FAT12

Understanding and Implementing the Classic File System in C_

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 2026-05-26

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.

© 2026 Björn Götz

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

Hello, Disk! **6**
- What Is FAT12? 6
- Creating a Virtual Disk 7
- Sectors and LBA 9
- A Tale of Four Regions 9
- Project Layout 10
- A Portable Foundation: The BlockDevice 10
- Reading Sector 0 16
- What Is Next 17

The Blueprint (BPB) **19**
- Boot Sector Structure 19
- Mapping Bytes to Structs 26
- The FAT12 Library 27
- The Info Command 29

Debugging 31
Production Notes 31
What Is Next 32

The Root Directory **33**
What Is the Root Directory? 33
The DirectoryEntry Struct 33
Decoding Timestamps 33
Where Is the Root Directory? 33
Geometry Functions 33
Loading the Root Directory 34
The ls Command 34
Updating the Header 34
Building 34
Verification: Create Real Files 34
Debugging with a Hexdump 34
Production Notes 34
What Is Next 35

Reading Files **36**
What the FAT Actually Is 36
Where the FAT Lives 36
Data Region Geometry 36
Reading an Entry 36
Loading the FAT 36
File API 36
The cat Command 37
Updating the Header 37
Building 37
Verification: Create a Real File 37
Hexdump Verification 37
Multi-Cluster File 37
Production Notes 37
FAT Family: FAT16 and FAT32 38
What Is Next 38

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 working `cat` command 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 four parts that build on each other:

Foundation – What is FAT12 and how do we pry open a raw disk? We set up the toolchain, build a portable `BlockDevice` abstraction, read our first sector, then decode every byte of the boot sector into C structs – the geometry calculations that power the entire filesystem.

Part I: Reading – Navigate the File Allocation Table and its famously awkward 12-bit entries. Walk the root directory. 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.

Foundation

A filesystem driver needs three things: a disk to experiment on, a way to talk to it, and a complete understanding of the metadata that governs every operation. This part delivers all three. We will format a real FAT12 volume, build a portable block device layer, read our first raw sector, and decode the boot sector into C structs all the way down to the geometry calculations that drive the entire filesystem. By the end we will know exactly where the FAT, root directory, and data region live on any FAT12 disk, and we will be ready to navigate them blindfolded in Part I.

Part I: Reading

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. Here is what we will create:

- `include/block_device.h` – abstract interface for reading and writing sectors
- `include/debug.h` – debug print macro so we can trace execution
- `platform/cli/file_block_device.h` – file-backed implementation header
- `platform/cli/file_block_device.c` – file-backed implementation code
- `platform/cli/debug.c` – debug print implementation
- `platform/cli/main.c` – a small program that reads the volume label from disk

Six files. By the end of this book they 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.

Every FAT volume is carved into four regions: a **boot sector** that describes the disk geometry, a **File Allocation Table** that tracks which clusters belong to which file, a **root directory** that maps names to first clusters, and a **data region** that holds the file bytes. We will build each piece from the ground up, starting with a real disk to experiment on.

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:

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

Both commands produce no output when they succeed – silence means our disk is ready.

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:

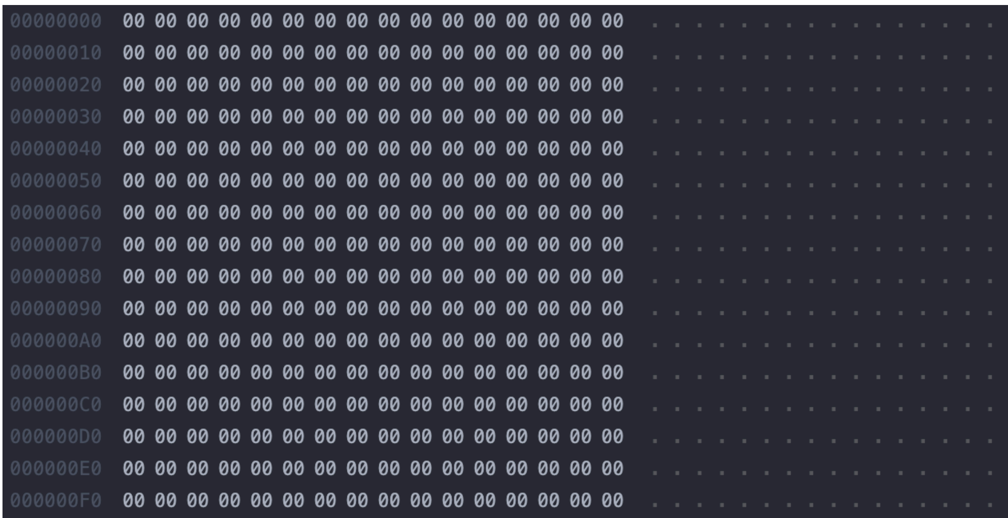

Figure 1. Empty disk – all zeros

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:

Region	Purpose
Boot sector	Filesystem metadata – including the BIOS Parameter Block (BPB) – and space for a bootloader

Region	Purpose
File Allocation Tables	Two (sometimes more) redundant maps that track which clusters are in use and how they chain together
Root directory	A fixed-size table of 32-byte directory entries: name, size, first cluster, timestamps
Data region	The actual contents of every file, stored cluster by cluster

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
│   ├── block_device.h
│   └── debug.h
├── platform/
│   └── cli/                # CLI tool platform code
│       ├── file_block_device.h
│       ├── file_block_device.c
│       ├── debug.c
│       └── main.c
├── src/                    # Shared FAT12 library (populated in later chapters)
└── disk.img                # Our FAT12 volume
```

The `include/` directory is the shared backbone – headers that every platform in this book uses unchanged. The `platform/cli/` directory is the CLI tool's platform layer: the file-backed block device, a debug output implementation, and the entry point. When we add the kernel platform in Chapter 12, it gets its own `platform/kernel/` directory alongside `cli/`, and the library code in `src/` stays exactly the same.

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 12, 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_block_size()` – bytes per sector on disk
- `block_device_block_count()` – total sector count on the disk

Every read and write call takes a buffer pointer. Make sure it is at least `block_count * block_size` bytes, or the I/O will overflow. (Spoiler: it will, and that is a bug we will not debug here.)

Now create `include/block_device.h`:

```
1  #ifndef BLOCK_DEVICE_H
2  #define BLOCK_DEVICE_H
3
4  #include <stdint.h>
5
6  /*
7   * Opening a block device is platform-specific – each platform provides
8   * its own function (e.g. file_block_device_open, ata_block_device_open)
9   * with the signature appropriate for that environment.
10  * This header only defines the common operations on an already-opened device.
11  */
12
```

```c
/*
 * 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 block_count,
    void* buffer
);

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

void block_device_close(BlockDevice* device);
uint32_t block_device_block_size(BlockDevice* device);
uint64_t block_device_block_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`:

```
#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 12, 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/`:

```
#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:

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

#define BLOCK_SIZE 512

struct BlockDevice {
    FILE* file;
};

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 block_count,
    void* buffer
) {
    DBG_PRINT("[ block_device ] read %" PRIu32 " block(s) starting at %" PRIu64
    ↪  "\n", block_count, lba);
    size_t total_bytes = block_count * BLOCK_SIZE;
    off_t offset = lba * BLOCK_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);
```

```
32         return 0;
33     }
34 
35     int block_device_write(
36         BlockDevice* device,
37         uint64_t lba,
38         uint32_t block_count,
39         const void* buffer
40     ) {
41         DBG_PRINT("[ block_device ] write %" PRIu32 " block(s) starting at %"
           ↪  PRIu64 "\n", block_count, lba);
42         size_t total_bytes = block_count * BLOCK_SIZE;
43         off_t offset = lba * BLOCK_SIZE;
44         DBG_PRINT("[ file         ] write %zu bytes at 0x%lx\n", total_bytes,
           ↪  offset);
45         fseeko(device->file, offset, SEEK_SET);
46         fwrite(buffer, 1, total_bytes, device->file);
47         return 0;
48     }
49 
50     void block_device_close(BlockDevice* device) {
51         DBG_PRINT("[ block_device ] close\n");
52         fclose(device->file);
53         free(device);
54     }
55 
56     uint32_t block_device_block_size(BlockDevice* device) {
57         (void)device;
58         return BLOCK_SIZE;
59     }
60 
61     uint64_t block_device_block_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 / BLOCK_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 12, 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`:

```
#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;
}
```

Compile and run it:

```
gcc -Wall -Wextra -iquote include -iquote platform/cli -o fat12
↪ platform/cli/main.c platform/cli/file_block_device.c platform/cli/debug.c
./fat12
```

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:

```
1 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.

What Is Next

We now have `include/block_device.h`, `include/debug.h`, `platform/cli/file_block_device.h`, `platform/cli/file_block_device.c`, `platform/cli/debug.c`, and `platform/cli/main.c` – 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.

The Blueprint (BPB)

Chapter 1 ended with us reading the volume label from raw bytes at offset 43. 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.

Here is what we will create:

- `src/layout.h` – packed C structs matching the on-disk boot sector layout
- `include/fat12.h` – public API declarations for the FAT12 library
- `src/fat12.c` – boot sector reading, volume info, and geometry calculations

We also update `platform/cli/main.c` to wire in an `info` command that prints every field we decode.

Boot Sector Structure

The boot sector itself can be split into several parts:

Offset	Size	Section	Description
0	3	Jump instruction	`JMP` + `NOP`, jumps over header to boot code
3	8	OEM name	ASCII identifier of formatter
11	25	BPB	BIOS Parameter Block (disk geometry & layout)
36	26	Extended BPB	Drive info, volume ID, label, FS type string
62	448	Boot code	Bootloader program
510	2	Signature	`0x55AA` boot sector marker

Offset	Size	Section	Description

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.

```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded Text
00000000  EB 3C 90 42 53 44 20 20 34 2E 34 00 02 40 01 00  . < . B S D     4 . 4 . . @ . .
00000010  02 00 02 00 00 F0 0C 00 20 00 20 00 00 00 00 00  . . . . . . . .   .   . . . . .
```

Figure 3. 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:

```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded Text
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 09 11 7D 13 4E 45 57 5F 4C  x.....)..}.NEW_L
00000030  41 42 45 4C 20 20 46 41 54 31 32 20 20 20 FA 31  ABEL  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  ................
00000100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA  ..............U.
```

Figure 4. 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:

Offset	Size	Field	Description
11	2	Bytes per sector	Size of a single sector in bytes. Almost always 512.
13	1	Sectors per cluster	Number of sectors that form one cluster – the minimum allocation unit. Cluster size = `bytes_per_sector * sectors_per_cluster`.
14	2	Reserved sector count	Number of sectors before the first FAT. Minimum 1 (the boot sector itself).
16	1	Number of FATs	How many copies of the FAT exist. Almost always 2 – the second is a backup.
17	2	Root entry count	Maximum number of 32-byte entries in the root directory.
19	2	Total sectors (16-bit)	Total sectors on the volume, if fewer than 65536. Otherwise set to 0.
21	1	Media descriptor	Identifies the media type: `0xF8` for fixed disks, `0xF0` for removable media (3.5“ floppy, etc.).
22	2	FAT size (sectors)	Number of sectors occupied by each FAT.

Offset	Size	Field	Description
24	2	Sectors per track	Geometry from CHS addressing – how many sectors per track on the physical disk.
26	2	Number of heads	Geometry from CHS addressing – how many read/write heads the physical disk has.
28	4	Hidden sectors	Count of sectors before the boot sector (e.g. from a master boot record). Zero on our volume.
32	4	Total sectors (32-bit)	Total sectors on the volume if the 16-bit field is zero (volumes >= 65536 sectors).

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:

Offset	Size	Field	Description
36	1	Drive number	Physical drive number (`0x00` for floppy, `0x80` for first hard disk).
37	1	Reserved (NT flags)	Used by Windows NT for flags. Zero on our volume.
38	1	Boot signature	`0x29` signals that the next three fields (volume ID, label, filesystem type) exist. Any other value means they should be ignored.
39	4	Volume ID	A unique identifier for the volume, randomly generated during format.
43	11	Volume label	Human-readable name for the disk – this is what we read in Chapter 1.
54	8	File system type	Human-readable string identifying the FAT version – `FAT12   ` (padded with spaces).

Boot Signature

Bytes 510–511 contain the boot signature `0x55 0xAA`. This two-byte marker tells the BIOS that the sector is bootable. Without it, the BIOS skips the device.

CHS Addressing

Cylinder-**h**ead-**s**ector addressing was an early method for locating data on a hard disk drive.

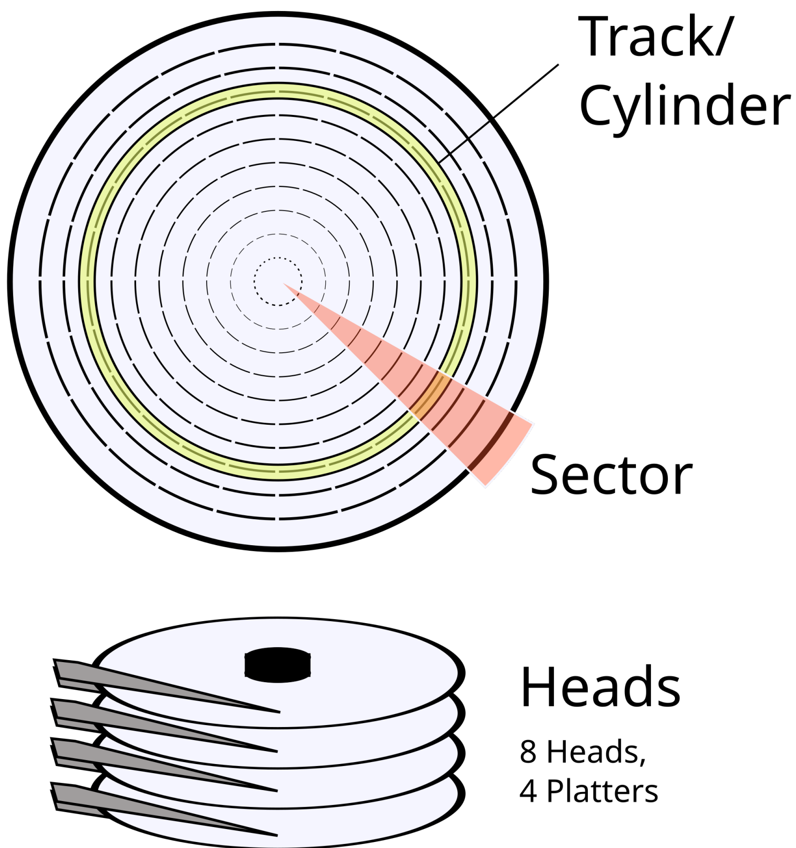

Figure 5. CHS addressing

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. The BPB fields `sectors_per_track` and `number_of_heads` were originally used by the BIOS to convert between CHS addresses and the disk's physical geometry.

On modern hardware – and our virtual disk – these values are vestigial. The disk controller translates between LBA and the internal geometry trans-

parently, and our code never needs CHS. The fields exist in the BPB only for compatibility with legacy software.

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;

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
```

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`:

```
#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;
```

```c
    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 block_size = block_device_block_size(disk);

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

    BootSector result;
    memcpy(&result, buffer, sizeof(BootSector));

    free(buffer);

    return result;
}

VolumeInfo fat12_volume_info(BlockDevice* disk)
{
    BootSector s = fat12_read_boot_sector(disk);
    VolumeInfo info = {0};

    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 with proper null-terminated strings for the character arrays.

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("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:

```
1  gcc -Wall -Wextra -iquote include -iquote platform/cli -o fat12-cli
   ↪ platform/cli/main.c src/fat12.c platform/cli/file_block_device.c
   ↪ platform/cli/debug.c
2  ./fat12-cli info
```

We should see:

```
1  OEM Name: mkfs.fat
2  Volume Label: MYDISK
3  File System Type: FAT12
4  Bytes Per Sector: 512
5  Sectors Per Cluster: 2
6  Reserved Sector Count: 1
7  Number of FATs: 2
8  Root Entry Count: 512
9  Total Sectors: 8192
10 Media Descriptor: 0xf8
11 FAT Size (sectors): 12
12 Sectors Per Track: 32
13 Number of Heads: 2
14 Hidden Sectors: 0
15 Drive Number: 0x80
16 Boot Signature: 0x29
17 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 boot signature.** A real driver checks the 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 unsupported values like 128 or 1024.

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 finally open the File Allocation Table itself. We will learn how 12-bit entries encode cluster chains, write the odd-even extraction logic that has tripped up programmers for decades, and walk our first chain from start to end.

The Root Directory

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

What Is the Root Directory?

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

The Attribute Byte

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

The DirectoryEntry Struct

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

Decoding Timestamps

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

Where Is the Root Directory?

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

Geometry Functions

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

Loading the Root Directory

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

The ls Command

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

Updating the Header

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

Building

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

Verification: Create Real Files

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

Debugging with a Hexdump

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

Production Notes

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

What Is Next

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

Reading Files

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

What the FAT Actually Is

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

Where the FAT Lives

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

Data Region Geometry

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

Reading an Entry

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

Loading the FAT

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

File API

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

The cat Command

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

Updating the Header

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

Building

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

Verification: Create a Real File

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

Hexdump Verification

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

Multi-Cluster File

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

Production Notes

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

FAT Family: FAT16 and FAT32

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

What Is Next

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