```
023    U8            Reserved1;
024    U8            BootSignal;
025    U32           VolumeID;
026    U8            VolumeLabel[11];
027    U8            FilSysType[8];
028    //------ FAT16 영역 ------
029
030    U8            BootCodeArea[448];
031
032    U16           Signature;
033  }FAT16_BPB;
034  #pragma pack()
035
036  typedef struct _VOL_struct{
037    U32           Drive;
038    U32           VolBeginSec;
039    U32           FirstDataSec;
040    U32           RootDirSec;
041    U32           RootEntCnt;
042    U32           RootDirSecCnt;
043    U32           FATSize;
044    U32           FATStartSec;
045    U32           TotalClusCnt;
046    U32           TotalSec;
047    U32           DataSecSize;
048    U32           ClusterSize;
049    U32           SecPerClus;
050  }VolStruct;
051
052  U32  HDD_read (U8 drv, U32 SecAddr, U32 blocks, U8* buf);
053  U32  get_BPB_info(FAT16_BPB* BPB, VolStruct* pVol);
054
055  VolStruct gVol;
056
057  int main(void) {
058    U8          buf[512];
059
060    gVol.Drive        = 0x2;
061    gVol.VolBeginSec   = 0x0;
062
063    if( HDD_read(gVol.Drive, gVol.VolBeginSec, 1, buf) == 0 ){
```

```
064         printf("Boot Sector Read Failed \n");
065         return 1;
066     }
067
068     if (get_BPB_info((FAT16_BPB*)buf, &gVol) == 0){
069         printf("It is not FAT16 File System \n");
070         return 1;
071     }
072
073     printf("[[[[[[[[ Volume Information ]]]]]]]] \n");
074
075     printf("Total Sector      = %d sectors \n", gVol.TotalSec);
076     printf("FAT size          = %d sectors \n", gVol.FATSize);
077     printf("Root Dir Sector   = %d        \n", gVol.RootDirSec);
078     printf("Root Dir Sector Count = %d    \n", gVol.RootDirSecCnt);
079     printf("First Data Sector = %d        \n", gVol.FirstDataSec);
080     printf("Data Sector Count = %d Sectors \n", gVol.DataSecSize);
081     printf("Total Cluster     = %d         \n", gVol.TotalClusCnt);
082     printf("Size of Cluster   = %d         \n", gVol.ClusterSize);
083     return 0;
084 }
085
086 U32 get_BPB_info(FAT16_BPB* BPB, VolStruct* pVol) {
087     if(BPB->RootEntCnt == 0 || BPB->Signature != 0xAA55)
088         return 0;
089
090     // Get Total Sector
091     if( BPB->TotSec16 != 0 )    pVol->TotalSec = BPB->TotSec16;
092     else                        pVol->TotalSec = BPB->TotSec32;
093
094     // Get FAT Size
095     pVol->FATSize = BPB->FATs16;
096
097     // Get FAT Start Sector
098     pVol->FATStartSec = pVol->VolBeginSec + BPB->RsvdSecCnt;
099
100     // Get Root Dir Entry Count
101     pVol->RootEntCnt = BPB->RootEntCnt;
102
103     // Get Root Dir Sector
104     pVol->RootDirSec = pVol->VolBeginSec + BPB->RsvdSecCnt
```

```
                    + (BPB->NumFATs * BPB->FATs16);
105
106     // Get Root Dir Sector Count
107     pVol->RootDirSecCnt = (( BPB->RootEntCnt * 32)
                        + (BPB->BytsPerSec - 1)) / BPB->BytsPerSec;
108
109     // GET FAT Start Sector
110     pVol->FirstDataSec = pVol->VolBeginSec + BPB->RsvdSecCnt
            + (BPB->NumFATs * pVol->FATSize) + pVol->RootDirSecCnt;
111
112     // Get Size of Data Area
113     pVol->DataSecSize = pVol->TotalSec - (BPB->RsvdSecCnt
           + (BPB->NumFATs * pVol->FATSize) + pVol->RootDirSecCnt);
114
115     // Get Total Cluster Count
116     pVol->TotalClusCnt = pVol->DataSecSize / BPB->SecPerClus;
117
118     // Get Size of Cluster
119     pVol->ClusterSize = BPB->SecPerClus * BPB->BytsPerSec;
120
121     // Get Sector Per Cluster
122     pVol->SecPerClus = BPB->SecPerClus;
123
124     return 1;
125 }
```

소스해설

- **5~34행** : FAT16 파일시스템의 부트 레코드의 구조체 선언이다. 부트 레코드의 섹터를 읽어서 BPB 구조체에 집어넣으면 각각의 변수에 해당하는 값이 들어가도록 해놓았다. 이렇게 편하게 한 번에 집어넣기 위해서는 컴파일러가 최적화 과정에서 구조체 변수들 사이에 최적화를 위한 패딩(Padding)을 집어넣는 것을 필수적으로 막아야 한다. 위 소스 중에서 Visual C++에게 구조체 패딩을 넣지 말라고 선언하는 것이 바로 #pragma pack(1)과 #progma pack( )이라는 구문이다.

  위 구조체에서 문자열을 저장하는 변수를 제외한 모든 변수가 unsigned형으로 선언된 것을 주의하라. FAT 파일시스템에서는 문자열을 제외하곤 전부 unsigned형만 사용한다.

- **36~50행** : 볼륨의 전반적인 정보를 담는 구조체이다. 읽어 들인 볼륨에 대한 상태와 정보를 담고 있으면서 여러 함수에서 편하게 참조하려고 만든 구조체이다.

- **60~61행** : 전역 변수인 gVol 구조체에 읽어 들일 드라이브와 볼륨의 시작 섹터를 지정한다. 이 소스에서는 드라이브를 2번으로 고정했지만 여러분의 PC 상황에 따라서 적절하게 변경하도록 한다. 자신이 접근하고자 하는 저장장치의 번호를 알고 싶다면 Windows의 "디스크 관리" 메뉴에 가서 확인하자.
- **63~66행** : 저장장치로부터 부트 레코드 섹터를 읽은 후 buf에 담는다.
- **68~71행** : buf를 읽어서 처리한 후 gVol 변수에 담는다. 이 소스는 FAT16 파일시스템만 인식하도록 코딩되어 있기 때문에 그 외의 파일시스템이 오면 에러를 내도록 했다.
- **73~83행** : 읽어 들인 부트 레코드에서 얻어낸 여러 정보들을 화면에 출력한다.
- **87행** : 읽어 들인 부트 레코드의 Root Entry Count 항목이 0인지를 조사한다. FAT16은 이 항목이 반드시 0이 아니어야 하므로 파일시스템이 FAT16인지 아닌지를 알아내기에 좋은 항목이다. 물론 이 항목이 0이라고 해서 FAT16이라고 곧바로 단정하기에는 무리가 있지만, 이번 실습에서는 이 항목만으로 판단을 한다. 추후에 파일시스템을 판단하는 더욱 견고한 방법을 알아볼 것이다. 또한 Signature 항목의 값이 0xAA55인지 확인한다.
- **90~92행** : 이제부터가 실질적으로 부트 레코드의 정보를 얻어내는 코드이다. 이 코드는 해당 볼륨의 총 섹터 수를 얻어낸다. 반드시 Total Sector 16이나 Total Sector 32 항목 중 하나만 값을 가져야 하므로 이 코드처럼 짜는 게 맞다.
- **95행** : FAT 영역의 섹터 수를 얻어낸다. FAT16 파일시스템에서는 반드시 FATs16 항목에 FAT 영역의 크기를 담고 있다.
- **98행** : 루트 디렉토리의 Directory Entry 개수를 얻어낸다. Directory Entry에 대해서는 나중에 자세히 설명할 것이다.
- **101행** : 루트 디렉토리의 시작 섹터를 구한다. FAT16에서는 루트 디렉토리의 위치가 FAT 영역 바로 뒤쪽으로 정해져 있으므로 위치를 알기가 매우 쉽다.

  Root Directory 시작 섹터 = 볼륨의 시작 섹터 + 예약된 영역의 크기 + FAT 영역의 크기

  위 공식처럼 구하면 쉽게 루트 디렉토리의 시작 위치를 알아낼 수 있다. FAT32 파일시스템의 경우에는 조금 더 복잡한데, 나중에 살펴볼 것이다.
- **107행** : 루트 디렉토리의 섹터 수를 얻어낸다. 얻어내는 방법은 우선 Root Entry Count 항목에 32를 곱한다(왜 32를 곱하는지는 나중에 알게 된다. 간단히 설명하자면 Directory Entry의 크기가 32Byte이기 때문이다). 이를 통해서 루트 디렉토리의 총 바이트 수를 얻어 온다. 여기서 Bytes Per Sector 항목으로 나누면 루트 디렉토리가 총 몇 개의 섹터를

사용하는지 알 수 있게 된다.

이 항목은 FAT16일 때만 의미가 있다. FAT32일 때는 RootEntCnt 값이 항상 0이므로 계산을 해보면 언제나 0이 나오게 되어 있다.

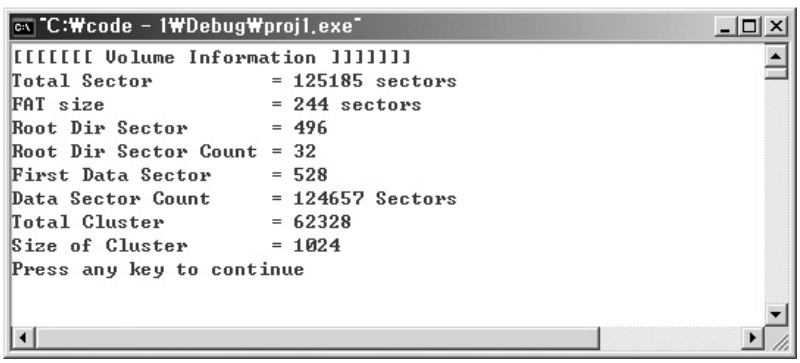
- **110행** : 첫 번째 데이터 섹터를 구하는 코드이다. 부트 레코드 항목 중에는 데이터 영역의 시작 섹터 위치에 대한 항목이 없으므로 위의 공식처럼 다른 항목들을 이용해서 직접 구해내야 한다. 데이터 영역 앞쪽에 있는 부분의 크기를 모두 더하면 데이터 영역의 시작 섹터를 알 수 있다.
- **113행** : 데이터 영역의 섹터 수를 얻어낸다. 이 항목 역시 부트 레코드에 존재하지 않으므로 아래의 공식을 써서 직접 구해야 한다. 공식은 그다지 어렵지 않다.

> 데이터 영역의 섹터 수 = 총 섹터 수– (예약된 영역 + FAT 영역 + 루트 디렉토리 섹터 수)

- **116행** : 볼륨의 총 클러스터 개수를 구한다. 클러스터 단위로 관리되는 영역은 데이터 영역뿐이므로 '데이터 영역의 총 섹터 수 / 클러스터당 섹터 수'를 하면 총 클러스터 개수가 나오게 된다.
- **119행** : 클러스터의 크기를 구한다.
- **122행** : 클러스터가 몇 개의 섹터를 차지하는지를 구한다.

실행 화면

아래는 프로그램의 실행 화면이다.



[그림 3-13] 예제 실행 화면