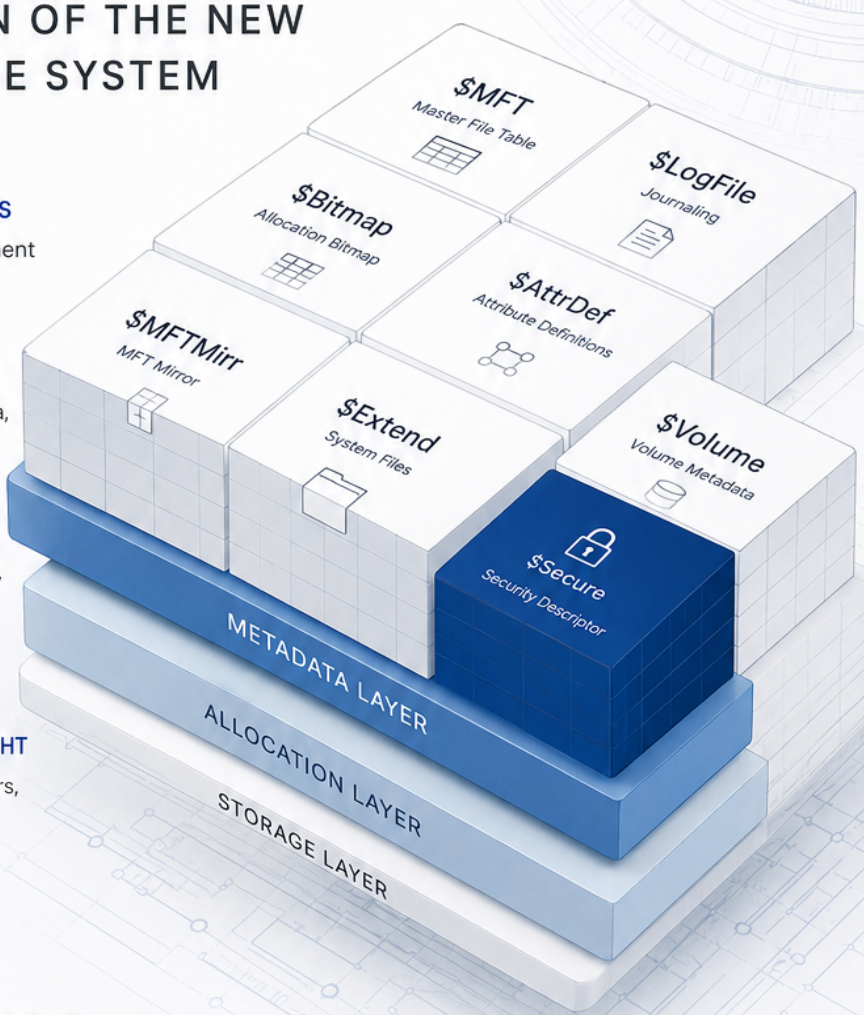
THE NTFS FILE SYSTEM

ARCHITECTURE, INTERNALS, AND IMPLEMENTATION OF THE NEW TECHNOLOGY FILE SYSTEM



FROM FIRST PRINCIPLES
Understand every component and design decision

ON-DISK ARCHITECTURE
Explore structures, metadata, and data management

FULL IMPLEMENTATION
Build a working NTFS parser, reader, writer, allocator, directory manager, journal, and more

PRODUCTION-LEVEL INSIGHT
For developers, OS engineers, and advanced computer science students

JACOB WANG

The NTFS File System

Architecture, Internals, and Implementation of the New Technology File System

Steve T. Publications

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

This version was published on 2026-07-13

Contents

Architecture, Internals, and Implementation of the New Technology File System

This book teaches the NTFS filesystem from first principles through full implementation. You will learn how every component works on disk, why the design choices were made, and how to build a working NTFS parser, reader, writer, allocator, directory manager, journaling subsystem, and metadata handler. Each chapter builds directly on the previous one, culminating in a minimal but functional NTFS implementation. The book is written for experienced software developers, operating system engineers, and computer science students who want deep, production-level understanding of one of the most sophisticated filesystems ever designed.

Introduction: The Architecture of Persistence

In July 1993, Microsoft shipped Windows NT 3.1 with a filesystem called NTFS, the New Technology File System. At that time, FAT was still the dominant filesystem on personal computers, and HPFS was IBM's answer for OS/2. NTFS was designed for enterprise reliability, scalability, and security, and it has remained the default filesystem for Windows ever since, now running on billions of machines worldwide.

Thirty years later, NTFS remains relevant not because Microsoft forced it upon users, but because the architecture is genuinely excellent. The central idea is deceptively simple: everything is a file. The allocation bitmap, the journal, the security database, even the Master File Table that tracks all other files, are themselves files described by the same MFT. This single abstraction means one set of mechanisms handles both your data and the filesystem's own bookkeeping, and it shapes every design decision that follows.

Why NTFS Matters

NTFS introduced features that were revolutionary for their time and remain important today: journaling for crash recovery, access control lists for fine-grained security, transparent compression, sparse files, disk quotas, reparse points, and B+ tree directory indexing. The attribute-based metadata model is more flexible than the fixed-field inode approach used by Unix filesystems. The runlist encoding for non-resident data is more compact than FAT's linked-list chains or even modern extent-based designs for fragmented files.

Understanding NTFS matters for several reasons. If you work on Windows systems, you live on NTFS every day. If you build storage software, filesystem forensics tools, data recovery utilities, or cross-platform filesystem drivers, NTFS knowledge is essential. If you study filesystem design, NTFS offers a masterclass in how to build a coherent, self-consistent architecture where every part reinforces the others.

The Book's Approach

This book takes an incremental, implementation-driven approach. Each chapter introduces concepts, explains the on-disk structures, and provides production-quality C code that you can compile and run. The code starts with basic type definitions and boot sector parsing, then grows to handle MFT records, attributes, runlists, directories, allocation, journaling, and eventually the complete set of NTFS features.

You will need familiarity with C programming, binary data structures, little-endian byte ordering, and basic operating system concepts. Prior filesystem experience is helpful but not required; the book explains everything from scratch. The code assumes a Unix-like development environment, though the concepts apply equally to Windows development.

How to Read This Book

Read the chapters in order. Each chapter depends on structures and functions defined in earlier chapters. If you are already familiar with some NTFS concepts, you can skim the explanatory sections, but the code examples build incrementally and each one references earlier definitions.

The book does not include exercises or quizzes. It is a reference and a tutorial combined, written to be read from cover to cover and then kept on your shelf for years of consulting. Every chapter is self-contained enough to use as a standalone reference, but the cumulative effect of reading them in sequence is what produces genuine mastery.

Let us begin at the beginning, at the very first bytes of an NTFS volume, where the boot sector tells us everything we need to know to start reading the rest of the filesystem.

Chapter 1: Volume Layout and the Boot Sector

Every NTFS volume begins with a single sector, typically 512 bytes, called the Partition Boot Sector or PBS. This sector serves two purposes simultaneously: it contains bootstrap code that can load the operating system, and it carries the filesystem's geometric parameters in a structure called the BIOS Parameter Block, or BPB. The BPB is the key to everything that follows. Without it, we cannot locate the Master File Table, we cannot determine cluster boundaries, and we cannot read any file on the volume.

Sectors, Clusters, and Volume Geometry

Before we examine the boot sector structure, we need to understand the three levels of addressable storage in NTFS: sectors, clusters, and logical cluster numbers.

A sector is the smallest unit the disk hardware understands. For decades this was 512 bytes, and most NTFS volumes still use this value. Modern drives report 4096-byte physical sectors, but NTFS typically abstracts them as 512-byte logical sectors for compatibility. The sector size is stored in the BPB and must be a power of two between 512 and 4096.

A cluster is the smallest unit NTFS allocates to files. NTFS never allocates partial clusters; if a file is one byte, it still consumes an entire cluster (unless the data fits inside the MFT record as a resident attribute, which we will cover later). Cluster size is always a power of two, ranging from 512 bytes (one sector) to 64 KB on older Windows versions, and up to 2 MB on Windows 10 version 1709 and later. The default cluster size depends on the volume size, with larger volumes using larger clusters to reduce metadata overhead.

The relationship between sectors and clusters is expressed as sectors per cluster, a single byte in the BPB. A value of 8 means each cluster contains 8 sectors, so with 512-byte sectors, the cluster size is 4096 bytes (4 KB).

A Logical Cluster Number, or LCN, is the address of a cluster on the volume. LCN 0 is the first cluster, which contains the boot sector. LCN 1 is the second cluster, and so on. The total number of clusters equals the total sectors divided by the sectors per cluster.

The relationship between byte offsets, sector numbers, and cluster numbers is straightforward but fundamental:

```
sector_number = byte_offset / bytes_per_sector
cluster_number = sector_number / sectors_per_cluster
byte_offset = cluster_number * sectors_per_cluster * bytes_per_sector
```

These conversions appear throughout the codebase, so we define them as inline functions from the start.

The Boot Sector Structure

The boot sector is exactly one sector in size, typically 512 bytes. Its layout is fixed and well-defined. The first three bytes are an x86 jump instruction that skips over the BPB data to the bootstrap code. Bytes 3 through 10 contain the OEM identifier, which for NTFS is always the ASCII string “NTFS “ (the word NTFS followed by four space characters). This is the magic number that tells us we are looking at an NTFS volume.

Following the OEM ID, the BPB occupies bytes 11 through 83. The BPB fields mix legacy FAT compatibility fields with NTFS-specific 64-bit values. Several fields are reserved and must be zero on NTFS volumes, remnants of the FAT filesystem structure that the boot sector format evolved from.

After the BPB, bytes 84 through 510 contain the bootstrap code, which loads the NTLDR (or BOOTMGR on Vista and later) loader. The final two bytes at offset 510 must be the signature word 0xAA55, indicating a valid boot sector.

The complete layout looks like this:

```
Offset  Size  Field
------  ----  -----
0x00    3     Jump instruction (EB 52 90)
0x03    8     OEM ID ("NTFS    ")
0x0B    2     Bytes per sector (WORD)
0x0D    1     Sectors per cluster (BYTE)
0x0E    2     Reserved sectors (always 0)
0x10    3     Unused (always 0)
0x13    2     Unused (always 0)
0x15    1     Media descriptor (0xF8 = fixed disk)
0x16    2     Unused (always 0)
0x18    2     Sectors per track (WORD)
0x1A    2     Number of heads (WORD)
0x1C    4     Hidden sectors (DWORD)
0x20    4     Unused (always 0)
0x24    4     EBPB unused (typically 0x80008000)
0x28    8     Total sectors in volume (LLONG)
0x30    8     $MFT cluster number (LLONG)
0x38    8     $MFTMirr cluster number (LLONG)
0x40    1     Bytes/Clusters per File Record Segment (SIGNED BYTE)
0x41    3     Unused (always 0)
0x44    1     Bytes/Clusters per Index Buffer (SIGNED BYTE)
0x45    3     Unused (always 0)
0x48    8     Volume serial number (LLONG)
0x50    4     Checksum (unused, typically 0)
0x54    426   Bootstrap code
0x1FE   2     Boot signature (0xAA55)
```

BIOS Parameter Block Fields

The most critical BPB fields deserve detailed explanation.

Bytes per sector at offset 0x0B is a WORD (16-bit little-endian integer). The value is almost always 512 (0x0200 in little-endian). NTFS supports sector sizes up to 4096, but compatibility with older tools and boot code means 512 remains the default.

Sectors per cluster at offset 0x0D is a single byte. Valid values are powers of two: 1, 2, 4, 8, 16, 32, 64, or 128. A value of 8 (the most common) with 512-byte sectors gives 4 KB clusters. On very small volumes (< 260 MB), the cluster size is 512 bytes (1 sector). On very large volumes, Windows may choose 64 KB or larger.

Total sectors at offset 0x28 is an 8-byte little-endian integer. This is the first field that distinguishes NTFS from FAT in the BPB layout, where the equivalent

field was only 4 bytes and limited volumes to roughly 2 TB. The 64-bit total sectors field allows NTFS volumes up to 2^64 clusters, which at 4 KB per cluster equals 8 exabytes. In practice, Windows implementations impose lower limits: 16 TB on older systems, 256 TB on Windows 8 and later, and up to 8 PB on modern Windows with 2 MB clusters.

$MFT cluster number at offset 0x30 tells us where the Master File Table begins. On Windows NT 4.0 and Windows 2000, this was typically cluster 4 (immediately after the boot sector's two clusters). Starting with Windows XP, Microsoft moved the MFT to LCN 786,432 (0x0C0000), roughly one-third into the volume, to give the MFT room to grow without colliding with user data allocated from the beginning of the volume. This change was made because the early placement caused the MFT to become fragmented as the volume filled up.

$MFTMirr cluster number at offset 0x38 points to a small backup copy of the first four MFT records. On Windows 2000, this was placed near the middle of the volume. Starting with Windows 7, it moved to cluster 2, immediately after the boot sector, because the mirror is only useful if the primary MFT's first sector is damaged, and placing it far away made recovery from widespread corruption less likely.

Bytes or Clusters per File Record Segment at offset 0x40 is a signed byte with an unusual encoding. A positive value (1 to 127) means the number of clusters per MFT record. A negative value means the MFT record size in bytes is 2 raised to the absolute value of the field. The most common value is -10 (0xF6), which means 2^10 = 1024 bytes per MFT record. If the cluster size is larger than the desired MFT record size, the negative encoding is necessary because you cannot have a fraction of a cluster.

Bytes or Clusters per Index Buffer at offset 0x44 uses the same signed-byte encoding for the size of index blocks used in B+ tree directory structures. The value of 1 (meaning one cluster) is most common, giving 4 KB index blocks with 4 KB clusters.

Parsing the Boot Sector (Code)

Now we write the first code: a boot sector parser that reads the BPB fields and validates the volume structure. We define the C structures first, then the parsing function.

```c
/* ntfs_types.h - Fundamental NTFS type definitions */
#ifndef NTFS_TYPES_H
#define NTFS_TYPES_H

#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

/* All NTFS on-disk structures use little-endian byte order.
 * We define explicit types to make this clear and avoid
 * endianness bugs on big-endian systems. */

typedef uint8_t  u8;
typedef uint16_t u16;
typedef uint32_t u32;
typedef uint64_t u64;
typedef int8_t   s8;
typedef int16_t  s16;
typedef int32_t  s32;
typedef int64_t  s64;

/* NTFS uses a specific end-of-attribute marker */
#define ATTR_END_MARK    0xFFFFFFFFu

/* MFT record magic signature */
#define MFT_MAGIC        0x454C4946u  /* "FILE" in little-endian */

/* Boot sector magic signature */
#define BOOT_SIGNATURE   0xAA55

/* Maximum filename length in UTF-16 code units */
#define NTFS_MAX_FILENAME_LEN  255

/* Return codes for NTFS operations */
typedef enum {
    NTFS_OK = 0,
    NTFS_ERROR_IO,
    NTFS_ERROR_CORRUPT,
    NTFS_ERROR_NOT_FOUND,
    NTFS_ERROR_INVALID_PARAM,
    NTFS_ERROR_NO_MEMORY,
    NTFS_ERROR_UNSUPPORTED
} ntfs_status_t;

/* Volume geometry descriptor */
typedef struct {
    u16 bytes_per_sector;       /* Typically 512 */
    u8  sectors_per_cluster;    /* Power of 2, typically 8 */
```

```
    u32 sectors_per_track;      /* CHS geometry (legacy) */
    u16 heads;                  /* CHS geometry (legacy) */
    u32 hidden_sectors;         /* Sectors before this partition */
    u64 total_sectors;          /* Total sectors in the volume */
    u64 mft_cluster;            /* LCN of first $MFT cluster */
    u64 mft_mirr_cluster;       /* LCN of first $MFTMirr cluster */
    s8  clusters_per_file_record; /* Negative = log2(bytes), positive = cluster
    ↪  count */
    s8  clusters_per_index_buffer;
    u64 volume_serial_number;   /* Volume serial (8 bytes) */
} ntfs_geometry_t;

/* Derived geometry values computed from the BPB */
typedef struct {
    ntfs_geometry_t bpb;        /* Raw BPB fields */
    u32 cluster_size;           /* bytes_per_sector * sectors_per_cluster */
    u64 total_clusters;         /* total_sectors / sectors_per_cluster */
    u64 volume_size_bytes;      /* total_sectors * bytes_per_sector */
    u32 mft_record_size;        /* Computed from clusters_per_file_record */
    u32 index_block_size;       /* Computed from clusters_per_index_buffer */
} ntfs_volume_info_t;

#endif /* NTFS_TYPES_H */
```

Next, the boot sector structure and parsing function:

```
/* ntfs_boot.h - Boot sector structures and parser declarations */
#ifndef NTFS_BOOT_H
#define NTFS_BOOT_H

#include "ntfs_types.h"

/* The boot sector is exactly one sector (typically 512 bytes).
 * We define the packed structure to match the on-disk layout. */

#pragma pack(push, 1)

typedef struct {
    u8  jump[3];              /* x86 jump instruction: EB 52 90 */
    char oem_id[8];           /* "NTFS    " (padded with spaces) */
    u16 bytes_per_sector;     /* Bytes per sector (typically 512) */
    u8  sectors_per_cluster;  /* Sectors per cluster (power of 2) */
    u16 reserved_sectors;     /* Always 0 for NTFS */
    u8  fat_count;            /* Always 0 for NTFS */
    u16 root_entries;         /* Always 0 for NTFS */
    u16 small_sector_count;   /* Always 0 for NTFS */
    u8  media_descriptor;     /* 0xF8 = fixed disk */
    u16 sectors_per_fat;      /* Always 0 for NTFS */
    u16 sectors_per_track;    /* Sectors per track (CHS) */
```

```
    u16 num_heads;            /* Number of heads (CHS) */
    u32 hidden_sectors;       /* Hidden sectors before partition */
    u32 int_thirty_two;       /* Always 0 for NTFS */
    /* Extended BPB follows */
    u64 total_sectors;        /* Total sectors in volume (64-bit) */
    u64 mft_cluster;          /* First cluster of $MFT file */
    u64 mft_mirr_cluster;     /* First cluster of $MFTMirr file */
    s8  clusters_per_file_record; /* MFT record size encoding */
    u8  unused_41[3];         /* Always 0 */
    s8  clusters_per_index_buffer; /* Index block size encoding */
    u8  unused_45[3];         /* Always 0 */
    u64 volume_serial;        /* Volume serial number */
    u32 checksum;             /* Unused, typically 0 */
    u8  bootstrap_code[426];  /* Bootstrap code (NTLDR/BOOTMGR loader) */
    u16 boot_signature;       /* Must be 0xAA55 */
} ntfs_boot_sector_t;

#pragma pack(pop)

/* Function declarations */
ntfs_status_t ntfs_parse_boot_sector(
    const u8 *sector_data,
    size_t sector_size,
    ntfs_volume_info_t *out_info);

u32 ntfs_compute_mft_record_size(s8 encoding, u32 cluster_size);
u32 ntfs_compute_index_block_size(s8 encoding, u32 cluster_size);

/* Conversion helpers */
static inline u64 byte_offset_to_lcn(u64 offset, u32 cluster_size) {
    return offset / cluster_size;
}

static inline u64 lcn_to_byte_offset(u64 lcn, u32 cluster_size) {
    return lcn * (u64)cluster_size;
}

static inline u64 byte_offset_to_sector(u64 offset, u16 bytes_per_sector) {
    return offset / bytes_per_sector;
}

static inline u64 sector_to_byte_offset(u64 sector, u16 bytes_per_sector) {
    return sector * (u64)bytes_per_sector;
}

#endif /* NTFS_BOOT_H */
```

Now the implementation:

```
/* ntfs_boot.c - Boot sector parsing and validation */
#include "ntfs_boot.h"
#include <string.h>

/* Compute MFT record size from the signed-byte encoding.
 * Positive value = number of clusters.
 * Negative value = 2^|value| bytes. */
u32 ntfs_compute_mft_record_size(s8 encoding, u32 cluster_size) {
    if (encoding > 0) {
        /* Positive: number of clusters per MFT record */
        return (u32)encoding * cluster_size;
    } else {
        /* Negative: 2^|encoding| bytes */
        u32 abs_val = (encoding < 0) ? (u32)(-encoding) : 0;
        if (abs_val > 31) return 0; /* Sanity check */
        return (u32)1 << abs_val;
    }
}

/* Compute index block size from the signed-byte encoding.
 * Same rules as MFT record size. */
u32 ntfs_compute_index_block_size(s8 encoding, u32 cluster_size) {
    if (encoding > 0) {
        return (u32)encoding * cluster_size;
    } else {
        u32 abs_val = (encoding < 0) ? (u32)(-encoding) : 0;
        if (abs_val > 31) return 0;
        return (u32)1 << abs_val;
    }
}

/* Parse and validate the boot sector.
 * Returns NTFS_OK on success, or an error code.
 * The caller must provide a buffer of at least 512 bytes. */
ntfs_status_t ntfs_parse_boot_sector(
    const u8 *sector_data,
    size_t sector_size,
    ntfs_volume_info_t *out_info)
{
    if (!sector_data || !out_info) {
        return NTFS_ERROR_INVALID_PARAM;
    }

    /* The boot sector must be at least 512 bytes */
    if (sector_size < 512) {
        return NTFS_ERROR_CORRUPT;
    }

    /* Zero-fill the output structure */
    memset(out_info, 0, sizeof(ntfs_volume_info_t));
```

```
51
52      /* Check the boot signature first */
53      u16 sig = sector_data[510] | (u16)(sector_data[511]) << 8;
54      if (sig != BOOT_SIGNATURE) {
55          return NTFS_ERROR_CORRUPT;
56      }
57
58      /* Verify the OEM ID is "NTFS    " */
59      if (memcmp(sector_data + 3, "NTFS    ", 8) != 0) {
60          return NTFS_ERROR_CORRUPT;
61      }
62
63      /* Read the BPB fields. We read them individually rather than
64       * casting the pointer to avoid alignment issues on strict
65       * alignment architectures. All values are little-endian. */
66
67      out_info->bpb.bytes_per_sector =
68          sector_data[0x0B] | (u16)(sector_data[0x0C]) << 8;
69
70      out_info->bpb.sectors_per_cluster = sector_data[0x0D];
71
72      out_info->bpb.sectors_per_track =
73          sector_data[0x18] | (u16)(sector_data[0x19]) << 8;
74
75      out_info->bpb.heads =
76          sector_data[0x1A] | (u16)(sector_data[0x1B]) << 8;
77
78      out_info->bpb.hidden_sectors =
79          sector_data[0x1C] | (u32)(sector_data[0x1D]) << 8 |
80          (u32)(sector_data[0x1E]) << 16 | (u32)(sector_data[0x1F]) << 24;
81
82      /* Total sectors (64-bit at offset 0x28) */
83      out_info->bpb.total_sectors =
84          sector_data[0x28] | (u64)(sector_data[0x29]) << 8 |
85          (u64)(sector_data[0x2A]) << 16 | (u64)(sector_data[0x2B]) << 24 |
86          (u64)(sector_data[0x2C]) << 32 | (u64)(sector_data[0x2D]) << 40 |
87          (u64)(sector_data[0x2E]) << 48 | (u64)(sector_data[0x2F]) << 56;
88
89      /* $MFT cluster number (64-bit at offset 0x30) */
90      out_info->bpb.mft_cluster =
91          sector_data[0x30] | (u64)(sector_data[0x31]) << 8 |
92          (u64)(sector_data[0x32]) << 16 | (u64)(sector_data[0x33]) << 24 |
93          (u64)(sector_data[0x34]) << 32 | (u64)(sector_data[0x35]) << 40 |
94          (u64)(sector_data[0x36]) << 48 | (u64)(sector_data[0x37]) << 56;
95
96      /* $MFTMirr cluster number (64-bit at offset 0x38) */
97      out_info->bpb.mft_mirr_cluster =
98          sector_data[0x38] | (u64)(sector_data[0x39]) << 8 |
99          (u64)(sector_data[0x3A]) << 16 | (u64)(sector_data[0x3B]) << 24 |
100         (u64)(sector_data[0x3C]) << 32 | (u64)(sector_data[0x3D]) << 40 |
```

```
        (u64)(sector_data[0x3E]) << 48 | (u64)(sector_data[0x3F]) << 56;

    /* Clusters per file record segment at offset 0x40 */
    out_info->bpb.clusters_per_file_record = (s8)sector_data[0x40];

    /* Clusters per index buffer at offset 0x44 */
    out_info->bpb.clusters_per_index_buffer = (s8)sector_data[0x44];

    /* Volume serial number (64-bit at offset 0x48) */
    out_info->bpb.volume_serial =
        sector_data[0x48] | (u64)(sector_data[0x49]) << 8 |
        (u64)(sector_data[0x4A]) << 16 | (u64)(sector_data[0x4B]) << 24 |
        (u64)(sector_data[0x4C]) << 32 | (u64)(sector_data[0x4D]) << 40 |
        (u64)(sector_data[0x4E]) << 48 | (u64)(sector_data[0x4F]) << 56;

    /* Validate bytes per sector */
    u16 bps = out_info->bpb.bytes_per_sector;
    if (bps < 512 || bps > 4096 || (bps & (bps - 1)) != 0) {
        return NTFS_ERROR_CORRUPT;
    }

    /* Validate sectors per cluster */
    u8 spc = out_info->bpb.sectors_per_cluster;
    if (spc == 0 || (spc & (spc - 1)) != 0) {
        return NTFS_ERROR_CORRUPT;
    }

    /* Compute derived values */
    out_info->cluster_size = bps * spc;
    out_info->total_clusters = out_info->bpb.total_sectors / spc;
    out_info->volume_size_bytes = out_info->bpb.total_sectors * bps;
    out_info->mft_record_size =
        ntfs_compute_mft_record_size(
            out_info->bpb.clusters_per_file_record,
            out_info->cluster_size);
    out_info->index_block_size =
        ntfs_compute_index_block_size(
            out_info->bpb.clusters_per_index_buffer,
            out_info->cluster_size);

    /* Sanity checks on computed values */
    if (out_info->mft_record_size < 256) {
        return NTFS_ERROR_CORRUPT;
    }
    if (out_info->total_clusters == 0) {
        return NTFS_ERROR_CORRUPT;
    }

    return NTFS_OK;
}
```

The parsing function reads each field individually from the raw byte buffer rather than casting to a struct pointer. This avoids alignment issues on architectures that require natural alignment for multi-byte types, and it makes the little-endian byte ordering explicit. In production code you might use helper macros or compiler builtins like `__builtin_bswap16` for endianness conversion, but the manual approach shown here is portable and clear.

Volume Identification and Signatures

NTFS uses several signatures to identify valid structures. The boot sector signature 0xAA55 at offset 0x1FE is the most basic check. The OEM ID "NTFS " at offset 0x03 confirms the filesystem type. Later we will encounter the MFT record magic "FILE" (0x454C4946 in little-endian) and various other signatures throughout the filesystem structures.

The volume serial number at offset 0x48 is an 8-byte value assigned when the volume is formatted. On the command line, Windows displays only the lower 32 bits in the familiar XXXX-XXXX format (e.g., "Volume Serial Number is A4E1-5DFC"). The full 64-bit value can be used to uniquely identify a volume across systems.

With the boot sector parsed, we now know the cluster size, the total volume size, the MFT record size, and most importantly, where the Master File Table lives on disk. The next chapter reads the MFT and begins decoding the actual file records that describe every file on the volume.

Chapter 2: The Master File Table

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

MFT Architecture and Location

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

File Record Segment Header

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

Attribute Types and the Type Registry

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

Resident Attributes

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

Non-Resident Attributes

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

Parsing File Records (Code)

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

Chapter 3: Core Attributes - Standard Information, Filename, Security

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

The $STANDARD_INFORMATION Attribute

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

The $FILE_NAME Attribute and Unicode Filenames

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

Security Descriptors and DACLs

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

SID Structure and Parsing

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

Building the Metadata Handler (Code)

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

Chapter 4: Data Attributes and Runlists

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

The $DATA Attribute

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

Runlist Encoding and Decoding

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

Parsing Non-Resident Data (Code)

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

Sparse Files and Zero Runs

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

Compression on NTFS

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

Chapter 5: Directories and B+ Tree Indexing

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

The $INDEX_ROOT Attribute

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

B+ Tree Architecture in NTFS

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

Index Entries and Collation Rules

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

The $INDEX_ALLOCATION Attribute

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

Directory Traversal (Code)

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

Chapter 6: Allocation Structures - Bitmaps, Clusters, and the Allocator

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

The $BITMAP Attribute

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

Cluster Allocation Strategies

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

Free Space Search Algorithms

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

Bad Cluster Management ($BAD_CLUSTERS)

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

Building the Allocator (Code)

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

Chapter 7: Links, Reparse Points, and Extended Attributes

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

Hard Links and MFT Reference Counting

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

Symbolic Links and Junction Points

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

Reparse Points and the Filter Architecture

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

Alternate Data Streams (ADS)

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

Object IDs and Extended Attributes

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

Chapter 8: Transaction Logging and Recovery

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

The NTFS Log File ($LogFile)

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

Write-Ahead Logging Protocol

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

Log Record Formats

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

Crash Recovery Algorithm

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

Building the Journaling Subsystem (Code)

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

Chapter 9: Special Files and System Metadata

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

The Metadata File Registry

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

$Volume and Volume Flags

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

$AttrDef - Attribute Definitions

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

$Quota - Disk Quotas

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

$UpCase and $Extend Tables

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

Chapter 10: The USN Journal and Change Tracking

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

USN Journal Architecture

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

USN Record Formats

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

Querying the Journal

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

Forensic and Monitoring Applications

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

Chapter 11: Encryption, Quotas, and Advanced Features

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

Encrypting File System (EFS) Architecture

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

FVEK and DEK Encryption Scheme

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

Disk Quota Enforcement

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

Volume Caching Considerations

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

Consistency Checking (chkdsk Logic)

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

Chapter 12: Version Evolution and Compatibility

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

NTFS 1.0 - Windows NT 3.1

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

NTFS 2.0-3.0 - Major Feature Additions

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

NTFS 3.1 and Later Evolutions

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

Cross-Version Compatibility Matrix

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

Interoperability with Other Platforms

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

Conclusion: The Complete Implementation

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

What We Built

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

What a Production Implementation Needs

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

Where to Go Next

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

References

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