9

2024년 10월 29일 화요일 오전 10:05

저장소 전략 수립

저장소 옵션

임베디드 장치에서의 저장소는 솔리드 스테이트 스토리지(solid state storage)를 의미

특성	NOR 플래시	NAND 플래시
삭제 블록 크기	일반적으로 128KiB	16KiB ~ 256KiB
삭제 동작	블록을 지우면 모든 비트가 1로 설정	블록을 지우면 모든 비트가 1로 설정
프로그램 단위	워드 단위로 프로그래밍	2 KiB 또는 4 KiB 페이지 단위로 프로그래밍
삭제 사이클 수	보통 100K에서 1M 사이	**SLC :** 최대 100K 사이클 **MLC :** 약 3K ~ 10K 사이클 **TLC :** 약 1K ~ 3K 사이클
신뢰성	삭제 사이클 후 블록 손상 가능	**SLC :** 높은 신뢰성 **MLC 및 TLC :** 상대적으로 낮음
데이터 접근 방식	직접 CPU 주소 공간에 매핑 가능 (XIP 가능)	RAM에 복사 후 접근해야 함
오류 정정코드	필요 없음 (단순 구현 가능)	**SLC :** 일반적으로 간단한 해밍 코드 사용 **MLC :** BCH 코드 또는 LDPC 코드 사용 **TLC :** BCH 코드 또는 LDPC 코드 사용
OOB(Out of Baud) 영역	없음	페이지 당 추가 메모리 영역 필요
용도	부트로더 코드 저장, 커널, 루트 파일시스템 등	SSD, USB 드라이브 등
비용	상대적으로 비쌈	저렴하고 대용량

삭제 블록(deletion block)은 플래시 메모리에서 데이터를 지우는 최소 단위
플래시 메모리는 일반적으로 비트 단위로 데이터를 쓰고 읽을 수 있지만, 데이터를 지울 때는 블록 단위로 지워야 함

- 섹터(Sector) : 하드 디스크의 최소 단위 저장공간
- 페이지(Page) : 플래시 메모리의 최소 단위 저장공간
- 블록(Block) : 여러 개의 페이지가 모인 단위

XIP (Execute In Place) 란?
메모리에서 데이터를 읽어 실행하는 방식
데이터를 RAM으로 복사할 필요 없이 직접 메모리에서 실행할 수 있게 해줌

BCH (Bose-Chaudhuri-Hocquenghem)
LDPC (저밀도 패리티 검사, Low-Density Parity Check)

CFI (Common Flash Interface) 란?
플래시 메모리 장치의 표준 인터페이스를 정의하는 규격
규격은 다양한 플래시 메모리 칩과 시스템 간의 호환성을 높이기 위해 개발

<NOR 플래시>

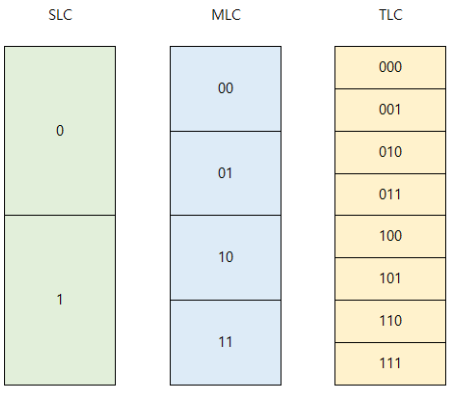
NOR 플래시 칩에는 CFI(Common Flash Interface)라고 불리는 표준 레지스터 레벨의 인터페이스가 있음

<NAND 플래시>

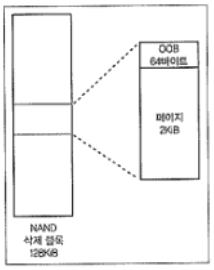
1세대: 메모리 셀당 1비트 저장, **SLC(Single Level Cell)**
2세대: 메모리 셀당 2비트저장, **MLC(Multi Level Cell)**
3세대: 메모리 셀당 3비트 저장, **TLC(Tri Level Cell)**

칩과 칩 사이의 데이터 전송은 비트 플립이 발생하기 쉬우며 비트 플립은 오류 정정 코드(ECC, Error Collection Code)를 사용해 탐지하고 수정할 수 있음

- **SLC**:
  - 일반적으로 32바이트의 메인 저장소당 1바이트 OOB를 사용, 2KiB 페이지는 OOB가 64바이트, 4KiB 페이지는 128바이트
- **MLC와 TLC**:
  - 더 복잡한 ECC와 메타데이터를 수용하기 위해 더 큰 OOB 영역을 갖음



(128KiB 의 삭제 블록과 2KiB 페이지를 가진 칩의 구조)



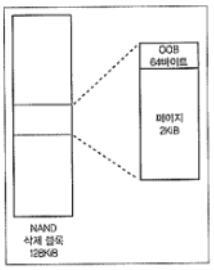


그림 9.1 OOB 영역

- 불량 블록 관리:
  - 제조사는 모든 블록을 테스트하고, 문제가 있는 블록은 OOB 영역에 배드 블록 플래그를 설정하여 표시
- 오류 검출 및 표시:
  - NAND 플래시 드라이버는 오류를 검출하고 불량으로 표시해야 함
- NAND 플래시 컨트롤러:
  - NAND 플래시 칩에 접근하려면 SoC(System on Chip)의 일부인 NAND 플래시 컨트롤러가 필요
    부트로더와 커널에 상응하는 드라이버가 필요
- ONFI:
  - ONFI는 NAND 플래시 칩용 표준 레지스터 레벨 인터페이스를 제공

배드 플래그(Bad Flag) 란?
플래시 메모리에서 특정 셀이나 블록이 더 이상 신뢰할 수 없거나 사용할 수 없음을 나타내는 표시

ONFI (Open NAND Flash Interface) 란?
NAND 플래시 메모리의 인터페이스 표준을 정의하는 조직

<컨트롤러가 내장된 플래시>

컨트롤러가 내장된 플래시는 하나 이상의 플래시 칩을 마이크로 컨트롤러와 결합하는 것을 의미
마이크로 컨트롤러는 섹터 크기가 작고 기존 파일시스템과 호환되는 이상적인 저장 장치를 제공
임베디드 시스템용 칩에는 SD 카드와 eMMC가 있음

- **컨트롤러가 내장된 플래시**

  컨트롤러가 내장된 플래시는 플래시 칩과 마이크로컨트롤러를 결합한 저장 장치
  섹터 크기가 작고 기존 파일 시스템과 호환되며, 주요 임베디드 시스템용 칩에는 SD 카드와 eMMC가 있음

  1. 멀티미디어 카드와 시큐어 디지털 카드

  - **멀티미디어카드 (MMC)**
    1997년 플래시 메모리를 이용해 패키징된 저장소로 소개
  - **시큐어 디지털 카드 (SD)**
    1999년에 MMC 기반으로 암호화와 DRM을 추가하여 탄생

- **용도:** 디지털 카메라, 음악 플레이어 등 가전제품에 사용
- **기능**: 최신 SD 카드 사양은 더 작은 패키징과 높은 용량을 제공
- **하드웨어 인터페이스: MMC와 SD 카드의 인터페이스는 유사하여 풀 사이즈 SD 슬롯에 풀 사이즈 MMC 사용 가능**
- **인터페이스:** 초기에는 1비트 SPI 사용, 최근 카드는 4비트 인터페이스 사용

  - 512바이트의 섹터로 메모리 읽기와 쓰기 명령어 제공

그림 9.2 SD 카드 패키지

2. eMMC (embedded MMC)

- 마더보드에 납땜 가능한 형태로 패키징된 MMC 메모리
인터페이스: 데이터 전송을 위해 4비트 또는 8비트 인터페이스 사용

3. 기타 종류의 플래시

- **컴팩트플래시 (CF):**
- 초기 컨트롤러가 내장된 플래시 기술로, PCMCIA 하드웨어 인터페이스 사용
- 병렬 ATA 인터페이스를 통해 메모리에 접근
- X86 기반 싱글 보드 컴퓨터 및 카메라 기기에서 사용

CMCIA(Personal Computer Memory Card International Association) 란?
개인용 컴퓨터에 메모리 카드 및 다른 장치를 연결하기 위한 표준 인터페이스

- **USB 플래시 드라이브:**
- USB 인터페이스를 통해 접근되며, USB 대용량 스토리지 스펙을 따름
- 일반적으로 FAT 파일 시스템으로 포맷됨
- 임베디드 시스템에서 데이터 전송에 주로 사용

- **유니버설 플래시 저장 장치:**
- eMMC와 같이 마더보드에 탑재되며, 빠른 시리얼 인터페이스를 갖고 있음
  eMMC보다 더 빠른 데이터 속도를 지원하고 SCSI 디스크 명령어 세트를 지원

부트로더에서 플래시 메모리 접근하기

<U-Boot와 NOR 플래시>

- **드라이버 위치:** U-Boot 소스 코드의 `drivers/mtd` 폴더에 NOR CFI 칩을 지원하는 드라이버 존재
- **주요 명령어:**
  - `erase`: 메모리 삭제
  - `cp.b`: 바이트 단위 데이터 복사

(0x40000000에서 0x48000000까지 매핑돼 있는 NOR 플래시 메모리를 가지고 있으며 0x40040000에서 시작하는 4MiB는 커널 이미지라고 가정, 다음과 같은 U-Boot 명령어로 새로운 커널을 플래시에 로드)

```
=> tftpboot 100000 uImage
=> erase 40040000 403fffff
=> cp.b 100000 40040000 $(filesize)
```

<U-Boot와 NAND 플래시>

- **드라이버 위치:** `drivers/mtd/nand` 폴더에 NAND 플래시를 위한 드라이버 존재
- **주요 명령어:**
  - `erase`: 메모리 블록 삭제
  - `write`: 데이터를 메모리에 기록
  - `read`: 메모리에서 데이터 읽기
- Nand write는 배드로 표시된 블록을 건너 뜀

(0x82000000에 로드한 후 옵셋 0x280000으로 시작하는 플래시에 배치되는 것을 보여줌)

```
=> tftpboot 82000000 uImage
=> nand erase 280000 400000
=> nand write 82000000 280000 $(filesize)
```

<U-Boot와 MMC, SD, eMMC>

- **드라이버 위치:** U-Boot 소스 코드의 `drivers/mmc` 폴더에 여러 MMC 컨트롤러의 드라이버 존재
- **주요 명령어:**
  - `mmc read`, `mmc write`: 원시 데이터 접근 및 원시 커널과 파일시스템 이미지를 다룰 수 있게 해줌
- **파일 시스템 지원:**
  - FAT32, ext 파일 시스템에서 파일 읽기 가능

U-Boot 는 NOR, NAND, 컨트롤러가 내장된 플래시에 접근하기 위해 드라이버가 필요함
드라이버 선택
- SoC에 따라 NOR 칩이나 플래시 컨트롤러 선택
- 리눅스에서 원시 NOR/NAND 플래시에 접근하려면 추가 소프트웨어 레이어 필요

리눅스에서 플래시 메모리 접근

MTD(Memory Technology Device) 서브 시스템
- 원시 NOR와 NAND 플래시 메모리를 관리하는 시스템
- 플래시 메모리 블록을 읽고, 지우고, 쓰기 위한 기본 인터페이스를 제공

컨트롤러가 내장된 플래시
- 특별한 하드웨어 인터페이스를 처리하기 위한 드라이버가 필요

- **MMC/SD 카드 및 eMMC:** `mmcblk` 드라이버를 사용
- **CompactFlash 및 하드 드라이브:** SCSI 디스크 드라이버(`sd`)를 사용
- **USB 플래시 드라이브:** `sd` 드라이버와 함께 `usb_storage` 드라이버를 사용

<MTD>

- **MTD의 세가지 레이어**

1. **함수 코어 세트**: MTD의 기본 기능을 제공하는 함수들
2. **칩 드라이버 세트**: 다양한 종류의 플래시 메모리 칩을 위한 드라이버
3. **사용자 레벨 드라이버**: 플래시 메모리를 문자 장치나 블록 장치로 표시하는 드라이버

칩 드라이버

- **레벨**: 칩 드라이버는 가장 낮은 레벨에서 플래시 칩과의 인터페이스를 담당
- **NOR 플래시 칩**: CFI(공통 플래시 인터페이스) 표준을 지원하는 드라이버가 필요하며, 사용하지 않는 비호환 칩도 필요
- **NAND 플래시 칩**: 현재 사용되는 NAND 플래시 컨트롤러를 위한 드라이버가 필요하며, 이 드라이버는 보드 지원 패키지(BSP)의 일부로 공급
- driver/mtd/nand 디렉터리에는 현재 주류 커널에 약 40개 이상의 NAND 플래시 드라이버가 포함

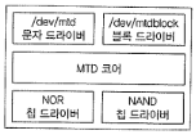


그림 9.3 MTD 레이어

- **MTD 파티션**
  MTD에서는 파티션의 사이즈와 위치를 지정하기 위한 몇가지 방법이 있음
  a. CONFIG_MTD_CMDLINE_PARTS 를 사용하는 커널 명령줄을 통한 방법
  b. CONFIG_MTD_QF_PARTS를 사용하는 장치 트리를 통한 방법
  c. 플랫폼 매핑 드라이버를 이용하는 방법

a. CONFIG_MTD_CMDLINE_PARTS 를 사용하는 커널 명령줄을 통한 방법
커널 명령줄 옵션은 mtdparts

(rivers/mtd/cmdlinepart.c의 리눅스 소스코드에 다음과 같이 정의)

```
mtdparts=<mtddef>[;<mtddef]
<mtddef> := <mtd-id>:<partdef>[,<partdef>]
<mtd-id> := unique name for the chip
<partdef> := <size>[@<offset>][<name>][ro][lk]
<size> := size of partition OR "-" to denote all remaining space
<offset> := offset to the start of the partition; leave blank to follow the
    previous partition without any gap
<name> := '(' NAME ')'
```

(ex. 128MiB의 플래시 칩이 5개의 파티션으로 분할되는 경우)

```
mtdparts=:512k(SPL)ro,780k(U-Boot)ro,128k(U-BootEnv), 4m(Kernel),-(Filesystem)
```

- 512k(SPL)ro:
  - 512KB 크기의 SPL (Secondary Program Loader) 파티션, 읽기 전용(ro).
- 780k(U-Boot)ro:
  - 780KB 크기의 U-Boot 부트로더 파티션, 읽기 전용(ro).
- 128k(U-BootEnv):
  - 128KB 크기의 U-Boot 환경 설정 파티션.
- 4m(Kernel):
  - 4MB 크기의 커널 파티션.
- -(Filesystem):
  - 나머지 공간은 파일시스템을 위한 파티션으로 사용

- 각 칩에 대한 정보는 세미콜론(;)으로 구분
- 각 파티션은 쉼표(,)로 분리되며, 크기는 바이트, KiB, MiB로 나타내고, 파티션 이름은 괄호로 둘러싸여 있음

(/proc/mtd 파일을 읽으면 구성 정보의 요약을 볼 수 있음)

```
# cat /proc/mtd
dev: size erasesize name
mtd0: 00080000 00020000 "SPL"
mtd1: 000C3000 00020000 "U-Boot"
mtd2: 00020000 00020000 "U-BootEnv"
mtd3: 00400000 00020000 "Kernel"
mtd4: 07A9D000 00020000 "Filesystem"
```

mtd0: 00080000 00020000 "SPL"
- **dev**: mtd0는 이 파티션의 디바이스 이름, 시스템에서 첫 번째 MTD 장치를 의미
- **size**: 00080000은 이 파티션의 크기를 **16진수**로 나타낸 것, 이를 **10진수**로 바꾸면 524288 바이트, 즉 **512KB**
- **erasesize**: 00020000은 이 파티션에서 하나의 "블록"을 지우는 데 필요한 최소 크기, 이를 **10진수**로 바꾸면 131072 바이트, 즉 **128KB**
- **name**: "SPL"은 이 파티션의 이름, 보통 **Secondary Program Loader**를 저장하는 파티션

(/sys/class/mtd에는 삭제 블록 크기와 페이지 크기를 포함한 각 파티션에 대한 자세한 정보가 있으며,
mtdinfo를 사용해 다음과 같이 자세한 요약 정보를 볼 수 있음)

```
# mtdinfo /dev/mtd0
mtd0
Name:            SPL
Type:            nand

Eraseblock size:       131072 bytes, 128.0 KiB
Amount of eraseblocks:     4 (524288 bytes, 512.0 KiB)
Minimum input/output unit size: 2048 bytes
Sub-page size:         512 bytes
OOB size:         64 bytes
Character device major/minor:  90:0


 Bad blocks are allowed:    true
 Device is writable:    false
```

---

b. CONFIG_MTD_QF_PARTS를 사용하는 방법 (CONFIG_MTD_QF_PARTS를 사용하는 장치 트리를 통한 방법)

(장치 트리를 통해 명령줄 예제와 동일한 파티션을 생성)

```
nand@0,0 {
  #address-cells = <1>;
  #size-cells = <1>;
  partition@0 {
    label = "SPL";
    reg = <0 0x80000>;
  };
  partition@80000 {
    label = "U-Boot";
    reg = <0x80000 0xc3000>;
  };
  partition@143000 {
    label = "U-BootEnv";
    reg = <0x143000 0x20000>;
  };          .
  partition@163000 {
    label = "Kernel";
    reg = <0x163000 0x400000>;
  };
  partition@563000 {
    label = "Filesystem";
    reg = <0x563000 0x7a9d000>;
  };
};
```

---

c. 플랫폼 매핑 드라이버를 사용하는 방법 (BSP 내에서 코드로 파티션 설정)

(arch/arm/mach-omap2/board-omap3beagle.c에서 가져온 후 예제처럼 mtd_partition 구조 안의
플랫폼 데이터로 파티션 정보를 코딩하는 것)

```
static struct mtd_partition omap3beagle_nand_partitions[] = {
  {
    .name = "X-Loader",

   .offset = 0,
   .size = 4 * NAND_BLOCK_SIZE,
   .mask_flags = MTD_WRITEABLE, /* force read-only */
 },
 {
   .name = "U-Boot",
   .offset = 0x80000;
   .size = 15 * NAND_BLOCK_SIZE,
   .mask_flags = MTD_WRITEABLE, /* force read-only */
 },
 {
   .name = "U-Boot Env",
   .offset = 0x260000;
   .size = 1 * NAND_BLOCK_SIZE,
 },
 {
   .name = "Kernel",
   .offset = 0x280000;
   .size = 32 * NAND_BLOCK_SIZE,
 },
 {
   .name = "File System",
   .offset = 0x680000;
   .size = MTDPART_SIZ_FULL,
  },
};
```

플랫폼 데이터는 더 이상 지원하지 않으며, 장치 트리를 사용하기 위해 업데이트 하지 못하는 기존 SoC용 BSP에서만 사용

<MTD 장치 드라이버>

메이저 번호
- 장치 드라이버를 구분하는 번호

마이너 번호
- 특정 장치 인스턴스를 구분하는 번호

- 90의 메이저 번호를 가진 **문자 장치**
  MTD 파티션 번호마다 N:/dev/mtdN(마이너번호=N*2)와 /dev/mtdNro(마이너번호=N=2+1)의 2개 장치 노드가 있음
  후자는 전자의 읽기 전용 버전

- 31의 메이저 번호와 N의 마이너 번호를 가진 **블록 장치** (장치 노드들은 /dev/mtdblockN 형태)

- **MTD 문자 장치, mtd**
  문자 장치는 기저의 플래시 메모리에 바이트 배열로 접근할 수 있으므로 플래시를 읽고 쓸 수 있음
  또한 블록을 지우고 NAND 칩의 OOB영역을 관리할 수 있는 여러 ioctl 함수를 구현

  (include/uapi/mtd/mtd-abi.h)

MEMGETINFO	기본적인 MTD 문자 장치 정보를 얻음
MEMERASE	MTD 파티션의 블록을 지움
MEMWRITEOOB	OOB 데이터를 씀
MEMREADOOB	OOB 데이터를 읽음
MEMLOCK	칩을 잠금
MEMUNLOCK	칩을 잠금 해제
MEMGETREGIONCOUNT	소거 영역의 개수를 가져옴 만약 파티션에 다른 크기의 삭제 블록이 있다면 0이 아님
MEMGETREGIONINFO	MEMGETREGIONCOUNT가 0이 아니면 옵셋, 크기, 각 영역의 블록 개수를 가져오는 데 사용할 수 있음
MEMGETOOBSEL	사용 안함
MEMGETBADBLOCK	배드 블록 플래그를 가져옴
MEMSETBADBLOCK	배드 블록 플래그를 세팅

OTPSELECT	칩에서 지원한다면, OTP 모드로 세팅
OTPGETREGIONCOUNT	OTP 영역의 개수를 가져옴
OTPGTREGIONINFO	OTP 영역에 대한 정보를 가져옴
ECCGETLAYOUT	사용 안함

Ioctl 함수를 사용하는 플래시 메모리를 다루기 위해 mtd-utils로 알려진 유틸리티 프로그램 세트

flash_erase	일정 범위의 블록을 지움
flash_lock	일정 범위의 블록을 잠금
flash_unlock	일정 범위의 블록을 잠금 해제
nanddump	NAND 플래시에서 메모리를 덤프 선택적으로 OOB 영역을 포함할 수 있며, 배드블럭은 건너 뜀
nandtest	NAND 플래시를 테스트하고 진단
nandwrite	배드 블록들을 건너 뛰면서 파일의 데이터를 NAND 플래시에 씀

=> 플래시 메모리에서 새로운 내용을 쓰기 전에 항상 플래시 메모리를 지워야 함

- **MTD 블록 장치 (mtdblock)**

목적: 플래시 메모리를 블록 장치로 제공하여 파일 시스템을 포맷하고 마운트하는 데 사용
특징: 신뢰성이 있는 파일 저자오를 위한 플래시 변환 레이어가 없음
사용 사례: 안정적인 NOR 플래시 메모리 위에 SquashFS와 같은 읽기 전용 파일 시스템을 마운트하는 경우에 적합

- **커널 Oops를 MTD에 로깅**

정의: 커널 에러(oops)는 보통 klogd와 syslogd 데몬을 통해 순환 메모리 버퍼나 파일에 로깅
문제점: 재부팅 후 링 버퍼의 로그가 사라지며, 파일에 기록된 로그도 손실될 수 있음
해결 방법:
  - **MTD에 로깅**:
    MTD 파티션에 순환 로그 버퍼로 **oops**와 **커널 패닉**을 저장하여 더 안전하게 로그를 보존

설정 방법:
  커널 설정: CONFIG_MTD_OOPS 활성화
  커널 명령줄에 console=ttyMTD<N> 추가 (여기서 N은 로그를 쓸 MTD 장치 번호)

- **NAND 메모리 시뮬레이션**

기능: 시스템 RAM을 사용해 NAND 칩을 에뮬레이트
목적: 물리적 NAND 메모리에 접근하지 않고도 NAND를 인식해야 하는 코드를 테스트
특징: 배드 블록, 비트 플립 등의 에러를 시뮬레이트하여 실제 플래시 메모리를 이용하기 어려운 코드를 테스트할 수 있음

<MMC 블록 드라이버>

사용: MMC/SD 카드 및 eMMC 칩에 접근하기 위해 mmcblk 블록 드라이버를 사용
요구 사항: BSP의 일부분으로 사용 중인 MMC 어댑터와 매치되는 호스트 컨트롤러가 필요
소스 코드 위치: 드라이버는 drivers/mmc/host에 위치
파티션 테이블: MMC 저장소는 fdisk와 같은 유틸리티를 사용하여 하드디스크와 동일한 방법으로 분할

플래시 메모리를 위한 파일시스템

플래시 메모리는 특유의 쓰기 제한과 배드 블록 관리, 웨어 레벨링 등을 처리하는 방식 때문에 일반적인 저장 장치와는 다르게 관리가 필요

<플래시 변환 레이어 (FTL, Flash Translaton Layer)>

- **플래시 변환 레이어의 특징**

1. **하위 할당:** 파일 시스템은 보통 512바이트 섹터의 작은 할당 단위로 해야 가장 잘 동작함, 이는 128KiB 이상의 플래시 삭제 블록보다 훨씬 작음. 따라서 삭제 블록은 많은 양의 공간 낭비를 없애기 위해 더 작은 단위로 세분화 해야 함

2. **가비지 컬렉션garbage collection:** 파일시스템이 얼마 동안 사용된 후 하위 할당의 결과로 삭제 블록은 정상적인 데이터와 그렇지 못한 데이터들이 섞이게 됨
   전체 삭제 블록만을 해제할 수 있으므로, 빈 공간을 다시 되찾기 위한 방법은 정상적인 데이터를 하나의 영역에 합치고 이제 비어 있는 삭제 블록을 빈 공간으로 리턴하는 것,이를 가비지 컬렉션이라고 하며, 일반적으로 백그라운드 스레드로 구현됨

3. **웨어 레벨링wsar levellng:** 각 블록은 삭제 사이를 횟수에 제한이 있음
   칩의 수명을 최대로 늘리기 위해 각 블록이 거의 같은 횟수만큼 삭제되도록 데이터를 이동하는 것은 중요

4. **배드 블록 처리bad block handling:** NANI 플래시 칩에서 배드로 표시된 볼록은 사용하지 말아야 하고, 만약 정상 블록이 삭제되지 않는다면 배드 블록으로 표시해야 함

5. **견고함robustness:** 임베디드 장치는 경고 없이 파워 오프나 리셋될 가능성이 있다. 그래서 일반적으로 저널ounal이나 트랜잭션 로그를 결합시킴으로써 어떤 파일시스템이라도 손상 없이 대응할 수 있어야 함

플래시 변환 레이어를 배치하기 위한 여러가지 방법

1. 파일 시스템에서: JFFS2, YAFFS2, UBIFS와 마찬가지로
2. 블록 장치 드라이버에서: UNIFS와 의존 관계인 UBI 드라이버는 플래시 변환 레이어의 일부를 구현
3. 장치 컨트롤러에서: 컨트롤러가 내장된 플래시 장치와 마찬가지로

NOR와 NAND 플래시 메모리를 위한 파일시스템

대용량 저장소용으로 원시 플래시 칩을 사용하기 위해 기본 기술의 특성과 잘 맞는 파일 시스템을 사용 해야 하는데,
세가지 파일 시스템이 있음

1. **JFFS2(Journalling Flash File Sysytem 2):**
   리눅스용 최초의 플래시 파일시스템 이었으며, 지금도 여전히 사용중
   NOR와 NAND 메모리용으로 동작하지만 마운트 중에는 속도가 매우 느림

2. **YAFFS2(Yet Another Flash File System 2):**
   JFFS2와 비슷하지만 NAND 플래시 메모리를 위한 파일시스템

3. **UBIFS(Unsorted Block Image File System):**
   신뢰성 있는 플래시 파일시스템을 만들기 위해 UBI 블록 드라이버와 연계해 동작
   NOR와 NAND 메모리 모두에 잘 동작

=> 세가지 모두 플래시 메모리의 공통 인터페이스로 MTD 를 사용

<JFFS2>

구조: MTD를 사용하여 플래시 메모리에 접근하는 로그 구조 파일 시스템. 변경 사항은 노드로 순차적으로 기록됨
노드: 파일명 수정, 데이터 변경 사항 등을 포함

JFFS2는 삭제 블록을 다음 세가지 유형으로 분류

1. **Free:** 노드가 전혀 없음
2. **Clean:** 유효한 노드들만 포함
3. **Dirty:** 최소 하나 이상의 사용하지 않는 노드를 포함

- **오픈 블록:** 업데이트를 받을 준비가 된 블록
  만약 파워가 종료되거나 시스템이 리셋되면, 잃어버릴 수 있는 데이터는 오픈 블록에 마지막으로 쓰임

효과적으로 플래시 칩의 저장 용량을 증가시키기 위해 노드는 기록되는 대로 압축
사용 가능한 블록의 수가 기준 점 아래로 떨어지면, 가비지 컬렉터 커널 스레드가 시작돼 더티 블록을 검색하고
유효한 노드를 오픈 블록에 복사한 다음 더티 블록을 해제

- **요약 노드**
  JFFS2의 단점 및 개선 사항

  단점
  - **온 칩 인덱스 없음:**
    마운트 시 전체 로그 파일을 읽어야 하므로 디렉터리 구조를 추정해야 함
  - **긴 마운트 시간:**
    파티션 크기에 비례해 시간이 걸리며, 메가바이트당 약 1초 소요. 결과적으로 수십 혹은 수백 초가 걸릴 수 있음

  개선 사항 (리눅스 2.6.15)
  - 요약 노드 도입: 마운트 중 스캔 시간을 줄이기 위해 삭제 블록 끝에 정보를 저장
  - 효과: 마운트 시 필요한 정보가 포함되어 데이터 처리량이 줄어들어 마운트 시간이 두 배에서 다섯 배 정도 감소
  - 활성화: 커널 구성 요소 `CONFIG_JFFS2_SUMMARY`로 설정 가능

- **클린 마커**
  모든 비트가 1로 세팅된 삭제된 블록은 1로 쓰여진 블록들과 구분할 수 없음
  후자는 메모리 셀을 새로 고치지 않았기 때문에 지워질 때까지 다시 프로그램을 할 수 없음
  JFFS2 는 이런 두가지 상황을 구별할 수 있는 클린 마커라는 매커니즘을 사용
  성공적으로 블록을 삭제한 후, 블록의 시작 부분이나 블록의 첫번째 페이지 OOB 영역에서 클린 마커가 쓰임

- **JFFS2 파일 시스템 만들기**

  빈 JFFS2 파일시스템은 프리 블록으로 구성돼 있기 때문에 포맷하는 단계 없음
  예를 들어, MTD 파티션 6을 포맷하려면 장치에 다음과 같은 명령어를 입력

```
# flash_erase -j /dev/mtd6 0 0
# mount -t jffs2 mtd6 /mnt
```

  fresh_erase의 -j 옵션은 클린 마커를 추가하고, 빈 파일시스템으로서 jffs2 타입 파티션으로 마운트
  한 번 마운트되면, 다른 파일시스템처럼 동일하게 취급

mkfs.jffs2를 사용해 JFFS2 포맷으로 파일을 생성하고 sumtool을 사용해 요약 노드들을 추가함으로써
개발 시스템의 스테이징 영역에서 직접 파일시스템 이미지를 생성 가능

Ex. 128KB(ex20008) 크기의 삭제 블록과 요약 노드가 있는 NAND 플래시 장치의 경우
rootfs에 파일 이미지를 만들려면 다음 두 명령어를 사용

```
$ mkfs.jffs2 -n -e 0x20000 -p -d ~/rootfs -o ~/rootfs.jffs2
$ sumtool -n -e 0x20000 -p -i ~/rootfs.jffs2 -o ~/rootfs-sum.jffs2
```

-p 옵션: 전체 삭제 블록의 횟수를 추가하기 위해 이미지 파일 끝에 패딩을 추가
-n 옵션: 이미지 안에 클린 마커를 생성하지 못하게 함
사용 권한과 소유권을 세팅하기 위해 -D [장치 테이블]을 추가함으로써 mkfs.jffs2의 옵션으로 장치 테이블 사용 가능

부트로더에서 플래시 메모리로 이미지를 프로그래밍 가능
예를 들어 파일시스템 이미지를 0x82000000 주소로 램에 로드하며, 그것은 플래시 칩 시작으로부터 0x163000번째 바이트에서 시작됨
길이가 0x7a9d000 바이트인 플래시 파티션에서 로드하려는 경우, U-Boot 명령어는 다음과 같음

```
nand erase clean 163000 7a9d000
nand write 82000000 163000 7a9d000
```

Mtd 드라이버를 이용해 리눅스에서 다음과 같이 동일한 작업을 수행할 수 있음

```
# flash_erase -j /dev/mtd6 0 0
# nandwrite /dev/mtd6 rootfs-sum.jffs2
```

JFFS2 루트 파일 시스템으로 부팅하려면, JFFS2는 자동 검출을 할 수 없으므로 커널 명령줄에서 파티션과 rootfstpye을 알려줘야 함

<YAFFS2>

YAFFS는 JFFS2처럼 동일한 설계 원칙을 따르고 있는 로그 구조 파일 시스템
더빠른 마운트 시간 스캔, 간단하고 빠른 가비지 컬렉션을 갖고 있으며 압축하지 않는다는 점
저장소 사용의 효율성이 떨어지는 대신 읽기와 쓰기의 퍼포먼스를 향상 시킨다는 것을 의미

YAFFS는 리눅스 뿐만 아니라 다양한 운영체제에 포팅되고, 듀얼 라이선스이므로 리눅스와의 호환을 위한 GPLv2와 다른 운영체제를 위한 상용 라이선스를 갖고 있음

(YAFF2를 가져오고 커널에 패치)

```
$ git clone git://www.aleph1.co.uk/yaffs2
$ cd yaffs2
$ ./patch-ker.sh c m <path to your link source>
```

- **YAFFS2 파일 시스템 만들기**

런타임에 YAFFS2 파일 시스템을 만들기 위해 파티션을 지우고 마운트만 하면 됨, 단 클린 마커를 사용하지 말아야 함

```
# flash_erase /dev/mtd/mtd6 0 0
# mount -t yaffs2 /dev/mtdblock6 /mnt
```

파일 시스템 이미지를 생성하기 위해 가장 간단한 방법은 mkyaffs2 도구를 사용하는 것

```
$ mkyaffs2 -c 2048 -s 64 rootfs rootfs.yaffs2
```

-c: 페이지 크기
-s: OOB 크기

타깃의 리눅스 셸 프롬프트에서 이미지를 MTD 파티션에 복사하기

```
# flash_erase /dev/mtd6 0 0
# nandwrite -a /dev/mtd6 rootfs.yaffs2
```

YAFFS2 파일 시스템을 부트하기 위해 커널 명령줄에 다음을 추가

```
root=/dev/mtdblock6 rootfstype=yaffs2
```

원시 NOR와 NAND 플래시를 위한 파일시스템 관련 주제를 알아보는 동안 좀 더 최신인 옵션 중 하나를 살펴봄
이 파일 시스템은 UBI 파일 드라이버 위에서 실행

<UBI와 UBIFS>

UBI 드라이버는 배드 블록을 처리하고 웨어 레벨링을 관리하는 플래시 메모리용 볼륨 관리자

- **UBI(Unsorted Block Image)**

- **목적**: 물리적 삭제 블록과 논리적 삭제 블록을 매핑하여 신뢰할 수 있는 플래시 메모리 뷰를 제공
- **배드 블록 처리**: 블록이 삭제되지 않으면 배드로 마크되어 매핑에서 삭제
- **LEB 헤더**: LEB(논리적 삭제 블록)의 헤더는 PEB(물리적 삭제 블록)가 삭제된 횟수를 카운트하고, 이를 기반으로 매핑을 조정하여 균형 잡힌 삭제를 수행
- **MTD 레이어**: MTD 레이어를 통해 플래시 메모리에 접근하며, 여러 개의 UBI 볼륨으로 나눌 수 있음
- **웨어 레벨링**: 두 개의 파일 시스템(정적 데이터와 변동 데이터)이 있을 경우, 단일 MTD 파티션에서 UBI 볼륨을 사용하면 웨어 레벨링이 두 영역에 걸쳐 수행되어 플래시 메모리 수명이 늘어남

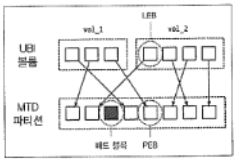


그림 9.4 UBI 볼륨

이런 방식으로 웨어 레벨링과 배드 블록 처리 충족

UBI용으로 MTD 파티션을 준비하기 위해 JFFS2와 YAFFS2의 flash_erase를 사용하지 않는 대신
PED 헤더에 저장돼 있는 삭제 횟수를 보존하는 ubiformat 유틸리티를 사용

```
# ubiformat /dev/mtd6 -s 2048
ubiformat: mtd0 (nand), size 134217728 bytes (128.0 MiB),
1024 eraseblocks of 131072 bytes (128.0 KiB),
min. I/O size 2048 bytes
```

ubiattach 명령어를 사용해 다음과 같이 준비된 MTD 파티션에 UBI 드라이버를 로드할 수 있음

```
  # ubiattach -p /dev/mtd6 -O 2048
  UBI device number 0, total 1024 LEBs (130023424 bytes, 124.0 MiB),
  available 998 LEBs (126722048 bytes, 120.9 MiB),
  LEB size 126976 bytes (124.0 KiB)
```

위 명령어는 UBI 볼륨에 접근할 수 있는 장치 노드인 /dev/ubi0을 생성
여러 다른 MTD 파티션에서 ubiattach 를 사용할 수 있으며 /dev/ubi1,/dev/ubi2 등을 통해 접근할 수 있음

Ubiformat 실행 이후 처음 MTD 파티션으로 접속하면 볼륨이 없을 것인데, unimkvol 을 사용하면 볼륨을 생성할 수 있음

```
# ubimkvol /dev/ubi0 -N vol_1 -s 32MiB
Volume ID 0, size 265 LEBs (33648640 bytes, 32.1 MiB),
LEB size 126976 bytes (124.0 KiB), dynamic, name "vol_1",
alignment 1
# ubimkvol /dev/ubi0 -N vol_2 -m
Volume ID 1, size 733 LEBs (93073408 bytes, 88.8 MiB),
LEB size 126976 bytes (124.0 KiB), dynamic, name "vol_2",
alignment 1
```

=> /dev/ubi0_0과 /dev/uni0_1이라는 2개의 노드를 가진 장치가 만들어짐

uninfo를 사용하면 현재 상황 확인 가능

```
# ubinfo -a /dev/ubi0
ubi0
Volumes count: 2
Logical eraseblock size: 126976 bytes, 124.0 KiB
Total amount of logical eraseblocks: 1024 (130023424 bytes,



124.0 MiB)
Amount of available logical eraseblocks: 0 (0 bytes)
Maximum count of volumes 128
Count of bad physical eraseblocks: 0
Count of reserved physical eraseblocks: 20
Current maximum erase counter value: 1
Minimum input/output unit size: 2048 bytes
Character device major/minor: 250:0
Present volumes: 0, 1

Volume ID: 0 (on ubi0)
Type: dynamic
Alignment: 1
Size: 265 LEBs (33648640 bytes, 32.1 MiB)
State: OK
Name: vol_1
Character device major/minor: 250:1
----------------------------------
Volume ID: 1 (on ubi0)
Type: dynamic
Alignment: 1
Size: 733 LEBs (93073408 bytes, 88.8 MiB)
State: OK
Name: vol_2
Character device major/minor: 250:2
```

이 시점에는 용량이 각각 32MiB와 88MiB 인 2개의 UBI 볼륨을 포함하는 128 MiB MTD 파티션이 있음
사용할 수 있는 총 저장 용량은 32 MiB와 88.8MiB 를 더한 120.8 MiB 임
남은 공간인 7.2 MiB는 각 PEB시작 부분의 UBI 헤더에 할당되고, 칩의 수명이 남아있는 동안 배드 블록을 매핑하기 위해 예약돼 있음

- **UBIFS**

UBIFS (Unsorted Block Image File System)

- **목적**: UBI 볼륨을 사용하여 견고한 파일 시스템을 생성하며, 하위 할당과 가비지 컬렉션을 통해 플래시 변환 레이어를 제공

  - JFFS2, YAFFS2와 달리 칩에 인덱스 정보를 저장하여 마운팅 속도가 빠르지만 UBI 볼륨 추가에는 시간 소요
  - 일반 디스크 파일 시스템처럼 write-back 캐싱을 사용하여 쓰기 속도가 빠르지만, 전원이 꺼지면 데이터 손실 위험
- **해결책**: fsync(2)와 fdatasync(2) 함수를 신중하게 사용하여 결정적인 시점에 파일 데이터를 플래시
- **저널링**: 전원이 꺼질 때 빠른 복구를 위한 저널이 존재
- **볼륨 마운트**: UBI 볼륨을 생성한 후 /dev/ubi0_0와 같은 장치 노드를 사용해 볼륨을 마운트할 수 있음

전체 파티션에 볼륨 이름을 더한 장치 노드를 사용해 마운트 가능 (아래 예제)

```
# mount -t ubifs ubi0:vol_1 /mnt
```

UBIFS 용 파일 시스템 이미지를 생성하는 것은 두 단계로 이뤄짐
Mkfs.ubifs 를 사용해 UBIFS 이미지를 생성하고, ubinize를 사용해 UBI 볼륨에 임베디드
첫번째 단계에서 mkfs.ubifs는 -m의 페이지 크기, -e의 UBI LEB 크기, -c 의 볼륨에서의 최대 삭제 불록 횟수 정보가 필요

(rootfs 디렉터리의 내용을 가져와 rootfs.ubi 라는 이름의 UBIFS 이미지를 생성하려면 다음과 같이 입력)

```
$ mkfs.ubifs -r rootfs -m 2048 -e 124KiB -c 256 -o rootfs.ubi
```

두번째 단계는 이미지의 각 볼륨이 지닌 특성을 알려주는 ubinize에 대한 구성 파일을 생성해야 함
구성 파일 예제는 vol_1과 vol_2 2개의 볼륨을 생성

```
[ubifsi_vol_1]
mode=ubi
image=rootfs.ubi
vol_id=0
vol_name=vol_1


vol_size=32MiB
vol_type=dynamic

[ubifsi_vol_2]
mode=ubi
image=data.ubi
vol_id=1
vol_name=vol_2
vol_type=dynamic
vol_flags=autoresize
```

두번째 볼륨은 auto-resize 플래그를 갖고 있어 MTD 파티션에 남아 있는 공간을 할당해 확장됨
오직 하나의 볼륨만이 이 플래그를 사용할 수 있음
이 정보를 갖고 ubinize라는 PEB 크기를 -p, 페이지 크기를 -m, 하위 페이지 크기를 -s로 해서 -o 의 매개 변수인 이름을 가진 이미지 파일을 생성

```
$ ubinize -o ~/ubi.img -p 128KiB -m 2048 -s 512 ubinize.cfg
```

해당 이미지를 타깃에 설치하기 위해 타깃에서 다음 명령어를 입력

```
# ubiformat /dev/mtd6 -s 2048
# nandwrite /dev/mtd6 /ubi.img
# ubiattach -p /dev/mtd6 -O 2048
```

만약 UBIFS 루트 파일시스템으로 부팅하려면, 다음 커널 명령줄 매개변수를 제공해야 함

```
ubi.mtd=6 root=ubi0:vol_1 rootfstype=ubifs
```

컨트롤러가 내장된 플래시를 위한 파일시스템

< 플래시 벤치 >

플래시 메모리를 최적으로 사용하려면 삭제 블록 크기와 페이지 크기를 알아야 함
플래시 벤치: 플래시 메모리 장치의 성능을 평가하고 측정하기 위한 도구 또는 벤치마크

```
$ sudo ./flashbench -a /dev/mmcblk0 --blocksize=1024
align 536870912 pre 4.38ms on 4.48ms post 3.92ms diff 332µs
align 268435456 pre 4.86ms on 4.9ms post 4.48ms diff 227µs
align 134217728 pre 4.57ms on 5.99ms post 5.12ms diff 1.15ms
align 67108864 pre 4.95ms on 5.03ms post 4.54ms diff 292µs
align 33554432 pre 5.46ms on 5.48ms post 4.58ms diff 462µs
align 16777216 pre 3.16ms on 3.28ms post 2.52ms diff 446µs
align 8388608 pre 3.89ms on 4.1ms post 3.07ms diff 622µs
align 4194304 pre 4.01ms on 4.89ms post 3.9ms diff 940µs
align 2097152 pre 3.55ms on 4.42ms post 3.46ms diff 917µs
align 1048576 pre 4.19ms on 5.02ms post 4.09ms diff 876µs
```

이 경우 플래시벤치는 여러 2의 거듭제곱 경계 직전과 직후 1024바이트 블록을 읽음

< discard 와 TRIM >

파일을 삭제하면 수정되는 디렉터리 노드만 저장소에 기록되고 파일의 내용을 담고 있는 섹터들은 변경되지 않고 남아있음
플래시 변환 레이어가 디스크 컨트롤러 안에 있으면, 이 디스크 섹터 그룹은 더 이상 유용한 데이터가 포함돼 있지 않다는 것을 알지 못하므로 쓸모없는 데이터를 복사하게 됨
SCSI와 SATA스펙은 TRIM 명령어가 있고, MMC는 비슷한 명령어인 ERASE를 갖고 있음
리눅스에서 해당 기능 discard이며, discard를 사용하려면 지원하는 저장 장치가 필요하며 리눅스 장치 드라이버도 필요

블록 시스템 큐 파라미터 (/sys/block/<block device>/queue/)

discard_granularity	장치의 내부 할당 유닛의 크기
discard_max_bytes	한꺼번에 discard 할 수 있는 최대 바이트 수
discard_zeroes_data	만약 1이면, discard되는 데이터는 0으로 설정

만약 장치나 장치 드라이버가 discard를 지원하지 않는다면, 이 값은 모두 0으로 설정됨

파일 시스템이 마운트될 때 -o discard 옵션을 mount 명령어에 추가함으로써 discard를 활성화할 수 있음
또한 어떻게 파티션을 마운트 했는지와는 별개로 util-linux 패키지의 일부분인 fstrim 명령어를 사용해 명령줄에 강제로 discard 할 수 있음

fstrim 은 마운트된 파일시스템에서 동작하므로 루트 파일시스템, /를 트림하기 위해 다음과 같이 입력

```
# fstrim -v /
/: 2061080704 bytes were trimmed
```

Verbose 옵션인 -v를 사용해 잠재적으로 사용 가능한 바이트 수를 출력

<ext4>

- **목적**: 리눅스 데스크톱을 위한 주요 파일 시스템
- **특징**:
  - 저널링 기능을 통해 갑작스러운 종료로부터 빠르고 쉽게 복구 가능
  - 컨트롤러가 내장된 플래시 장치에 적합하며, eMMC 저장소를 가진 안드로이드 장치에서 선호됨
- **옵션**: 장치가 discard를 지원하는 경우 -o discard 옵션으로 마운트 가능

런타임에 ext4 파일시스템을 포맷하고 만들려면 다음과 같이 입력

```
# mkfs.ext4 /dev/mmcblk0p2
# mount -t ext4 -o discard /dev/mmcblk0p1 /mnt
```

-B 로 블록 크기를, -b로 이미지의 블록 수를 설정

```
$ genext2fs -B 1024 -b 10000 -d rootfs rootfs.ext4
```

Genext2fs는 -D [file table]로 장치 테이블을 사용해 파일 권한과 소유권을 설정

이름에서 알 수 있듯이 실제로 Ext2 포맥으로 이미지 생성

다음과 같이 tune2fs를 이용해 Ext4로 업그레이드할 수 있음

```
$ tune2fs -j -J size=1 -O filetype,extents,uninit_bg,dir_index
\
rootfs.ext4
$ e2fsck -pDf rootfs.ext4
```

Yocto 프로젝트와 Buildroot 둘다 Ext4 포맷의 이미지를 만들 때 위와 동일한 단계들을 사용

<F2FS>

F2FS 는 컨트롤러가 내장된 플래시 장치를 위해 설계된 로그 구조 파일시스템
F2FS는 페이지와 삭제 블록 크기를 고려한 다음 경계에 데이터를 정렬하려고 함
로그 포맷은 전원 종료 시 복원력을 제공하며 일부 테스트에서는 ex4보다 두배 향상된 쓰기 성능을 보여줌

Mfs2.f2fs도구는 -1 레이블인 빈 F2FS 파일 시스템을 생성

```
# mkfs.f2fs -l rootfs /dev/mmcblock0p1
# mount -t f2fs /dev/mmcblock0p1 /mnt
```

<FAT16/32>

- **포맷:**
  SD 카드와 USB 플래시 메모리는 대부분 FAT32로 포맷되어 있으며, 일부 마이크로컨트롤러는 FAT32 접근 패턴에 최적화
- **부트로더:**
  T1 OMAP 기반 칩들은 2단계 부트로더를 위한 FAT 파티션이 필요
- **단점:**
  FAT 포맷은 손상되기 쉽고 저장 공간을 효율적으로 활용하지 못하여 중요한 파일 저장에는 적합하지 않음

- 리눅스는 msdos 파일 시스템을 통해 FAT16을 지원하고, vfat 파일 시스템을 통해 FAT32와 FAT16을 모두 지원

두번째 MMC 하드웨어 어댑터에 SD 카드를 마운트하려면 다음과 같이 입력

```
# mount -t vfat /dev/mmcblock1p1 /mnt
```

읽기 전용 압축 파일시스템

데이터를 압축하는 것은 모든 것을 담을 수 있을만큼 충분한 공간을 갖고 있지 않은 경우에 유용
JFFS2와 UBIFS 는 기본적으로 즉시 데이터 압축을 함
리눅스는 romfs, cramfs, squashfs 등 여러 방법 지원

<SquashFS>

squashFS 파일시스템은 cramfs를 대체할 목적으로 만들어짐

Mksquashfs를 사용해 파일시스템 이미지를 만들고 그것을 플래시 메모리에 설치

```
$ mksquashfs rootfs rootfs.squashfs
```

만든 파일시스템은 읽기 전용이므로 런타임에 파일을 수정할 수 있는 매커니즘이 없음

- 업데이트는 전체 파티션을 삭제하고 새로운 이미지로 프로그래밍하는 방식
- SquashFS는 배드 블록을 인식할 수 없으며, 신뢰할 수 있는 MTD 볼륨을 만들기 위해 UBI를 사용하여 NAND 플래시를 사용할 수 있음
- **커널 구성**:
  CONFIG_MTD_UBI_BLOCK을 설정하여 각 UBI 볼륨용 읽기 전용 MTD 블록 장치를 생성

(두 개의 MTD 파티션과 그에 따른 mtdblock 장치를 보여주며, 두 번째 파티션은 읽기 전용 파일 시스템으로 사용되는 UBI 볼륨을 만들기 위해 활용)

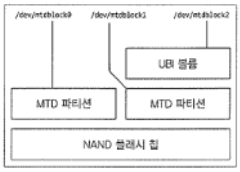


그림 9.5 UBI 볼륨

임시 파일시스템

수명이 짧거나 재부팅 후 중요하지 않은 파일들은 대부분 /tmp 안에 저장하고 영국적인 저장소와 격리시키는 것이 적절한 방법
tmpfs를 이용해 마운트 함으로써 임시 램 기반의 파일시스템을 만들 수 있음

```
# mount -t tmpfs tmp_files /tmp
```

Procfs와 sysfs처럼 tmpfs와 연결된 장치 노드가 없으므로 이전에서 자리 표시자 문자열로 tmp_files를 파라미터로 제공해야 함
사용된 메모리의 양은 파일이 생성디고 삭제됨에 따라 증가하거나 줄어들 것
기본 최대 사이즈는 물리적 램의 절반
대부분의 경우 tmpfs가 그렇게 커지면 문제가 될 수 있으므로 -o size 파라미터로 제한을 걸어두는 것이 좋음

```
# mount -t tmpfs -o size=1m tmp_files /tmp
```

/tmp뿐 아니라 /var의 서브디렉터리는 휘발성의 데이터를 포함하며 각각의 서브디렉터리에 별도로 파일시스템을 만들거나 좀 더 경제적으로 심볼릭 링크를 사용하는 것이 tmpfs를 사용하는 좋은 방법

```
/var/cache -> /tmp
/var/lock -> /tmp
/var/log -> /tmp
/var/run -> /tmp
/var/spool -> /tmp
/var/tmp -> /tmp
```

Yocto 프로젝트에서 /run과 /var/volatile 은 다음과 같이 심볼릭 링크가 가리키는 tmpfs 마운트

```
/tmp -> /var/tmp
/var/lock -> /run/lock
/var/log -> /var/volatile/log
/var/run -> /run
/var/tmp -> /var/volatile/tmp
```

런타임에 콘텐츠의 손상이 발생할 수 있음

읽기 전용 루트 파일시스템 만들기

읽기전용으로 루트 파일시스템을 만드는 것은 우연히 발생될 수 있는 덮어 쓰기를 막기 때문에 중요

쓰기 가능한 몇가지 파일과 디렉터리

- **/etc/resolv.cof:**
  이 파일은 DNS 네임서버들의 주소를 기록하기 위한 네트워크 구성 스크립트에 의해 쓰여짐
  정보가 휘발성이므로 간단히 임시 디렉터리에 연결된 심볼릭 링크로 만들어야 함

- **/etc/passwd:**
  이 파일은 /etc/group, /etc/shadow, /etc/gshadow와 함께 사용자, 그룹 이름, 암호를 저장함
  영구 저장소 영역에 심볼릭 링크를 만들어야 함

- **/var/lib:**
  많은 애플리케이션이 이 디렉터리에 쓸 수 있고 또한 여기서 영구적인 데이터를 보관할 수 있도록 기대
  한가지 해결책은 부팅 시에 파일의 기본 세트를 tmpfs 파일시스템으로 복사한 다음 /var/lib 을 새로운 디렉터리로 마운트하는 것

```
$ mkdir -p /var/volatile/lib
$ cp -a /var/lib/* /var/volatile/lib
$ mount --bind /var/volatile/lib /var/lib
```

부트 스크립트들 중 하나에 다음과 같은 명령어를 입력해 작업 수행 가능

- **/var/log:**
  syslog와 다른 데몬들이 로그를 저장하는 장소
  간단한 해결 방법은 tmpfs를 사용해 /var/log를 마운트함으로써 모든 로그 메시지를 휘발성화하는 것

파일시스템 선택

저장소 요구 사항들을 다음 세가지 범주로 나눌 수 있음

영구적인 읽기-쓰기 데이터	런타임 구성, 네트워크 파라미터, 암호, 데이터 로그, 사용자 데이터
영구적인 읽기 전용 데이터	루트 파일시스템과 같은 내용이 변하지 않는 프로그램들, 라이브러리, 구성 파일들
휘발성 데이터	/tmp 와 같은 임시 저장소

읽기-쓰기 저장소를 위한 선택 사항

NOR	UBIFS 또는 JFFS2
NAND	UBIFS, JFFS2 또는 YAFFS2
eMMC	ext4또는 F2FS

읽기 전용 저장소의 경우, ro 속성으로 마운트된 이것들 중 하나를 이용할 수 있음