FAT12

Understanding and Implementing the Classic File System in C_

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 2026-06-05

To my mother – for believing in me. To my father – for always being there, whenever I needed him. To my fiancée – for her endless love.

Contents

Preface **1**
Why FAT12? 1
Who This Book Is For 1
What You Will Build 1
What You Will Learn 1
A Note on Style 2
How to Read This Book 2
About the Author 3
Feedback 3

Prerequisites **4**
Operating System 4
C Compiler and Dependencies 4
Editor and Hex Editor 4
Getting the Code (Optional) 4

Part I: Reading **6**

Chapter 1: Hello, Disk! **7**
What Is FAT12? 7
Creating a Virtual Disk 8
Sectors and LBA 9
A Tale of Four Regions 10
Project Layout 10
A Portable Foundation: The BlockDevice 11
Reading Sector 0 17
Tracing the Translation 19
Production Notes 20
What Is Next 20

Chapter 2: The Blueprint (BPB) **22**

Boot Sector Structure 22
Mapping Bytes to Structs 29
The FAT12 Library 30
The Info Command 33
Debugging 34
Production Notes 35
What Is Next 35

Chapter 3: The Root Directory 36
What Is the Root Directory? 36
The DirectoryEntry Struct 38
Decoding Timestamps 39
Where Is the Root Directory? 40
Geometry Functions 41
Loading the Root Directory 42
Formatting an 8.3 Name 43
The ls Command 46
Updating the Header 49
Building 51
Verification: Create Real Files 51
Debugging with a Hexdump 52
Production Notes 53
What Is Next 54

Chapter 4: Reading Files 55
What the FAT Actually Is 55
Where the FAT Lives 55
Data Region Geometry 55
Reading an Entry 55
Loading the FAT 55
Reading a Cluster Chain 55
File API 56
The cat Command 56
Updating the Header 56
Building 56
Verification: Create a Real File 56
Hexdump Verification 56
Multi-Cluster File 56
Production Notes 57

FAT Family: FAT16 and FAT32 57
What Is Next 57

Part II: Writing **58**

Chapter 5: Full Path Walking **59**
Subdirectories in FAT12 59
Reading a Subdirectory from a Cluster Chain 59
Splitting a Path into Tokens 59
Finding an Entry by Name 59
The Centerpiece: `fat12_resolve_path` 59
Wiring into `fat12_opendir` 59
Wiring into `fat12_open` 60
Updating the CLI Commands 60
Building 60
Verification: Nested Directories 60
Production Notes 60
The FAT Family: FAT32 Root Cluster 60
What Is Next 60

Chapter 6: Writing Files **62**
The Write Path 62
Writing FAT Entries 62
Finding Free Clusters 62
Allocating a Chain 62
Writing Data Clusters 62
Syncing the FAT 62
Loading the Parent Directory 63
Finding a Free Slot 63
Formatting 8.3 Names 63
Creating a Directory Entry 63
Updating the File Struct 64
Extending fat12_open 64
Adding fat12_write 64
Updating fat12_close 64
Updating the Header 64
Updating the cat Command 64
The Write CLI Command 64
Building 65

Verification 65
Production Notes 65
FAT Family: Writing Across FAT Variants 65
What Is Next 65

Chapter 7: Creating Directories 66
What We Need to Write 66
Splitting a Path into Parent and Name 66
The `fat12_mkdir` Function 66
Updating the Header 66
The `mkdir` CLI Command 66
Building 66
Verification 67
Production Notes 67
FAT Family: Directories Across FAT Variants 67
What Is Next 67

Chapter 8: Deleting Files and Directories 68
The Deleted Entry Marker 68
Freeing a Cluster Chain 68
The `fat12_remove` Function 68
The `fat12_rmdir` Function 68
Updating the Header 68
The `rm` and `rmdir` CLI Commands 68
Building 69
Verification 69
Production Notes 69
FAT Family: Deleting Across FAT Variants 69
What Is Next 69

Chapter 9: Renaming and Moving 70
The Two Cases 70
The Remaining Helper 70
The `fat12_rename` Function 70
Updating the Header 72
The `mv` CLI Command 72
Building 72
Verification 72
Production Notes 72

FAT Family: Rename and Move Across FAT Variants 72
What Is Next 72

Part III: Advanced **74**

Chapter 10: Formatting a Disk **75**
What We Are Building 75
Computing the Layout 75
Updating the Header 76
The `format` CLI Command 76
Building 76
Verification 76
Production Notes 76
FAT Family: Formatting Across FAT Variants 76
What Is Next 76

Chapter 11: Into the Kernel **78**
What Changes, What Does Not 78
The Freestanding Environment 78
ATA PIO Block Device 78
The Kernel Entry Point 79
Linking at 1 MB 79
kernel_main 79
Building 79
Running in QEMU 79
Debugging Tips 79
Production Notes 79
FAT Family: Multiplatform Payoff 80
What Is Next 80

Conclusion **81**
The Architecture 81
What Is Next 81
Thank You 81

Preface

No journaling. No extended attributes. No access control lists. Just a file allocation table, directory entries, and data clusters. That is exactly why we start here.

Why FAT12?

FAT12 is not a historical curiosity – it is still the standard filesystem for floppy disk images in QEMU and VirtualBox, and variants of FAT are used by EFI System Partitions and embedded firmware because the format is simple and universally supported. And here is the real payoff: once you understand FAT12, you already understand most of FAT16 and FAT32. The FAT family shares the same overall design – a BIOS Parameter Block (BPB), directory-entry structures, and cluster-chain allocation through a File Allocation Table. The major differences are mainly wider fields, larger address spaces, and some structural extensions in FAT32. The whole family shares a blueprint.

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 have skimmed filesystem tutorials before, but this time you want to actually *write* one, from the boot sector up to a filesystem driver running in QEMU.

What You Will Build

By the end of this book we will have a complete FAT12 library that reads, writes, deletes, creates directories, renames, and formats – plus a CLI tool to exercise every operation on our host, and a 32-bit i386 kernel that boots in QEMU and lists files. The FAT12 code is pure C with zero architecture dependencies. It talks to storage through five operations. Implement those five for our hardware and the library works anywhere!

What You Will Learn

The book is split into three parts that build on each other:

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 every byte of the boot sector into C structs — the geometry calculations that power the entire filesystem. Walk the root directory, navigate the File Allocation Table and its famously awkward 12-bit entries, then read a file from disk to screen, start to finish.

Part II: Writing — Resolve full paths through nested directories. Write and allocate files. Create directories. Delete. Rename. Make the filesystem do work instead of just answering questions.

Part III: Advanced — Format a disk from scratch — write a boot sector, twin FATs, and root directory by hand. Then take the same library code and boot it in a 32-bit kernel over ATA PIO, printing to VGA text mode in QEMU. Filesystem code that runs on bare metal.

Each chapter closes with debugging tips and a *Production* callout that flags what a real-world driver would do differently — consider them cheat codes for later.

A Note on Style

This is not production code — and that is by design. Defensive checks, error recovery, long filename support, field-by-field validation — all skipped on purpose so the FAT12 logic stays front and center. Every I/O call *can* fail; we ignore those return values so we see the filesystem, not the error handling noise. The code works — it handles real FAT12 volumes and has been tested on disk images.

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. The debugging tips are not

optional; they are the point.

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 in your Leanpub library. My goal is to make this book both valuable and genuinely fun to read.

Prerequisites

Before writing code, we need a working environment. Here is exactly what that looks like.

Operating System

All commands in this book target Linux. If you are not on Linux, use Docker or a Linux VM.

C Compiler and Dependencies

We need GCC[1] and `dosfstools` (which provides `mkfs.fat`). On Debian-based Linux: `sudo apt install build-essential dosfstools`.

Editor and Hex Editor

We will read and edit code throughout the book and occasionally inspect raw disk data. I use VS Code[2] with the Hex Editor extension[3], but any editor and hex viewer works.

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[4] is available:

[1] https://gcc.gnu.org/
[2] https://code.visualstudio.com/
[3] https://marketplace.visualstudio.com/items?itemName=ms-vscode.hexeditor
[4] https://github.com/SoftwareFreak1/FAT12

```
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.

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. By the end of this chapter we will have built that ring: a real FAT12 volume, the code to talk to it, and a program that pries open its first sector and reads a field.

Six source files and a build script are all we need: an abstract `BlockDevice` interface so we can swap disk implementations without touching filesystem logic, a file-backed version of that interface for our dev machine, a debug macro we will lean on through every chapter, and a small `main.c` that glues them together into a CLI tool. By the end of this book those same files will have grown into a complete FAT12 library, a CLI tool, and a bootable kernel. But that is the future. Right now we focus on the first step: getting those bytes off the disk.

What Is FAT12?

FAT12 is the grandparent of the FAT family. Microsoft introduced FAT in 1977 for Standalone Disk BASIC, and FAT12 became the filesystem of early MS-DOS floppy disks in the early 1980s – the era when floppy disks were how you shipped software. Its name comes from the File Allocation Table, where each entry is exactly 12 bits wide – a famously awkward packed size that does not align to byte boundaries, requiring the cross-byte boundary unpacking logic we will feel firsthand in Chapter 3. Small entries keep overhead low, but they also limit the volume to roughly 4,084 usable clusters – so the maximum size depends on the cluster size, often up to around 32 MiB in common configurations. That is why FAT16 and FAT32 later took over for larger disks.

Why start here instead of jumping straight to something modern? Because FAT12 is the perfect teacher:

- **Small volumes** – we work with 1–4 MiB disk images. Every structure fits in a handful of sectors we can hexdump and understand end to end.
- **Simple allocation** – clusters chain through a single table. No extents, no block groups, no B-trees. Just a linked list wearing a clever disguise.

- **Flat root directory** – fixed size, right after the FATs. No tree walking to find `/`.
- **Still everywhere** – FAT12 is the most commonly used filesystem for floppy disk images in QEMU, VirtualBox, and similar emulators. The entire FAT family – from FAT12 through FAT32 – is natively supported by UEFI and appears in embedded firmware. Learn it once and you will recognize it everywhere.

Endianness gotcha: FAT12 stores all multi-byte values – cluster pointers, sector counts, everything – in **little-endian** format. Since x86 is also little-endian, our code reads them directly without byte-swapping. That is convenient, but not portable: if you ever adapt this code for a big-endian architecture (a PowerPC firmware, say), every multi-byte read needs a swap. We flag it now so it does not bite us later.

Creating a Virtual Disk

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

Create a 4 MiB FAT12 disk image with two commands:

```
dd if=/dev/zero of=disk.img bs=1M count=4         # create empty 4 MiB file
mkfs.fat -F 12 -s 2 -S 512 -n MYDISK disk.img     # format as FAT12
```

The output should look like this (exact values may differ):

```
4+0 records in
4+0 records out
4194304 bytes (4.2 MB, 4.0 MiB) copied
mkfs.fat 4.2 (2021-01-31)
```

The `-s 2` flag sets 2 sectors per cluster (the allocation unit), and `-S 512` sets 512 bytes per sector – the standard values we will use throughout this book. The `-n MYDISK` flag writes a volume label, which we will read back from the raw bytes before this chapter ends.

Before formatting, the file is nothing but zeros:

```
00000000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figure 1. Empty disk — all zeros

After `mkfs.fat` runs, the first few dozen sectors spring to life. The boot sector appears, the FAT tables take shape, and the root directory is carved out. That is our FAT12 volume:

```
00000000  EB 3C 90 42 53 44 20 20 34 2E 34 00 02 40 01 00  .<.BSD  4.4..@..
00000010  02 00 02 00 00 F0 0C 00 20 00 20 00 00 00 00 00  ........ . ......
00000020  78 FD 03 00 00 00 29 08 1E 1E 35 4D 59 44 49 53  x.....)...5MYDIS
00000030  4B 20 20 20 20 20 46 41 54 31 32 20 20 20 FA 31  K     FAT12   .1
00000040  C0 8E D0 BC 00 7C FB 8E D8 E8 00 00 5E 83 C6 19  .....|......^...
00000050  BB 07 00 FC AC 84 C0 74 06 B4 0E CD 10 EB F5 30  .......t.......0
00000060  E4 CD 16 CD 19 0D 0A 4E 6F 6E 2D 73 79 73 74 65  .......Non-syste
00000070  6D 20 64 69 73 6B 0D 0A 50 72 65 73 73 20 61 6E  m disk..Press an
00000080  79 20 6B 65 79 20 74 6F 20 72 65 62 6F 6F 74 0D  y key to reboot.
00000090  0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figure 2. Disk after mkfs.fat — boot sector and FAT tables written

Sectors and LBA

Disks do not deal in bytes – they deal in fixed-size blocks called **sectors**. Every sector in this book is 512 bytes. No exceptions.

We address sectors by number – 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). Everything we build starts from a sector number.

A Tale of Four Regions

FAT12 volumes are organized in sequential regions starting at the beginning of the volume:

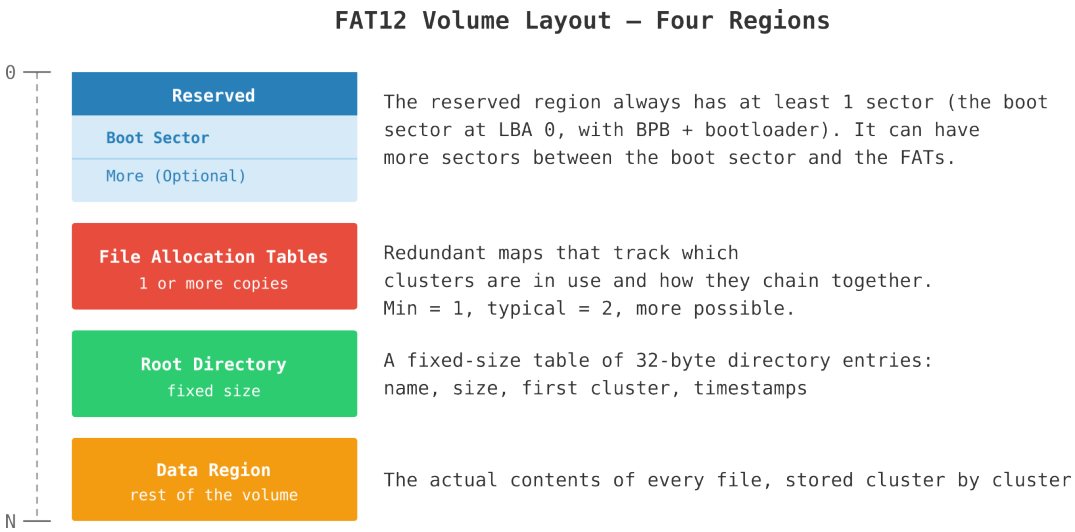

Figure 3. FAT12 volume layout – four regions

Every operation in this book – reading a file, listing a directory, deleting an entry – follows the same pattern: read the geometry once from the boot sector, then walk the FAT and directory structures to reach the data. One blueprint, many operations.

Project Layout

Let's create the directory structure that will hold everything we build across the chapters:

```
.
├── include/              # Platform-independent headers
├── platform/             # Platform-dependent code & entry points
│   ├── cli/              # CLI tool
│   └── kernel/           # Kernel (added in Ch. 11)
├── src/                  # Shared FAT12 library
└── disk.img              # Our FAT12 volume
```

The `include/` directory is the shared backbone – headers that every platform in this book uses unchanged. The `platform/` directory holds platform-dependent code and entry points – one subdirectory per platform, each implementing the `BlockDevice` interface and its own `main()` entry point. The CLI tool in `platform/cli/` uses a file-backed block device; the kernel in `platform/kernel/` (added in Chapter 11) uses ATA PIO. The library code in `src/` stays exactly the same regardless of which platform it runs on.

A Portable Foundation: The BlockDevice

Real operating systems never talk to disk hardware directly from filesystem code. Instead they define an abstract interface – a block device layer – 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. We follow the same pattern: the FAT12 library sees five operations, and swapping the platform means swapping a single `.c` file at link time.

Why bother? We could scatter `fread` and `fwrite` through every function – but then the code would be chained to files forever. When we run the same filesystem code on bare hardware in Chapter 11, every line would need rewriting. With the device abstracted away, we write the FAT logic once and it works everywhere.

The five operations:

- `block_device_read()` – read one or more sectors by LBA
- `block_device_write()` – write one or more sectors by LBA
- `block_device_close()` – release resources
- `block_device_sector_size()` – bytes per sector on disk
- `block_device_sector_count()` – total sector count on the disk

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. (Spoiler: it will, and that is a bug we will not debug here.)

The architecture we are building – platform, library, block device – looks like this from the top down:

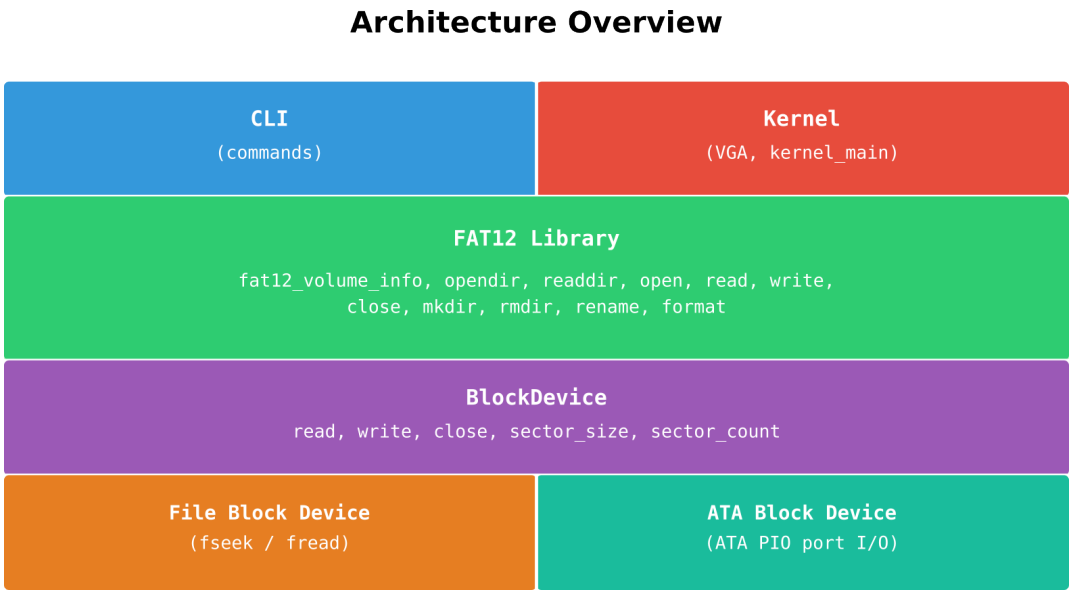

Figure 4. FAT12 library architecture

The `BlockDevice` sits at the bottom, with two implementations: the file-backed device we will build right now (left), and the ATA PIO driver we will write for the kernel in Chapter 11 (right). The FAT12 Library in the middle implements every filesystem operation – reading directories, walking cluster chains, writing data – using only the five block device calls. The platform layer on top mirrors the same split: the CLI tool uses the file-backed device during development, and the kernel uses ATA PIO on bare metal. The library never knows what is below it; the platform never cares how filesystem operations work internally. Swap the bottom layer and the same library runs on a different platform. Add a new command and the library does the heavy lifting.

Now create `include/block_device.h`:

```c
#ifndef BLOCK_DEVICE_H
#define BLOCK_DEVICE_H

#include <stdint.h>

/*
 * Opening a block device is platform-specific — each platform provides
 * its own function (e.g. file_block_device_open, ata_block_device_open)
 * with the signature appropriate for that environment.
 * This header only defines the common operations on an already-opened device.
 */

/*
 * All operations return 0 on success, -1 on error.
 * This enables production error handling patterns,
 * but the book suppresses checks for readability.
 * The focus is on FAT12, not on raw block device
 * communication or error handling.
 */

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
);

void block_device_close(BlockDevice* device);
uint32_t block_device_sector_size(BlockDevice* device);
uint64_t block_device_sector_count(BlockDevice* device);

#endif
```

Notice that 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. A real 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 simpler: one set of definitions per platform, and the linker picks whichever `.c` file we compile. The CLI tool links `file_block_device.c`, the kernel links its own ATA driver – and the library in the middle never changes.

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

While setting up infrastructure, let's add a utility we will use throughout the book. Create `include/debug.h`:

```
#ifndef DEBUG_H
#define DEBUG_H

#define DEBUG 0

#if DEBUG
    void debug_print(const char* format, ...);
    #define DBG_PRINT(...) debug_print(__VA_ARGS__)

#else
    #define DBG_PRINT(...)
#endif

#endif
```

When `DEBUG` is `1`, `DBG_PRINT` routes its arguments to `debug_print()`, which each platform implements separately. When `DEBUG` is `0`, the macro expands to nothing and the log calls vanish from the binary. Verbose tracing while we develop, clean output when we ship – one `#define` to rule them both.

The `debug_print()` function declaration lives in the shared header, but the implementation is platform-specific. For the CLI tool, create `platform/-cli/debug.c`:

```c
#include <stdio.h>
#include <stdarg.h>
#include "debug.h"

void debug_print(const char* format, ...) {
    va_list args;
    va_start(args, format);
    printf("\033[36m");
    vprintf(format, args);
    printf("\033[0m");
    va_end(args);
}
```

ANSI escape codes like `\033[36m` (cyan foreground) work on modern terminal emulators – fine for the CLI tool. When the same library runs in the kernel in Chapter 11, the `platform/kernel/` directory will provide its own `debug_print`.

Now for the concrete implementation. Since we are working with a file-backed disk on our development machine, create `file_block_device.h` inside `platform/cli/`:

```c
#ifndef FILE_BLOCK_DEVICE_H
#define FILE_BLOCK_DEVICE_H

#include "block_device.h"

BlockDevice* file_block_device_open(const char* path);

#endif
```

And `file_block_device.c` in the same directory:

```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;
};

```

```c
BlockDevice* file_block_device_open(const char* path) {
    DBG_PRINT("[ block_device ] open file %s\n", path);
    BlockDevice* device = (BlockDevice*)malloc(sizeof(BlockDevice));
    device->file = fopen(path, "r+");
    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 %"
    ↪  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%lx\n", total_bytes, 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 %"
    ↪  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%lx\n", total_bytes,
    ↪  offset);
    fseeko(device->file, offset, SEEK_SET);
    fwrite(buffer, 1, total_bytes, device->file);
    return 0;
}

void block_device_close(BlockDevice* device) {
    DBG_PRINT("[ block_device ] close\n");
    fclose(device->file);
    free(device);
}

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

```
60
61  uint64_t block_device_sector_count(BlockDevice* device) {
62      off_t saved = ftello(device->file);
63      fseeko(device->file, 0, SEEK_END);
64      off_t size = ftello(device->file);
65      fseeko(device->file, saved, SEEK_SET);
66      return (uint64_t)(size / SECTOR_SIZE);
67  }
```

Each function translates a sector-level request into a file seek and read or write. The `malloc`-ed `BlockDevice` struct holds nothing but a `FILE*`. When we move to bare metal in Chapter 11, this same header swaps file I/O for ATA PIO commands — and the library code does not change by a single line. The write path stays unused until Chapter 7, but defining it here keeps the interface complete from the start. Incomplete interfaces have a way of staying incomplete.

The code above skips error-checking on `fread`, `fwrite`, `fopen`, and `malloc` — we note them here so you know where production code should do more.

Reading Sector 0

The boot sector is the volume's identity card. Let's pull it off the disk and read one field — the volume label — straight from the raw bytes. No parsing library, no structs, just a pointer into a buffer and a format string.

Create `platform/cli/main.c`:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <stdint.h>
4   #include "block_device.h"
5   #include "file_block_device.h"
6
7   int main(void) {
8       BlockDevice* device = file_block_device_open("disk.img");
9       if (device == NULL) {
10          fprintf(stderr, "error: could not open disk.img\n");
11          return 1;
12      }
13
14      uint8_t sector[512];
15      block_device_read(device, 0, 1, sector);
16
```

```
    printf("Volume label: %.11s\n", (const char*)&sector[43]);

    block_device_close(device);
    return 0;
}
```

Create `build.sh` – a build script that discovers every `.c` file in `src/` and `platform/cli/` automatically, so it never needs to change as we add files:

```
#!/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 -iquote include -iquote src -iquote platform/cli -o fat12-cli
↪  $SRC
```

Make it executable and run it:

```
chmod +x build.sh
./build.sh
./fat12-cli
```

The `-iquote` flag restricts the `#include "..."` search to the listed directories but does *not* touch `#include <...>`. Why the distinction? Without it, a project header named `string.h` could shadow the system `<string.h>` – and debugging *that* bug is not how anyone wants to spend an afternoon. The `-I` flag makes both quote and angle-bracket includes search project directories first; `-iquote` avoids that entirely.

We should see:

```
Volume label: MYDISK
```

There it is. The volume label lives at a fixed position inside the boot sector – offset 43, exactly 11 bytes. The field is space-padded, not null-terminated: short labels like `MYDISK` are filled with trailing spaces, which is why we use `%.11s` to print exactly 11 characters. In printf-speak, precision (`%.11s`) caps the output at 11 characters – unlike field width (`%11s`), which would pad shorter

strings to 11 but happily read past a missing terminator. A plain `%s` would keep marching through memory until it found a zero byte.

We asked for sector 0 — LBA 0, the very first block on the disk — copied its 512 bytes into a buffer, and pulled out 11 characters starting at byte 43. No parsing, no validation, no intermediate data structures. Just raw bytes from disk and the `-n MYDISK` flag we passed to `mkfs.fat` staring right back at us.

This works because `mkfs.fat` always writes the extended boot signature (`0x29`) at offset 38 — the marker that says the volume label and serial number fields are present, not to be confused with the `0xAA55` end-of-sector marker at offset 510. (If you hexdump the sector, you will see `0x80` at offset 36 — that is the drive number, which comes before the signature.) The next chapter explains the full boot sector layout, including which fields are always present and which depend on that signature check.

Tracing the Translation

The `file_block_device.c` functions we just wrote contain `DBG_PRINT` calls on every read and write — but they are compiled away by default (`#define DEBUG 0` in `include/debug.h`). Set it to `1` to watch the I/O in real time:

```
#define DEBUG 1
```

Recompile and run:

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

Now every sector request leaves a trace:

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

The `[ block_device ]` line tells us the caller requested LBA 0 — one sector. The `[ file ]` line shows the file-backed driver translating that LBA into a byte

offset: sector 0 starts at offset `0x0` in the file, sector 1 at `0x200` (512), sector 2 at `0x400`, and so on. Every LBA is just `lba × 512` bytes into the disk image.

This is the bottom layer of a three-level translation we will build across the book. Starting in Chapter 3, the FAT12 library translates **cluster numbers** to LBAs — and we will add `DBG_PRINT` calls there too, making every step from cluster □ LBA □ file offset visible. The `[ file ]` line you see above is the final step in that chain.

Set `DEBUG` back to `0` for now — the noise is not helpful until we have cluster chains to trace in Chapter 4.

Production Notes

- **I/O can fail.** Our `file_block_device` wraps a `FILE*`, but `read/write` on the device handle can still fail — disk full, hardware error, cable unplugged. Production code checks every return value from the five device operations.
- **`malloc` can fail.** The `BlockDevice` struct is heap-allocated. On a resource-constrained system a `malloc` failure would return `NULL`, and the caller should propagate the error. Our sample code assumes allocation succeeds.
- **The extended boot signature check is skipped.** We read the volume label at offset 43 unconditionally, relying on `mkfs.fat` always writing the extended boot signature `0x29` at offset 38. Production code on foreign-formatted media checks `boot_signature == 0x29` before trusting the label, volume ID, and filesystem-type fields.

What Is Next

We now have six files that add up to a program that pulled one field off a real FAT12 disk. The block device abstraction is in place, and we have read our first raw byte from the boot sector. Not bad for a first chapter.

In Chapter 2 we decode every field of the boot sector — the jump instruction, the OEM name, the full BPB, and the signature — and map them into real C structs. Once the geometry is parsed, a proper `info` command will print the volume's complete identity. Every computation in the later chapters — cluster chains, directory walking, file reads — flows from those numbers.

- **Debugging:** `xxd -l 64 disk.img` — hexdump the first 64 bytes of the boot sector. Spot the volume label at bytes 43–53.

Chapter 2: The Blueprint (BPB)

Chapter 1 ended with us reading the volume label from raw bytes. One field, pulled straight off the disk with no structs, no parsing – just a pointer and a format string. Now we build the full decoder: packed structs that mirror every byte of the boot sector, a function that reads it all off the disk, and an `info` command that prints every field.

We create three new files – a packed layout header that maps every byte of the boot sector to a C struct, a public API header for the library, and the library implementation that reads it all from disk – then wire an `info` command into the CLI tool.

Boot Sector Structure

The boot sector itself can be split into several parts:

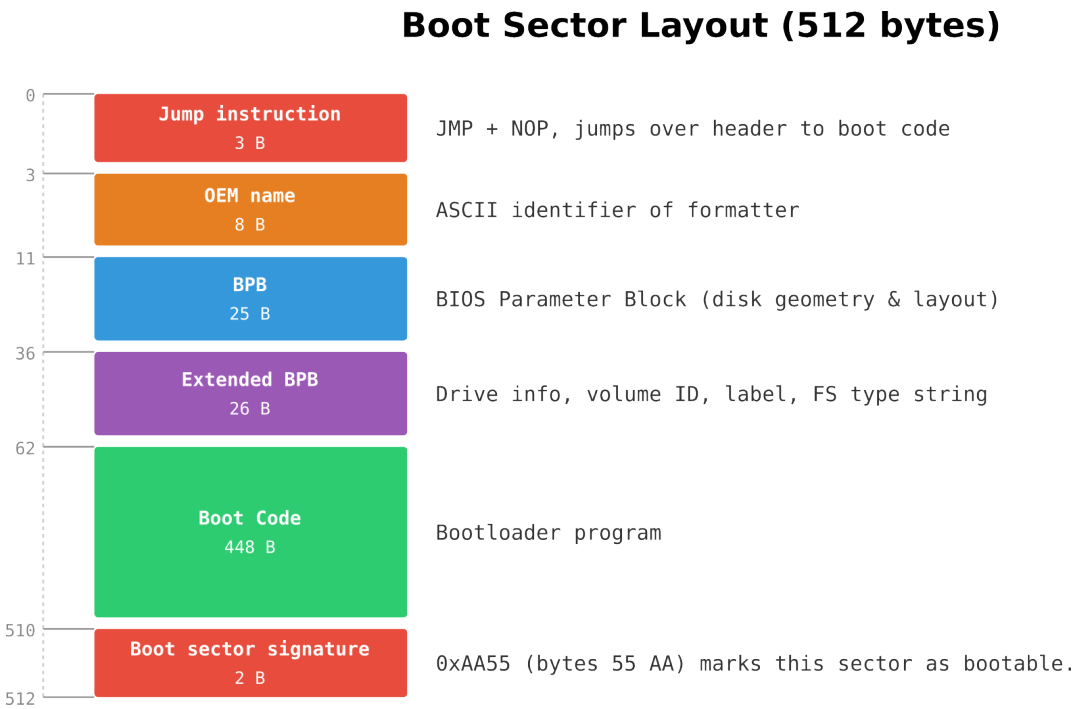

Figure 5. Boot sector layout

Bootloader and Jump Instruction

The BIOS reads the first 512 bytes of each storage medium to find a bootable device. When we format a disk with FAT12, that same sector serves two masters: it holds filesystem metadata and, optionally, bootloader code. To keep them from colliding, the first three bytes are reserved for a jump instruction that skips past the metadata and lands in the bootloader area.

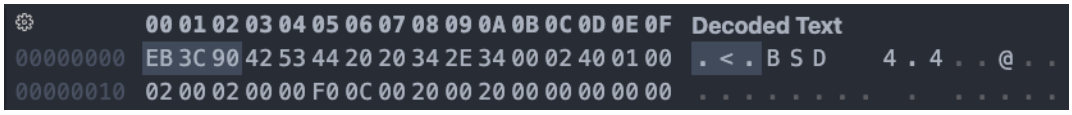

Figure 6. Jump instruction in hex editor

`EB 3C` is an x86 `JMP SHORT` instruction. `EB` is the opcode for a short jump, and `3C` (60 in decimal) is the offset from the address of the next instruction. Since the jump instruction is 2 bytes long, the next instruction would be at offset 2, so the jump target is `2 + 3C = 3E`. That offset lands us at byte 62 — right where the boot code area begins.

The third byte, `90`, is `NOP` (no operation). The jump has already skipped past it by the time execution reaches this byte, so the NOP is never executed. It is a placeholder to fill the reserved 3-byte slot.

When we look at offset `3E` in a hex editor, we can see the boot code — the actual program that would run if we booted from this disk:

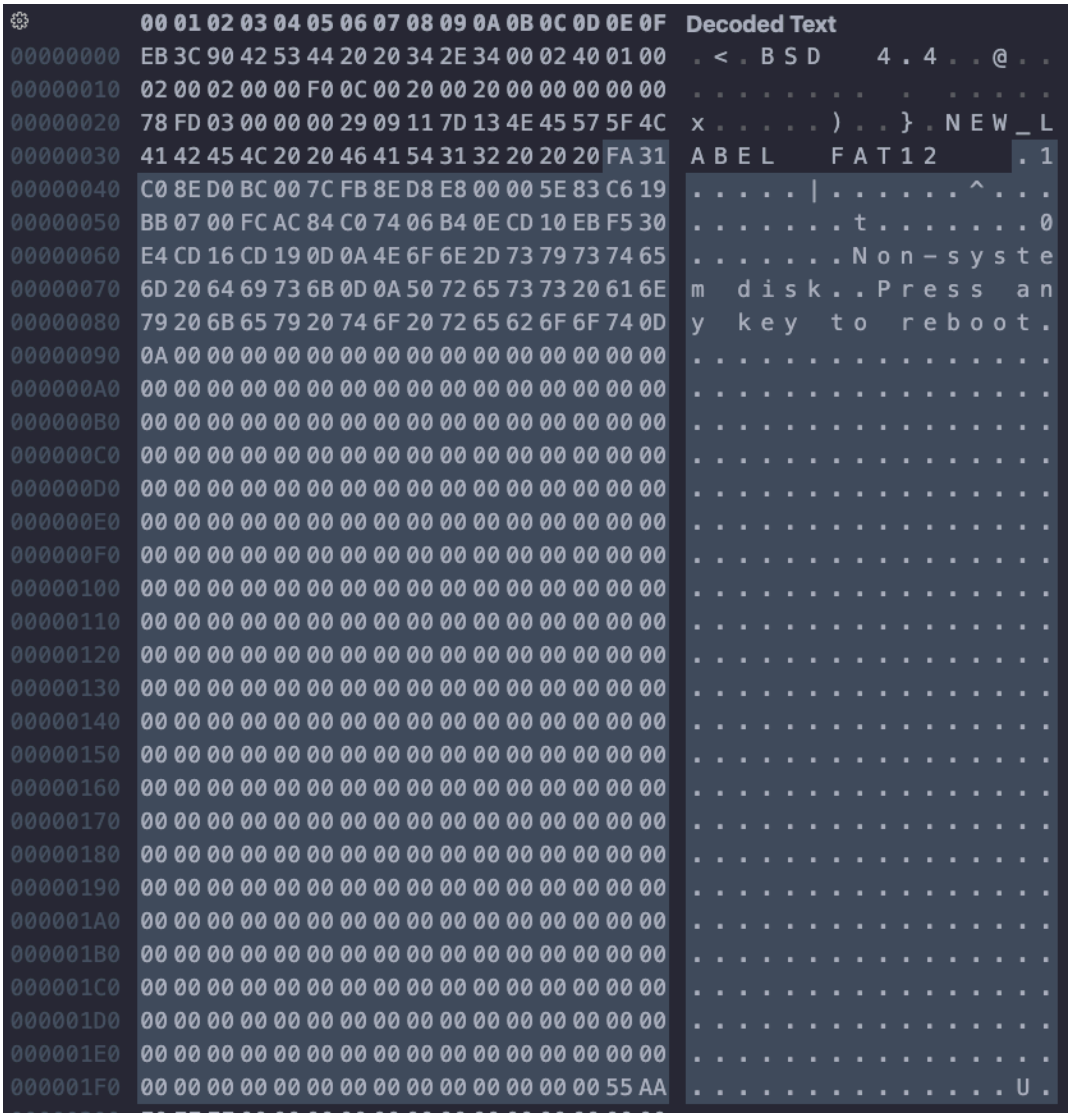

Figure 7. Bootloader code in hex editor

The boot code generated by `mkfs.fat` prints a message like "Non-system disk / Press any key to reboot" and waits for a keypress. We could replace it with a real bootloader, but that is beyond the scope of this book.

OEM Name

Bytes 3–10 are an 8-byte ASCII string identifying the tool that formatted the volume. `mkfs.fat` writes `mkfs.fat`, other tools write their own. It has no functional purpose.

BIOS Parameter Block (BPB)

Bytes 11–35 contain the BPB, a fixed-layout table that describes the disk geometry and filesystem structure. Every field is stored at a known offset with a known size:

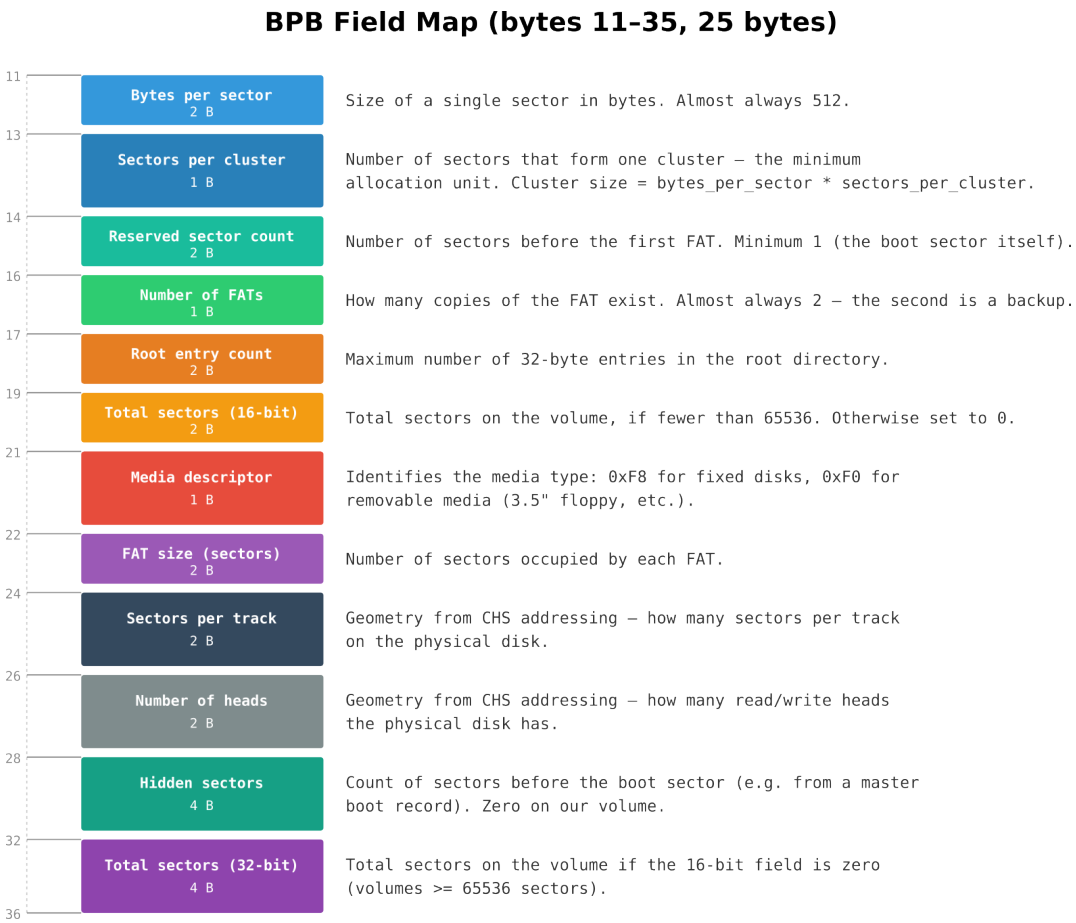

Figure 8. BPB field map

Note the two total sector fields. If the volume has fewer than 65536 sectors the 16-bit field at offset 19 holds the count and the 32-bit field at offset 32 is zero. If it exceeds 65535 sectors the 16-bit field is set to zero and the 32-bit field holds the real count. Our 4 MiB disk fits in 16 bits (8192 sectors), but the code handles both cases.

Every computation in the coming chapters — where the FAT starts, where the root directory lives, which cluster holds a given file — derives from these numbers.

Extended BPB

Bytes 36–61 were added after the original BPB and carry additional identification:

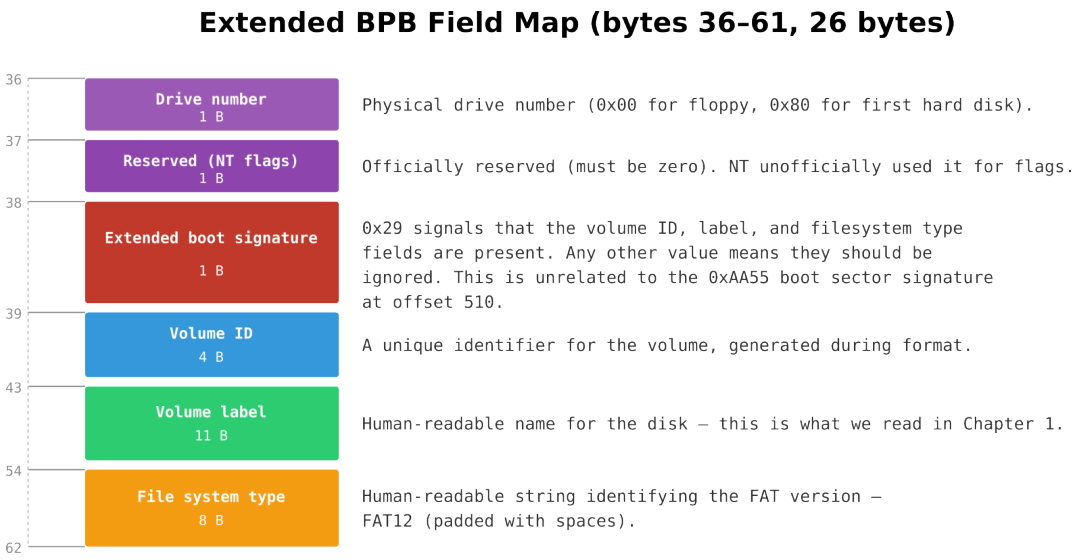

Figure 9. Extended BPB field map

Boot Sector Signature

Bytes 510–511 contain the boot sector signature `0x55 0xAA` (stored little-endian as `0xAA55`). This two-byte marker tells the BIOS that the sector is bootable. Without it, the BIOS skips the device. Do not confuse this with the extended boot signature (`0x29`) at offset 38 — they are separate fields with unrelated purposes.

CHS vs LBA

There are two ways to tell a disk controller which sector to read: CHS and LBA.

CHS (Cylinder-Head-Sector) was the original addressing scheme, inherited from the days when the operating system talked directly to disk drive hardware.

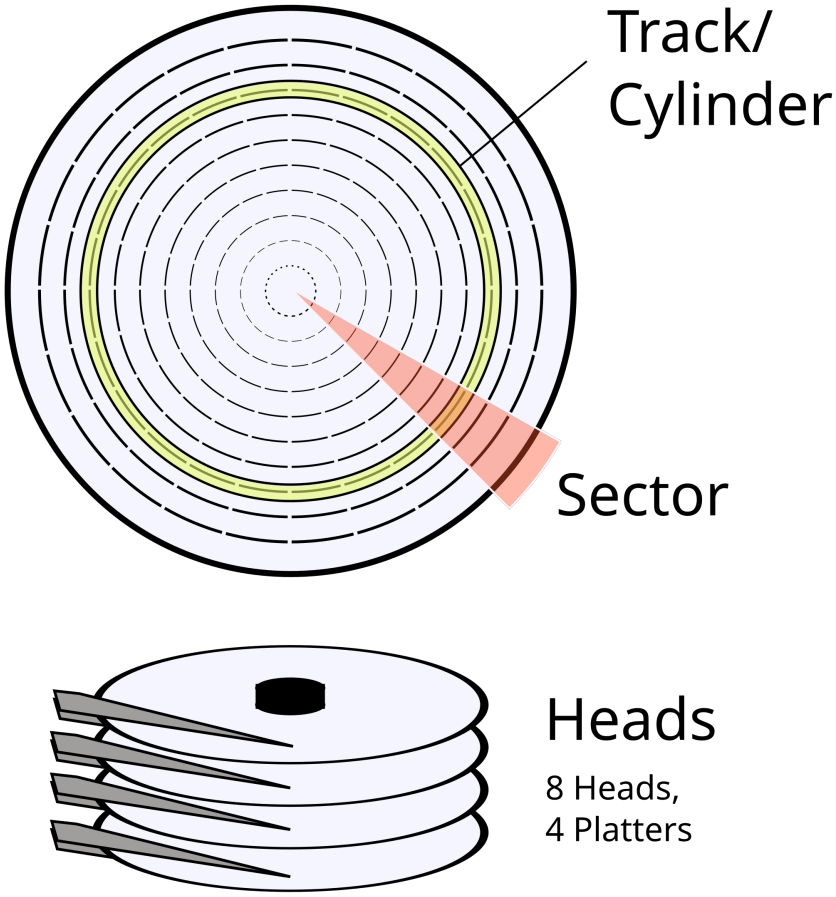

Figure 10. CHS addressing

Source: Wikipedia

It represents a 3D coordinate system: the head selects a platter surface, the cylinder (or track) is a circular band on that surface, and the sector is a block within that track. CHS addresses are awkward because the geometry depends on the physical drive — a sector expressed as cylinder 10, head 2, sector 6 means different things on different disks.

LBA (Logical Block Addressing) replaces the 3D coordinate with a simple linear index: sector 0, sector 1, sector 2, up to sector N – 1. The disk controller

translates the LBA to the physical platter, head, and sector internally. No geometry knowledge needed. LBA28 (28-bit LBA, supporting up to 128 GiB) is what the ATA controller in Chapter 11 uses.

The BPB fields `sectors_per_track` and `number_of_heads` are legacy CHS geometry values – they were originally used by the BIOS's INT 13h to convert CHS addresses to physical locations. On modern hardware and our virtual disk, these are vestigial. The disk controller transparently translates LBA to its internal geometry, and our code never touches CHS. The fields exist in the BPB only for compatibility with software written before LBA became universal. Every calculation in this book uses LBA – our code never reads the CHS geometry fields.

FAT12 Volume Layout

Here is how the BPB fields carve the disk into regions:

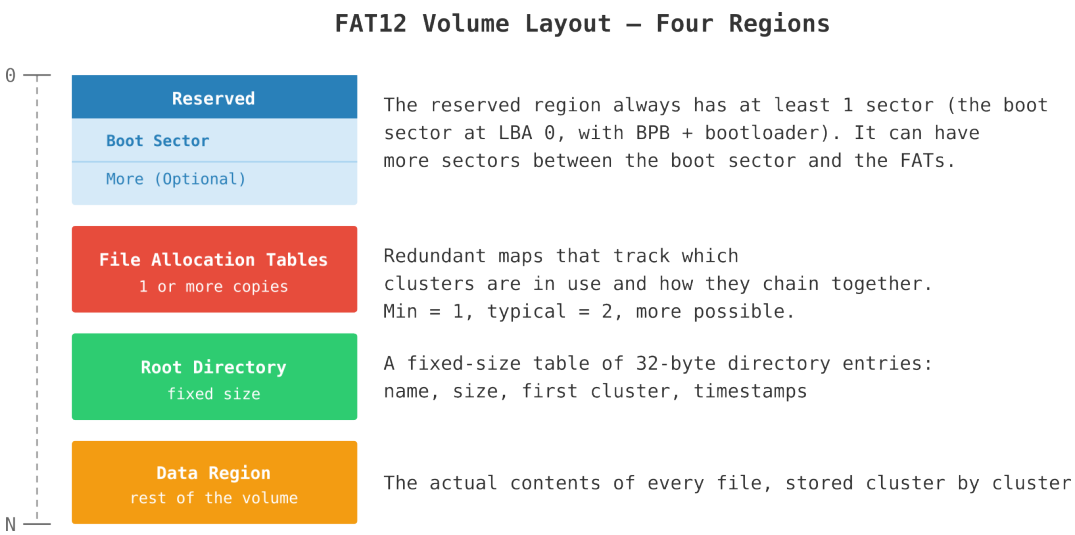

Figure 11. FAT12 volume layout

The reserved region starts at LBA 0. Its first sector is always the boot sector (BPB + bootloader). The rest of the reserved region – if `reserved_sector_count > 1` – sits between the boot sector and the FATs (on FAT12 it is almost always empty: `reserved_sector_count` is typically 1). After the reserved region come the FATs – 1 or more redundant copies, each `sectors_per_fat` sectors wide. The root directory follows right after the last FAT copy, occupying however many sectors it needs to hold `root_entry_count` entries at 32

bytes each. Everything from the data region onward holds file and subdirectory cluster chains.

These LBAs are the numbers our code will compute in every chapter ahead. Learn to read this map, and you already know where every structure lives on disk.

Mapping Bytes to Structs

Reading raw bytes by hand from a hex dump works for exploration, but our code needs a structured way to access every field. The solution is a packed C struct that mirrors the boot sector's exact binary layout.

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. The solution is `#pragma pack(push, 1)`, which tells the compiler to pack every field at the next available byte with no gaps.

Create `src/layout.h` with the three packed structs:

```
#ifndef LAYOUT_H
#define LAYOUT_H

#include <stdint.h>

#pragma pack(push, 1)

typedef struct {
    uint16_t bytes_per_sector;
    uint8_t sectors_per_cluster;
    uint16_t reserved_sector_count;
    uint8_t  num_fats;
    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;

```

```
23  typedef struct {
24      uint8_t  drive_number;
25      uint8_t  reserved_nt;
26      uint8_t  boot_signature;
27      uint32_t volume_id;
28      char volume_label[11];
29      char file_system_type[8];
30  } ExtendedBPB;
31
32  typedef struct {
33      uint8_t jump[3];
34      char oem_name[8];
35      BPB bpb;
36      ExtendedBPB extended_bpb;
37      uint8_t  boot_code[448];
38      uint16_t signature;
39  } BootSector;
40
41  #pragma pack(pop)
42
43  #endif
```

Let's walk through why the sizes line up. The jump instruction is 3 bytes (offset 0–2). The OEM name is 8 bytes (3–10), bringing us to offset 11 — exactly where the BPB starts. The BPB struct's 25 bytes (offsets 11–35) put us at 36, where `ExtendedBPB` picks up with its 26 bytes (36–61). Then `boot_code[448]` fills 62–509, and `signature` lands at bytes 510–511. Three-byte jump + 8-byte OEM + 25-byte BPB + 26-byte extended BPB + 448-byte code + 2-byte signature = 512. Every byte accounted for, no holes, no overlap.

The FAT12 Library

Now we need a header that declares the public API. Create `include/fat12.h`:

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

typedef struct {
    char oem_name[9];
    char volume_label[12];
    char file_system_type[9];
    uint16_t bytes_per_sector;
    uint8_t sectors_per_cluster;
    uint16_t reserved_sector_count;
    uint8_t num_fats;
    uint16_t root_entry_count;
    uint32_t total_sectors;
    uint8_t media_descriptor;
    uint16_t sectors_per_fat;
    uint16_t sectors_per_track;
    uint16_t number_of_heads;
    uint32_t hidden_sectors;
    uint8_t drive_number;
    uint8_t boot_signature;
    uint32_t volume_id;
} VolumeInfo;

VolumeInfo fat12_volume_info(BlockDevice* disk);

#endif
```

Every function that touches the disk takes a `BlockDevice*` parameter – no global state, no mount ceremony. Pass it a disk, get back a result. This will save us headaches when the library grows.

Now create `src/fat12.c`.

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

static BootSector fat12_read_boot_sector(BlockDevice* disk)
{
    uint32_t sector_size = block_device_sector_size(disk);

    void* buffer = malloc(sector_size);
    block_device_read(disk, 0, 1, buffer);
```

```
13
14      BootSector result;
15      memcpy(&result, buffer, sizeof(BootSector));
16
17      free(buffer);
18
19      return result;
20  }
21
22  VolumeInfo fat12_volume_info(BlockDevice* disk)
23  {
24      BootSector s = fat12_read_boot_sector(disk);
25      VolumeInfo info = {0};
26
27      memcpy(info.oem_name, s.oem_name, 8);
28      info.oem_name[8] = '\0';
29
30      memcpy(info.volume_label, s.extended_bpb.volume_label, 11);
31      info.volume_label[11] = '\0';
32
33      memcpy(info.file_system_type, s.extended_bpb.file_system_type, 8);
34      info.file_system_type[8] = '\0';
35
36      info.bytes_per_sector = s.bpb.bytes_per_sector;
37      info.sectors_per_cluster = s.bpb.sectors_per_cluster;
38      info.reserved_sector_count = s.bpb.reserved_sector_count;
39      info.num_fats = s.bpb.num_fats;
40      info.root_entry_count = s.bpb.root_entry_count;
41      info.total_sectors = s.bpb.total_sectors_16
42          ? s.bpb.total_sectors_16
43          : s.bpb.total_sectors_32;
44      info.media_descriptor = s.bpb.media;
45      info.sectors_per_fat = s.bpb.fat_size_16;
46      info.sectors_per_track = s.bpb.sectors_per_track;
47      info.number_of_heads = s.bpb.number_of_heads;
48      info.hidden_sectors = s.bpb.hidden_sectors;
49      info.drive_number = s.extended_bpb.drive_number;
50      info.boot_signature = s.extended_bpb.boot_signature;
51      info.volume_id = s.extended_bpb.volume_id;
52
53      return info;
54  }
```

`fat12_read_boot_sector` allocates a 512-byte buffer, reads sector 0 into it, then copies the bytes into a `BootSector` struct with `memcpy`. The packed struct guarantees that every field lines up with the correct on-disk offset. `fat12_volume_info` calls it, then copies each field into a `VolumeInfo` struct.

Notice the two representations of the same data. `BootSector` and its

nested structs (`BPB`, `ExtendedBPB`) mirror the disk exactly — `char` arrays with space-padded, non-null-terminated strings, just as `mkfs.fat` wrote them. The library uses these internally for all read and write operations. `VolumeInfo` is the public API — it widens those fixed-width char arrays to proper C strings with null terminators. The library never exposes raw disk structs; callers work with the sanitized version.

The Info Command

Now update `platform/cli/main.c` from Chapter 1 to wire in an `info` command:

```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"

int main(int argc, char *argv[]) {
    const char* disk_path = "disk.img";

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

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

    char* command = argv[1];

    if (strcmp(command, "info") == 0) {
        VolumeInfo info = fat12_volume_info(disk);

        printf("OEM Name: %s\n", info.oem_name);
        printf("Volume Label: %s\n", info.volume_label);
        printf("File System Type: %s\n", info.file_system_type);
        printf("Bytes Per Sector: %u\n", info.bytes_per_sector);
        printf("Sectors Per Cluster: %u\n", info.sectors_per_cluster);
        printf("Reserved Sector Count: %u\n", info.reserved_sector_count);
```

```
        printf("Number of FATs: %u\n", info.num_fats);
        printf("Root Entry Count: %u\n", info.root_entry_count);
        printf("Total Sectors: %u\n", info.total_sectors);
        printf("Media Descriptor: 0x%02x\n", info.media_descriptor);
        printf("FAT Size (sectors): %u\n", info.sectors_per_fat);
        printf("Sectors Per Track: %u\n", info.sectors_per_track);
        printf("Number of Heads: %u\n", info.number_of_heads);
        printf("Hidden Sectors: %u\n", info.hidden_sectors);
        printf("Drive Number: 0x%02x\n", info.drive_number);
        printf("Extended Boot Signature: 0x%02x\n", info.boot_signature);
        printf("Volume ID: 0x%08x\n", info.volume_id);
    }

    block_device_close(disk);
    return 0;
}
```

Build and run:

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

We should see:

```
OEM Name: mkfs.fat
Volume Label: MYDISK
File System Type: FAT12
Bytes Per Sector: 512
Sectors Per Cluster: 2
Reserved Sector Count: 1
Number of FATs: 2
Root Entry Count: 512
Total Sectors: 8192
Media Descriptor: 0xf8
FAT Size (sectors): 12
Sectors Per Track: 32
Number of Heads: 2
Hidden Sectors: 0
Drive Number: 0x80
Extended Boot Signature: 0x29
Volume ID: 0x????????    ← varies per format
```

The values may differ slightly depending on your `mkfs.fat` version — in particular, the `sectors_per_fat` and `number_of_heads` fields are geometry choices the formatter makes based on disk size. Every value is read directly from the boot sector bytes, and the `info` command is now a full decoder for the volume identity.

Debugging

Print every BPB field and compare against `mkfs.fat -v` output on your disk image. Also use `xxd -l 512 disk.img` to hexdump the full boot sector and verify each field's offset by hand against the tables above.

Production Notes

- **Check the extended boot signature.** A real driver checks the extended boot signature (`boot_signature == 0x29`) before reading the volume label, volume ID, and filesystem type. If the signature is missing, those fields contain garbage and should be ignored. Our code always reads them because `mkfs.fat` always writes the signature, but production code on foreign-formatted media needs this guard.
- **Validate `bytes_per_sector`.** It is almost always 512 in practice, but a real driver validates it and rejects values outside `{512, 1024, 2048, 4096}`.

What Is Next

We now have `src/layout.h`, `include/fat12.h`, and `src/fat12.c` — the boot sector is fully decoded into C structs, `./fat12-cli info` prints the volume's complete identity card. Every function takes an explicit `BlockDevice*` — no hidden global state, no mount ceremony. Pass a disk, get results.

In Chapter 3 we decode the root directory — a fixed table of 32-byte entries sitting right after the FATs — and build the directory iteration API that powers `ls`. Then in Chapter 4 we crack open the File Allocation Table itself: 12-bit entries, cluster chains, odd-even extraction logic, and the `cat` command that reads a file's contents by following its chain through the FAT.

Chapter 3: The Root Directory

Cluster chains and bit-level extraction are coming — but not yet. Before we wrestle with 12-bit entries, we take a victory lap: the root directory. It is a fixed table of 32-byte entries sitting right after the FATs, and reading it is almost boringly straightforward. Compute an address, read some sectors, print what you find.

That simplicity is the point. The root directory gives the numbered clusters on your disk something they otherwise lack: names. Everything we built in Chapter 2 tells us where things live. Now we see who is home.

By the end of this chapter we will have a `DirectoryEntry` struct in the layout header, a first-class directory iteration API (`opendir` / `readdir` / `closedir`), timestamp decoding from DOS format to readable strings, and an `ls` command that prints it all. `./fat12-cli ls /` lists every file and directory on the volume.

What Is the Root Directory?

Every FAT12 volume has a root directory: a fixed-size table of 32-byte entries sitting right after the last FAT copy. Unlike subdirectories (which we build in clusters later), the root directory is allocated at format time and never grows or shrinks. The BPB field `root_entry_count` (512 on our disk) tells us how many entries fit. If we fill it up, no new files 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, not free clusters, is the bottleneck.

Each entry describes one file, directory, or volume label. The 32 bytes break down like this:

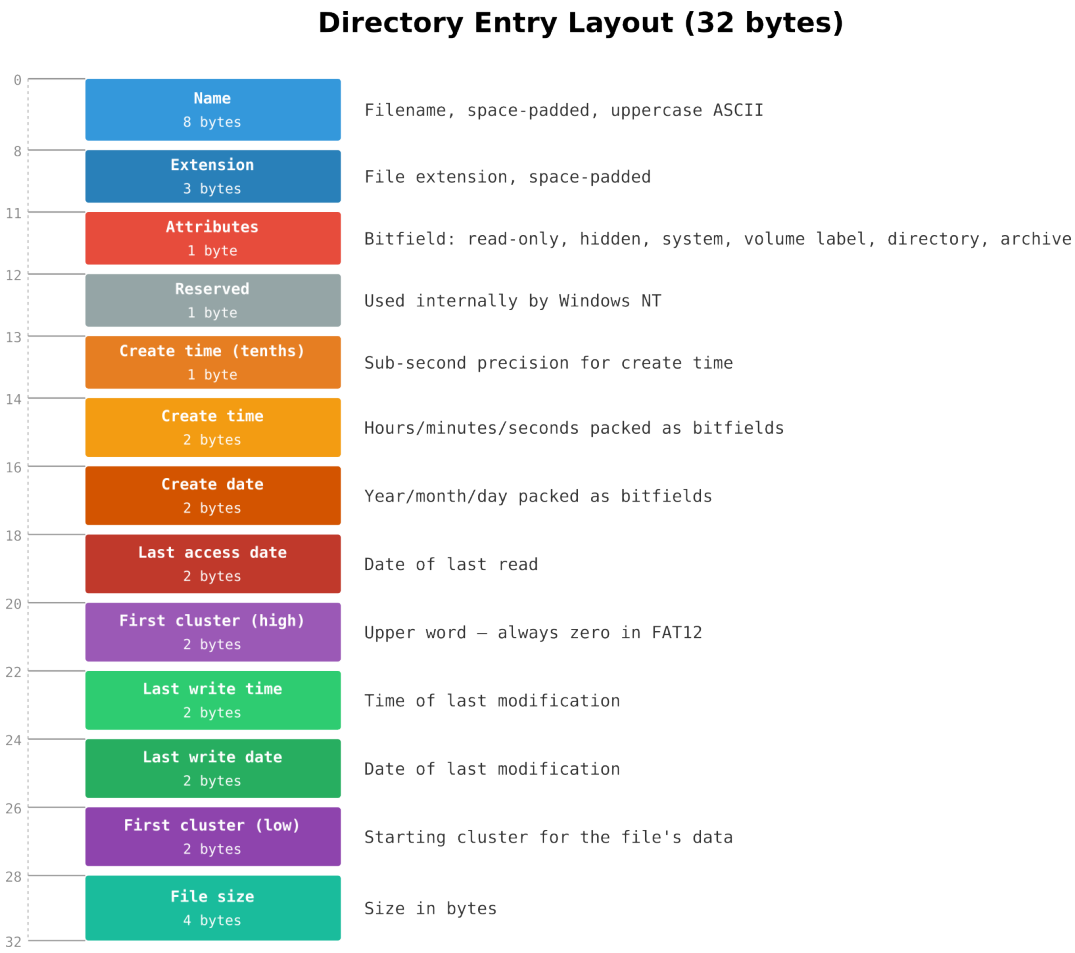

Figure 12. Directory entry layout

Together, the name and extension form the familiar 8.3 format — up to 8 characters for the name, a dot (implied, not stored), and up to 3 for the extension. Both are space-padded on disk and always uppercase.

The first cluster number (offset 26) is the bridge to Chapter 4 — it tells us where on disk the file's data begins. The file size (offset 28) tells us how many bytes to read. The attribute byte tells us what kind of entry we are looking at.

The Attribute Byte

The attribute byte is a bitfield. The bits that matter for our `ls` command:

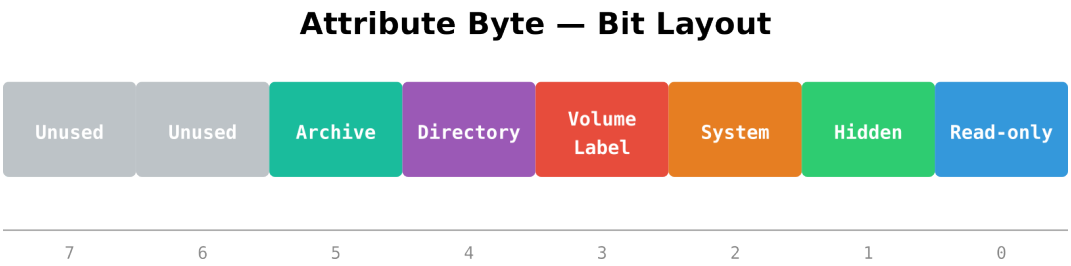

Figure 13. Attribute byte bit layout

The top two bits are unused (always 0). The **archive** bit marks a normal file – it is set when a file is created. Multiple attributes combine by OR-ing: `0x02 | 0x04 = 0x06` for a hidden system file.

When iterating entries, we filter out:

- Entries with `name[0] == 0x00` – no more entries after this
- Entries with `name[0] == 0xE5` – deleted entry
- Entries with `attr == 0x0F` – long filename fragment (not covered in this book – see "Long Filenames" below)

Long Filenames (LFNs)

Long filenames are an extension, introduced with Windows 95, that supports names longer than 8.3: `verylongfilename.txt` instead of `VERYLO~1.TXT`. The trick is repurposing otherwise-invalid directory entries – marked by `attr == 0x0F` – as UTF-16 fragments. Each fragment stores up to 13 characters and chains backward toward the real 8.3 entry. A single long name can span up to 20 fragments. A checksum ties the chain to its real entry so the OS can verify it belongs together, and a sequence number keeps the fragments in order.

It is a clever hack – but we skip these entries with `attr == 0x0F` and move on. Decoding LFN is UTF-16 bookkeeping orthogonal to core FAT12 concepts like cluster chains and the FAT itself. We leave it out because it does not teach anything about FAT12. Skipping LFN fragments is exactly the right thing to do when you are not decoding them – they are valid structures, and our code treats them as invisible.

Even without LFN support, every file remains accessible through its auto-generated 8.3 alias. A file named `Readme For Project.txt` appears as `README~1.TXT` (or similar) in its real 8.3 entry. The 8.3 entry is always present; the LFN fragments are just an overlay on top.

The DirectoryEntry Struct

Add this packed struct to `src/layout.h` after the `BootSector` definition, before `#pragma pack(pop)`:

```
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;
```

Decoding Timestamps

Each directory entry carries two timestamps: creation time (`create_time` / `create_date`) and last modification time (`last_write_time` / `last_write_date`). Both words use the same bit-packed layout, shown here:

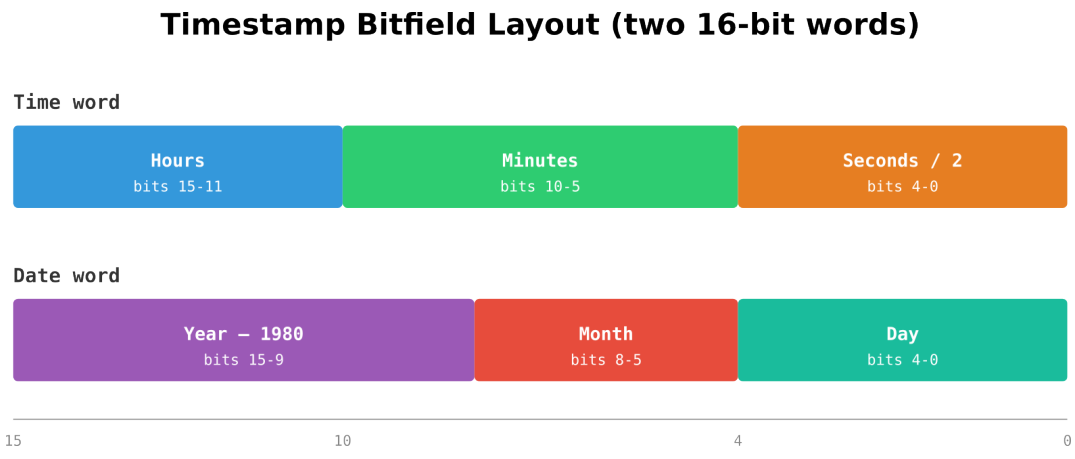

Figure 14. Timestamp bitfield layout

The time word packs hours (5 bits), minutes (6 bits), and seconds divided by 2 (5 bits) into 16 bits. The date word packs year minus 1980 (7 bits), month (4 bits), and day (5 bits) into 16 bits. Multiply the stored seconds value by 2 to get the real seconds, and add 1980 to the stored year to get the calendar year.

The timestamp fits in two 16-bit `uint16_t` fields because FAT12 was designed for DOS in the 1980s, when 2-byte values were cheap and second-level precision was plenty. A date like 2026-05-21 at 14:34:24 would be stored as:

- Time: `0x744C` — hours = 14 (`0x0E`), minutes = 34 (`0x22`), seconds = 24 : 2 = 12 (`0x0C`)
- Date: `0x5CB5` — year = 2026 – 1980 = 46 (`0x2E`), month = 5 (`0x05`), day = 21 (`0x15`)

We will add a static helper to `src/fat12.c` that decodes these packed fields into a convenient struct. Add it after `format_8_3_name`, alongside the other helpers:

```
static DosTimestamp decode_dos_timestamp(uint16_t time, uint16_t date)
{
    DosTimestamp 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;
}
```

`fat12_readdir` calls this helper for both timestamps, so every caller gets clean `DosTimestamp` fields — `entry.modify_time.year`, `entry.create_time.hours` — without touching a single bit operation. The bit-shifting logic lives in the library, shared by every platform. Only the final string formatting — converting year/month/day/hours/minutes/seconds into `"2026-05-21 14:34:24"` — is platform-specific, since it's a presentation detail rather than FAT12 logic. The CLI uses `snprintf`; a kernel would roll its own integer-to-string routine.

Where Is the Root Directory?

The root directory sits right after the last FAT copy. Here is the volume layout again — the third region is what we are reading now:

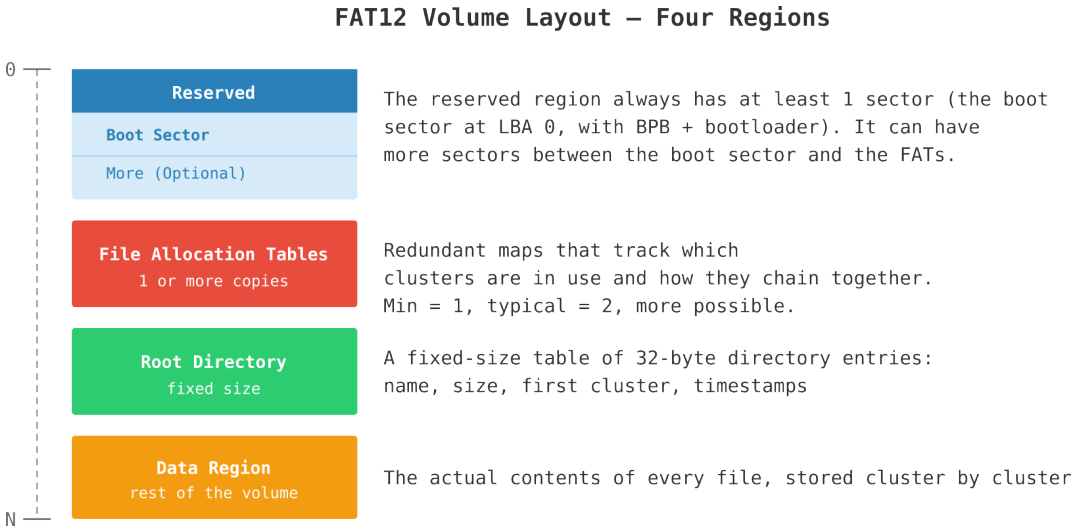

Figure 15. FAT12 volume layout

We already have all the numbers we need from the BPB:

```
Root directory LBA = reserved_sector_count + (num_fats * fat_size_16)
```

For our disk: `1 + (2 * 12) = 25`. The root directory starts at LBA 25.

How many sectors does it span? Each entry is 32 bytes, and we have `root_entry_count` of them:

```
Root directory size = ceil(root_entry_count * 32 / bytes_per_sector)
```

For our disk: `ceil(512 * 32 / 512) = 32` sectors. The root directory occupies LBA 25 through LBA 56.

After the root directory comes the data region — where file contents live. The formula we need to find our way there — `first_data_sector` — will join the library in Chapter 4, when we have the FAT machinery to use it.

Geometry Functions

We need two calculations right now — the root directory's size and its starting LBA. They live in `src/fat12.c` as `static` helpers, visible only inside this file. Open `src/fat12.c` and add these two functions just before the `fat12_root_directory` helper (which we will write next):

```
static uint32_t root_dir_sector_count(BootSector bs)
{
    return ((bs.bpb.root_entry_count * 32)
        + (bs.bpb.bytes_per_sector - 1))
        / bs.bpb.bytes_per_sector;
}

static uint32_t calculate_root_dir_lba(BootSector bs)
{
    return bs.bpb.reserved_sector_count +
           (bs.bpb.num_fats * bs.bpb.fat_size_16);
}
```

`root_dir_sector_count` computes how many sectors the root directory needs: each entry is 32 bytes, multiplied by `root_entry_count`, then rounded up to the nearest full sector. `calculate_root_dir_lba` starts at the end of the last FAT: `reserved_sector_count` (sector 1) + (`num_fats` × `fat_size_16`) = `1 + (2 × 12)` = LBA 25 on our disk.

One more geometry function — `first_data_sector` — will join the library in Chapter 4 when we have FAT entries to decode. Together these three functions cover every disk address we will ever need.

Loading the Root Directory

We still need a function that reads the root directory into memory, but it is now a `static` helper — an implementation detail, not part of the public API. Add it to `src/fat12.c` after the geometry functions:

```
static DirectoryEntry* fat12_root_directory(
    BlockDevice* disk,
    BootSector bs,
    uint32_t* count
)
{
    uint32_t sectors = root_dir_sector_count(bs);
    uint32_t lba = calculate_root_dir_lba(bs);
    uint32_t bytes = sectors * bs.bpb.bytes_per_sector;

    *count = bytes / sizeof(DirectoryEntry);

    DirectoryEntry* entries =
        (DirectoryEntry*)malloc(bytes);

    block_device_read(disk, lba, sectors, entries);

    return entries;
}
```

Same calculation, same allocation, same read — just guarded by `static` so only other functions inside `fat12.c` can call it.

Formatting an 8.3 Name

Before we define the public API, we need one more static helper. Every function that reads a directory entry needs to convert the raw 8.3 name fields (space-padded, no null terminator) into a clean C string. We will call this helper from both `fat12_readdir` and the functions in later chapters. Add it after `fat12_root_directory`:

```
static void format_8_3_name(const DirectoryEntry* raw, char* out)
{
    int i = 0;

    for (; i < 8 && raw->name[i] != ' '; i++)
        out[i] = raw->name[i];

    int has_ext = 0;
    for (int k = 0; k < 3; k++)
    {
        if (raw->ext[k] != ' ')
        {
            has_ext = 1;
            break;
```

```
        }
    }

    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 first loop copies the name bytes until it hits a space or the full 8 bytes. Then we check whether the extension has any non-space character — if it does, we add a dot and copy the extension bytes. The result is a null-terminated string like `"HELLO.TXT"` or `"MYDIR"`. In Chapter 6 we will write the inverse — `set_entry_name` — which packs a user string back into the on-disk 8.3 format with space padding.

Now the public API. Add a `struct Directory` definition before the function implementations:

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

This keeps track of our position as we iterate through directory entries. The full struct is hidden inside `fat12.c` — callers only see an opaque `Directory*` pointer.

Add these three functions after the struct:

```
Directory* fat12_opendir(BlockDevice* disk, const char* path)
{
    (void)path;

    BootSector bs = fat12_read_boot_sector(disk);

    uint32_t count;
    DirectoryEntry* entries =
        fat12_root_directory(disk, bs, &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)
{
    while (dir->offset < dir->count)
    {
        DirectoryEntry* raw = &dir->entries[dir->offset];

        if (raw->name[0] == 0x00) return -1;

        dir->offset++;

        if ((unsigned char)raw->name[0] == 0xE5) continue;
        if (raw->attr == 0x0F) continue;

        memset(out->name, 0, sizeof(out->name));
        format_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;
    }

    return -1;
}

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

```
51      free(dir);
52  }
```

`fat12_opendir` reads the boot sector into a `BootSector` struct, calls the static `fat12_root_directory` to load every entry, and wraps the result in a `Directory` struct. The `(void)path;` line silences the compiler's "unused parameter" warning – we accept a path for future subdirectory traversal but do not use it yet. Right now the function ignores the path and always reads the root directory. For now, pass `"/"` (or any string – validation is not yet in place). That parameter exists for later chapters when we add subdirectory traversal; then the same function will resolve the path and open whatever directory it points to.

`fat12_readdir` returns one entry at a time. It handles filtering internally: it skips deleted entries (`0xE5`) and long filename fragments (`0x0F`), and stops when it hits the end-of-directory marker (`0x00`). Each valid entry is passed to `format_8_3_name`, which produces a human-readable name like `HELLO.TXT` or `MYDIR`. The function also decodes both timestamps via `decode_dos_timestamp` – a static helper that unpacks the DOS bitfields into a `DosTimestamp` struct with clean `year`, `month`, `day`, `hours`, `minutes`, and `seconds` fields. The caller gets ready-to-use values like `entry.modify_time.year` without any bit operations. The function returns `0` on success and `-1` when there are no more entries.

Notice the order of checks inside the loop: `0x00` is checked *before* incrementing `dir->offset`, while `0xE5` and `0x0F` are checked *after*. That is deliberate. The `0x00` marker means "no more entries ever" – once we see it we return immediately without advancing, so a subsequent call sees the same marker and stops too. Deleted entries and LFN fragments are holes we skip past – we increment the offset and continue scanning, because a real entry might follow.

The `0xE5` check also casts `raw->name[0]` to `unsigned char`. On most platforms `char` is signed by default, and `0xE5` (229) does not fit in a signed byte (–128 to 127). Without the cast, sign-extension turns `0xE5` into `0xFFFFFFE5` during integer promotion, and the comparison fails – the entry would not be skipped, and our `ls` would show a garbage name. The cast ensures the comparison treats the byte as the unsigned value that it is.

`fat12_closedir` frees the entry buffer and the `Directory` struct. Every `opendir` must be paired with a `closedir`.

The ls Command

Now we wire everything together. Add an `ls` branch to `main()` that loads the root directory, iterates every entry, and prints the name, size, and last-write timestamp of each file.

Open `platform/cli/main.c` and replace it with:

```
#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,
                            DosTimestamp ts)
{
    snprintf(buf, size, "%04u-%02u-%02u %02u:%02u:%02u",
             ts.year, ts.month, ts.day,
             ts.hours, ts.minutes, ts.seconds);
}

int main(int argc, char *argv[]) {
    const char* disk_path = "./disk.img";

    if (argc < 2) return -1;

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

    char* command = argv[1];

    if (strcmp(command, "info") == 0) {
        VolumeInfo info = fat12_volume_info(disk);

        printf("OEM Name: %s\n", info.oem_name);
        printf("Volume Label: %s\n", info.volume_label);
        printf("File System Type: %s\n", info.file_system_type);
        printf("Bytes Per Sector: %u\n", info.bytes_per_sector);
        printf("Sectors Per Cluster: %u\n", info.sectors_per_cluster);
        printf("Reserved Sector Count: %u\n", info.reserved_sector_count);
        printf("Number of FATs: %u\n", info.num_fats);
        printf("Root Entry Count: %u\n", info.root_entry_count);
```

```
41          printf("Total Sectors: %u\n", info.total_sectors);
42          printf("Media Descriptor: 0x%02x\n", info.media_descriptor);
43          printf("FAT Size (sectors): %u\n", info.sectors_per_fat);
44          printf("Sectors Per Track: %u\n", info.sectors_per_track);
45          printf("Number of Heads: %u\n", info.number_of_heads);
46          printf("Hidden Sectors: %u\n", info.hidden_sectors);
47          printf("Drive Number: 0x%02x\n", info.drive_number);
48          printf("Extended Boot Signature: 0x%02x\n", info.boot_signature);
49          printf("Volume ID: 0x%08x\n", info.volume_id);
50      }
51      else if (strcmp(command, "ls") == 0) {
52          Directory* dir = fat12_opendir(disk, "/");
53
54          printf("Directory: \\\n\n");
55
56          DirEntry entry;
57          while (fat12_readdir(dir, &entry) == 0)
58          {
59              char time_str[20];
60              format_datetime(time_str, sizeof(time_str),
61                              entry.modify_time);
62
63              if (entry.attr == 0x08)
64              {
65                  printf("%-12s  %-8s  %s\n",
66                         entry.name, "<VOL>", time_str);
67              }
68              else if (entry.attr & 0x10)
69              {
70                  printf("%-12s  %-8s  %s\n",
71                         entry.name, "<DIR>", time_str);
72              }
73              else
74              {
75                  char size_str[16];
76                  snprintf(size_str, sizeof(size_str), "%u B", entry.size);
77                  printf("%-12s  %-8s  %s\n",
78                         entry.name, size_str, time_str);
79              }
80          }
81
82          fat12_closedir(dir);
83      }
84
85      block_device_close(disk);
86      return 0;
87  }
```

The `ls` branch calls `fat12_opendir(disk, "/")` to open the root directory, then loops through every entry with `fat12_readdir`. For each valid

entry, we check:

- `attr == 0x08` – volume label, print with `<VOL>` marker
- `attr & 0x10` – directory, print with `<DIR>` marker
- Otherwise – regular file, print name and size

The filtering of deleted entries, long filename fragments, and the end-of-directory marker is handled inside `fat12_readdir` – the command code stays clean and focused on output formatting.

The `%-12s` and `%-8s` format specifiers left-justify the entry name and type columns. 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`.

Updating the Header

The `root_dir_sector_count` and `calculate_root_dir_lba` helpers are `static` inside `fat12.c` – they do not appear in the header. We need to add the `DosTimestamp` struct, the `DirEntry` struct (with decoded timestamp fields), the opaque `Directory` type, and the three directory iteration functions. Open `include/fat12.h` and make these additions before the `#endif`:

```
/* Decoded DOS timestamp */
typedef struct {
    unsigned year;
    unsigned month;
    unsigned day;
    unsigned hours;
    unsigned minutes;
    unsigned seconds;
} DosTimestamp;

/* Directory entry (user-facing) */
typedef struct {
    char name[12];
    uint32_t size;
    uint8_t attr;
    DosTimestamp create_time;
    DosTimestamp modify_time;
} DirEntry;
```

```
19
20  typedef struct Directory Directory;
21
22  /* Directory iteration */
23  Directory* fat12_opendir(BlockDevice* disk, const char* path);
24  int fat12_readdir(Directory* dir, DirEntry* out);
25  void fat12_closedir(Directory* dir);
```

The full `include/fat12.h` should now look like:

```
1   #ifndef FAT12_H
2   #define FAT12_H
3   #include <stddef.h>
4   #include <stdint.h>
5   #include "block_device.h"
6
7   typedef struct {
8       char oem_name[9];
9       char volume_label[12];
10      char file_system_type[9];
11      uint16_t bytes_per_sector;
12      uint8_t sectors_per_cluster;
13      uint16_t reserved_sector_count;
14      uint8_t num_fats;
15      uint16_t root_entry_count;
16      uint32_t total_sectors;
17      uint8_t media_descriptor;
18      uint16_t sectors_per_fat;
19      uint16_t sectors_per_track;
20      uint16_t number_of_heads;
21      uint32_t hidden_sectors;
22      uint8_t drive_number;
23      uint8_t boot_signature;
24      uint32_t volume_id;
25  } VolumeInfo;
26
27  VolumeInfo fat12_volume_info(BlockDevice* disk);
28
29  /* Decoded DOS timestamp */
30  typedef struct {
31      unsigned year;
32      unsigned month;
33      unsigned day;
34      unsigned hours;
35      unsigned minutes;
36      unsigned seconds;
37  } DosTimestamp;
38
39  /* Directory entry (user-facing) */
```

```
typedef struct {
    char name[12];
    uint32_t size;
    uint8_t attr;
    DosTimestamp create_time;
    DosTimestamp modify_time;
} DirEntry;

typedef struct Directory Directory;

/* Directory iteration */
Directory* fat12_opendir(BlockDevice* disk, const char* path);
int fat12_readdir(Directory* dir, DirEntry* out);
void fat12_closedir(Directory* dir);

#endif
```

Building

Compile and run:

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

On a freshly formatted disk with no files, the output should be similar to:

```
Directory: \

MYDISK        <VOL>    2026-05-21 14:34:22
```

Just the volume label — there are no files yet. The timestamp shows when the disk was formatted; yours will be different.

Verification: Create Real Files

Time to put files on the disk and verify our reader can see them. Mount it via loopback and create some test files:

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

echo "Hello from FAT12!" | sudo tee /tmp/fatmnt/hello.txt > /dev/null
echo "FAT12 internals"  | sudo tee /tmp/fatmnt/readme.txt > /dev/null
sudo mkdir /tmp/fatmnt/mydir

sudo umount /tmp/fatmnt
```

These three commands mount the disk image, create a short text file, a second text file, and a subdirectory, then unmount. Now our code can see what the host wrote:

```
./fat12-cli ls /
```

Output (your timestamps will differ):

```
Directory: \

MYDISK        <VOL>    2026-05-21 14:34:22
HELLO.TXT     18 B     2026-05-21 14:34:24
README.TXT    16 B     2026-05-21 14:34:24
MYDIR         <DIR>    2026-05-21 14:34:24
```

Every file and directory we created is visible. The `fat12_readdir` function has joined the name and extension with a dot, so `HELLO.TXT` appears as a clean filename instead of the raw 8.3 split. The sizes match: `hello.txt` contains 18 bytes ("Hello from FAT12!" plus the newline that `echo` appends). The subdirectory shows `<DIR>` instead of a size. The timestamp on each entry is its last-write time — the moment the file was created or last modified, encoded by the operating system as a pair of bit-packed 16-bit words.

Debugging with a Hexdump

If the output looks wrong, dump the root directory sectors directly and decode an entry by hand:

```
xxd -s $((25 * 512)) -l 512 disk.img
```

The root directory LBA is 25 for our disk, so `-s $((25 * 512))` skips to byte 12800. The first 32 bytes after our volume label entry should decode to a valid directory entry. Match the bytes against the offset table at the start of this chapter:

- Bytes 0–7: filename, space-padded
- Bytes 8–10: extension, space-padded
- Byte 11: attribute (`0x20` = archive for our files)
- Bytes 26–27: first cluster (little-endian `uint16_t`)
- Bytes 28–31: file size (little-endian `uint32_t`)

Production Notes

- **Long filename entries.** See "Long Filenames (LFNs)" earlier in this chapter. Our loop skips the fragments; a production driver would decode them into proper long names.
- **Entry sorting.** The root directory is not sorted. Entries appear in the order they were created, and deleted entries leave holes. A production `ls` might sort alphabetically; ours prints in creation order.
- **`fat12_opendir` allocates memory.** Every call `malloc`s a buffer for the full root directory. On a resource-constrained system, this allocation could fail. A production version would check for `NULL` and propagate the error; ours assumes `malloc` succeeds.
- **The `file_size` field is only valid for regular files.** Directories and volume labels have size zero even if they appear to contain data. Always check the attribute byte before using `file_size`.
- **8.3 fields are not null-terminated.** The `name` and `ext` fields in `DirectoryEntry` are fixed-size arrays with no null terminator. Our `fat12_readdir` copies them into a null-terminated `DirEntry.name` before the caller sees them, so the `ls` command can use `%s` safely. If you ever access a `DirectoryEntry` directly, use precision specifiers like `%.8s` and `%.3s` or copy to a null-terminated buffer first.
- **Timestamp decoding lives in the library, formatting is platform-specific.** The `decode_dos_timestamp` helper in `fat12.c` does the

bit-shifting work and returns a `DosTimestamp` struct — every platform uses the same decoder. The `format_datetime` function then formats that struct as a string; on the CLI it uses `snprintf`, while a kernel would need its own integer-to-string routine.

What Is Next

We now have the root directory fully decoded. `./fat12-cli ls /` lists every entry on the disk. The static geometry helpers `root_dir_sector_count` and `calculate_root_dir_lba` are in `fat12.c`, ready for the next step.

In Chapter 4 we crack open the File Allocation Table itself. We will learn how 12-bit entries encode cluster chains, write the odd-even extraction logic, load the FAT into memory, walk a chain from start to end, and wire a `cat` command that reads a file's contents by following its chain through the FAT.