FAT12 BOOT SECTOR (OFFSET)

```
00  EB 3C 90          JMP BOOT
03  4D 53 34 4F 53    OEM NAME
0B  0201              BYTES PER SECTOR
0D  01                SECTORS PER CLUSTER
0E  02                RESERVED SECTORS
10  02                NUMBER OF FATs
11  00E0              ROOT ENTRIES
13  2880              TOTAL SECTORS (16)
15  F0                MEDIA DESCRIPTOR
16  009               SECTORS PER FAT
18  12
1A  02
1C  0000
20  0000000
24  80
25  00
26  29
27  12345678
2B  4E414D4520
36  4641543132
3E  55 AA
```

SECTOR 0 (HEX)

```
     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
000  EB 3C 90 4D 53 44 4F 53 20 20 20 02 01 01 02 02
010  00 E0 28 00 00 F2 00 00 00 00 00 00 00 00 00 00
030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
050  00 00 00 00 00 03 00 00 00 00 00 00 00 00 00 00
060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
070  00 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00
080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA
```

FAT12

Understanding and Implementing the Classic File System in C_

VOLUME LAYOUT

CLUSTER CHAIN (EXAMPLE)

FAT TABLE (12-BIT ENTRIES)

CLUSTER:	0	1	2	3	4	5	6	7	8	9	A	BC	D	E	F
VALUE:	FFF	FFF	003	004	007	FFF	FFF	FFF	000	000	000	000	000	000	000

HANDS-ON. LOW-LEVEL. REAL-WORLD.
BUILD A FAT12 DRIVER FROM THE GROUND UP.

Björn Götz

FAT12

Understanding and Implementing the Classic File System in C

Björn Götz

This book is available at https://leanpub.com/fat12

This version was published on 2026-08-16

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

To my beautiful wife, for brightening my life with her endless love; to my mother, for always believing in me; and to my father, for always being there whenever I needed him.

Contents

Preface

You want to understand how filesystems really work — from raw bytes on disk to reading and writing files. FAT12 is where that journey begins: simple enough to fully understand, real enough to run on bare metal.

Why FAT12?

FAT is everywhere. USB drives, SD cards, EFI System Partitions — every major operating system reads and writes it out of the box. It has survived for four decades because it is simple, predictable, and good enough for a huge range of devices.

We start with FAT12, the original and simplest variant. Once you understand FAT12, you already understand most of FAT16 and FAT32 — the core concepts transfer directly. And FAT12 itself strips everything to the essentials:

- **Small volumes** — 1–4 MiB disk images. Every structure fits in a handful of sectors we can hexdump end to end.
- **Simple allocation** — a single table links it all together. No extents, no block groups, no B-trees. Just a linked list.
- **Fixed root directory** — fixed position, fixed size, no tree walking to find `/`.

Who This Book Is For

You are building a hobby OS, or just tinkering because low-level systems are fun. You know C — pointers, structs, bitwise operations. You want to go all the way — from raw bytes on a disk to a working filesystem running inside your own OS. If that sounds like you, keep reading.

What We Will Build

By the end of this book we will have a complete FAT12 library implementing every major filesystem operation. We will build a CLI tool to test every operation as we go, and a kernel – proving the same library runs on bare metal. The library has no external dependencies – the same pure C code compiles for both the CLI tool and the kernel. Platform frontends connect through the File System API above; storage backends through the Block Device API below.

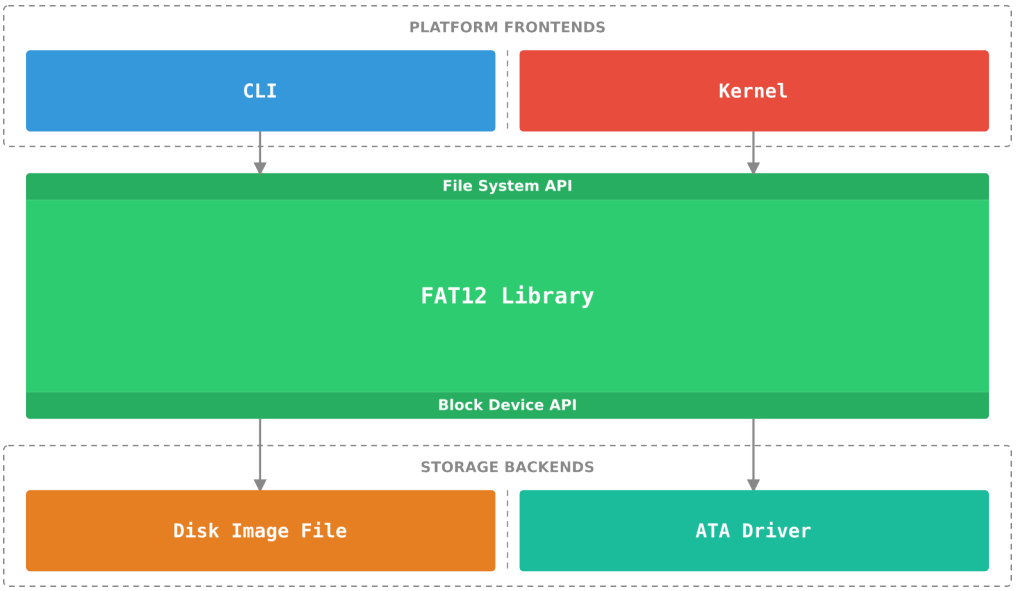

Figure 1. Architecture overview: CLI and Kernel share the same FAT12 library; platform frontends connect through the File System API, storage backends through the Block Device API

How This Book Is Structured

The book is split into three parts, plus a set of reference appendices:

Part I: Reading – From raw bytes to file contents. We set up the toolchain, build a portable `BlockDevice` abstraction, read our first sector, and decode the boot sector into C structs – revealing the geometry calculations that power the entire filesystem. Walk the root directory, navigate the File Allocation Table and its famously awkward 12-bit entries, read a file from disk to screen start to finish, and resolve full paths through nested directories.

Part II: Writing – Write and allocate files. Create directories, delete entries, rename across the tree. Every operation that reads a sector learns to write it back – and the order matters for crash safety.

Part III: Advanced – Format a disk from scratch: write a boot sector, twin FATs, and root directory by hand. Then take the same unmodified library and boot it in a 32-bit kernel over ATA PIO, printing to VGA text mode in QEMU. Platform code changes; the library does not.

Appendices – Reference material for after you finish reading: a field-by-field reference for the boot sector fields the library never touches, a FAT12/FAT16/FAT32 comparison, a cheat sheet collecting every on-disk structure and position formula, and a short introduction to how long filenames (LFN/VFAT) work – enough to extend the library yourself, though the book itself does not implement them.

A Note on Style

This is not production code – and that is by design. Long filename support, field-by-field validation, worst-case error recovery – skipped on purpose so the FAT12 logic stays front and center. In the library code, every block device I/O and memory allocation return value is unchecked; we trade those checks for readability so the filesystem logic does not drown in `if (result == -1) return -1` chains. Filesystem-level guards – path validation, attribute checks, bounds on cluster chains, cycle detection – *are* present, because those are part of the FAT12 logic itself, not noise around it. **Production Note** callouts throughout flag what a real-world driver would do differently.

Think of it as a starting point. Understand it, extend it, then harden it for your own use.

How to Read This Book

Read front to back. Every chapter builds on the last. Code appears with the explanation – type it, run it, hexdump it, break it.

About the Author

I am Björn Götz, a German software engineer who loves understanding low-level systems and explaining them to others. I wrote this book because I could not find a hands-on guide to FAT12 – just scattered forum posts and wiki pages. This is the book I wished I had when I started, and I hope it gives you the hands-on experience I was missing.

Feedback

Found an error? Have a question? Use the “Email the Author” link on the book’s Leanpub page. If you enjoyed the book, I would appreciate a review on Leanpub. My goal is to make this book both valuable and genuinely fun to read.

Prerequisites

Before we write any code, let's set up the environment. It takes a few minutes and gives us everything we need for the rest of the book.

Operating System

This book is based on and tested against x86 Ubuntu. Any other x86 Linux works too – just adapt the package manager and package names. If you prefer not to worry about that, the virtual machine and Docker options below set up an Ubuntu environment – they work on Windows, macOS, or any other Linux distro.

Virtual Machine

Download an Ubuntu Desktop image and run it in any x86 hypervisor. VirtualBox[1] is free if you need one. A full setup guide is out of scope for this book; there are plenty of tutorials online.

Docker

Install Docker by following the official guide[2] – a full installation guide is out of scope for this book. Once Docker is running, here is the command to start a container that works for this book:

[1] https://www.virtualbox.org/
[2] https://docs.docker.com/get-docker/

```
docker run \
  -it \
  --rm \
  --privileged \
  --platform linux/amd64 \
  -v /host/path/to/your/code:/workspace \
  -w /workspace \
  ubuntu \
  bash
```

Replace `/host/path/to/your/code` with the directory where you keep your project files. From the container shell, run the commands in the following sections to set up the tools one by one. The `--privileged` flag is required for loop device mounts (Chapter 3 onward), and `--platform linux/amd64` ensures x86 emulation on ARM hosts.[3]

The container shell is already root — every command shown in this book should drop the `sudo` prefix when run inside the container.

(I wrote and tested the entire book on an Apple Silicon MacBook — Docker with Rosetta 2 works without issues.)

C Compiler

This book uses GCC[4]. All code was tested against it. Another compiler might work, but I recommend sticking with GCC throughout the book.

```
sudo apt install build-essential
```

mkfs.fat

We also need `dosfstools`, which provides the `mkfs.fat` command for creating FAT disk images.

[3] On Apple Silicon, Docker Desktop handles x86 emulation transparently via Rosetta 2.
[4] https://gcc.gnu.org/

```
sudo apt install dosfstools
```

QEMU (Chapter 11 Only)

Chapter 11 boots the library inside a 32-bit i386 kernel. For that we need the x86 system emulator from QEMU. If you are using a VM or Docker container, install QEMU on the host – it needs access to KVM and hardware virtualization, which nested setups complicate.

Feel free to skip this step for now and come back to it when you reach Chapter 11.

See the official install guide for your platform: https://www.qemu.org/download/

Hex Dump Utility

We use `xxd` throughout the book to hexdump disk structures and verify what we read. Install it with:

```
sudo apt install xxd
```

Getting the Code (Optional)

You do not need to clone a repository to follow this book – every chapter lists the complete code for every file. Type it yourself and compile along.

If you want to browse the finished code or skip ahead, the repository[5] is available:

```
git clone https://github.com/SoftwareFreak1/FAT12
cd FAT12
```

Every chapter has its own directory (`chapter-01`, `chapter-02`, ...), so you can jump to any point.

[5] https://github.com/SoftwareFreak1/FAT12

Part I: Reading

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Chapter 1: Hello, Disk!

A filesystem is just bytes on a disk waiting for a decoder ring. In this chapter we lay the foundation: create a FAT12 disk image, build a portable block device abstraction that will carry us through the entire book, and read the volume label straight off the disk to prove it works.

Creating a Virtual Disk

We need a disk to experiment on, but not real hardware. A file works just as well – safer to play with, faster to create and discard, and immune to accidental wipes.

Luckily, creating one is straightforward on Linux. We first create an empty file the size of our disk (filled with zeros), then use `mkfs.fat` – the standard tool for formatting FAT volumes – to write a FAT12 filesystem with the volume label `MYDISK` onto it. (The command also sets some geometry parameters we will decode in the following chapters.) Put it all in a script called `create_disk.sh` in the project root so we can create a fresh disk image whenever we need one:

`create_disk.sh`

```
#!/bin/bash
dd if=/dev/zero of=disk.img bs=1M count=4
mkfs.fat -F 12 -s 2 -S 512 -n MYDISK disk.img
```

We first need to make the script executable, then run it:

```
chmod +x create_disk.sh
./create_disk.sh
```

We should now have `disk.img` – a 4 MiB disk image with a valid FAT12 filesystem. Running the command below gives us a peek at the first few bytes of what `mkfs.fat` actually wrote:

```
xxd -g 1 disk.img | head -6
```

```
00000000: eb 3c 90 6d 6b 66 73 2e 66 61 74 00 02 02 01 00   .<.mkfs.fat.....
00000010: 02 00 02 00 20 f8 0c 00 20 00 02 00 00 00 00 00   .... ... .......
00000020: 00 00 00 00 80 00 29 ca ff 9e 9c 4d 59 44 49 53   ......)....MYDIS
00000030: 4b 20 20 20 20 20 46 41 54 31 32 20 20 20 0e 1f   K     FAT12   ..
00000040: be 5b 7c ac 22 c0 74 0b 56 b4 0e bb 07 00 cd 10   .[|."."t.V.......
00000050: 5e eb f0 32 e4 cd 16 cd 19 eb fe 54 68 69 73 20   ^..2.......This
```

We can already see the volume label `MYDISK` inside it, clearly visible in the ASCII column at offset `0x002b`.

Sectors and LBA

A disk does not transfer data one byte at a time — it reads and writes in fixed-size blocks called **sectors**. The number of bytes in each sector depends on the hardware: floppy disks use 512, CD-ROMs use 2048, and modern NVMe drives use 4096. We call this the **sector size**. Whatever the hardware, it is always a power of 2.

We passed `-S 512` to `mkfs.fat` when initializing the FAT12 filesystem — this set the sector size to 512 bytes for our disk. The value is recorded on the volume and must match what the block device (more on this shortly) reports for the math to work. We will see in Chapter 2 exactly how the sector size is stored on the volume.

Every sector also has an address. We number them sequentially, starting from 0 — a scheme called Logical Block Addressing (LBA). Sector 0 is the very first block on the disk, sector 1 is the next, all the way to the end. A 4 MiB disk with 512-byte sectors has 8192 of them (0 through 8191), as you can see in the image below.

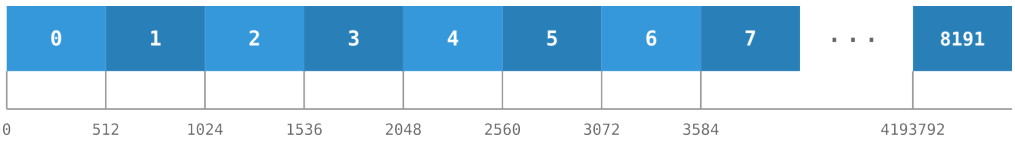

Figure 2. LBA addressing — sectors as numbered blocks in a linear array

The Block Device

Operating systems should never talk to disk hardware directly from filesystem code. Instead they define an abstract interface – a Block Device API – and each storage driver implements it. The filesystem calls `read` and `write` through that interface, never knowing or caring whether the underlying device is an ATA hard disk, an NVMe SSD, or a file on another filesystem. This is exactly what lets us swap the block device implementation without touching a single line of filesystem code – we will need this in Chapter 11 when we move the FAT12 library from a CLI tool into a kernel and replace file-backed I/O with real ATA disk access.

It is called a *block* device because the disk deals in fixed-size blocks; the naming is a bit confusing since on FAT systems we call them *sectors* rather than blocks, but "block device" is the established term across operating systems.

include/block_device.h

```c
#ifndef BLOCK_DEVICE_H
#define BLOCK_DEVICE_H

#include <stdint.h>

typedef struct BlockDevice BlockDevice;

int block_device_read(
    BlockDevice* device,
    uint64_t lba,
    uint32_t sector_count,
    void* buffer
);

int block_device_write(
    BlockDevice* device,
    uint64_t lba,
    uint32_t sector_count,
    const void* buffer
);

uint32_t block_device_sector_size(BlockDevice* device);
uint64_t block_device_sector_count(BlockDevice* device);
```

```
24 void block_device_close(BlockDevice* device);
25
26 #endif
```

Every read and write call takes a buffer pointer. Make sure it is at least `sector_count * sector_size` bytes, or the I/O will overflow the buffer – corrupting whatever memory follows. (A production driver validates buffer sizes; we skip that check to focus on FAT12.)

LBA uses `uint64_t`, not `uint32_t`. FAT12 volumes max out at roughly 64K sectors – well within 32 bits. The wider type is intentional: the same `BlockDevice` interface can back larger filesystems (FAT16, FAT32, or anything else) without an API break. A little foresight saves a lot of refactoring.

These functions are plain global symbols, not function pointers in a vtable. An OS juggling multiple devices (a SATA disk and a USB stick at the same time) would give each `BlockDevice` instance its own function table. This book keeps it simple: one set of definitions per platform, and the linker picks whichever `.c` file we compile. The CLI tool reads `disk.img`, the kernel reads a real disk – and the library in the middle never changes.

Every operation returns `int` (0 on success, -1 on error), enabling error handling in production. Our implementations always return 0 – we skip the checks to keep the FAT12 logic front and center.

Now for the concrete implementation on our development machine. Opening a block device is platform-specific – different backends need different arguments, like a file path here – so each block device provides its own open function. The header above only defines the common operations.

`platform/cli/file_block_device.h`

```c
#ifndef FILE_BLOCK_DEVICE_H
#define FILE_BLOCK_DEVICE_H

#include "block_device.h"

BlockDevice* file_block_device_open(const char* path);

#endif
```

platform/cli/file_block_device.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <inttypes.h>
#include "debug.h"
#include "block_device.h"

#define SECTOR_SIZE 512

struct BlockDevice {
    FILE* file;
};

BlockDevice* file_block_device_open(const char* path) {
    BlockDevice* device = (BlockDevice*)malloc(sizeof(BlockDevice));
    device->file = fopen(path, "r+");
    if (device->file == NULL) {
        free(device);
        return NULL;
    }
    return device;
}

int block_device_read(
    BlockDevice* device,
    uint64_t lba,
    uint32_t sector_count,
    void* buffer
) {
    DBG_PRINT("[ block_device ] read %" PRIu32 " sector(s) starting at LBA %"
    ↪  PRIu64 "\n", sector_count, lba);
    size_t total_bytes = sector_count * SECTOR_SIZE;
    off_t offset = lba * SECTOR_SIZE;
    DBG_PRINT("[ file         ] read %zu bytes at 0x%" PRIx64 "\n", total_bytes,
    ↪  (uint64_t)offset);
    fseeko(device->file, offset, SEEK_SET);
    fread(buffer, 1, total_bytes, device->file);
    return 0;
```

```
}

int block_device_write(
    BlockDevice* device,
    uint64_t lba,
    uint32_t sector_count,
    const void* buffer
) {
    DBG_PRINT("[ block_device ] write %" PRIu32 " sector(s) starting at LBA %"
    ↪  PRIu64 "\n", sector_count, lba);
    size_t total_bytes = sector_count * SECTOR_SIZE;
    off_t offset = lba * SECTOR_SIZE;
    DBG_PRINT("[ file         ] write %zu bytes at 0x%" PRIx64 "\n",
    ↪  total_bytes, (uint64_t)offset);
    fseeko(device->file, offset, SEEK_SET);
    fwrite(buffer, 1, total_bytes, device->file);
    return 0;
}

void block_device_close(BlockDevice* device) {
    fclose(device->file);
    free(device);
}

uint32_t block_device_sector_size(BlockDevice* device) {
    (void)device;
    return SECTOR_SIZE;
}

uint64_t block_device_sector_count(BlockDevice* device) {
    off_t saved = ftello(device->file);
    fseeko(device->file, 0, SEEK_END);
    off_t size = ftello(device->file);
    fseeko(device->file, saved, SEEK_SET);
    return (uint64_t)(size / SECTOR_SIZE);
}
```

`file_block_device_open` checks whether `fopen` actually succeeded before handing the device back. If `disk.img` does not exist (or we lack permission to open it), `fopen` returns `NULL` — we free the struct we just allocated and return `NULL` ourselves, so the caller's `if (device == NULL)` check further down catches the failure instead of dereferencing a device with a broken file handle.

Each function translates a sector-level request into a normal file operation. LBA numbers map directly to byte offsets — sector 0 starts at byte 0, sector 1 at byte 512, sector 2 at byte 1024, and so on. The formula is always `Byte Offset = LBA x Sector Size`. `fseeko` positions the file cursor at that byte offset,

then reads or writes `sector_count * SECTOR_SIZE` bytes from there. The disk image file is just a linear array of bytes from the file API's perspective; the LBA is just an index into that array scaled by the sector size.

You may have noticed the `DBG_PRINT` calls in the code – a printf wrapper we can flip on and off for debugging:

`include/debug.h`

```
#ifndef DEBUG_H
#define DEBUG_H

#if DEBUG
#include <stdio.h>
#include <stdarg.h>
#define DBG_PRINT(...) \
    do { \
        printf("\033[36m"); \
        printf(__VA_ARGS__); \
        printf("\033[0m"); \
    } while (0)
#else
#define DBG_PRINT(...)
#endif

#endif
```

When `DEBUG` is `1`, `DBG_PRINT` routes its arguments directly to `printf` with ANSI cyan escape codes for terminal visibility. When `DEBUG` is `0`, the macro expands to nothing and the log calls vanish from the binary. We will sprinkle `DBG_PRINT` calls throughout the library to trace what is happening inside – which sectors get read, which clusters get walked, which directory entries get matched – without cluttering the normal output. Notice the header never defines `DEBUG` itself – `#if DEBUG` on an undefined macro evaluates to `0` in C, no error, no warning.

Reading the Volume Label

Let's pull the volume label off the disk – just raw bytes into a buffer and a format string. The sector buffer is hardcoded to 512 bytes; this is demo code that chapter 2 will replace entirely. Chapter 2 will also explain why the label lives exactly where it is.

platform/cli/main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include "block_device.h"
#include "file_block_device.h"

int main(void) {
    BlockDevice* device = file_block_device_open("disk.img");
    if (device == NULL) {
        fprintf(stderr, "error: could not open disk.img\n");
        return 1;
    }

    uint8_t sector[512];
    block_device_read(device, 0, 1, sector);
    printf("Volume label: %.11s\n", (const char*)&sector[43]);
    block_device_close(device);

    return 0;
}
```

The last thing we need to do is compile the binary. Here is the script — it discovers every `.c` file automatically, so it never needs to change as we add files. It also reads a `DEBUG` environment variable and passes it straight to the compiler as a macro, so we can flip debug tracing on for a single build without editing `debug.h`:

build.sh

```sh
#!/bin/bash
SRC=""
for d in src platform/cli; do
    for f in "$d"/*.c; do
        [ -f "$f" ] && SRC="$SRC $f"
    done
done
gcc -Wall -Wextra -DDEBUG=${DEBUG:-0} -iquote include -iquote src -iquote
↪  platform/cli -o fat12-cli $SRC
```

`${DEBUG:-0}` is bash's default-value syntax: if the `DEBUG` environment variable is set when we run the script, its value is passed straight through as

`-DDEBUG=`; if it is unset, `0` is used instead. Either way, `build.sh` always defines `DEBUG` for us.

We first need to make the script executable, then run it:

```
chmod +x build.sh
./build.sh
```

We should now have an executable named `fat12-cli`. When we run it with:

```
./fat12-cli
```

we should see:

```
Volume label: MYDISK
```

There it is. The label field is exactly 11 bytes, not null-terminated — short labels are padded with spaces. `%.11s` prints exactly 11 characters without needing a terminator.

Tracing the Translation

Now we turn on debug mode and run it again — no file to edit, just an environment variable in front of the build:

```
DEBUG=1 ./build.sh
./fat12-cli
```

```
[ block_device ] read 1 sector(s) starting at LBA 0
[ file         ] read 512 bytes at 0x0
Volume label: MYDISK
```

The two debug lines trace the call: the FAT12 library requests 1 sector at LBA 0, and the file driver translates that to 512 bytes starting at offset 0 in the disk image.

Run `./build.sh` again without `DEBUG=1` to go back to a quiet build — but keep the trick in mind. Any time something looks off — an address translation error, a read that returns garbage — rebuild with `DEBUG=1 ./build.sh` and the full read/write path prints to the terminal. As we add more functionality in future chapters, we will add more debug calls so the trace builds up.

What's Next

We read one field off a raw disk. In Chapter 2 we decode that first sector in full, which packs the entire volume's geometry into 512 bytes — every byte mapped into real C structs, with debug output that prints the volume's full identity. That geometry is what drives every calculation in the rest of the book.

Chapter 2: The Boot Sector

Chapter 1 was the proof of concept: one field, pulled off the disk with nothing but a pointer and a format string. Now we build the real thing. The boot sector is the blueprint that defines every FAT12 volume — we will decode it with packed structs that mirror it byte-for-byte, read it all off the disk, and verify every field through debug output.

What Is the Boot Sector?

The boot sector is the first sector of every FAT volume (LBA 0) and always one sector wide. It is the disk's self-description — the first thing an OS reads to learn how the rest of the volume is laid out. Without it, the disk is just an undifferentiated blob of bytes.

Here is what it looks like:

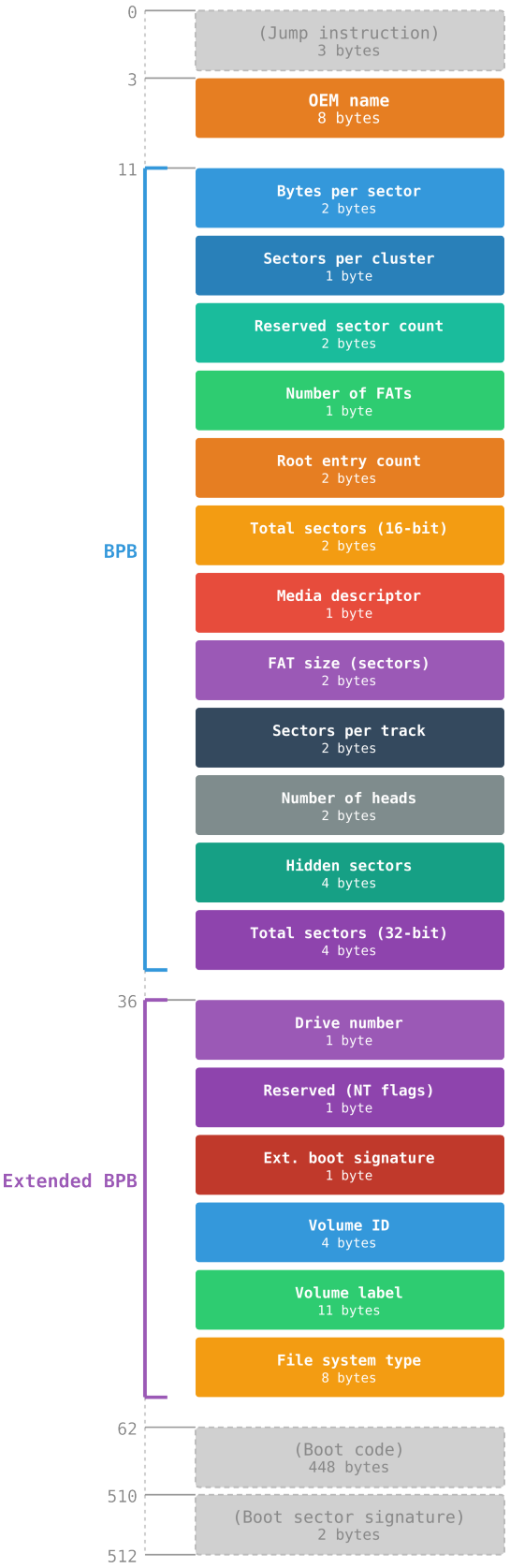

Figure 3. Boot sector layout

We will explain the fields the library actually reads and uses later in this chapter; the rest are covered in the appendix. For now it is enough to understand the boot sector's structure – what sits where and how big each piece is – so we can define structs that mirror it.

The centerpiece of the boot sector is the **BIOS Parameter Block (BPB)** at bytes 11–35 – a fixed-layout table that describes the disk geometry and filesystem structure.

It is called the BIOS Parameter Block because the original IBM PC BIOS read this data directly to configure the disk controller; before OS-level drivers existed, the BPB was how the BIOS learned the disk layout.

At bytes 36–61 comes the **Extended BPB**, added in later FAT revisions. It carries additional identification – volume tracking, human-readable labels, and filesystem type strings.

Mirroring the Layout in C

Our code needs a structured way to access every field. We could read raw bytes and access them by offset numbers, but the easiest approach is a packed C struct that mirrors the boot sector's exact binary layout – we can read it straight from the disk into the struct and access every field by name.

`src/layout.h`

```
1  #ifndef LAYOUT_H
2  #define LAYOUT_H
3
4  #include <stdint.h>
5
6  #pragma pack(push, 1)
7
8  typedef struct {
9      uint16_t bytes_per_sector;
10     uint8_t sectors_per_cluster;
11     uint16_t reserved_sector_count;
12     uint8_t  num_fats;
13     uint16_t root_entry_count;
```

```
    uint16_t total_sectors_16;
    uint8_t  media;
    uint16_t fat_size_16;
    uint16_t sectors_per_track;
    uint16_t number_of_heads;
    uint32_t hidden_sectors;
    uint32_t total_sectors_32;
} BPB;

typedef struct {
    uint8_t  drive_number;
    uint8_t  reserved_nt;
    uint8_t  boot_signature;
    uint32_t volume_id;
    char volume_label[11];
    char file_system_type[8];
} ExtendedBPB;

typedef struct {
    uint8_t jump[3];
    char oem_name[8];
    BPB bpb;
    ExtendedBPB extended_bpb;
    uint8_t  boot_code[448];
    uint16_t signature;
} BootSector;

#pragma pack(pop)

#endif
```

> **Memory Alignment:** When the compiler lays out a struct in memory it normally adds padding between members to satisfy alignment rules – a `uint16_t` after a `uint8_t` might get shifted to an even address. That would break our mapping, because the on-disk layout has no padding. `#pragma pack(push, 1)` tells the compiler to pack every field at the next available byte with no gaps. Notice we push before the first struct and pop after the last – the push/pop pair ensures the packing only applies to our on-disk structs and does not affect any other structs after the pop.

Endianness gotcha: FAT12 stores every multi-byte value in **little-endian** (least significant byte first). Since x86 is also little-endian, copying raw disk bytes straight into `bytes_per_sector`, `total_sectors_32`, and every other multi-byte field of our struct gives the correct value with no extra work. On a big-endian system (some embedded PowerPC or big-endian MIPS), that same raw copy would leave every multi-byte field byte-swapped. Throughout this book, whenever we read a value wider than one byte, we are relying on x86's endianness to match the disk's. The pattern for a big-endian CPU is always the same: grab the bytes in the little-endian order they appear on disk and convert to the CPU's native byte order before use, and reverse them again before writing back.

Mounting and Unmounting

Before we can read or write anything, we must mount the filesystem. Mounting is the process of opening a volume – reading its metadata, setting up internal state, and returning a handle that all subsequent operations use. We want to read the filesystem's metadata just once, when it is mounted, rather than hitting the disk again every time we need it. `fat12_mount` takes a block device, reads the boot sector into memory, and returns a `FAT12FS*` handle – the same pattern real OS kernels use. `fat12_umount` frees the handle when we are done.

`include/fat12.h`

```
#ifndef FAT12_H
#define FAT12_H
#include <stdint.h>
#include "block_device.h"

typedef struct FAT12FS FAT12FS;

FAT12FS* fat12_mount(BlockDevice* device);
void fat12_umount(FAT12FS* fs);

#endif
```

`src/fat12.c`

```c
#include <stdlib.h>
#include <string.h>
#include "block_device.h"
#include "debug.h"
#include "layout.h"
#include "fat12.h"

struct FAT12FS {
    BlockDevice* device;
    BootSector bs;
};

static BootSector read_boot_sector(BlockDevice* device)
{
    uint32_t sector_size = block_device_sector_size(device);
    void* buffer = malloc(sector_size);
    block_device_read(device, 0, 1, buffer);
    BootSector result;
    memcpy(&result, buffer, sizeof(BootSector));
    free(buffer);
    return result;
}

FAT12FS* fat12_mount(BlockDevice* device)
{
    FAT12FS* fs = malloc(sizeof(FAT12FS));
    fs->device = device;
    fs->bs = read_boot_sector(device);

    DBG_PRINT("[ fat12 ] OEM Name: %.8s\n", fs->bs.oem_name);
    DBG_PRINT("[ fat12 ] Bytes Per Sector: %u\n", fs->bs.bpb.bytes_per_sector);
    DBG_PRINT("[ fat12 ] Sectors Per Cluster: %u\n",
    ↪  fs->bs.bpb.sectors_per_cluster);
    DBG_PRINT("[ fat12 ] Reserved Sector Count: %u\n",
    ↪  fs->bs.bpb.reserved_sector_count);
    DBG_PRINT("[ fat12 ] Number of FATs: %u\n", fs->bs.bpb.num_fats);
    DBG_PRINT("[ fat12 ] Root Entry Count: %u\n", fs->bs.bpb.root_entry_count);
    DBG_PRINT("[ fat12 ] Total Sectors (16): %u\n",
    ↪  fs->bs.bpb.total_sectors_16);
    DBG_PRINT("[ fat12 ] Media Descriptor: 0x%02x\n", fs->bs.bpb.media);
    DBG_PRINT("[ fat12 ] FAT Size (sectors): %u\n", fs->bs.bpb.fat_size_16);
    DBG_PRINT("[ fat12 ] Sectors Per Track: %u\n",
    ↪  fs->bs.bpb.sectors_per_track);
    DBG_PRINT("[ fat12 ] Number of Heads: %u\n", fs->bs.bpb.number_of_heads);
    DBG_PRINT("[ fat12 ] Hidden Sectors: %u\n", fs->bs.bpb.hidden_sectors);
    DBG_PRINT("[ fat12 ] Total Sectors (32): %u\n",
    ↪  fs->bs.bpb.total_sectors_32);
    DBG_PRINT("[ fat12 ] Drive Number: 0x%02x\n",
    ↪  fs->bs.extended_bpb.drive_number);
```

```
44      DBG_PRINT("[ fat12 ] Boot Signature: 0x%02x\n",
        ↪  fs->bs.extended_bpb.boot_signature);
45      DBG_PRINT("[ fat12 ] Volume ID: 0x%08x\n", fs->bs.extended_bpb.volume_id);
46      DBG_PRINT("[ fat12 ] Volume Label: %.11s\n",
        ↪  fs->bs.extended_bpb.volume_label);
47      DBG_PRINT("[ fat12 ] File System Type: %.8s\n",
        ↪  fs->bs.extended_bpb.file_system_type);
48
49      return fs;
50  }
51
52  void fat12_umount(FAT12FS* fs)
53  {
54      free(fs);
55  }
```

On mount, we allocate a `FAT12FS` struct and initialize it with the block device — we will need it for every later read and write — and the boot sector, which `read_boot_sector` reads directly from sector 0 of the block device and copies into the packed struct. Before returning, we debug-print every field of the boot sector so we can verify it.

Now we rewrite the CLI `main` from Chapter 1 to mount and unmount the filesystem:

platform/cli/main.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include "block_device.h"
4   #include "fat12.h"
5   #include "file_block_device.h"
6
7   int main(void)
8   {
9       BlockDevice *device = file_block_device_open("disk.img");
10      if (device == NULL)
11      {
12          fprintf(stderr, "error: could not open disk.img\n");
13          return 1;
14      }
15
16      FAT12FS *fs = fat12_mount(device);
17      fat12_umount(fs);
18      block_device_close(device);
19
```

```
    return 0;
}
```

Build and run with debug output enabled – `DEBUG=1` in front of `./build.sh`:

```
DEBUG=1 ./build.sh
./fat12-cli
```

We should see:

```
[ block_device ] read 1 sector(s) starting at LBA 0
[ file         ] read 512 bytes at 0x0
[ fat12 ] OEM Name: mkfs.fat
[ fat12 ] Bytes Per Sector: 512
[ fat12 ] Sectors Per Cluster: 2
[ fat12 ] Reserved Sector Count: 1
[ fat12 ] Number of FATs: 2
[ fat12 ] Root Entry Count: 512
[ fat12 ] Total Sectors (16): 8192
[ fat12 ] Media Descriptor: 0xf8
[ fat12 ] FAT Size (sectors): 12
[ fat12 ] Sectors Per Track: 32
[ fat12 ] Number of Heads: 2
[ fat12 ] Hidden Sectors: 0
[ fat12 ] Total Sectors (32): 0
[ fat12 ] Drive Number: 0x80
[ fat12 ] Boot Signature: 0x29
[ fat12 ] Volume ID: 0x07e80101
[ fat12 ] Volume Label: MYDISK
[ fat12 ] File System Type: FAT12
```

Some fields can differ on your system – the Volume ID, for example, is a random serial `mkfs.fat` generates on every format – but most values should match exactly.

Set `DEBUG` back to `0` for now. The boot sector is now fully decoded – we will dig deeper into what each field actually means.

Four BPB fields need context from later chapters and are covered where they are used: **sectors per cluster** (Chapter 4), **number of FATs** (Chapter 3), **root entry count** (Chapter 3), and **FAT size** (Chapter 3). The fields the library actually reads for its own logic are explained below; the rest – legacy or purely informational – are covered in the appendix.

Bytes Per Sector

FAT predates the block device abstraction – real drivers configured the disk controller *from* the BPB, not the other way around. The BPB keeps the volume self-describing – without knowing the hardware. The FAT specification only allows 512, 1024, 2048, or 4096 for `bytes_per_sector`.

> **Production Note:** Verify the value is allowed and matches `block_device_sector_size()` – refuse to mount on mismatch or every operation silently corrupts data.

Total Sectors

`total_sectors_16` and `total_sectors_32` represent the total size of the volume in sectors. `total_sectors_32` was added at the end of the BPB later because 16 bits was not enough for larger disks. With 512-byte sectors, the 16-bit field caps at about 32 MB (65535 x 512), while the 32-bit field supports up to about 2 TB (4,294,967,295 x 512). The rule: if the volume fits in 16 bits, `total_sectors_16` holds the count and `total_sectors_32` is zero; otherwise `total_sectors_16` is zero and `total_sectors_32` holds the real count. This ensures backward compatibility with older FAT volumes.

Reserved Sector(s)

The reserved region starts at LBA 0. The boot sector is always its first sector, but it is not always the last – `reserved_sector_count` tells us how many sectors the reserved region spans. When it is 1, the reserved region *is* just the boot sector. When it is larger, the extra sectors typically hold a multi-sector bootloader. This count matters beyond the boot sector itself – every other structure on the volume sits after the reserved region, and we will lean on it to calculate their offsets in the chapters ahead, starting with the root directory in Chapter 3.

Figure 4. Reserved region layout – boot sector at LBA 0, optional extra sectors

Extended BPB

The Extended BPB carries fields appended after the original BPB:

- **Extended boot signature**: `0x29` signals the volume ID, label, and FS type fields are valid. Not to be confused with the `0xAA55` boot sector signature at offset 510.
- **Volume label**: A human-readable disk name assigned at format time. The OS displays it in file manager UIs (e.g., "My Disk (D:)") and some boot managers use it to locate the right partition. This is what we read in Chapter 1.

> **Production Note:** Chapter 1 read the volume label at offset 43 without checking the signature – that worked because `mkfs.fat` always writes it. A production driver checks `boot_signature == 0x29` before trusting those fields. Our code skips the check because we only ever read our own formatted disks, but production code on foreign-formatted media needs this guard.

A few more fields round out the boot sector – the media descriptor, hidden sectors (for partitioned disks), CHS geometry (`sectors_per_track`, `number_of_heads`), the rest of the Extended BPB (drive number, reserved/NT flags, volume ID, file system type), and the OEM name. The library never reads any of them for its own logic, and walking through each one here would turn this chapter into a spec listing instead of a story. If you're curious, Appendix: Boot Sector Reference Fields covers every one of them field-by-field.

Booting from FAT12

The boot sector got its name from the fact that the BIOS boots from it. When the BIOS finds the `0xAA55` signature at bytes 510–511, it loads the entire sector into memory and starts executing at byte 0. The CPU needs valid machine instructions there — but the BPB metadata occupies bytes 11 onward. So bytes 0–2 are reserved for a jump instruction that skips past the BPB and lands at offset 0x3E, where there is room for real boot code. The jump bytes, boot code area, and boot signature are opaque to FAT12 — they exist only for the BIOS boot process. The filesystem does not interpret them, validate them, or care what they contain. In our on-disk structs they exist only as offset padding — we never read or write them.

If we look at the first three bytes of our disk, we find the jump instruction `mkfs.fat` wrote:

```
xxd -g 1 disk.img | head -1
```

```
00000000: eb 3c 90 6d 6b 66 73 2e 66 61 74 00 02 02 01 00   .<.mkfs.fat.....
```

`eb 3c 90` — three bytes of x86 machine code that the CPU executes directly. The first two bytes (`eb 3c`) encode a short jump instruction in x86 assembly — the `3c` skips forward past the BPB and lands at offset 0x3E, where the boot code begins. The third byte (`90`) is a NOP that pads the slot to 3 bytes, making room for the alternative 3-byte near jump that some bootloaders use.

At offset 0x3E, `mkfs.fat` wrote a stub bootloader:

```
xxd -g 1 -s 0x3e -l 450 disk.img
```

```
0000003e: 0e 1f be 5b 7c ac 22 c0 74 0b 56 b4 0e bb 07 00   ...[|.".t.V.....
0000004e: cd 10 5e eb f0 32 e4 cd 16 cd 19 eb fe 54 68 69   ..^..2.......Thi
0000005e: 73 20 69 73 20 6e 6f 74 20 61 20 62 6f 6f 74 61   s is not a boota
0000006e: 62 6c 65 20 64 69 73 6b 2e 20 20 50 6c 65 61 73   ble disk.  Pleas
0000007e: 65 20 69 6e 73 65 72 74 20 61 20 62 6f 6f 74 61   e insert a boota
0000008e: 62 6c 65 20 66 6c 6f 70 70 79 20 61 6e 64 0d 0a   ble floppy and..
0000009e: 70 72 65 73 73 20 61 6e 79 20 6b 65 79 20 74 6f   press any key to
000000ae: 20 74 72 79 20 61 67 61 69 6e 20 2e 2e 2e 20 0d    try again ... .
000000be: 0a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000000ce: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000000de: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000000ee: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000000fe: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0000010e: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0000011e: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0000012e: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0000013e: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0000014e: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0000015e: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0000016e: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0000017e: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0000018e: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0000019e: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001ae: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001be: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001ce: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001de: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001ee: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001fe: 55 aa                                            U.
```

If booted from, this stub would print “This is not a bootable disk” and wait for a keypress before retrying. The last two bytes of the dump, `55 AA`, are the boot sector signature — the marker the BIOS checks before loading the sector at all.

What’s Next

The boot sector is fully decoded now — every field of the BPB and Extended BPB is a named struct member instead of a magic offset, and the debug output proves it by printing the volume’s complete identity straight off the disk.

But a boot sector alone doesn’t get us to a single file. In Chapter 3 we calculate where the root directory lives on disk, and read it to list every file and folder it holds.

Chapter 3: The Root Directory

Chapter 2 gave us the boot sector — the disk's blueprint, decoded into structs we can read by name. But a blueprint does not hand us a single file. Before we can read anything, we need to find it, and that is the root directory's job: a fixed table of entries at a position we can compute from the BPB fields we already know. The root directory gives us every file and folder's name, size, timestamps, and attributes — everything we need to know about it before we read a single byte of its contents.

By the end of this chapter we will have a first-class directory iteration API (`fat12_opendir` / `fat12_readdir` / `fat12_closedir`) and an `ls` command that uses it.

The Directory Entry

The root directory is the anchor of the entire filesystem — the top of the directory hierarchy. Every file and folder on the volume traces back to an entry in the root directory, or to a subdirectory that itself traces back to the root. Each entry describes one thing: a file, a subdirectory, or the volume label (more on that later).

The root directory itself is just an array of directory entries, each 32 bytes and structured like the diagram below:

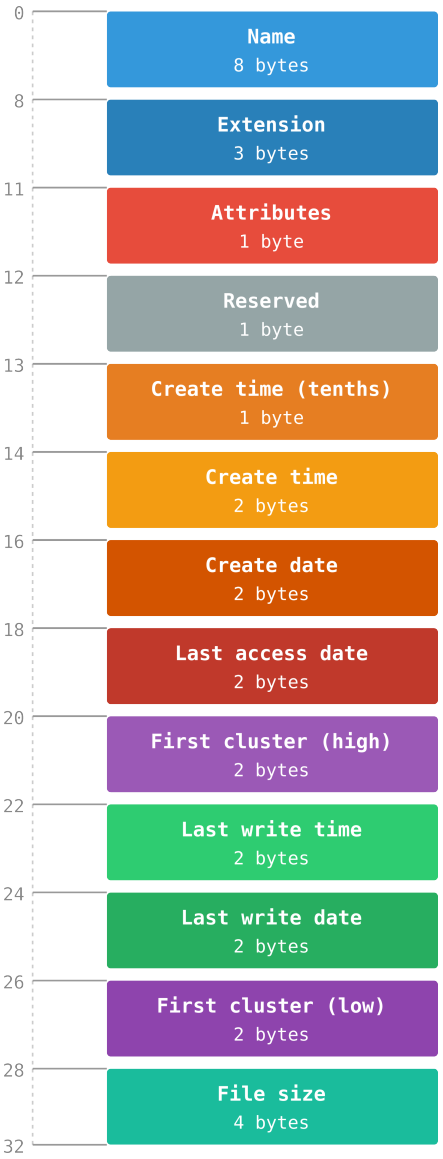

Figure 5. Directory entry layout

Most of these fields we will walk through in detail later in this chapter. Two matter right now: `file_size`, which tells us exactly how many bytes of the file's content to read, and `first_cluster`, which points to where that content actually begins on disk (more on that in Chapter 4).

Here is how the same layout maps to C. Add this packed struct after the `BootSector` definition, before `#pragma pack(pop)`:

src/layout.h

```
typedef struct
{
    char name[8];
    char ext[3];
    uint8_t attr;
    uint8_t reserved;
    uint8_t create_time_tenth;
    uint16_t create_time;
    uint16_t create_date;
    uint16_t last_access_date;
    uint16_t first_cluster_high;
    uint16_t last_write_time;
    uint16_t last_write_date;
    uint16_t first_cluster;
    uint32_t file_size;
} DirectoryEntry;
```

Position and Size

Before we can read the root directory, we need to know exactly where it lives on disk and how big it is. Here is an overview of the regions on a FAT12 volume.

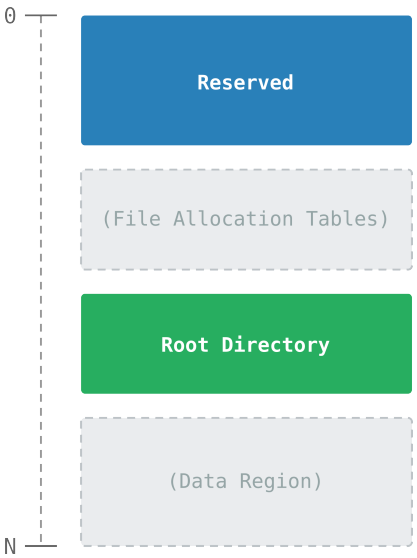

Figure 6. FAT12 volume layout – root directory highlighted

The reserved sector is where the boot sector lives, as we saw in the previous chapter. The File Allocation Tables (FATs, for short – the same FAT that gives FAT12 its name) and data region are shown, to give a correct overview of where the root directory is located – ignore them for now, they get their own treatment in Chapter 4.

As the diagram shows, the root directory sits after the FATs. To find where they start, we first need the reserved sector count – we already know this from the BPB (`reserved_sector_count`). That puts us at the start of the FATs.

We did not explain the FAT fields in Chapter 2, but we do here because we need them now. Two BPB fields tell us how big the file allocation tables are: `num_fats` (how many copies of the FAT exist) and `fat_size_16` (how many sectors each copy spans). The size of the file allocation tables, in sectors, is simply `num_fats * fat_size_16`. Add that to the reserved sector count and we land at the start of the root directory.

```
FATs start LBA      = reserved_sector_count
FATs size (sectors) = num_fats * fat_size_16
Root dir start LBA  = FATs start LBA + FATs size (sectors)
```

Now we need to know how many sectors the root directory itself occupies so we can load it into memory. The BPB gives us `root_entry_count` – the number of entries the root directory has space for. Multiply `root_entry_count * sizeof(DirectoryEntry)` to get the total byte size, then round up to a full sector because everything on disk is allocated in whole sectors:

```
Root directory size (sectors) = ceil(root_entry_count * sizeof(DirectoryEntry)
↪  / bytes_per_sector)
```

Unlike subdirectories (which we build later), the root directory is fixed size. The BPB field `root_entry_count` tells us how many entries fit. If we fill it up, no new entries can be created in the root – even if there is free space on the disk. The root directory is the one place on a FAT12 volume where the entry count is the bottleneck.

Let's translate these formulas directly into code. Extend the `FAT12FS` struct from Chapter 2 with cached fields, then compute them in `fat12_mount`:

src/fat12.c

```
struct FAT12FS {
    BlockDevice* device;
    BootSector bs;
    /* --- NEW: cached position and size, computed once on mount --- */
    uint32_t fat_lba;
    uint32_t fat_sectors;
    uint32_t root_dir_lba;
    uint32_t root_dir_sectors;
};
```

src/fat12.c

```
FAT12FS* fat12_mount(BlockDevice* device)
{
    FAT12FS* fs = (FAT12FS*)malloc(sizeof(FAT12FS));
    fs->device = device;
    fs->bs = read_boot_sector(device);

    /* --- NEW: cached position and size, computed once on mount --- */
    fs->fat_lba = fs->bs.bpb.reserved_sector_count;
    fs->fat_sectors = fs->bs.bpb.num_fats * fs->bs.bpb.fat_size_16;
    fs->root_dir_lba = fs->fat_lba + fs->fat_sectors;
    fs->root_dir_sectors = ((fs->bs.bpb.root_entry_count *
    ↪  sizeof(DirectoryEntry))
        + (fs->bs.bpb.bytes_per_sector - 1))
        / fs->bs.bpb.bytes_per_sector;

    DBG_PRINT("[ fat12 ] FAT start LBA: %u\n", fs->fat_lba);
    DBG_PRINT("[ fat12 ] FAT size (sectors): %u\n", fs->fat_sectors);
    DBG_PRINT("[ fat12 ] Root dir start LBA: %u\n", fs->root_dir_lba);
    DBG_PRINT("[ fat12 ] Root dir size (sectors): %u\n", fs->root_dir_sectors);

    return fs;
}
```

The formulas match the ones above — computed once, cached forever. `fat_lba` and `fat_sectors` will be used in Chapter 4 when we decode the File Allocation Table itself; for now they are the building blocks for `root_dir_lba`. This `fat12_mount` replaces Chapter 2's version wholesale — the field-by-field boot sector dump (`Bytes Per Sector`, `Media Descriptor`, and the rest) is gone, since its job was proving we decoded the boot sector correctly, and we already did. Flip `DEBUG` back to `1` and these four values print on every mount instead — a quick way to check your own by-hand math against what the library actually computed, instead of re-deriving the formulas from a hexdump every time.

Hexdumping the Root Directory

Now that we know where the root directory lives and what it contains, we will hexdump it to verify that. But first we need real entries on disk to look at, and for that we need a loopback mount – it treats our `disk.img` file as if it were a real block device, letting Linux's own FAT12 driver take over and giving us ordinary tools like `mkdir` and `echo` to create files and directories on it, exactly as it would on a physical disk. We will be able to write files with our own code soon enough; for now, this is the only way we have to put files on the disk, and it doubles as a check that our implementation reads them back correctly.

```
mkdir -p /tmp/fatmnt
sudo mount -o loop disk.img /tmp/fatmnt

echo "Hello from FAT12!" | sudo tee /tmp/fatmnt/HELLO.TXT
echo "Some text file"  | sudo tee /tmp/fatmnt/README.TXT
sudo mkdir /tmp/fatmnt/MYDIR

sudo umount /tmp/fatmnt
```

To find where the root directory starts, we need four BPB values – the same ones the debug output from Chapter 2 already prints:

```
[ fat12 ] Bytes Per Sector: 512
[ fat12 ] Reserved Sector Count: 1
[ fat12 ] Number of FATs: 2
[ fat12 ] FAT Size (sectors): 12
```

Now calculate the root directory byte offset using the formula from earlier:

```
Offset = (Reserved Sector Count + Number of FATs * FAT Size) * Bytes Per Sector
       = (1 + 2 * 12) * 512
       = 12800
```

Replace `<offset>` with your calculated value to dump 128 bytes (four entries) from the start of the root directory:

```
xxd -g 1 -s <offset> -l 128 disk.img
```

```
00003200: 4d 59 44 49 53 4b 20 20 20 20 20 08 00 00 4c 74   MYDISK     ...Lt
00003210: b5 5c 00 00 00 00 4c 74 b5 5c 00 00 00 00 00 00   .\....Lt.\......
00003220: 48 45 4c 4c 4f 20 20 20 54 58 54 20 00 00 4c 74   HELLO   TXT ..Lt
00003230: b5 5c 00 00 00 00 4c 74 b5 5c 02 00 12 00 00 00   .\....Lt.\......
00003240: 52 45 41 44 4d 45 20 20 54 58 54 20 00 00 4c 74   README  TXT ..Lt
00003250: b5 5c 00 00 00 00 4c 74 b5 5c 03 00 0f 00 00 00   .\....Lt.\......
00003260: 4d 59 44 49 52 20 20 20 20 20 20 10 00 00 4c 74   MYDIR      ...Lt
00003270: b5 5c 00 00 00 00 4c 74 b5 5c 04 00 00 00 00 00   .\....Lt.\......
```

The volume label comes first, followed by our three new entries — each exactly 32 bytes. Now let's break down every field and what it actually means.

The 8.3 Name Format

The 8.3 filename format was inherited from CP/M, the dominant operating system before DOS. The extension was not just a naming convention — it was a file type system. CP/M used the three-character suffix to identify what kind of file an entry was: `.COM` for executables, `.TXT` for text, `.BAS` for BASIC source code, and so on. The OS and applications relied on the extension to decide how to handle a file. Splitting name and extension into fixed-size fields also made directory scanning faster — the OS could check `ext[3]` directly without parsing a variable-length string. A file could still have no extension (just spaces in the `ext` field), in which case the dot is omitted on read.

The dot between name and extension is never stored on disk; it is implied. On read, software reconstructs it: if the extension has any non-space bytes, insert a dot before them.

Both fields are stored in ASCII, uppercased by convention, and space-padded (`0x20`) when the value is shorter than the field allows. A file named `README.TXT` appears on disk as `README  TXT` (two trailing spaces). A file with no extension, like `MYDIR`, stores `MYDIR   ` (six trailing spaces).

The valid character set for 8.3 names is limited: uppercase `A-Z`, digits `0-9`, and the special characters `$`, `%`, `'`, `-`, `_`, `@`, `~`, `` ` ``, `!`, `(`, `)`, `{`, `}`, `^`, `#` and `&`. Lowercase letters are not valid in the field itself — tools uppercase them before writing, so `readme.txt` on the command line ends up as `README.TXT` on disk. Case never matters for matching either way: FAT searches are case-insensitive, so

`README.TXT` and `readme.txt` are the same name. We will put that to work in Chapter 4. A space (byte `0x20`) is never valid inside a name or extension – it appears only as padding.

The first byte of a directory entry name, `name[0]`, has two special values.

`0x00` marks the end of the directory – no more entries after this. `0xE5` marks a deleted entry that can be reused. We will use both when we implement the directory iteration. Let's define them as constants:

src/fat12.c

```
#define NAME_END      0x00
#define NAME_DELETED  0xE5
```

One gotcha: byte `0xE5` is also a valid lead byte for Kanji characters in the Shift-JIS code page used in Japan. To resolve the conflict, the FAT specification requires the filesystem to store `0x05` instead of `0xE5` in `name[0]`. The byte `0x05` is the ASCII ENQ control character – a non-printable code that was never valid in filenames (the valid character set listed above doesn't include control characters). So there's no ambiguity: `0x05` can only mean "escaped `0xE5`." On read, software checks for `0x05` and replaces it with `0xE5` in the reconstructed string.

We want to show the filename as one zero-terminated string (`name.ext`). To do that we need to convert the raw 8.3 format, so we introduce this little helper function:

src/fat12.c

```c
static void decode_8_3_name(const DirectoryEntry* raw, char* out)
{
    int i = 0;

    /* 0x05 escape — real first byte is 0xE5 */
    if (raw->name[0] == 0x05)
        out[i++] = (char)0xE5;

    /* Copy name bytes until padding or end */
    for (; i < 8 && raw->name[i] != ' '; i++)
        out[i] = raw->name[i];

    /* If extension exists, insert dot and copy non-space bytes */
    int has_ext = raw->ext[0] != ' ';
    if (has_ext)
    {
        out[i] = '.';
        i++;
        for (int k = 0; k < 3; k++)
        {
            if (raw->ext[k] != ' ')
            {
                out[i] = raw->ext[k];
                i++;
            }
        }
    }

    out[i] = '\0';
}
```

The Attribute Byte

The attribute byte is a bitfield — each bit is its own independent flag. The table below lists every possible flag:

Bit	Flag	Meaning
0	Read-only	Blocks modification and deletion — protects important files from being changed or removed by accident. We enforce it when we delete files in Chapter 8.
1	Hidden	Hides the entry from normal directory listings, keeping clutter out of casual browsing.
2	System	Marks the entry as an OS-critical file. Tools hide it like `Hidden` and often refuse to touch it without an explicit override — extra protection for files the boot process depends on.
3	Volume Label	The entry stores the volume label instead of a real file — a convenient reuse of the existing 32-byte entry format instead of a dedicated structure.
4	Directory	The entry is a subdirectory.
5	Archive	Set when the file is created or modified, cleared by backup software after a backup runs. Before this flag existed, backup software had to compare timestamps or checksums across the entire disk to find changed files — expensive and slow. The archive bit gives a cheap answer: if it's set, the file changed since the last backup.
6-7	Reserved	Always 0 — set aside in the original spec for flags that were never added.

Some combinations of flags are worth calling out on their own, since their meaning is not just the sum of the individual bits:

Flags Combined	Meaning
Directory = 0 & Volume Label = 0	A regular file. Read-only, Hidden, System, and Archive may be set in any combination – only these two bits determine whether an entry counts as a regular file.
Read-only = 1 & Hidden = 1 & System = 1 & Volume Label = 1	Long filename (LFN / VFAT) fragment – a combination no real entry would ever use, making it a safe sentinel. LFNs support file names longer than 8.3 allows; each fragment entry stores a piece of the long name, and a short 8.3 alias is always available alongside it. We skip these fragment entries entirely, which is enough to avoid showing garbage in directory listings. LFN support is not core FAT12 and is not implemented in this book, but you can add it yourself later – see Appendix: Long Filename Support (LFN / VFAT) for an introduction to how they work.

We define constants for each flag in `fat12.h`. These names are self-documenting – no need to remember hex values while reading attribute bytes:

include/fat12.h

```
#define FAT12_ATTR_READ_ONLY    0x01
#define FAT12_ATTR_HIDDEN       0x02
#define FAT12_ATTR_SYSTEM       0x04
#define FAT12_ATTR_VOLUME_ID    0x08
#define FAT12_ATTR_DIRECTORY    0x10
#define FAT12_ATTR_ARCHIVE      0x20
```

One more constant is worth defining now: `FAT12_ATTR_LONG_NAME`, the LFN sentinel value from the table above. It is an internal implementation detail rather than part of the public API, so it lives in `fat12.c` instead:

`src/fat12.c`

```
#define FAT12_ATTR_LONG_NAME    0x0F
```

Decoding Timestamps

Each directory entry carries three time-related fields: creation time (`create_time` / `create_date`), last access date (`last_access_date`), and last modification time (`last_write_time` / `last_write_date`). Each pair shares the same two bit-packed 16-bit words.

The format for the time word:

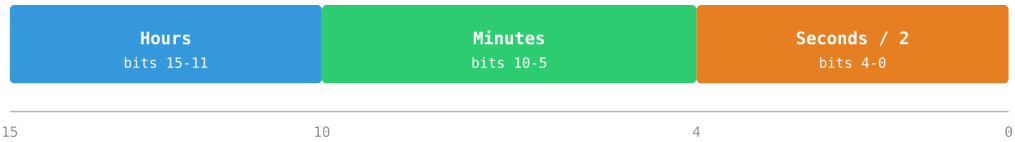

Figure 7. Time word bitfield layout

The format for the date word:

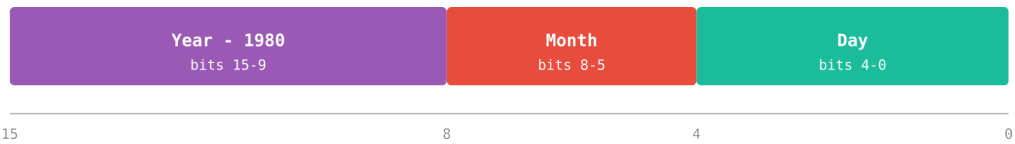

Figure 8. Date word bitfield layout

The directory entry has two more timestamp fields: `create_time_tenth`, a byte holding 0–199 centiseconds of sub-second creation precision, and `last_access_date`, which records the date a file was last read. Like most operating systems, we leave both unused.

Decoding a time word and a date word gives us six separate values, so we need somewhere to put them. Add this struct to `fat12.h`:

include/fat12.h

```
typedef struct {
    unsigned year;
    unsigned month;
    unsigned day;
    unsigned hours;
    unsigned minutes;
    unsigned seconds;
} Timestamp;
```

Now we can write the function that fills it in:

src/fat12.c

```
static Timestamp decode_dos_timestamp(uint16_t time, uint16_t date)
{
    Timestamp ts;
    ts.hours   = (time >> 11) & 0x1F;
    ts.minutes = (time >> 5)  & 0x3F;
    ts.seconds = (time & 0x1F) * 2;
    ts.day     = date & 0x1F;
    ts.month   = (date >> 5) & 0x0F;
    ts.year    = ((date >> 9) & 0x7F) + 1980;
    return ts;
}
```

Directory Iteration

Before we implement the directory iteration functions, let's define the public API in `fat12.h`. The types and function signatures go here – the implementation follows in `fat12.c`:

include/fat12.h

```
typedef struct {
    char name[13];
    uint32_t size;
    uint8_t attr;
    Timestamp create_time;
    Timestamp modify_time;
} DirEntry;

typedef struct Directory Directory;

Directory* fat12_opendir(FAT12FS* fs, const char* path);
int fat12_readdir(Directory* dir, DirEntry* out);
void fat12_closedir(Directory* dir);
```

With the public interface declared, we can build the implementation piece by piece, starting with how we get the root directory's entries off disk in the first place.

src/fat12.c

```
static DirectoryEntry* read_root_directory(
    FAT12FS* fs,
    uint32_t* count
)
{
    uint32_t bytes = fs->root_dir_sectors * fs->bs.bpb.bytes_per_sector;
    *count = bytes / sizeof(DirectoryEntry);
    DirectoryEntry* entries = (DirectoryEntry*)malloc(bytes);
    block_device_read(fs->device, fs->root_dir_lba, fs->root_dir_sectors,
    ↪  entries);

    return entries;
}
```

The root directory lives at a fixed location — Chapter 2 already computed it (`root_dir_lba`, `root_dir_sectors`) — so `read_root_directory` reads it in one shot straight into a freshly allocated buffer.

That buffer is raw and unfiltered — it still contains deleted slots and long-filename fragments mixed in with real entries. The next two functions turn that raw array into something we can iterate safely.

src/fat12.c

```c
static int is_deleted_entry(const DirectoryEntry* entry)
{
    // Cast to unsigned char: 0xE5 doesn't fit a signed char (-128 to 127),
    // so without the cast, sign-extension turns it into 0xFFFFFFE5 and the
    // comparison fails.
    return (unsigned char)entry->name[0] == NAME_DELETED;
}

static int next_active_entry(
    DirectoryEntry* entries,
    uint32_t count,
    uint32_t* offset,
    DirectoryEntry** out
)
{
    while (*offset < count)
    {
        DirectoryEntry* raw = &entries[*offset];

        /* End-of-directory marker — no more entries after this */
        if (raw->name[0] == NAME_END) return 0;

        (*offset)++;

        /* Deleted entry */
        if (is_deleted_entry(raw)) continue;
        /* Long filename fragment — not a real entry */
        if (raw->attr == FAT12_ATTR_LONG_NAME) continue;

        *out = raw;
        return 1;
    }

    return 0;
}
```

`next_active_entry` walks the array from `*offset` forward and hands back the next real entry, skipping everything deleted and any long-filename fragment along the way. Filtering happens once, here, instead of at every call site — every caller of `fat12_readdir` only ever sees real, live entries.

With loading and filtering in place, we need somewhere to keep track of where we are mid-scan, and a way to resolve a path down to a starting point:

src/fat12.c

```c
// Sentinel meaning "the root directory"
#define ROOT_DIR_CLUSTER 0

struct Directory {
    DirectoryEntry* entries;
    uint32_t count;
    uint32_t offset;
};

static int resolve_path(
    FAT12FS* fs,
    const char* path,
    DirectoryEntry* out
)
{
    (void)fs;
    const char* p = path;
    if (*p == '/') p++;

    if (*p == '\0')
    {
        out->attr = FAT12_ATTR_DIRECTORY;
        out->first_cluster = ROOT_DIR_CLUSTER;
        return 0;
    }

    return -1;
}
```

`Directory` is defined here in `fat12.c`, not in the header — `fat12.h` only forward-declares it. Callers get back an opaque `Directory*` from `fat12_opendir` and hand it to `fat12_readdir`/`fat12_closedir` without ever seeing what is inside, the same pattern C's own `FILE*` uses. That lets us change how we track iteration state later without breaking anyone who calls the API.

`ROOT_DIR_CLUSTER` is just a name for `0` — cluster zero is never a valid data cluster, so using it as a sentinel for "this is the root directory" is safe and conventional in every FAT implementation. `resolve_path` puts it to use: for the path `"/"`, it hands back a synthetic entry whose `first_cluster` is `ROOT_DIR_CLUSTER`, which is how the rest of the code recognizes "this is root." Every other path returns `-1` for now — the `fs` parameter goes unused for now, but the signature stays fixed so the resolver can grow into real subdirectory traversal in Chapter 5 without changing every caller.

Now we have everything `fat12_opendir`, `fat12_readdir`, and `fat12_closedir` need — this is where it all comes together into the public API, following the same open/read/close shape as POSIX's `opendir/readdir/closedir`:

src/fat12.c

```
Directory* fat12_opendir(FAT12FS* fs, const char* path)
{
    DirectoryEntry resolved;
    if (resolve_path(fs, path, &resolved) != 0)
        return NULL;
    if (!(resolved.attr & FAT12_ATTR_DIRECTORY))
        return NULL;

    uint32_t count;
    DirectoryEntry* entries = read_root_directory(fs, &count);
    Directory* dir = (Directory*)malloc(sizeof(Directory));
    dir->entries = entries;
    dir->count = count;
    dir->offset = 0;

    return dir;
}

int fat12_readdir(Directory* dir, DirEntry* out)
{
    DirectoryEntry* raw;
    if (!next_active_entry(dir->entries, dir->count, &dir->offset, &raw))
        return -1;

    memset(out->name, 0, sizeof(out->name));
    decode_8_3_name(raw, out->name);

    out->size = raw->file_size;
    out->attr = raw->attr;
    out->create_time = decode_dos_timestamp(raw->create_time,
    ↪  raw->create_date);
    out->modify_time = decode_dos_timestamp(raw->last_write_time,
    ↪  raw->last_write_date);

    return 0;
}

void fat12_closedir(Directory* dir)
{
    free(dir->entries);
```

```
    free(dir);
}
```

`fat12_opendir` resolves the path, rejects anything that is not a directory, then loads and wraps the entries in a fresh `Directory`. `fat12_readdir` leans entirely on `next_active_entry` for filtering and just decodes whatever it finds. `fat12_closedir` frees what `fat12_opendir` allocated – every `opendir` call needs a matching `closedir`, or the entry buffer leaks.

The ls Command

Now that we have the directory iteration API, we will extend `main` from Chapter 2 – which just mounts and unmounts – to detect the `ls` command and use the directory iteration API to print the contents of the root directory. This is the boilerplate we will extend in the coming chapters to add more CLI commands:

`platform/cli/main.c`

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include "block_device.h"
#include "fat12.h"
#include "file_block_device.h"

static void format_datetime(char *buf, size_t size,
                            Timestamp ts)
{
    snprintf(buf, size, "%04u-%02u-%02u %02u:%02u:%02u", ts.year, ts.month,
    ↪  ts.day, ts.hours, ts.minutes, ts.seconds);
}

static int cmd_ls(FAT12FS *fs, int argc, char *argv[])
{
    if (argc < 3)
    {
        fprintf(stderr, "usage: %s ls <path>\n", argv[0]);
        return 1;
    }

    const char *path = argv[2];
    Directory *dir = fat12_opendir(fs, path);
```

```
    if (dir == NULL)
    {
        fprintf(stderr, "error: directory not found\n");
        return 1;
    }

    DirEntry entry;
    while (fat12_readdir(dir, &entry) == 0)
    {
        char time_str[20];
        char type_str[16];
        format_datetime(time_str, sizeof(time_str), entry.modify_time);

        if (entry.attr == FAT12_ATTR_VOLUME_ID)
            snprintf(type_str, sizeof(type_str), "<VOL>");
        else if (entry.attr & FAT12_ATTR_DIRECTORY)
            snprintf(type_str, sizeof(type_str), "<DIR>");
        else
            snprintf(type_str, sizeof(type_str), "%u B", entry.size);

        printf("%-12s  %-8s  %s\n", entry.name, type_str, time_str);
    }

    fat12_closedir(dir);
    return 0;
}

int main(int argc, char *argv[])
{
    if (argc < 2)
    {
        fprintf(stderr, "usage: %s <command>\n", argv[0]);
        return 1;
    }

    BlockDevice *device = file_block_device_open("disk.img");
    if (device == NULL)
    {
        fprintf(stderr, "error: could not open disk image\n");
        return 1;
    }

    FAT12FS *fs = fat12_mount(device);
    if (fs == NULL)
    {
        fprintf(stderr, "error: could not mount filesystem\n");
        block_device_close(device);
        return 1;
    }

```

```
75      char *command = argv[1];
76      int ret = 0;
77
78      if (strcmp(command, "ls") == 0)
79          ret = cmd_ls(fs, argc, argv);
80      else
81      {
82          fprintf(stderr, "unknown command: %s\n", command);
83          ret = 1;
84      }
85
86      fat12_umount(fs);
87      block_device_close(device);
88
89      return ret;
90  }
```

The `cmd_ls` function requires a path argument, then calls `fat12_opendir` and loops through every entry with `fat12_readdir`, printing `<VOL>` for a volume label, `<DIR>` for a directory, or the size in bytes for a regular file.

The `DirEntry` name is already formatted into a readable string like `HELLO.TXT` or `MYDIR`, with trailing spaces stripped and a dot between the name and extension. After the size or type marker, each line shows the file's last-write timestamp formatted by `format_datetime`.

Entries print in on-disk order, not alphabetically — the same order `next_active_entry` walks the array, with the holes left by deleted entries simply skipped over.

With `cmd_ls` wired into `main`, let's see it work. We already have a disk with real entries on it — the same `HELLO.TXT`, `README.TXT`, and `MYDIR` we created for the hexdump earlier. Compile and run:

```
./build.sh
./fat12-cli ls /
```

Output (your timestamps will differ):

```
MYDISK        <VOL>     2026-05-21 14:34:22
HELLO.TXT     18 B      2026-05-21 14:34:24
README.TXT    15 B      2026-05-21 14:34:24
MYDIR         <DIR>     2026-05-21 14:34:24
```

Every entry we hand-decoded from the hexdump earlier is visible here too — our own code now parses the same bytes we read by hand. The sizes match: `HELLO.TXT` contains 18 bytes ("Hello from FAT12!" plus the newline that `echo` appends), `README.TXT` contains 15. The subdirectory shows `<DIR>` instead of a size, and the volume label shows `<VOL>`. The timestamp on each entry is its last-write time.

> **Volume Label Duplication:** When a volume label is set, it can appear in two places. The primary location is this root directory entry with the volume label attribute — reusing the existing 32-byte directory entry format as a ready-made slot for the label instead of inventing a new on-disk structure just for one string. The secondary location is a dedicated volume label field inside the Extended BPB. The root directory entry is the source of truth: the FAT spec expects drivers to update that field whenever the volume label entry in the root directory is created or renamed, keeping the BPB copy in sync rather than treating it as an independent record.

What's Next

We can now list the root directory's contents — names, sizes, types, timestamps, all correctly decoded. `resolve_path` only understands `"/"` for now; Chapter 5 extends it into full path walking so `fat12_opendir` works on subdirectories too.

But a directory entry only tells us *where* a file begins, not how far it goes. That's the `first_cluster` field we teased earlier in this chapter: it points to the first cluster of a file's data, and nothing else in the entry says what comes next.

Finding out is the job of the File Allocation Table — the structure the whole filesystem is named after. Chapter 4 introduces it: how clusters chain together in the data region, how to follow that chain to read a file end to end, and the `cat` command that puts it to use.

Chapter 4: Reading Files

At the end of Chapter 3, `ls` printed `HELLO.TXT` on screen — name, size, timestamp, all decoded straight from a 32-byte directory entry. But one field went unused: `first_cluster`, a bare number. That number alone doesn't get us the file's contents — it's just a starting point. How do we turn it into the actual bytes on disk?

By the end of this chapter, a `cat` command will read a file's full contents from disk, and we'll verify the output byte-for-byte against the original.

But first, we need to understand what a cluster is and how it relates to a file's contents.

Clusters: The Allocation Units

`first_cluster` points somewhere inside the **data region** — the largest part of the volume. The data region is nothing more than one long array of numbered, equally-sized chunks — these are the clusters:

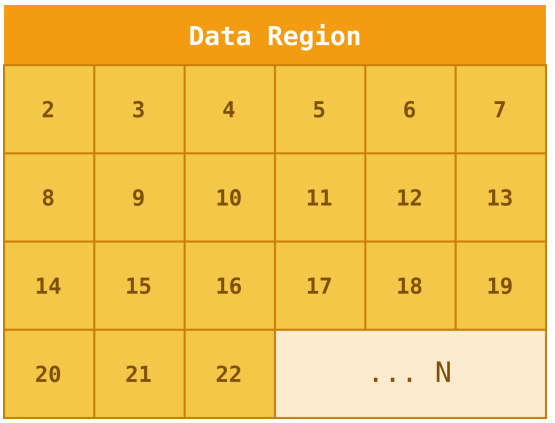

Figure 9. Data region as numbered cluster chunks

Each cluster is a contiguous span of sectors — the smallest amount of disk space a file can occupy. Our 4 MiB disk uses 2 sectors per cluster because

that is what the `-s 2` flag in `create_disk.sh`'s `mkfs.fat` command set back in Chapter 1. The formatter records the value in the BPB's `sectors_per_cluster` field, which we listed in Chapter 2 but have not needed until now — so each cluster is `sectors_per_cluster * bytes_per_sector = 2 * 512 = 1024` bytes. Every file on disk consumes a whole number of clusters, even a tiny file — the remaining bytes in the last cluster are allocated to the file whether they hold data or not.

> **Why clusters instead of sectors?** A 4 MiB disk has nearly 8000 sectors. Tracking 8000 individual allocation units would bloat the FAT to 12 KiB for 12-bit entries. Clusters reduce the bookkeeping: with 2 sectors per cluster, the same volume has about 4000 clusters, and the FAT shrinks to 6 KiB. The trade-off is internal fragmentation — a 1-byte file wastes 1023 bytes — but on small volumes, the FAT size savings are worth it.

Clusters 0 and 1 are reserved — they exist as bookkeeping entries but have no corresponding space in the data region. We will see why when we get to the FAT. The first real cluster on disk is cluster 2, and it maps to the very first sector of the data region.

Locating Data on Disk

We know clusters live inside the data region — now we need to find exactly where that region starts on disk.

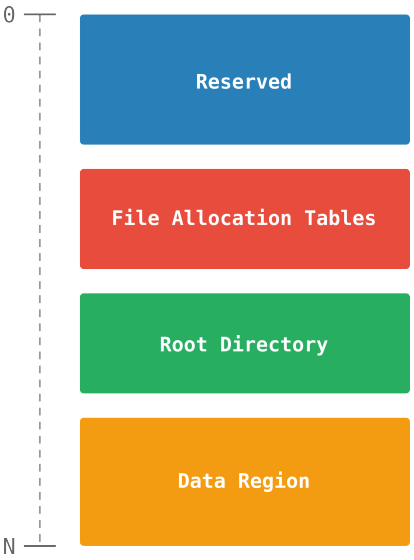

Figure 10. FAT12 volume layout – full

As the diagram above shows, the data region starts right after the root directory. From Chapter 3 we have `root_dir_lba` and `root_dir_sectors` cached in `FAT12FS` – rebuilding Chapter 3's code with `DEBUG=1` prints them on every mount:

```
[ fat12 ] Root dir start LBA: 25
[ fat12 ] Root dir size (sectors): 32
```

The first data sector is simply:

```
Data region start LBA = Root dir start LBA + Root directory size (sectors)
                      = 25 + 32
                      = 57
```

So cluster 2 – the first real cluster – maps directly to Data region start LBA, and every cluster after it maps to its own LBA the same way.

> **Why fixed-size clusters?** Because every cluster is the same size, "where is cluster N" collapses into the one-line formula above: multiply and add. A filesystem built on variable-sized allocation regions would need to store each region's size somewhere and walk a list to find any given piece of a file

> – no direct formula, no constant-time lookup. The price is the same internal fragmentation we saw earlier (a tiny file still eats a whole cluster) – one design choice, both effects.

Let's see it on a real file. Reuse the exact command from Chapter 3 – `<offset>` is still the root directory's start:

```
xxd -g 1 -s <offset> -l 128 disk.img
```

```
00003200: 4d 59 44 49 53 4b 20 20 20 20 20 08 00 00 4c 74   MYDISK     ...Lt
00003210: b5 5c 00 00 00 00 4c 74 b5 5c 00 00 00 00 00 00   .\....Lt.\......
00003220: 48 45 4c 4c 4f 20 20 20 54 58 54 20 00 00 4c 74   HELLO   TXT ..Lt
00003230: b5 5c 00 00 00 00 4c 74 b5 5c 02 00 12 00 00 00   .\....Lt.\......
00003240: 52 45 41 44 4d 45 20 20 54 58 54 20 00 00 4c 74   README  TXT ..Lt
00003250: b5 5c 00 00 00 00 4c 74 b5 5c 03 00 0f 00 00 00   .\....Lt.\......
00003260: 4d 59 44 49 52 20 20 20 20 20 20 10 00 00 4c 74   MYDIR       ...Lt
00003270: b5 5c 00 00 00 00 4c 74 b5 5c 04 00 00 00 00 00   .\....Lt.\......
```

Focus on the third and fourth rows – `HELLO.TXT`'s entry. We already know the last four bytes of its second row, `12 00 00 00` – that is `file_size`. The two bytes right before it, `02 00`, are `first_cluster`: that is cluster 2. (If your own dump shows a different value, use that instead of 2 below – it depends on the exact order you created files in Chapter 3.)

Plug that into the Cluster LBA formula:

```
Cluster LBA = Data region start LBA + (cluster - 2) * Sectors per cluster
            = 57 + (2 - 2) * 2
            = 57
```

Let's convert that LBA into a byte offset, the same way we found the root directory's offset in Chapter 3, and go look:

```
Byte offset = Cluster LBA * Bytes per sector
            = 57 * 512
            = 29184
```

We only read `file_size` bytes – anything past that is unused slack space left over in the cluster, and FAT12 makes no promise about what is sitting there, so we ignore it entirely. Replace `<offset>` and `<file_size>` with your own values – here that's 29184 and 18:

```
xxd -g 1 -s <offset> -l <file_size> disk.img
```

```
00007200: 48 65 6c 6c 6f 20 66 72 6f 6d 20 46 41 54 31 32  Hello from FAT12
00007210: 21 0a                                           !.
```

There it is – `HELLO.TXT`'s own bytes, sitting exactly where the formula predicted. We have not written a line of file-reading code yet, and we already know exactly where this file's data lives.

We cache the data region's start position in `FAT12FS` and compute it once on mount:

src/fat12.c

```c
struct FAT12FS {
    BlockDevice* device;
    BootSector bs;
    uint32_t fat_lba;
    uint32_t fat_sectors;
    uint32_t root_dir_lba;
    uint32_t root_dir_sectors;
    /* --- NEW: first data-region LBA, computed once on mount --- */
    uint32_t first_data_lba;
};

// ...

FAT12FS* fat12_mount(BlockDevice* device)
{
    FAT12FS* fs = malloc(sizeof(FAT12FS));
    fs->device = device;
    fs->bs = read_boot_sector(device);
    fs->fat_lba = fs->bs.bpb.reserved_sector_count;
```

```
    fs->fat_sectors = fs->bs.bpb.num_fats * fs->bs.bpb.fat_size_16;
    fs->root_dir_lba = fs->fat_lba + fs->fat_sectors;
    fs->root_dir_sectors = ((fs->bs.bpb.root_entry_count *
    ↪  sizeof(DirectoryEntry))
        + (fs->bs.bpb.bytes_per_sector - 1))
        / fs->bs.bpb.bytes_per_sector;
    /* --- NEW: first data-region LBA, computed once on mount --- */
    fs->first_data_lba = fs->root_dir_lba + fs->root_dir_sectors;

    DBG_PRINT("[ fat12 ] FAT start LBA: %u\n", fs->fat_lba);
    DBG_PRINT("[ fat12 ] FAT size (sectors): %u\n", fs->fat_sectors);
    DBG_PRINT("[ fat12 ] Root dir start LBA: %u\n", fs->root_dir_lba);
    DBG_PRINT("[ fat12 ] Root dir size (sectors): %u\n", fs->root_dir_sectors);
    /* --- NEW: first data-region LBA --- */
    DBG_PRINT("[ fat12 ] First data LBA: %u\n", fs->first_data_lba);

    return fs;
}
```

Mount now traces `First data LBA` alongside the FAT and root-directory positions we started printing in Chapter 3. Flip `DEBUG` on and rebuild:

```
./build.sh
./fat12-cli ls /
```

We should see:

```
[ fat12 ] FAT start LBA: 1
[ fat12 ] FAT size (sectors): 24
[ fat12 ] Root dir start LBA: 25
[ fat12 ] Root dir size (sectors): 32
[ fat12 ] First data LBA: 57
```

57 — exactly the Data region start LBA we hand-computed above, confirmed by the library itself. Set `DEBUG` back to `0` when you are done.

We also need a helper that converts a cluster number to its LBA:

`src/fat12.c`

```c
static uint32_t data_cluster_to_lba(FAT12FS* fs, uint16_t cluster)
{
    uint32_t lba = fs->first_data_lba + ((cluster - 2) *
    ↪  fs->bs.bpb.sectors_per_cluster);
    DBG_PRINT("[ fat12 ] cluster %u -> LBA %u\n", cluster, lba);
    return lba;
}
```

The Cluster Chain

`HELLO.TXT` fit comfortably in a single 1024-byte cluster, but not every file is that lucky. When a file outgrows one cluster, it gets split into cluster-sized pieces and scattered across the data region. Now we have a new problem: how do we know which clusters belong to which file, and in which order do they need to be assembled? The File Allocation Table (FAT) solves this — it links those scattered pieces together into one logical file. The FAT is a flat array of entries, one per cluster. Each entry is 12 bits wide — exactly one and a half bytes, an odd size for something so fundamental (we will see exactly why that stride matters once we get to decoding entries). Each cluster has an entry in the FAT that stores the number of the next cluster in the chain. The last cluster's entry holds an end-of-chain marker. Walking this chain reassembles the file's data in order.

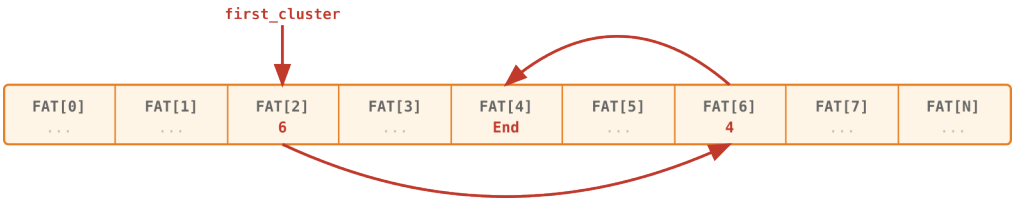

Figure 11. FAT array: walking a cluster chain from first_cluster to end-of-chain

The diagram above shows exactly that: each cluster's slot points to the next. Picture a hypothetical file whose directory entry gives `first_cluster = 2` — we will decode a real file's actual FAT bytes by hand shortly. Follow the chain step by step:

1. Start at cluster 2. FAT[2] points to 6.
2. Jump to cluster 6. FAT[6] points to 4.
3. Jump to cluster 4. FAT[4] holds the end-of-chain marker.

The chain is 2 -> 6 -> 4 — the order the file's pieces need to be read in, even though the clusters themselves are scattered across the data region:

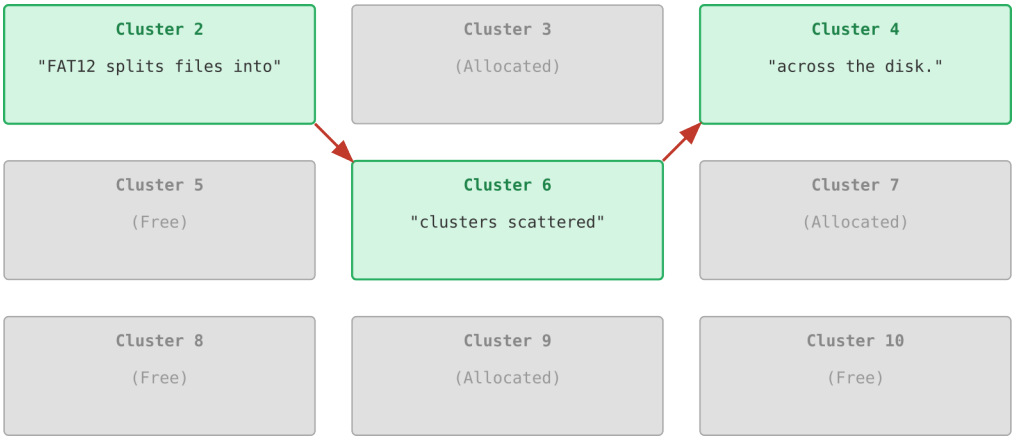

Figure 12. Data region: the same chain followed through the actual clusters

Following that chain through the data region, we read the content sequentially from these three clusters and concatenate them into the complete file:

```
"FAT12 splits files into " + "clusters scattered " + "across the disk."
```

Every cluster on the volume has a corresponding entry in the FAT. As we noted earlier, clusters 0 and 1 are reserved – now we can see why. They have entries in the FAT but no corresponding space in the data region. FAT[0] stores the media descriptor in its low 8 bits with the remaining bits set to 1 (`0xFF8` for a hard disk), and FAT[1] is set to `0xFFF`. The media descriptor in FAT[0] is a historical artifact – the earliest FAT filesystems had no boot-sector field for it, so the FAT itself was the only place to record the media type. The BPB later added the `media` byte to the boot sector, making FAT[0]'s copy redundant, but it stayed for backward compatibility. Our own code never even needs the media descriptor itself – we don't branch on media type anywhere – so FAT[0] and FAT[1] are dead weight for us either way; we can ignore both and go straight to the entries that matter. Data clusters start at index 2. Every other entry just stores the next cluster number, as described above – but a handful of specific values carry special meaning instead:

Value (hex)	Name	Description
`0x000`	Free cluster	Available for allocation

Value (hex)	Name	Description
`0xFF7`	Bad cluster	Physically damaged, so the filesystem skips it permanently (found by formatting/repair tools like `fsck.fat`, not the driver at runtime; modern drives remap bad sectors in firmware, so the OS rarely sees this today)
`0xFF8–0xFFF`	End-of-chain marker	Last cluster in a file's chain. `0xFFF` is the value every tool actually writes today — Microsoft's own drivers and Linux's `mkfs.fat` both agree on it. The rest of the range is legacy: old third-party DOS disk utilities once wrote other values here for the same meaning, so a conformant reader still tests the whole range (`>= 0xFF8`) on read, even though nothing writes anything but `0xFFF` anymore

We know what these values mean. Now for the hard part: actually pulling one out of the raw bytes for a given cluster. It is simple in principle, but that odd 12-bit stride finally catches up with us here — decoding an entry takes more care than anything we have done with this disk so far.

Decoding the 12-Bit Packing

Let's make this hands-on with a real file — but `HELLO.TXT` won't do: it fit in a single cluster, so its FAT entry is just an end-of-chain marker, never a real multi-cluster chain we can decode by hand. We need a file too big for one

cluster. We still have the disk from Chapter 3 — no need to start over, just add one more file. Type the name in uppercase, exactly as shown below: some Linux `vfat` driver versions write an extra long-filename (LFN) entry ahead of a mixed- or lowercase name even when it fits 8.3, which would shift every hexdump offset in the walkthrough below. Chapter 3 already established that FAT12 matching itself is case-insensitive, so typing it uppercase costs us nothing and sidesteps the issue entirely.

```
mkdir -p /tmp/fatmnt
sudo mount -o loop disk.img /tmp/fatmnt

# 1024 bytes of 'A', then 1024 of 'B', then 1024 of 'C' — three clusters, easy to tell apart in the output
printf '%1024s' '' | tr ' ' 'A' > /tmp/fatmnt/BIGFILE.TXT
printf '%1024s' '' | tr ' ' 'B' >> /tmp/fatmnt/BIGFILE.TXT
printf '%1024s' '' | tr ' ' 'C' >> /tmp/fatmnt/BIGFILE.TXT

sudo umount /tmp/fatmnt
```

`BIGFILE.TXT` becomes the fifth root directory entry, right after `MYDIR` — 128 bytes past the root directory's start, since the four entries ahead of it take up 32 bytes each: `12800 + 128 = 12928`. Reuse the same command from Chapter 3 again — `<offset>` is still the root directory's start — just ask for 160 bytes (five entries) instead of 128, so `BIGFILE.TXT` shows up alongside the files we already know:

```
xxd -g 1 -s <offset> -l 160 disk.img
```

```
00003200: 4d 59 44 49 53 4b 20 20 20 20 20 08 00 00 4c 74   MYDISK     ...Lt
00003210: b5 5c 00 00 00 00 4c 74 b5 5c 00 00 00 00 00 00   .\....Lt.\......
00003220: 48 45 4c 4c 4f 20 20 20 54 58 54 20 00 00 4c 74   HELLO   TXT ..Lt
00003230: b5 5c 00 00 00 00 4c 74 b5 5c 02 00 12 00 00 00   .\....Lt.\......
00003240: 52 45 41 44 4d 45 20 20 54 58 54 20 00 00 4c 74   README  TXT ..Lt
00003250: b5 5c 00 00 00 00 4c 74 b5 5c 03 00 0f 00 00 00   .\....Lt.\......
00003260: 4d 59 44 49 52 20 20 20 20 20 20 10 00 00 4c 74   MYDIR       ...Lt
00003270: b5 5c 00 00 00 00 4c 74 b5 5c 04 00 00 00 00 00   .\....Lt.\......
00003280: 42 49 47 46 49 4c 45 20 54 58 54 20 00 00 4c 74   BIGFILE TXT ..Lt
00003290: b5 5c 00 00 00 00 4c 74 b5 5c 05 00 00 0c 00 00   .\....Lt.\......
```

Same trick as before: focus on the last two rows, `BIGFILE.TXT`'s entry. The last four bytes are `file_size` (`00 0c 00 00` = 3072, our 3 KiB file), and the two

bytes right before it are `first_cluster – 05 00`, cluster 5. Cluster 5 alone only holds 1024 bytes, and the file is 3072 – the rest has to be somewhere, and the FAT is what tells us where. `fat_lba` is 1, so the FAT starts at byte offset `1 * 512 = 512`. Replace `<fat_offset>` with your calculated value:

```
xxd -g 1 -s <fat_offset> -l 16 disk.img
```

```
00000200: f8 ff ff ff ff ff ff 6f 00 07 f0 ff 00 00 00 00
```

Right now these sixteen bytes are just noise, with nothing obviously marking where cluster 5's entry ends and cluster 6's begins. Here is why: FAT12 packs each entry into exactly 12 bits – the odd stride we teased earlier – so entries do not line up with byte boundaries the way everything else on this disk has so far. Think of them as sliding windows across the byte array: cluster 0's entry covers byte 0 plus the low nibble of byte 1, cluster 1's entry covers the high nibble of byte 1 plus byte 2, and so on.

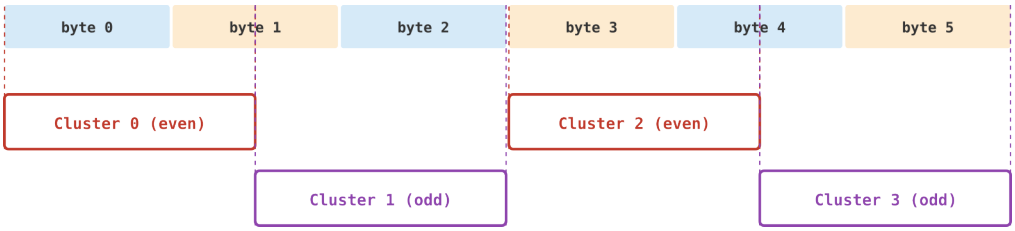

Figure 13. 12-bit entries as 1.5-byte sliding windows across the FAT byte array

Finding an entry starts with a simple offset: `offset = cluster + cluster/2`. `first_cluster` is 5, so the chain starts at `offset = 5 + 5/2 = 7` – that's why we zoom into offsets 7-11 from the 16-byte dump above.

Hex hides the bit-level picture. Let's translate those five bytes to binary first:

```
xxd -b -s 519 -l 5 disk.img
```

```
00000207: 01101111 00000000 00000111 11110000 11111111          o....
```

Decoding an entry is always the same two steps: combine the two bytes at an entry's offset into a single 16-bit value, then keep only 12 of those 16 bits – which 12 depends on whether the cluster number is even or odd. An even cluster keeps the low 12 bits of that value; an odd cluster keeps the high 12 bits, shifted down by 4. That is also why the byte at `offset + 1` is shared between two clusters: half its bits belong to the even cluster's entry, the other half to the odd cluster's entry. The one catch is step one – combining two bytes into a value is not as simple as reading them left to right, as we are about to see.

Here's the catch: FAT12 writes every multi-byte value little-endian – the byte at the *lower* offset holds the *least* significant half of the value, the byte at the *higher* offset holds the *most* significant half. Take the small value `6`: written the way we would naturally read it, most-significant byte first, it is `00000000 00000110`. Little-endian flips which byte comes first – the same value is stored on disk as `00000110 00000000`, low byte first. That means the raw bytes are stored in the opposite order from how we would write the number by hand. `xxd` shows them exactly as stored, so reading `6f 00` left to right and expecting it to spell out the value directly gives the wrong answer – the real value swaps them: `00 6f` = `0x006f`. Now let's put both steps together for real: swap each cluster's own pair of bytes into that 16-bit value, then trim it the same way as before – keep the low 12 bits for an even cluster, the high 12 bits (shifted down by 4) for an odd one:

```
Cluster 5 (odd), bytes 7-8:
01101111 00000000     as they sit on disk
00000000 01101111     bytes swapped – the real register value, 0x006f
0000 0000 0110        drop the last nibble   =  0x006

Cluster 6 (even), bytes 9-10:
00000111 11110000     as they sit on disk
11110000 00000111     bytes swapped – the real register value, 0xf007
     0000 0000 0111   drop the first nibble  =  0x007

Cluster 7 (odd), bytes 10-11:
11110000 11111111     as they sit on disk
11111111 11110000     bytes swapped – the real register value, 0xfff0
1111 1111 1111        drop the last nibble   =  0xFFF
```

Here is the same thing we just did by hand, shown visually. It looks more involved than the sliding-window diagram earlier in the chapter – that one

drew clusters as clean, non-overlapping spans and quietly left little-endian byte order out of the picture, to keep the first introduction simple. This diagram puts it back in: the raw byte order and the swap it forces, exactly as we just worked through by hand:

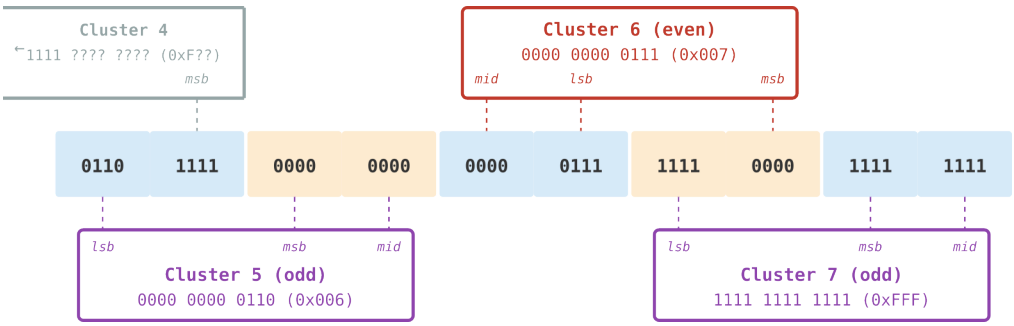

Figure 14. Worked example: decoding clusters 5, 6, and 7 from BIGFILE.TXT's raw FAT bytes

Cluster 5's entry is `6`, cluster 6's is `7`, and cluster 7's is `0xFFF` — the end-of-chain marker. The chain is `5 -> 6 -> 7 -> end`, decoded entirely by hand. It happens to be contiguous here only because nothing has been deleted from this disk yet, and because Linux's kernel FAT driver — which actually wrote `BIGFILE.TXT`, since we created it through the loopback mount — happened to scan for free clusters from the start and hand out the next one it found. That is a common allocator strategy, not a guarantee FAT12 itself makes; a different driver is free to hand out clusters in a different order even on an otherwise-empty disk — chains scatter exactly like the toy example earlier in this chapter, and the FAT is what makes the file readable correctly either way.

Reusing the Cluster LBA and byte offset formulas from earlier:

```
Cluster LBA = Data region start LBA + (cluster - 2) * Sectors per cluster
Byte offset = Cluster LBA * Bytes per sector
```

Plugged in for clusters 5, 6, and 7:

```
Cluster 5: Cluster LBA = 57 + (5 - 2) * 2 = 63    Byte offset = 63 * 512 = 32256
Cluster 6: Cluster LBA = 57 + (6 - 2) * 2 = 65    Byte offset = 65 * 512 = 33280
Cluster 7: Cluster LBA = 57 + (7 - 2) * 2 = 67    Byte offset = 67 * 512 = 34304
```

Read 16 bytes from the start of each cluster:

```
xxd -g 1 -s 32256 -l 16 disk.img   # cluster 5, LBA 63
xxd -g 1 -s 33280 -l 16 disk.img   # cluster 6, LBA 65
xxd -g 1 -s 34304 -l 16 disk.img   # cluster 7, LBA 67
```

```
00007e00: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00008200: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
00008600: 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
```

A's, then B's, then C's — the file reassembled by hand, cluster by cluster, exactly as we typed it. Everything from here on is just teaching the computer to do what we just did with `xxd` and a calculator.

Loading the FAT

To read FAT entries, the table needs to be in memory first. Its position and size are already cached in `FAT12FS` from Chapter 3 — `fat_lba` and `fat_sectors` — so we know exactly where to read.

The right place to load it is `fat12_mount`, which so far only reads the boot sector. Almost every operation touches the FAT repeatedly, so loading the whole thing once avoids hundreds of tiny disk reads — and at 6 KiB for our 4 MiB volume, the memory cost is negligible.

While we are in there, let's cache one more number: the total count of data clusters on the volume. Every cluster chain we walk from here on needs a way to tell a legitimate cluster number from an invalid one — a corrupted directory entry could claim a file starts at cluster 9999 on a volume that only has 4000 data clusters, and nothing about that FAT entry value alone gives it away.

Add the FAT buffer and the cluster count to `FAT12FS`, then update `fat12_mount` to read the FAT and cache the cluster count after the boot sector. Mount already knows the volume's total sector count, so it is the natural place to compute the cluster count once instead of recomputing it on every chain walk — that is what the small `total_data_clusters` helper does, right above where mount calls it:

src/fat12.c

```
struct FAT12FS {
    BlockDevice* device;
    BootSector bs;
    uint32_t fat_lba;
    uint32_t fat_sectors;
    uint32_t root_dir_lba;
    uint32_t root_dir_sectors;
    uint32_t first_data_lba;
    /* --- NEW: in-memory copy of the FAT, and the volume's total data cluster count --- */
    uint8_t* fat;
    uint32_t data_cluster_count;
};

// ...

static uint32_t total_data_clusters(FAT12FS* fs)
{
    uint32_t total_sectors = fs->bs.bpb.total_sectors_16
        ? fs->bs.bpb.total_sectors_16
        : fs->bs.bpb.total_sectors_32;

    uint32_t data_sectors = total_sectors - fs->first_data_lba;

    return data_sectors / fs->bs.bpb.sectors_per_cluster;
}

FAT12FS* fat12_mount(BlockDevice* device)
{
    FAT12FS* fs = malloc(sizeof(FAT12FS));
    fs->device = device;
    fs->bs = read_boot_sector(device);
    fs->fat_lba = fs->bs.bpb.reserved_sector_count;
    fs->fat_sectors = fs->bs.bpb.num_fats * fs->bs.bpb.fat_size_16;
    fs->root_dir_lba = fs->fat_lba + fs->fat_sectors;
    fs->root_dir_sectors = ((fs->bs.bpb.root_entry_count *
    ↪  sizeof(DirectoryEntry))
        + (fs->bs.bpb.bytes_per_sector - 1))
        / fs->bs.bpb.bytes_per_sector;
    fs->first_data_lba = fs->root_dir_lba + fs->root_dir_sectors;
    /* --- NEW: total data cluster count, cached once so chain walks never recompute it --- */
    fs->data_cluster_count = total_data_clusters(fs);

    DBG_PRINT("[ fat12 ] FAT start LBA: %u\n", fs->fat_lba);
    DBG_PRINT("[ fat12 ] FAT size (sectors): %u\n", fs->fat_sectors);
    DBG_PRINT("[ fat12 ] Root dir start LBA: %u\n", fs->root_dir_lba);
    DBG_PRINT("[ fat12 ] Root dir size (sectors): %u\n", fs->root_dir_sectors);
    DBG_PRINT("[ fat12 ] First data LBA: %u\n", fs->first_data_lba);
    /* --- NEW --- */
```

```
48      DBG_PRINT("[ fat12 ] Data cluster count: %u\n", fs->data_cluster_count);
49
50      /* --- NEW: read the whole FAT into memory once --- */
51      uint32_t fat_bytes = fs->bs.bpb.fat_size_16 * fs->bs.bpb.bytes_per_sector;
52      fs->fat = (uint8_t*)malloc(fat_bytes);
53      block_device_read(device, fs->fat_lba, fs->bs.bpb.fat_size_16, fs->fat);
54
55      return fs;
56  }
```

And free it in `fat12_umount`:

src/fat12.c

```
1   void fat12_umount(FAT12FS* fs)
2   {
3       free(fs->fat);
4       free(fs);
5   }
```

The FAT is now available to every function. Notice we only ever read the FAT at `fat_lba` — never the copies after it. FAT volumes usually keep more than one copy of the table (`num_fats` in the BPB, almost always 2): a primary and a backup, written identically and kept in sync by the driver on every update. It's pure redundancy against a bad sector — if a media defect corrupts one copy, the other survives at a different physical location. Since both copies always hold the same data, we only ever need to read one.

The FAT is in memory now. Let's write the function that reads a single entry from it — the same `offset = cluster + cluster/2` rule we just did by hand:

src/fat12.c

```c
static uint16_t find_next_cluster(
    FAT12FS* fs,
    uint16_t cluster
)
{
    uint32_t offset = cluster + (cluster / 2);

    if (cluster & 1)
    {
        /* odd cluster: shift right by 4 */
        return ((fs->fat[offset + 1] << 8) | fs->fat[offset]) >> 4;
    }
    else
    {
        /* even cluster: mask low 12 bits */
        return ((fs->fat[offset + 1] << 8) | fs->fat[offset]) & 0x0FFF;
    }
}
```

The even branch forms a little-endian `uint16_t` from two bytes and masks off the high nibble – the same math we used for cluster 6. The odd branch shifts right by 4 instead, dropping the previous entry's leftover nibble – the same math we used for clusters 5 and 7.

Locating a File

Before we can read a file, we must find it. A path like `/HELLO.TXT` names a single entry in the root directory. The filesystem must scan the directory's 32-byte entries, match the name (8.3 format), and return the entry – which carries the `first_cluster` we need to start the chain walk.

This version only searches the root directory. Adding subdirectory support here would bloat the chapter with directory traversal before we have seen the core read path in action. Chapter 5 extends this to full path walking so `/DATA/NOTES.TXT` resolves correctly through nested directories. For now, only bare filenames (`HELLO.TXT`) or root-absolute paths (`/HELLO.TXT`) work – passing a subdirectory path like `/MYDIR/FILE.TXT` returns "file not found."

Two new helpers make the lookup possible: `str_upper` uppercases a name in place, and `find_by_name_in_entries` scans a directory entry array for a case-insensitive name match.

src/fat12.c

```
static void str_upper(char* s)
{
    for (int i = 0; s[i] != '\0'; i++)
    {
        if (s[i] >= 'a' && s[i] <= 'z')
            s[i] -= 32; // ASCII: lowercase letters are 32 above uppercase
    }
}

static DirectoryEntry* find_by_name_in_entries(
    DirectoryEntry* entries,
    uint32_t count,
    const char* name
)
{
    uint32_t i = 0;
    DirectoryEntry* raw;
    while (next_active_entry(entries, count, &i, &raw))
    {
        if (raw->attr == FAT12_ATTR_VOLUME_ID) continue;

        char entry_name[13];
        decode_8_3_name(raw, entry_name);
        str_upper(entry_name);

        if (strcmp(entry_name, name) == 0)
            return raw;
    }
    return NULL;
}
```

We already have `resolve_path` from Chapter 3 — it only recognized the root path `"/"` so far, returning `-1` for everything else. Now we extend it to also look up a regular file by name:

src/fat12.c

```c
static int resolve_path(
    FAT12FS* fs,
    const char* path,
    DirectoryEntry* out
)
{
    const char* p = path;
    if (*p == '/') p++;

    if (*p == '\0')
    {
        out->attr = FAT12_ATTR_DIRECTORY;
        out->first_cluster = ROOT_DIR_CLUSTER;
        return 0;
    }

    char name[13];
    strncpy(name, p, 12);
    name[12] = '\0';
    str_upper(name);

    uint32_t count;
    DirectoryEntry* entries = read_root_directory(fs, &count);
    if (entries == NULL) return -1;

    DirectoryEntry* raw = find_by_name_in_entries(entries, count, name);
    if (raw != NULL)
    {
        memcpy(out, raw, sizeof(DirectoryEntry));
    }

    free(entries);
    return raw != NULL ? 0 : -1;
}
```

The root-path check at the top is untouched. The only change is what happens after it — and that's where the `fs` parameter, unused until now, finally does real work.

Chapter 3 returned `-1` for anything past the root check. Now we copy the remaining path into `name`, uppercase it, and load the root directory into memory via `read_root_directory`. We walk each entry with `next_active_entry`, which skips deleted and LFN entries (long filenames — see the LFN Support appendix if you ever spot one of these fragments in a hexdump and want to know what its bytes mean); `find_by_name_in_entries` additionally skips volume labels. On a match, the entry is copied into `out` and the function returns 0; otherwise -1.

> **Why normalize case?** FAT12 names are stored uppercase-only on disk — lowercase is not a legal byte value in the raw `DIR_Name` field. The spec still requires every search operation — find, open, create, delete, rename — to treat names case-insensitively, so we uppercase both sides before comparing: the caller's input and the name decoded from the directory entry. `HELLO.TXT`, `Hello.txt`, and `hello.txt` all resolve to the same on-disk file, since only the uppercase form can ever actually be stored. Case-differing variants can never coexist as separate entries.

Reading a Cluster Chain

A file's data lives scattered across the disk, linked together by FAT entries. To read it, we start at the first cluster, follow the chain until we hit the end-of-chain marker, and read each cluster along the way, translating cluster numbers to disk positions. Both reading file data and loading subdirectory contents follow this same pattern.

This pattern appears in every chapter from here forward, so extract it into a shared static helper. Data clusters start at index 2, as we saw earlier — let's finally give that number a name. We already cached the volume's data cluster count on mount, so the bounds check is a single comparison against `fs->data_cluster_count`:

src/fat12.c

```
#define CLUSTER_FIRST           0x002

// ...

static uint8_t* read_cluster_chain(
    FAT12FS* fs,
    uint16_t first_cluster,
    uint32_t* out_bytes
)
{
    uint32_t cluster_bytes = fs->bs.bpb.sectors_per_cluster *
    ↪  fs->bs.bpb.bytes_per_sector;
    uint32_t max_bytes = 0;
```

```
13      uint32_t capacity = 0;
14      uint8_t* all = NULL;
15      uint16_t cluster = first_cluster;
16
17      while (cluster >= CLUSTER_FIRST && (uint32_t)(cluster - 2) <
        ↪  fs->data_cluster_count)
18      {
19          uint32_t lba = data_cluster_to_lba(fs, cluster);
20          DBG_PRINT("[ fat12 ] read: cluster %u\n", cluster);
21
22          uint8_t* buf = (uint8_t*)malloc(cluster_bytes);
23         block_device_read(fs->device, lba, fs->bs.bpb.sectors_per_cluster, buf);
24
25          uint32_t needed = max_bytes + cluster_bytes;
26          if (needed > capacity)
27          {
28              capacity = capacity ? capacity * 2 : 4096;
29              if (capacity < needed) capacity = needed;
30              all = (uint8_t*)realloc(all, capacity);
31          }
32
33          memcpy(all + max_bytes, buf, cluster_bytes);
34          max_bytes += cluster_bytes;
35          free(buf);
36
37          cluster = find_next_cluster(fs, cluster);
38      }
39
40      *out_bytes = max_bytes;
41      return all;
42  }
```

The function walks the chain cluster by cluster, growing a single buffer to hold the whole file. On the first cluster it allocates a 4096-byte buffer; whenever another cluster would overflow it, `realloc` doubles the buffer — or grows straight to the needed size when one more doubling is not enough. Because the buffer grows instead of being a fixed array, there is no read-size ceiling built in: a file that occupies every data cluster on the volume still loads completely, and the read path depends on nothing beyond `malloc`, `realloc`, `memcpy`, and `block_device_read`. Every cluster read triggers the LBA mapping from `data_cluster_to_lba` followed by the block device read.

The end-of-chain marker (`0xFFF`, or any value `>= 0xFF8`) is nowhere near a valid data cluster on any real volume, so it fails the `fs->data_cluster_count` check and stops the traversal — same for the bad-cluster marker `0xFF7` and the unused `0xFF0-0xFF6` range. One bounds check against the volume's own

cluster count catches all of them, plus outright corruption: a directory entry that claims a file starts at cluster 9999 on a 4000-cluster volume won't walk past the data region either.

Putting It All Together

We have all the building blocks – cluster chains, directory entries, the 12-bit packing, and how to locate a file. Now we put it all together to read a file's contents. Callers should not need to know any of that – the internal navigation and filesystem concepts should stay hidden behind a simple open-read-close interface. This pattern mirrors the generic file system API that mainstream operating systems expose – `open`, `read`, `close` – so it feels familiar to anyone who's used a filesystem API before.

`include/fat12.h`

```
typedef struct File File;

File* fat12_open(FAT12FS* fs, const char* path, const char* mode);
uint32_t fat12_read(File* file, void* buffer, uint32_t size);
void fat12_close(File* file);
```

The `File` struct fields are an implementation detail, so the struct definition lives in `fat12.c`, not the header. Callers only see the opaque `typedef struct File File;` declaration.

`src/fat12.c`

```
struct File
{
    FAT12FS* fs;
    uint8_t* data;
    uint32_t size;
    uint32_t position;
};

File* fat12_open(FAT12FS* fs, const char* path, const char* mode)
{
    if (mode == NULL || strcmp(mode, "r") != 0)
        return NULL;

    DirectoryEntry resolved;
    if (resolve_path(fs, path, &resolved) != 0)
        return NULL;
    if (resolved.attr & FAT12_ATTR_DIRECTORY)
        return NULL;

    uint32_t chain_bytes;
    uint8_t* data = NULL;

    if (resolved.file_size > 0)
    {
        data = read_cluster_chain(fs, resolved.first_cluster, &chain_bytes);
        if (data == NULL)
        {
            return NULL;
        }
    }

    File* file = (File*)malloc(sizeof(File));
    file->fs = fs;
    file->data = data;
    file->size = resolved.file_size;
    file->position = 0;
    return file;
}

uint32_t fat12_read(File* file, void* buffer, uint32_t size)
{
    if (file->position >= file->size) return 0;

    uint32_t remaining = file->size - file->position;
    if (size < remaining) remaining = size;

    memcpy(buffer, file->data + file->position, remaining);
    file->position += remaining;

    return remaining;
```

```
}

void fat12_close(File* file)
{
    free(file->data);
    free(file);
}
```

`fat12_open` accepts a `mode` parameter – `"r"` for reading, which is all we support now. We guard against any mode other than `"r"`, returning `NULL` if the caller asks for something else. The `mode` parameter is here so the API is ready when Chapter 6 adds write support. The function calls `resolve_path` to locate the entry, checks that it is not a directory (opening a directory as a file would read garbage), then calls `read_cluster_chain` to pull the entire file into memory. Note the two sizes: `chain_bytes` is the size of the full cluster-aligned allocation (always a multiple of the cluster size), while `file_size` from the directory entry is the actual bytes the file uses. `read` stops at `file_size`, ignoring any data beyond it. An empty file (`file_size == 0`) skips the chain walk entirely – `data` stays `NULL` and `fat12_read` returns 0 immediately since `position >= size`.

`fat12_read` copies up to `size` bytes from the pre-loaded data buffer into `buffer` and returns the number of bytes actually read (less than `size` means end of file). No cluster walking, no disk reads – `read_cluster_chain` already did the work when the file was opened.

`fat12_close` frees the data buffer and the `File` struct.

> **Production Note:** `fat12_read` copies from a pre-loaded buffer – our `fat12_open` loads the entire file into memory via `read_cluster_chain`, so `fat12_read` is just a `memcpy` with no per-call disk I/O. We chose this approach to keep things simple. A production driver would use a sliding window: read clusters on demand and evict old ones. That approach needs more bookkeeping but handles large files without exhausting memory.

The cat Command

The `cat` command becomes an open-loop-close:

platform/cli/main.c

```
static int cmd_cat(FAT12FS *fs, int argc, char *argv[])
{
    if (argc < 3)
    {
        fprintf(stderr, "usage: fat12-cli cat <path>\n");
        return 1;
    }

    const char *path = argv[2];
    File *file = fat12_open(fs, path, "r");
    if (file == NULL)
    {
        fprintf(stderr, "error: file not found\n");
        return 1;
    }

    uint8_t buf[512];
    uint32_t bytes;
    while ((bytes = fat12_read(file, buf, sizeof(buf))) > 0)
        fwrite(buf, 1, bytes, stdout);

    fat12_close(file);
    return 0;
}
```

Then add this branch after the `ls` branch in the dispatch chain inside `main()`:

platform/cli/main.c

```
    else if (strcmp(command, "cat") == 0)
        ret = cmd_cat(fs, argc, argv);
```

We already have `BIGFILE.TXT` sitting on disk from earlier in this chapter, where we read it by hand with `xxd` and a calculator. Let's see our own code do the same work. Rebuild with `DEBUG=1` to see the full trace:

```
DEBUG=1 ./build.sh && ./fat12-cli cat /BIGFILE.TXT
```

A three-cluster file produces output like this — the mount-time trace (boot sector, FAT, and root directory loads) prints first, then the per-cluster reads:

```
[ block_device ] read 1 sector(s) starting at LBA 0
[ file         ] read 512 bytes at 0x0
[ fat12 ] FAT start LBA: 1
[ fat12 ] FAT size (sectors): 24
[ fat12 ] Root dir start LBA: 25
[ fat12 ] Root dir size (sectors): 32
[ fat12 ] First data LBA: 57
[ fat12 ] Data cluster count: 4067
[ block_device ] read 12 sector(s) starting at LBA 1
[ file         ] read 6144 bytes at 0x200
[ block_device ] read 32 sector(s) starting at LBA 25
[ file         ] read 16384 bytes at 0x3200
[ fat12 ] cluster 5 -> LBA 63
[ fat12 ] read: cluster 5
[ block_device ] read 2 sector(s) starting at LBA 63
[ fat12 ] cluster 6 -> LBA 65
[ fat12 ] read: cluster 6
[ block_device ] read 2 sector(s) starting at LBA 65
[ fat12 ] cluster 7 -> LBA 67
[ fat12 ] read: cluster 7
[ block_device ] read 2 sector(s) starting at LBA 67
AAAAAAAAAA... (1024 A's)
BBBBBBBBBB... (1024 B's)
CCCCCCCCCC... (1024 C's)
```

Each cluster logs two steps: the LBA mapping first, then the block read. Structure resolution and data transfer are separated — for each of the three clusters we see exactly where on disk it lives and how it gets read into memory.

What's Next

`first_cluster` is not a mystery anymore — it is the first stop on a chain we now know how to walk, cluster by cluster, all the way to the last byte of the file. We can read files — but only from the root directory. Chapter 5 fixes that: we retrofit path resolution so every operation we have built so far works at any depth.

Chapter 5: Full Path Walking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Directories: Files in Disguise

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The Path Walking Algorithm

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Retrofitting Path Walking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

What's Next

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Part II: Writing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Chapter 6: Writing Files

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Manipulating FAT Entries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Cluster Allocation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Writing the Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Directory Entry Creation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Putting It All Together

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`include/fat12.h`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The Write CLI Command

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/cli/main.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/cli/main.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

What's Next

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Chapter 7: Creating Directories

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

What Are `.` and `..`?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Extending a Directory Chain

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Finding or Extending a Slot

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Directory Manipulation Helpers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Updating `create_dir_entry`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Updating `fat12_close`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The `fat12_mkdir` Function

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Helper: Creating Directory Contents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`include/fat12.h`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The `mkdir` CLI Command

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/cli/main.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/cli/main.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Bonus: Copy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

What's Next

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Chapter 8: Deleting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Freeing the cluster chain

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Marking a directory entry deleted

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

When is a directory empty?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Putting it all together

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`include/fat12.h`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The `rm` CLI Command

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/cli/main.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/cli/main.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Verifying Deletion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

What's Next

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Chapter 9: Moving (and Renaming)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The Two Cases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Updating `.` and `..`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The `fat12_move` Function

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`include/fat12.h`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The `mv` CLI Command

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/cli/main.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/cli/main.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Computing the Geometry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The Boot Sector

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The File Allocation Tables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The Root Directory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`src/fat12.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The Data Region

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The `format` CLI Command

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/cli/main.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/cli/main.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/cli/main.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

What's Next

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Chapter 11: Into the Kernel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The Freestanding Environment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/kernel/lib.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Debug Logging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The ATA Block Device Driver

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/kernel/ata_block_device.h`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/kernel/ata_block_device.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

VGA Text Mode

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/kernel/vga.h`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/kernel/vga.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The Kernel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/kernel/main.c`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Booting the Kernel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The Kernel Entry Point

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`platform/kernel/boot.s`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Linking at 1 MB

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`kernel.ld`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Building the Kernel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`build_kernel.sh`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Running in QEMU

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

`run_kernel.sh`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

What's Next

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Appendixes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Appendix: Boot Sector Reference Fields

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Hidden Sectors (Partitioning)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

CHS (Cylinder-Head-Sector) Addressing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Media Descriptor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Extended BPB — Additional Fields

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

OEM Name (Original Equipment Manufacturer)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Appendix: FAT Family Reference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

FAT Entry Widths and Values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Root directory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

BPB

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Volume Size Limits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Appendix: FAT12 Cheat Sheet

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Disk Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Position Formulas

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Boot Sector (512 bytes)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Jump and OEM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

BIOS Parameter Block

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Extended BPB

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Boot Code and Signature

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Total struct size: 512 bytes (one sector)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Directory Entry (32 bytes)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Special `name[0]` values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Attribute byte bitfield

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

FAT12 Entry (12 bits)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Byte layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Special values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Reserved entries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Timestamp Format

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Time word

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Date word

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Example

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

8.3 Filename Format

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Quick Reference: Cluster Math

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Appendix: Long Filename Support (LFN / VFAT)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Overview

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

LFN Entry Structure (32 bytes)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Sequence Number

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Character Encoding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Checksum Algorithm

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

8.3 Alias Generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Reading LFN Entries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Writing LFN Entries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Deleting LFN Entries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Moving and Renaming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Worked Example

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Step 1: Generate 8.3 alias

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Step 2: Compute checksum

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Step 3: Split into chunks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Step 4: Write to disk

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

The Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

What's Next

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.

Thank You

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fat12.