An In-Depth Guide to FAT32 for File Recovery Software Development

Part 1: Foundations of Low-Level Disk Access

The development of file recovery software requires a fundamental departure from conventional programming practices. Standard file input/output (I/O) operations are mediated by the operating system, which presents a high-level, abstracted view of data as a logical hierarchy of files and directories. This abstraction, while convenient for general application development, conceals the underlying physical layout of data on the storage medium. To recover data that the operating system no longer recognizes—such as deleted files or data from a corrupted volume—it is imperative to bypass this abstraction and interact directly with the raw, uninterpreted bytes stored on the device. This foundational section details the physical organization of data on a disk and the specific programming techniques required to achieve this low-level access. This shift in perspective, from the logical file to the physical sector, is the absolute prerequisite for analyzing and manipulating file system structures for the purpose of data recovery.

1.1 The Physical Landscape: Sectors, Clusters, and Logical Block Addressing (LBA)

At the lowest level, data on a storage device like a USB pendrive is organized into discrete units. The smallest addressable unit of storage on a disk is called a **sector**. For the vast majority of modern storage devices, including those formatted with FAT32, the sector size is fixed at 512 bytes.[1]

While the sector is the smallest unit the hardware can read or write, the file system manages space in larger units called **clusters**. A cluster is the smallest allocatable unit of disk space and consists of a fixed, power-of-two number of contiguous sectors.[1] For example, a cluster might be composed of 1, 2, 4, 8, 16, 32, 64, or 128 sectors. The size of a cluster is a compromise: larger clusters reduce the size of the metadata needed to manage the disk (the File Allocation Table) but increase the amount of wasted space for small files, a phenomenon known as slack space.[3]

To locate these units, modern systems use a simple and efficient addressing scheme called **Logical Block Addressing (LBA)**. LBA treats the entire storage device as a linear array of sectors, numbered sequentially from 0 to N-1, where N is the total number of sectors on the device.[2] This model superseded the older and more complex Cylinder/Head/Sector (CHS) addressing scheme, which was tied to the physical geometry of hard disk platters and imposed significant limitations on disk capacity.[3] All subsequent analysis and calculations in this guide will be based on the LBA model.

1.2 Bypassing the Operating System: Principles of Raw Disk Access

Standard file system APIs are designed to prevent applications from accessing arbitrary locations on a disk, particularly areas that are not currently allocated to a file. This is a critical security and stability feature. However, for data recovery, this is precisely what is required. The data of a deleted file often remains physically present on the disk in what the file system now considers "unallocated space".[4] To access this data, or to inspect the file system's own metadata structures for forensic analysis, the software must circumvent the standard file I/O layer.[6]

On the Windows operating system, this is achieved by opening a special device path that provides a handle directly to the storage device driver. There are two primary forms of this path [8]:

- \\.\PhysicalDriveN: This path provides access to the entire physical disk, starting from the Master Boot Record (MBR) at LBA 0. N is the zero-based index of the physical disk in the system (e.g., PhysicalDrive0, PhysicalDrive1).
- \\.\X:: This path provides access to a specific volume or partition on a disk, identified by its drive letter X. This access begins at the first sector of that partition, which for a FAT32 volume is its Volume Boot Record (VBR).

Accessing these device paths requires elevated privileges. The recovery software must be run with Administrator rights to successfully open a handle to a raw device.[8]

1.3 Implementation Primer: Achieving Raw Disk Access with Python

on Windows

Python's built-in functions can be used to perform raw disk access on Windows, provided the correct device paths and file modes are used. The key is to treat the raw device as a binary file.

To obtain a file handle to a physical device, one can use the open() function. The device path must be correctly formatted, and the mode should be for binary reading ('rb') or binary reading and writing ('rb+').[8]

Example 1: Opening a Logical Volume (e.g., E:)

Python

```python
# Administrator privileges are required to run this code.
volume_path = r'\\.\E:'
try:
    with open(volume_path, 'rb') as disk:
        # The 'disk' object is now a handle to the raw volume.
        # Read the first sector (512 bytes), which is the VBR.
        vbr_data = disk.read(512)
        print(f"Successfully read {len(vbr_data)} bytes from {volume_path}")
except PermissionError:
    print("Permission denied. Please run the script as an Administrator.")
except FileNotFoundError:
    print(f"Device {volume_path} not found.")
except Exception as e:
    print(f"An error occurred: {e}")
```

Once a handle is obtained, the .seek() and .read() methods are used to navigate the disk. The .seek() method moves the read/write cursor to a specific byte offset from the beginning of the device. To move to a specific LBA, the offset is calculated as offset_in_bytes = lba_address * bytes_per_sector. The .read() method then reads a specified number of bytes from that position.[9]

Example 2: Seeking to and Reading a Specific Sector

Python

```python
# Assuming a sector size of 512 bytes.
BYTES_PER_SECTOR = 512
LBA_TO_READ = 6 # The typical location of the backup boot sector.

# Code to open the disk handle as in Example 1...
# with open(volume_path, 'rb') as disk:
#     target_offset = LBA_TO_READ * BYTES_PER_SECTOR
#     disk.seek(target_offset)
#     sector_data = disk.read(BYTES_PER_SECTOR)
#     print(f"Read {len(sector_data)} bytes from LBA {LBA_TO_READ}")
```

A common pitfall when attempting to write to a raw device is the IOError: [Errno 22] Invalid argument. This error often occurs if the write operation is not aligned to a sector boundary or if the size of the data being written is not a multiple of the sector size.[8] While our primary focus is on reading for recovery, understanding these constraints is crucial for any potential repair functionality.

For more advanced interactions, such as locking a volume to prevent other processes from writing to it during analysis, the pywin32 library can be used to interface with the Windows API directly. However, for read-only recovery operations, the built-in open() function is sufficient and simpler.[7]

Part 2: Anatomy of a FAT32 Volume

Having established the means to access the raw sectors of a USB pendrive, the next step is to interpret the data they contain. A FAT32 volume is not an arbitrary collection of bytes; it is a highly structured entity with a specific layout. Understanding this layout is akin to learning the grammar of a new language. The Volume Boot Record (VBR), located at the very beginning of the volume, acts as a self-describing blueprint. It contains a set of parameters that define the geometry of the entire file system, enabling software to calculate the precise location and size of every other critical component. This self-contained design is what makes FAT32 portable and relatively straightforward to parse.

2.1 The Grand Architecture: Reserved, FAT, and Data Regions

A FAT32 volume is partitioned into three distinct, sequential regions [1]:

1. **Reserved Region:** This region always starts at the beginning of the volume (LBA 0). Its primary component is the Volume Boot Record (VBR). For FAT32, this region is larger than in previous FAT versions and also contains other vital structures, such as the File System Information (FSInfo) sector and a backup copy of the boot sector.[2] The size of this region is variable and is specified within the VBR itself.
2. **FAT Region:** This region immediately follows the Reserved Region. It contains the File Allocation Tables. FAT32 volumes almost universally have two identical copies of the FAT to provide redundancy against data corruption.[2] The second FAT is a mirror of the first.
3. **Data Region:** This is the largest part of the volume and follows the FAT Region. It is where the actual content of files and the metadata for directories are stored. Space in this region is managed in units of clusters.[14] The root directory of a FAT32 volume is located within this region as a standard cluster chain.[17]

2.2 The Volume Boot Record (VBR): The Filesystem's Rosetta Stone

The VBR is the single most important data structure on a FAT32 volume. It is always located in the first sector of the partition (LBA 0).[3] It contains boot code, an OEM identifier, and, most importantly, the **BIOS Parameter Block (BPB)**. The BPB is a data structure that provides the fundamental parameters needed to interpret the rest of the volume, such as sector size, cluster size, and the locations of the FATs.[1] A valid VBR is identified by a 2-byte signature, 0x55AA, located at the very end of the sector (byte offset 510).[16]

The multi-byte integer fields within the BPB are stored in **little-endian** byte order, meaning the least significant byte is stored first. When parsing these fields, the bytes must be read and interpreted accordingly. The following table details the critical fields within the FAT32 BPB.

Byte Offset (Hex)	Field Name	Size (Bytes)	Description
0x0B	BPB_BytsPerSec	2	Bytes Per Sector. Almost universally 512 for USB drives.[1]
0x0D	BPB_SecPerClus	1	Sectors Per Cluster. Must be a power of

			2 (1, 2, 4,..., 128).[1]
0x0E	BPB_RsvdSecCnt	2	Number of Reserved Sectors. This defines the size of the Reserved Region and the starting LBA of the first FAT.[1] Usually 32 for FAT32.
0x10	BPB_NumFATs	1	Number of File Allocation Tables. Almost always 2.[1]
0x11	BPB_RootEntCnt	2	Must be 0 for FAT32. The root directory is a cluster chain, not a fixed-size area.[1]
0x13	BPB_TotSec16	2	Must be 0 for FAT32. The 4-byte field at 0x20 is used instead.
0x15	BPB_Media	1	Media Descriptor. 0xF8 for fixed media (hard disks) and 0xF0 for removable media.[1]
0x16	BPB_FATSz16	2	Must be 0 for FAT32. The 4-byte field at 0x24 is used instead.[1]
0x20	BPB_TotSec32	4	Total count of sectors on the volume. This is the 32-bit version of

			the field.[16]
0x24	BPB_FATSz32	4	The 32-bit count of sectors occupied by ONE FAT.[1]
0x2C	BPB_RootClus	4	The cluster number of the first cluster of the root directory. This is usually 2.[15]
0x30	BPB_FSInfo	2	Sector number of the FSInfo structure within the reserved area. Typically 1.[1]
0x32	BPB_BkBootSec	2	Sector number of the backup copy of the boot sector. Typically 6.[1]
0x1FE	Signature_word	2	The boot signature, which must be 0xAA55 (read as 55 AA in memory).[16]

Table 1: FAT32 BIOS Parameter Block (BPB) Layout

From these few fields, a program can derive the location of every major region on the volume:

- **Start of First FAT (LBA):** BPB_RsvdSecCnt
- **Start of Second FAT (LBA):** BPB_RsvdSecCnt + BPB_FATSz32
- **Start of Data Region (LBA):** BPB_RsvdSecCnt + (BPB_NumFATs * BPB_FATSz32)

The FAT32 Reserved Region also contains two other important structures:

- **FSInfo Sector:** Usually located at LBA 1, this sector stores hints for the operating system, such as the last known number of free clusters and the cluster number where the OS should start searching for a free cluster. While useful for improving write performance, this information can be out of sync with the actual FAT and should be treated as advisory, not authoritative, during recovery.[1]
- **Backup Boot Sector:** A key reliability feature of FAT32 is the presence of a backup copy of the first three sectors of the volume (the boot sector code), typically located starting

at LBA 6.[1] This provides a critical path for recovery if the primary VBR at LBA 0 becomes corrupted or unreadable. A robust recovery tool must not assume the VBR at LBA 0 is valid; it should verify its integrity (e.g., check the 0x55AA signature) and, if it is corrupt, attempt to read the backup VBR from the location specified in BPB_BkBootSec. If the primary VBR is damaged but the backup is intact, the volume can often be made accessible again simply by copying the backup over the primary.

2.3 The File Allocation Table (FAT): A Map of the Data

The File Allocation Table is the heart of the file system's data management. It is effectively a large array that functions as a map for the Data Region. The location of the first FAT is given by BPB_RsvdSecCnt. The total size of the FAT region is calculated by multiplying BPB_NumFATs by BPB_FATSz32.[2]

The FAT is an array of 32-bit (4-byte) entries. Each entry in the FAT corresponds one-to-one with a cluster in the Data Region. The value stored in a FAT entry describes the status of its corresponding cluster. The first two entries (FAT and FAT) are reserved and do not correspond to any data cluster. Data clusters are numbered starting from 2, which corresponds to the third entry in the FAT (FAT).[3]

The FAT is used to track which clusters are free and to link together the clusters that belong to a single file, forming a **cluster chain**. The values in a FAT entry are interpreted as follows [3]:

- 0x00000000: The cluster is free and available for allocation.
- 0x00000002 to 0x0FFFFFF6: The cluster is in use, and the value is the cluster number of the *next* cluster in the file's chain.
- 0x0FFFFFF7: The cluster is a "bad cluster" marked by the disk utility and should not be used.
- 0x0FFFFFF8 to 0x0FFFFFFF: This is an End of Chain (EOC) marker, indicating that this is the last cluster in the file's chain.

To read a file that occupies multiple clusters, the software first finds the file's starting cluster number from its directory entry. It then looks up that cluster's entry in the FAT. If the value is an EOC marker, the file has only one cluster. If the value points to another cluster, the software reads that next cluster and repeats the process, following the chain through the FAT until it encounters an EOC marker.

Part 3: Files, Directories, and Metadata

With an understanding of the volume's high-level structure, the focus now shifts to the Data Region, where files and directories are stored. Every file and directory on the volume is represented by a 32-byte data structure called a **directory entry**. These entries contain all the metadata about the object, such as its name, size, attributes, and, most critically, the starting cluster number where its data begins. The directory entry serves as the bridge between the logical concept of a "file" and its physical location on the disk. While the user-friendly long filename is a prominent feature, it is the associated, older-style Short File Name (SFN) entry that holds the authoritative metadata. All essential information—timestamps, attributes, file size, and the starting cluster—resides exclusively in this SFN entry. The long filename entries are merely descriptive additions; without their corresponding SFN entry, they are meaningless, and the file's data becomes disconnected from its name.

3.1 The 32-Byte Directory Entry: An In-Depth Analysis

Directory entries are stored within the data clusters allocated to a directory. Each entry is a fixed 32 bytes long.[21] To read a directory's contents, software reads the cluster(s) belonging to that directory and parses them as a sequence of these 32-byte structures. The table below provides a detailed breakdown of the Short File Name (SFN) directory entry.

Byte Offset (Hex)	Field Name	Size (Bytes)	Description
0x00	DIR_Name	11	Short File Name (SFN) in 8.3 format (8 bytes for name, 3 for extension). The first byte has special meanings: 0xE5 = Deleted Entry, 0x00 = End of Directory List, 0x2E = . or .. entry.[3]
0x0B	DIR_Attr	1	File Attributes. A bitmask indicating properties of the

			entry. See sub-table below.[21]
0x0C	DIR_NTRes	1	Reserved for use by Windows NT.
0x0D	DIR_CrtTimeTenth	1	Millisecond stamp at file creation time.
0x0E	DIR_CrtTime	2	Time file was created.
0x10	DIR_CrtDate	2	Date file was created.
0x12	DIR_LstAccDate	2	Last accessed date.
0x14	DIR_FstClusHI	2	High two bytes of the entry's first cluster number.[2]
0x16	DIR_WrtTime	2	Time of last write.
0x18	DIR_WrtDate	2	Date of last write.
0x1A	DIR_FstClusLO	2	Low two bytes of the entry's first cluster number.[3]
0x1C	DIR_FileSize	4	32-bit file size in bytes. This value is 0 for directories.[22]

Table 2: FAT32 Directory Entry Structure

The DIR_Attr byte at offset 0x0B is a bitfield where each bit represents a specific attribute:

Bit	Value (Hex)	Attribute

0	0x01	ATTR_READ_ONLY
1	0x02	ATTR_HIDDEN
2	0x04	ATTR_SYSTEM
3	0x08	ATTR_VOLUME_ID (Volume Label)
4	0x10	ATTR_DIRECTORY
5	0x20	ATTR_ARCHIVE

If the attribute byte has the special value 0x0F, which corresponds to ATTR_READ_ONLY | ATTR_HIDDEN | ATTR_SYSTEM | ATTR_VOLUME_ID, the entry is not a standard file or directory entry. Instead, it is a **Long File Name (LFN)** entry, which is discussed in section 3.3.

The 32-bit starting cluster number for the file or directory is formed by combining the high and low two-byte fields: Starting Cluster = (DIR_FstClusHI << 16) | DIR_FstClusLO.

3.2 Navigating the Hierarchy: From the Root Directory to Subdirectories

In older FAT systems, the root directory was stored in a special, fixed-size area. A significant improvement in FAT32 is that the root directory is a standard cluster chain, just like any other directory. This allows it to grow to any size.[15] The starting cluster number for the root directory is stored in the BPB_RootClus field of the VBR, which is typically cluster 2.[15]

To list the contents of any directory (including the root), the process is as follows:

1. Obtain the directory's starting cluster number (from the VBR for the root, or from its parent directory's entry for a subdirectory).
2. Calculate the LBA of that cluster. The formula is: FirstSectorofCluster = StartOfDataRegionLBA + ((ClusterNumber - 2) * SectorsPerCluster).
3. Read all sectors belonging to that cluster.
4. Iterate through the data in 32-byte chunks, parsing each as a directory entry.
5. If a directory entry's first byte is 0x00, this marks the end of the directory listing, and no further entries are valid.

6. To handle directories that span multiple clusters, follow the cluster chain for the directory in the FAT until an EOC marker is found.

Subdirectories (but not the root directory) contain two special entries: . ("dot") and .. ("dot dot"). The . entry points to the starting cluster of the directory itself, while the .. entry points to the starting cluster of the parent directory. These allow for relative path navigation within the file system hierarchy.[21]

3.3 The Duality of Filenames: Short File Names (SFN) and Long File Names (LFN)

FAT32 supports two types of filenames simultaneously for backward compatibility. The **Short File Name (SFN)**, also known as an 8.3 name, is the legacy format from MS-DOS, consisting of up to 8 characters for the name and 3 for the extension.[3] This SFN is stored directly in the DIR_Name field of the primary 32-byte directory entry.

To support modern, longer filenames, FAT employs a clever but "kludgey" system of **Long File Name (LFN)** entries.[3] When a file with a long name is created, the file system generates a corresponding SFN (e.g., LONGFI~1.TXT for My Long Filename.txt) and stores it in a standard directory entry. The actual long name is then stored, in reverse order, in one or more LFN entries that are placed immediately *before* the SFN entry in the directory listing.[21]

These LFN entries are distinguished by having the special 0x0F attribute.[3] Each LFN entry contains:

- A sequence number in the first byte. The first LFN entry (closest to the SFN entry) is numbered 1, the next is 2, and so on. The last LFN entry (furthest from the SFN entry) has its sequence number OR-ed with 0x40.
- Portions of the long filename, stored in 16-bit Unicode (UTF-16) format, spread across three non-contiguous fields within the 32-byte entry.[3]
- A checksum calculated from the SFN, which links the LFN entries to their corresponding SFN entry.[21]

To reconstruct a long filename, software must start at the SFN entry, then read the preceding directory entries backwards. It collects the filename characters from each LFN entry until it finds the one marked as the last in the sequence. The characters are then concatenated in the correct order to form the full, original long filename.

Part 4: The Art and Science of File Recovery

The preceding sections have established the theoretical foundation of the FAT32 file system. This knowledge is now applied to the practical challenge of data recovery. The effectiveness of any recovery attempt hinges on a precise understanding of what occurs during file deletion. The process is not one of erasure, but of de-referencing. The file's metadata is altered to mark it as deleted, and the space it occupied is marked as available. This leaves the actual data intact, at least temporarily, creating a window of opportunity for recovery.

There are two fundamentally different philosophies for approaching this task. The first, metadata-based undeletion, is an elegant and precise method that leverages the remnants of the file system's own structures to restore a file. It is fast and can preserve original filenames, but it is fragile and depends entirely on the integrity of the deleted file's directory entry. The second, file carving, is a brute-force approach that ignores the file system entirely, instead scanning the raw disk for recognizable data patterns. It is robust and can find files even when all metadata is lost, but it is slow, computationally intensive, and loses all context, such as filenames and directory structure. An effective recovery tool must not choose one over the other but should instead implement a tiered strategy, using the appropriate technique based on the state of the file system and the nature of the data loss.

4.1 The Deletion Process: A Forensic Examination

When a file is deleted from a FAT32 volume under a standard operating system like Windows, the data is not immediately overwritten with zeros. Instead, the file system performs two key modifications to mark the file as deleted and its space as free [23]:

1. **Directory Entry Modification:** The first byte of the file's Short File Name (SFN) directory entry is changed to the hexadecimal value 0xE5 (the lowercase Greek letter sigma).[3] This single-byte flag signals to the file system that the entry is no longer in use and can be overwritten by a new entry when a new file is created in that directory. The remaining 31 bytes of the directory entry, which contain the rest of the filename, the file size, attributes, and the starting cluster number, are typically left unchanged.
2. **File Allocation Table (FAT) Zeroing:** The chain of entries in the FAT that corresponded to the file's allocated clusters is cleared. Each entry in the chain is overwritten with 0x00000000, marking those clusters as free and available for new data.[25] This action effectively severs the link between the file's clusters, which is particularly problematic for fragmented files.

A critical nuance exists specifically for FAT32 file systems that complicates recovery. While the classic FAT deletion process leaves the starting cluster number intact in the directory entry, some operating systems (notably Windows versions from Vista onwards) have been observed to perform an additional step: zeroing out the high word of the starting cluster number (DIR_FstClusHI at offset 0x14) upon deletion.[27] This action effectively destroys the pointer to the start of the file's data for any file that began at or after cluster 65,536 (2^{16}). For large modern USB drives, this applies to the vast majority of the storage area. This behavior severely hinders simple undeletion techniques, as the starting point of the file's data is lost even if the rest of the directory entry is perfectly preserved. A robust recovery tool must be designed to anticipate this possibility and cannot rely solely on the presence of a valid starting cluster in a deleted entry.

4.2 Recovery Strategy I: Metadata-Based Undeletion

This strategy attempts to reverse the deletion process by using the information remaining in the 0xE5-marked directory entry. It is most effective for recently deleted, non-fragmented files.

Algorithm for Contiguous File Undeletion:

1. **Scan for Deleted Entries:** Navigate to the target directory (or scan all directories, including the root) and read its contents cluster by cluster. Iterate through the 32-byte entries, looking for any that begin with the 0xE5 marker.
2. **Parse Metadata:** For each 0xE5 entry found, parse the remaining 31 bytes to extract the filename (with the first character missing), the file attributes (DIR_Attr), the file size (DIR_FileSize), and the starting cluster number (reconstructed from DIR_FstClusHI and DIR_FstClusLO).
3. **Validate Starting Cluster:** This is a crucial step. Check if the recovered starting cluster number is valid. A valid cluster number must be greater than or equal to 2 and less than the total number of clusters on the volume. If the high word (DIR_FstClusHI) was zeroed during deletion, this check will likely fail for files on larger volumes, rendering this method ineffective for that specific file. If the cluster number is 0 or 1, the entry is invalid.[28]
4. **Calculate Data Location and Size:** If the starting cluster is valid, calculate the number of clusters the file occupies using the formula: NumClusters = ceil(DIR_FileSize / BytesPerCluster). Then, calculate the starting LBA of the file's data on the disk.
5. **Extract Data:** Read NumClusters worth of contiguous clusters from the disk, starting from the calculated LBA. This assumes the file was not fragmented. Since the FAT chain has been zeroed, there is no way to follow the links for a fragmented file using this method.
6. **Save Recovered File:** Write the extracted data to a new file in a safe location (i.e., on a

different storage device to avoid overwriting other recoverable data). The filename can be reconstructed with a placeholder for the first character (e.g., _ILE.TXT), which the user can later correct.

Challenges:

- **Filename Ambiguity:** If two deleted files in the same directory had names differing only in the first character (e.g., REPORT.DOC and DEPORT.DOC), they will both appear as ?EPORT.DOC. Without additional information, such as a user-provided SHA-1 hash of the original file to verify against the recovered content, it can be impossible to distinguish between them.[29]
- **Fragmentation:** This method's greatest weakness is fragmentation. It can only recover the first fragment of a non-contiguous file, resulting in a truncated and likely corrupted recovery.

4.3 Recovery Strategy II: File Carving

When metadata is missing or unreliable (e.g., the directory entry has been overwritten, or the starting cluster was zeroed), file carving is the necessary alternative. This technique operates independently of the file system, treating the entire disk (or its unallocated space) as a raw sequence of bytes.[4]

Principles of Signature-Based Recovery:

File carving works by scanning for **file signatures**, also known as "magic numbers." These are unique sequences of bytes found at the beginning (header) and sometimes the end (footer) of files of a specific type. For example, all JPEG files begin with the hexadecimal bytes FF D8 FF E0 or FF D8 FF E1, and they end with FF D9.[30] By identifying these signatures, a tool can "carve out" the data between them.

Implementation Steps:

1. **Build a Signature Database:** The first step is to create a database of known file signatures for the types of files the tool should recover. This database should contain the file type, common extensions, the header signature, and the footer signature (if one exists and is reliable).[31]

File Type	Extension	Header (Hex Signature)	Footer (Hex Signature)

JPEG Image	.jpg,.jpeg	FF D8 FF E0 or FF D8 FF E1	FF D9
PNG Image	.png	89 50 4E 47 0D 0A 1A 0A	49 45 4E 44 AE 42 60 82
GIF Image	.gif	47 49 46 38 37 61 or 47 49 46 38 39 61	00 3B
PDF Document	.pdf	25 50 44 46	25 25 45 4F 46
ZIP Archive	.zip,.docx,.xlsx	50 4B 03 04	50 4B 05 06 (Central Directory)

Table 3: Common File Signatures for Carving

2. **Scan the Device:** Read the USB drive sector by sector (or in larger, more efficient chunks) into a memory buffer.
3. **Search for Headers:** Search the buffer for any of the header signatures defined in the database.
4. **Begin Carving:** When a header is found, record its starting offset. Begin writing the data from that point forward into a temporary recovery file.
5. **Find the End of the File:** Continue reading from the source device and writing to the recovery file. This process stops when one of three conditions is met:
    - A corresponding footer signature is found. This is the ideal scenario and results in the most accurate recovery.[30]
    - A new header signature for another file is found. This often indicates the end of the current file and the beginning of the next.
    - A pre-defined maximum file size is reached. This is a safety mechanism to prevent the recovery process from creating enormous files if a footer is missing or corrupted.

Limitations:
File carving is a powerful but imperfect technique. Its primary drawbacks are:

- **Loss of Metadata:** It cannot recover original filenames, timestamps, or directory structures. Recovered files are typically given generic names like file0001.jpg, file0002.pdf, etc..[34]
- **Fragmentation:** Like simple undeletion, standard carving techniques perform poorly with fragmented files. The tool will typically carve only the first contiguous fragment of the file, from the header until the fragment ends, resulting in an incomplete file.[5] Advanced carving algorithms exist that use file-structure-specific heuristics to attempt to

reassemble fragments, but this is a significantly more complex problem.[5]

Part 5: Synthesis and Future Work

The journey from understanding raw disk sectors to implementing file recovery algorithms is a significant undertaking. This guide has provided the necessary theoretical and practical knowledge for Phase 1: developing a functional file recovery tool for the FAT32 file system. This final section synthesizes this knowledge into a concrete development roadmap and looks ahead to the more advanced challenges that lie beyond this initial scope.

5.1 A Roadmap for Your Recovery Tool

A structured, modular approach is recommended for developing the recovery software. Each component can be built and tested independently before being integrated into the final application.

- **Step 1: Core Library for Raw Disk Access:** The foundation of the entire application is a module dedicated to low-level disk I/O. This module should contain functions or a class that can:
  - Enumerate available physical drives and logical volumes on the system.
  - Open a specified device using the \\.\PhysicalDriveN or \\.\X: path with the necessary permissions.
  - Provide reliable methods to seek() to a specific byte offset or LBA and read() a specified number of bytes into a buffer.
- **Step 2: VBR and BPB Parser:** Create a dedicated parser that reads the first sector (LBA 0) of a volume and populates a data structure (e.g., a Python class or dictionary) with the parameters from the BIOS Parameter Block. This parser must correctly handle little-endian byte order. The resulting object will serve as the configuration for all subsequent operations on that volume. It should also include logic to check for and use the backup VBR if the primary one is corrupted.
- **Step 3: FAT and Directory Traversal Engine:** Implement the logic for navigating the file system. This includes:
  - A function to read and cache the File Allocation Table.
  - A function that, given a starting cluster number, can follow a cluster chain through the FAT.
  - A function to parse the 32-byte entries within a directory's data cluster, capable of reconstructing both Short and Long File Names.

- **Step 4: Metadata-Based Undelete Module:** Build the first recovery module based on the algorithm in Section 4.2. This module will use the directory traversal engine to scan for 0xE5-marked entries and attempt to recover them based on their stored metadata. It must be designed to handle the case where the high word of the starting cluster may have been zeroed.
- **Step 5: File Carving Module:** Implement the second recovery module based on the signature-based carving algorithm from Section 4.3. This will require:
    - A configurable database of file signatures (headers and footers).
    - An efficient engine for scanning large amounts of raw data for these byte patterns.
- **Step 6: User Interface (UI):** Develop a user interface to control the recovery process. A command-line interface (CLI) is a practical starting point.[29] The UI should allow the user to:
    - Select the target drive to scan.
    - Choose the recovery method (e.g., a "quick scan" for metadata-based undeletion or a "deep scan" for file carving).
    - Specify a safe output directory for recovered files.

5.2 Beyond Phase 1: Advanced Challenges

Mastering FAT32 recovery is a significant achievement, but it is also a gateway to more complex challenges in the field of data recovery and digital forensics.

- **Fragmented File Recovery:** The most significant limitation of the methods described is their inability to reliably recover fragmented files once the FAT chain is lost. Advanced carving techniques, sometimes called SmartCarving or heuristic-based recovery, attempt to solve this by using knowledge of a file type's internal structure to validate and link non-contiguous clusters. For example, pointers or size markers within a file's data can provide clues about where the next fragment might be located. This is an active area of research and is considerably more complex than simple header/footer carving.[5]
- **Directory Structure Recovery:** Recovering a single deleted file is one challenge; reconstructing an entire deleted directory tree is another. This requires not only recovering the directory's own data clusters (which contain the list of entries) but also recursively applying recovery techniques to each file and subdirectory entry found within.
- **Exploring Other File Systems:** The principles of low-level analysis learned with FAT32 are applicable to other file systems, but the specific structures are vastly different.
    - **exFAT:** A successor to FAT32, designed to overcome its file and volume size limitations. It is common on large SD cards and external drives.[38]
    - **NTFS (New Technology File System):** The standard file system for modern Windows operating systems. It is far more complex than FAT32, featuring a central metadata file called the Master File Table (MFT), journaling for improved reliability,

and support for file permissions and encryption.[34] Recovery from NTFS involves parsing the MFT records rather than simple directory entries.

By successfully completing this Phase 1 project, a developer will have built not only a useful tool but also a deep, practical understanding of how data is organized and managed at the lowest levels—a skill set that is foundational to the entire field of computer science.

Works cited

1. Overview of FAT32 · 9-FileSystem - jennyzhang0215, accessed on October 26, 2025, https://jennyzhang0215.gitbooks.io/9-filesystem/overview-of-fat32.html
2. FAT32 File Structure, accessed on October 26, 2025, https://cscie92.dce.harvard.edu/fall2025/slides/FAT32%20File%20Structure.pdf
3. Unit 10 – The FAT32 File System - Washburn University, accessed on October 26, 2025, https://cislinux2.washburn.edu/zzmech/cm203/units/Unit10-TheFAT32FileSystem.pdf
4. en.wikipedia.org, accessed on October 26, 2025, https://en.wikipedia.org/wiki/File_carving#:~:text=File%20carving%20is%20the%20process,in%20the%20existing%20file%20system.
5. File carving - Wikipedia, accessed on October 26, 2025, https://en.wikipedia.org/wiki/File_carving
6. File carving | Infosec, accessed on October 26, 2025, https://www.infosecinstitute.com/resources/digital-forensics/file-carving/
7. RAW disk access with C? : r/C_Programming - Reddit, accessed on October 26, 2025, https://www.reddit.com/r/C_Programming/comments/1fssnf9/raw_disk_access_with_c/
8. Is it possible to get writing access to raw devices using python with ..., accessed on October 26, 2025, https://stackoverflow.com/questions/7135398/is-it-possible-to-get-writing-access-to-raw-devices-using-python-with-windows
9. Workday/raw-disk-parser: A tool to interact with Windows ... - GitHub, accessed on October 26, 2025, https://github.com/Workday/raw-disk-parser
10. Need a simple way to direct read HDD sector / raw data in Python - Reddit, accessed on October 26, 2025, https://www.reddit.com/r/Python/comments/7cynt3/need_a_simple_way_to_direct_read_hdd_sector_raw/
11. owenlo/ReadDisk-Python: Read a single sector of a physical disk using Python - GitHub, accessed on October 26, 2025, https://github.com/owenlo/ReadDisk-Python
12. Reading and Writing to Raw Disk Sectors - CodeProject, accessed on October 26, 2025, https://www.codeproject.com/articles/Reading-and-Writing-to-Raw-Disk-Sectors
13. Raw Input Overview - Win32 apps | Microsoft Learn, accessed on October 26,

2025,
https://learn.microsoft.com/en-us/windows/win32/inputdev/about-raw-input
14. FAT32: Structure, Limitations & Advantages | StudySmarter, accessed on October 26, 2025, https://www.studysmarter.co.uk/explanations/computer-science/computer-systems/fat32/
15. FAT32 Boot Sector and Bootstrap - Active@ File Recovery, accessed on October 26, 2025, https://www.file-recovery.com/recovery-FAT32-boot-sector.htm
16. Volume Boot Sector Format of FAT - Network Intelligence, accessed on October 26, 2025, https://www.networkintelligence.ai/blogs/volume-boot-sector-format-of-fat/
17. en.wikipedia.org, accessed on October 26, 2025, https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#:~:text=FAT32%20stores%20the%20root%20directory,the%20Data%20Region%20starts%20here.&text=This%20is%20where%20the%20actual,up%20most%20of%20the%20partition.
18. FAT Boot Sector, accessed on October 26, 2025, https://averstak.tripod.com/fatdox/bootsec.htm
19. Everything I Know About FAT 2013, accessed on October 26, 2025, https://kcall.co.uk/fat/index.html
20. Raw File System Analysis (FAT32 File Recovery) - NTCore, accessed on October 26, 2025, https://ntcore.com/raw-file-system-analysis-fat32-file-recovery/
21. Directory Structure, accessed on October 26, 2025, https://averstak.tripod.com/fatdox/dir.htm
22. 11: FATs and Directory Entries | COMPSCI 365/590F | Digital Forensics (Spring 2017), accessed on October 26, 2025, https://people.cs.umass.edu/~liberato/courses/2017-spring-compsci365/lecture-notes/11-fats-and-directory-entries/
23. A Complete Guide to FAT32 Data Recovery, accessed on October 26, 2025, https://www.cnwrecovery.com/html/fat32.html
24. Detailed Guide: FAT32 File Recovery on Windows, accessed on October 26, 2025, https://www.powerdatarecovery.com/hard-drive-recovery/fat32-data-recovery.html
25. HDD GURU FORUMS • View topic - How to recover FAT table?, accessed on October 26, 2025, https://forum.hddguru.com/viewtopic.php?f=25&t=32535
26. FAT Data Recovery - Analyst, accessed on October 26, 2025, https://bakerst221b.com/docs/ttp/data-recovery/file-systems/fat/
27. Deleting files on FAT32 USB changes the first cluster information - Stack Overflow, accessed on October 26, 2025, https://stackoverflow.com/questions/1146612/deleting-files-on-fat32-usb-changes-the-first-cluster-information
28. fat32 - Can cluster chains end with 0 or 1? - Stack Overflow, accessed on October 26, 2025, https://stackoverflow.com/questions/48966441/can-cluster-chains-end-with-0-or-1

29. A simple file recovery tool for FAT32 filesystems - GitHub, accessed on October 26, 2025, https://github.com/shan18/FAT32-File-Recovery
30. File Carver - Wise Forensics, accessed on October 26, 2025, https://wise-forensics.com/2024/09/27/file-carver/
31. File Carving - Digital Forensics : TryHackMe Walkthrough | by RosanaFSS | Medium, accessed on October 26, 2025, https://medium.com/@RosanaFS/file-carving-digital-forensics-tryhackme-walkthrough-8d0ed2f879ec
32. Gary Kessler's File Signature Table - eSecurity Institute, accessed on October 26, 2025, https://www.esecurityinstitute.com/gary-kesslers-file-signature-table/
33. What is file carving and its basic techniques? - Educative.io, accessed on October 26, 2025, https://www.educative.io/answers/what-is-file-carving-and-its-basic-techniques
34. File Carving – What It Is and How to Get Started - eForensics Magazine, accessed on October 26, 2025, https://eforensicsmag.com/file-carving-what-it-is-and-how-to-get-started/
35. Recover/Carve FAT32 file system metadata (file names etc.) without partition table, accessed on October 26, 2025, https://superuser.com/questions/1768698/recover-carve-fat32-file-system-metadata-file-names-etc-without-partition-tab
36. Data Carving: Signature-Based Data Recovery, accessed on October 26, 2025, https://www.starusrecovery.com/articles/data-carving-signature-based-data-recovery.html
37. wahlflo/pyFileCarving: A python cli script for simple file carving - GitHub, accessed on October 26, 2025, https://github.com/wahlflo/pyFileCarving
38. What Is FAT32? - Coursera, accessed on October 26, 2025, https://www.coursera.org/articles/fat32