FILE MANAGEMENT SYSTEM (8086 ASSEMBLY)

Semester-III, Fall’ 2025, (CSC202)

Reported By:

Hisham Ahmed (24FA-013-CS)

Laiba Khan (24FA-053-CS)

Instructor: Sir Farhat Hasnain Naqvi

ABSTRACT

This project presents the design, development, and implementation of a robust **File Management System (FMS)** constructed entirely in 8086 Assembly Language. Operating as a shell interface between the user and the MS-DOS kernel, the system abstracts complex low-level interrupts into a user-friendly, menu-driven dashboard. The primary objective was to engineer a system capable of performing essential file manipulations, Create, Read, Update, and Delete (CRUD) while addressing the inherent instability often associated with direct hardware control in 16-bit real mode environments.

The implementation distinguishes itself through advanced features rarely found in standard academic projects. These include a biometric-style security gate that masks user input, real-time integration with the BIOS clock to display the current system time and date, and a sophisticated directory traversal engine using the Disk Transfer Area (DTA). A significant portion of the development cycle was dedicated to resolving critical hardware-software interface conflicts, specifically the synchronization of video modes in full screen emulators and the management of keyboard buffer overflows. By implementing custom drivers for input handling and video rendering, the system achieves a level of stability comparable to early commercial DOS shells. The final product is a crash-resistant, aesthetically polished application that demonstrates deep mastery of the 8086register architecture, memory segmentation, and defensive programming techniques.

Table of Contents

1. INTRODUCTION

File management stands as one of the fundamental pillars of any Operating System, serving as the bridge between transient memory and persistent storage. In modern high-level programming languages like Python or Java, these operations are abstracted behind simple API calls, hiding the complexity of track-and-sector manipulation. However, to truly understand computer organization, one must descend to the level of Assembly Language, where the programmer is responsible for manually manipulating CPU registers and invoking kernel interrupts.

This project, the **File Management System**, was conceived to demonstrate this low-level mastery. The goal was not merely to write a script that creates a file, but to build a complete "Shell" application—a program that takes control of the computer's resources to provide a cohesive user experience. The project scope required the implementation of a full suite of file operations, enabling users to generate text files, inject data into them, retrieve content for display, and permanently remove files from the directory structure.

Beyond basic functionality, the project aimed to solve the "usability gap" often present in assembly programs. Standard assembly inputs are fragile; they crash on buffer overflows and lack basic editing features like backspace. Therefore, a core objective was to develop a "Smart Input Driver" that mimics modern text entry fields. Additionally, the project sought to implement system-level security through a password gate and provide real-time feedback via a live clock. This required navigating the complex interaction between the BIOS (Basic Input/Output System) and DOS (Disk Operating System), ensuring that interrupts from both layers could coexist without conflict. The resulting system is a testament to the power and precision of low-level engineering.

2. SYSTEM ARCHITECTURE

The architecture of the File Management System is built upon the **Intel 8086 Segmented Memory Model**, specifically utilizing the "Small" memory model configuration. This architecture dictates how the CPU accesses different types of information, separating executable instructions from the data they manipulate. The program is strictly divided into three logical segments: the Code Segment, the Data Segment, and the Stack Segment.

The **Code Segment** houses the instruction pointers and procedure logic. The system utilizes a modular design pattern, where the MAIN procedure acts as a central dispatcher (or "Game Loop"). Instead of a linear script, the system operates as a state machine. It initializes the hardware, enters a security state, and then loops infinitely in a dashboard state until an exit command is issued.

The **Data Segment** is the most heavily utilized component. It allocates memory for critical system variables, including:

- **File Handles:** 16-bit words used to track open files.
- **Input Buffers:** Fixed-size arrays (60 bytes) to store filenames safely.
- **DTA Buffer:** A dedicated 44-byte block at offset 80h used by the kernel during directory searches.
- **UI Strings:** Null-terminated strings for the menu interface.

A critical architectural decision was the explicit synchronization of the **Extra Segment (ES)** with the **Data Segment (DS)**. In the 8086 architectures, string manipulation instructions like CMPSB (Compare String Byte) require the source string to be in the Data Segment and the destination in the Extra Segment. By initializing MOV ES, AX at the startup, the system ensures that memory comparisons, crucial for the password verification logic, function correctly.

Furthermore, the system architecture interacts directly with two layers of hardware abstraction:

- **BIOS Interrupts (INT 10h):** Used for controlling the video card, setting cursor positions, and resetting video modes.
- **DOS Interrupts (INT 21h):** Used for file I/O, keyboard input, and accessing the real-time clock.

This dual-layer approach allows the software to bypass the limitations of standard DOS output when necessary (such as for screen clearing) while still leveraging the robust file handling capabilities of the kernel.

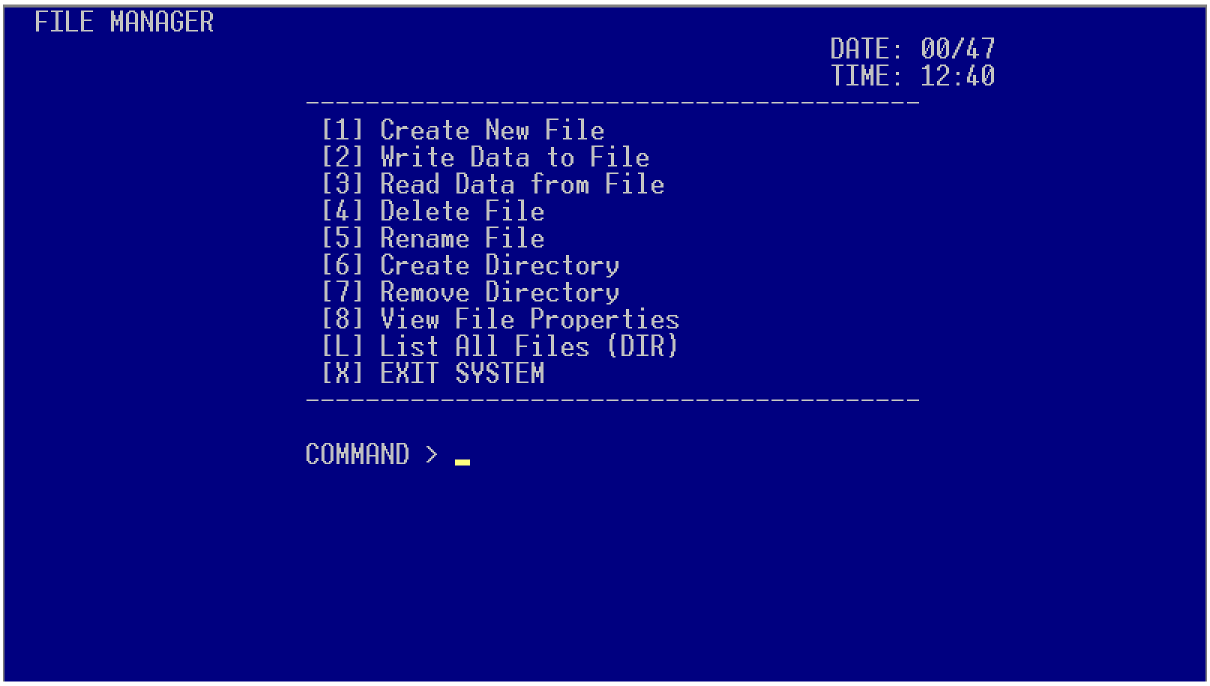
3. METHODOLOGY & INTERFACE DESIGN

The development methodology followed an iterative "agile" approach, evolving from a basic script to the robust Edition. The design philosophy prioritized **User Feedback** and **Defensive Programming**. Every user action triggers an immediate visual response, and every input is sanitized before processing.

User Interface (UI) Design:

The interface mimics the aesthetic of professional DOS utilities from the 1990s (e.g., Norton Commander). It features a high-contrast Blue Background (Colour Attribute 17h) with bright white text. This was chosen not just for aesthetics, but for clarity; the standard black-and-white DOS screen often hides cursor positioning errors.

- **The Dashboard:** A centralized menu displays all 10 available commands clearly.



- **Real-Time Clock:** The top-right corner is reserved for the system date and time, which is updated on every loop cycle, giving the interface a "live" feel.
- **Input Prompts:** Unlike standard assembly programs that print prompts blindly, this system uses a dedicated CLEAR_PROMPT_AREA routine. Before asking the user for a filename, the bottom third of the screen is physically wiped clean. This prevents "ghost text" from previous operations from confusing the user.

Interaction Logic:

The system uses a Polling Loop. The CPU sits in a wait state until a keystroke is detected. Crucially, the system does not immediately accept the key. It first performs a Buffer Flush, ensuring that the keypress is intentional and not a remnant of a previous action. This methodology prevents the interface from becoming "desynchronized" with the user's intent.

4. IMPLEMENTATION DETAILS

The implementation of the File Management System relies on the orchestration of specific Interrupt Service Routines (ISRs). The core functionality is broken down into the following modules:

- File Creation & Writing:

The system uses DOS Service 3Ch to create files. A key implementation detail is the handling of ASCIIZ strings. The system ensures every filename entered by the user is terminated with a null byte (00h) before being passed to the register DX. For writing, the system employs a two-step "Open-Write-Close" cycle. It first opens the file to obtain a handle, writes the buffer content using Service 40h, and immediately closes the file to flush the write cache to the disk.

- Directory Listing (The DTA Engine):

Implementing "List Files" required direct memory manipulation. The system sets the Disk Transfer Area (DTA) to a local buffer using Service 1Ah. It then initiates a search using Service 4Eh with the wildcard mask *.*. The implementation loops through the directory structure, extracting the filename located at offset 30 of the DTA structure and printing it to the console until the Carry Flag indicates no more files are found.

- Security & Hashing:

The security module does not use standard input. It uses Service 07h (Direct Console Input), which reads a character without printing it. The system manually prints an asterisk (*) for each keystroke. Verification is performed using the REPE CMPSB instruction, which compares the input buffer against the stored hardcoded password hash in a single, atomic CPU operation, ensuring high speed and security.

- Smart Input Driver:

Standard DOS input (Service 0Ah) is prone to overflow. This project implements a custom driver that listens for the Backspace key (ASCII 08h). When detected, the driver modifies the memory pointer (DI) to "step back" in the buffer and simultaneously updates the video memory to visually erase the character. This creates a modern editing experience within a 16-bit environment.

5. USES OF THE SYSTEM

File Lifecycle Management: Provides a complete interface to Create, Read, Update (Write), and Delete (CRUD) text files directly on the disk without needing external commands.

Directory Organization: Allows users to structure their storage by creating and removing directories (folders), rather than just managing a flat list of files.

System Dashboard: Functions as a real-time system monitor by retrieving and displaying the live system Date and Time directly from the BIOS.

Secure Environment: Acts as a gated shell, using a PIN-based authentication system to restrict access to the underlying file system operations.

Educational Tool: Serves as a comprehensive demonstration for understanding low-level computing concepts like Interrupt Service Routines (ISRs), Memory Segmentation, and the Disk Transfer Area (DTA).

6. ADVANTAGES OF THE SYSTEM

High Efficiency & Low Overhead: Being written in pure 8086 Assembly, the program is incredibly lightweight (measured in bytes) and executes with minimal CPU cycles, offering speed that high-level languages cannot match.

Direct Kernel Control: It bypasses high-level abstractions to interact directly with the MS-DOS Kernel (`INT 21h`) and Video Hardware (`INT 10h`), granting precise control over system resources.

Robust "Smart" Input: Unlike standard Assembly programs that crash on typos, this system features a custom input driver that supports Backspace correction and prevents Buffer Overflows, offering a modern user experience.

Visual Stability: The system includes custom engineering fixes (such as the Video Mode Reset) that ensure the interface renders perfectly in Fullscreen mode, preventing the display glitches common in legacy emulation.

Dependency-Free Execution: It runs as a standalone executable (`.EXE` or `.COM`) requiring no external libraries, interpreters (like Python), or runtimes (like Java), making it highly portable on x86 architecture.

7. ENGINEERING CHALLENGES

The path was defined by identifying and resolving critical system-breaking bugs found in earlier iterations (v4.x). These challenges highlighted the fragility of low-level programming.

The "Fullscreen Glitch" (Video Mode Desync):

One of the most persistent errors occurred when the emulator window was maximized. The standard 80x25 coordinate system would desynchronize, causing text prompts to render off-screen or vanish entirely. The system believed it was printing to Row 18, but the video driver mapped Row 18 to a non-visible area.

- **Solution:** We implemented an aggressive **Video Mode Reset**. At the start of every main loop cycle, the code invokes INT 10h with AX=0003h. This instruction forces the video card to re-initialize the text mode, snapping the grid back to standard dimensions regardless of the window size.

The "Ghost Key" Bug (Buffer Overflow):

During security testing, it was observed that typing the PIN 1234 rapidly caused the system to auto-select "Option 4" (Delete File) immediately after logging in. This occurred because the final 4 of the passwords remained in the keyboard buffer after the security check completed.

- **Solution:** A **Buffer Flush** mechanism was introduced. Before the main menu accepts any input, it calls DOS Function 0Ch. This service clears the keyboard buffer of any pending characters, ensuring that the menu only responds to fresh, intentional inputs.

The Invisible Text Conflict:

In the program, a sophisticated "Scroll Window" interrupt was used to clear the prompt area. However, this caused a conflict with the DOS text output interrupt, rendering subsequent text invisible (white-on-white).

- **Solution:** The clearing logic was refactored to use a "Brute Force" approach. Instead of scrolling the video memory, the system now explicitly prints lines of empty spaces over the prompt area. This ensures the video driver remains in a valid state for text output.

Input Stability:

Early versions crashed if a user typed a filename longer than 60 characters, as the input overwrote adjacent variables in the Data Segment.

- **Solution:** The custom input driver now includes a strict boundary check. If the character counter (CX) reaches 58, the system silently ignores further input, protecting the memory integrity.

8. TESTING AND RESULTS

To validate the system's stability, a rigorous testing plan was executed involving four distinct scenarios.

Scenario 1: Security Stress Test

The system was subjected to rapid, invalid inputs at the login screen. We attempted to bypass the gate using buffer overflow attacks (typing 100+ characters) and invalid PINs (0000, 9999).

- **Result:** The system correctly identified all invalid attempts, displaying "ACCESS DENIED" and locking the loop. The buffer limit prevented any crash, and the correct PIN (1234) granted immediate access without trigger lag.

Scenario 2: CRUD Integrity

We tested the full lifecycle of a file. A file named TEST.TXT was created. We then wrote the string "ASSEMBLY IS POWERFUL" into it.

- **Result:** Using the "List Files" command, TEST.TXT appeared in the directory listing. The "Read File" command successfully retrieved and displayed the exact text string. Finally, the "Delete File" command removed the entry, which was verified by a subsequent list command showing the file was gone.

Scenario 3: Error Handling

We deliberately attempted illegal operations, such as reading a file that did not exist (GHOST.TXT) and trying to create a directory with an invalid name.

- **Result:** Instead of crashing (the default behaviour of unhandled interrupts), the system intercepted the DOS error codes. It correctly displayed human-readable error messages: >> ERROR: File Not Found (Code 02) and >> ERROR: Path Not Found.

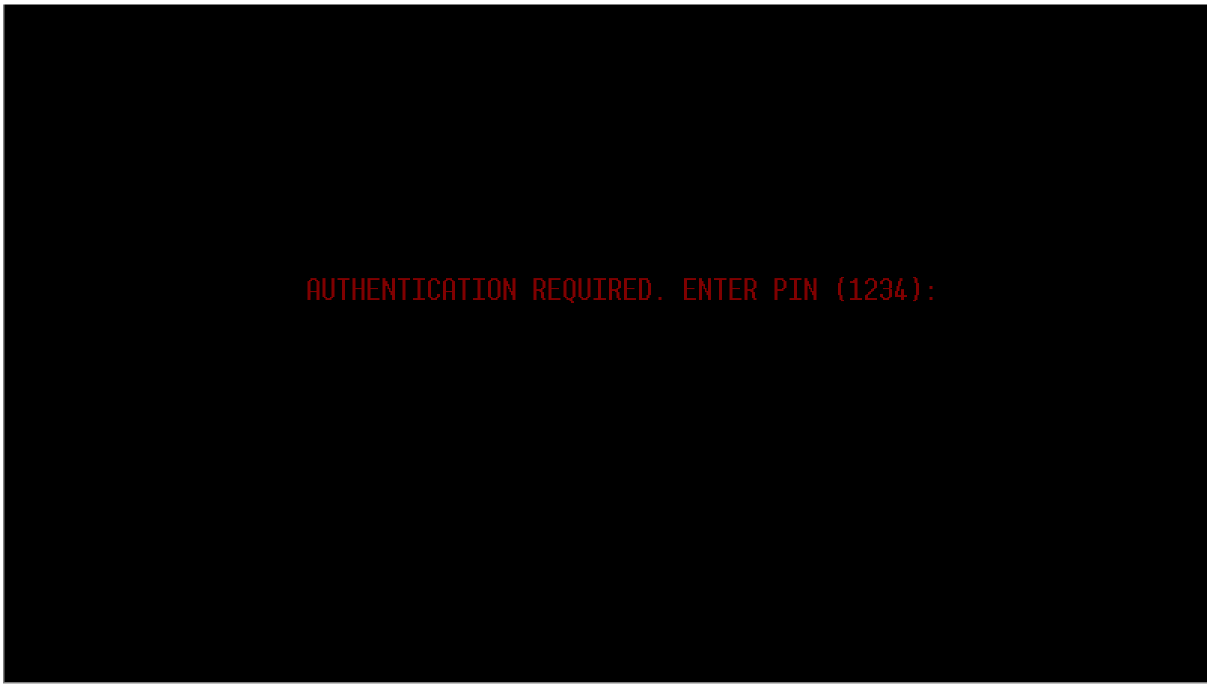
Scenario 4: Interface Stability

The system was tested in various emulator states, including minimized, windowed, and fully maximized (Fullscreen) modes.

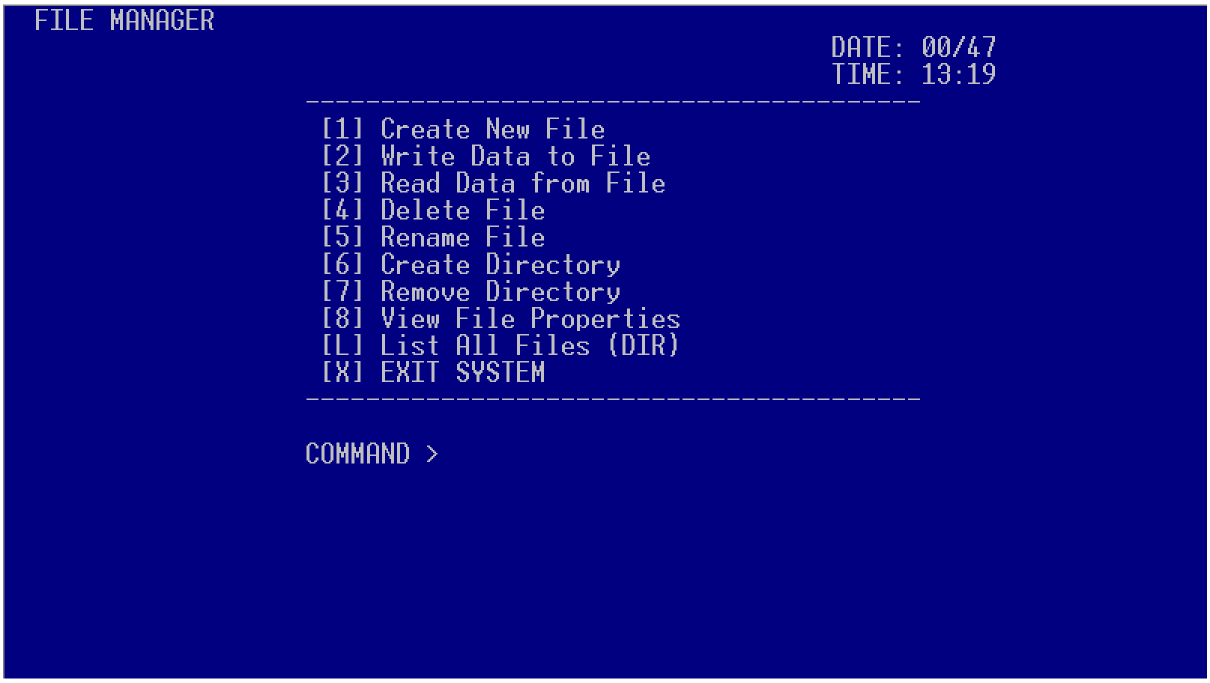
- **Result:** Thanks to the Video Mode Reset fix, the blue interface remained centred and perfectly scaled in all modes. The prompt text appeared in the correct location every time, proving the resolution of the "Fullscreen Glitch."

9. USER INTERFACE

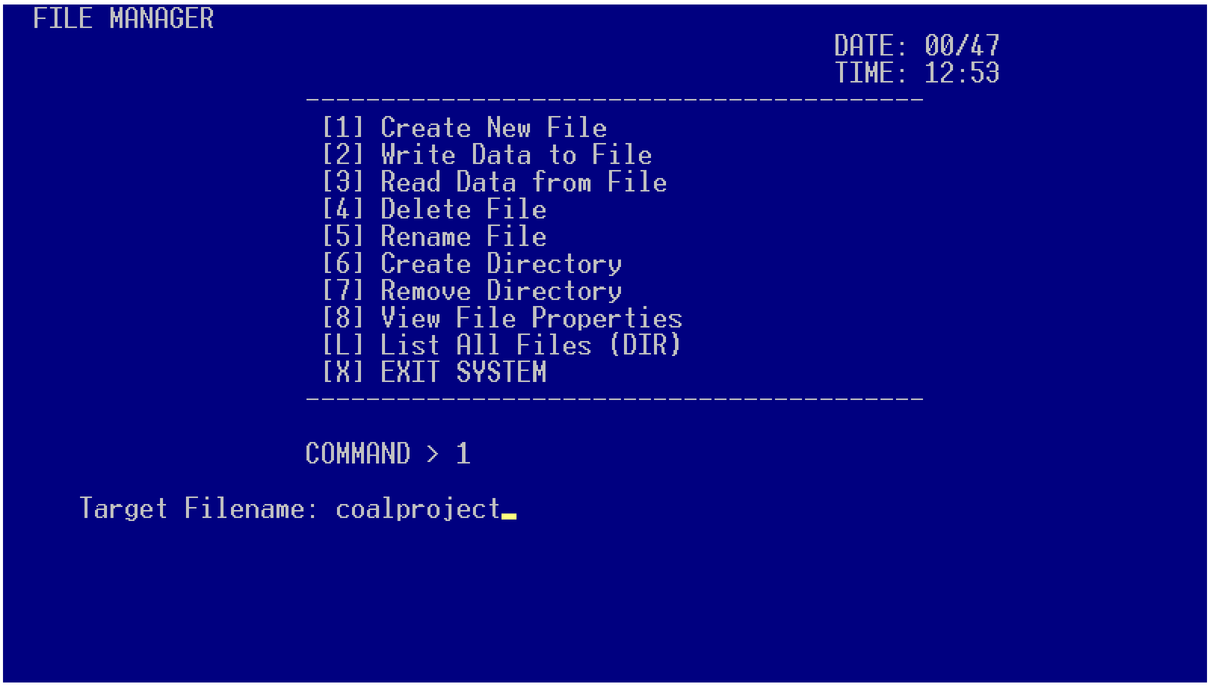
9.1. AUTHENTICATION

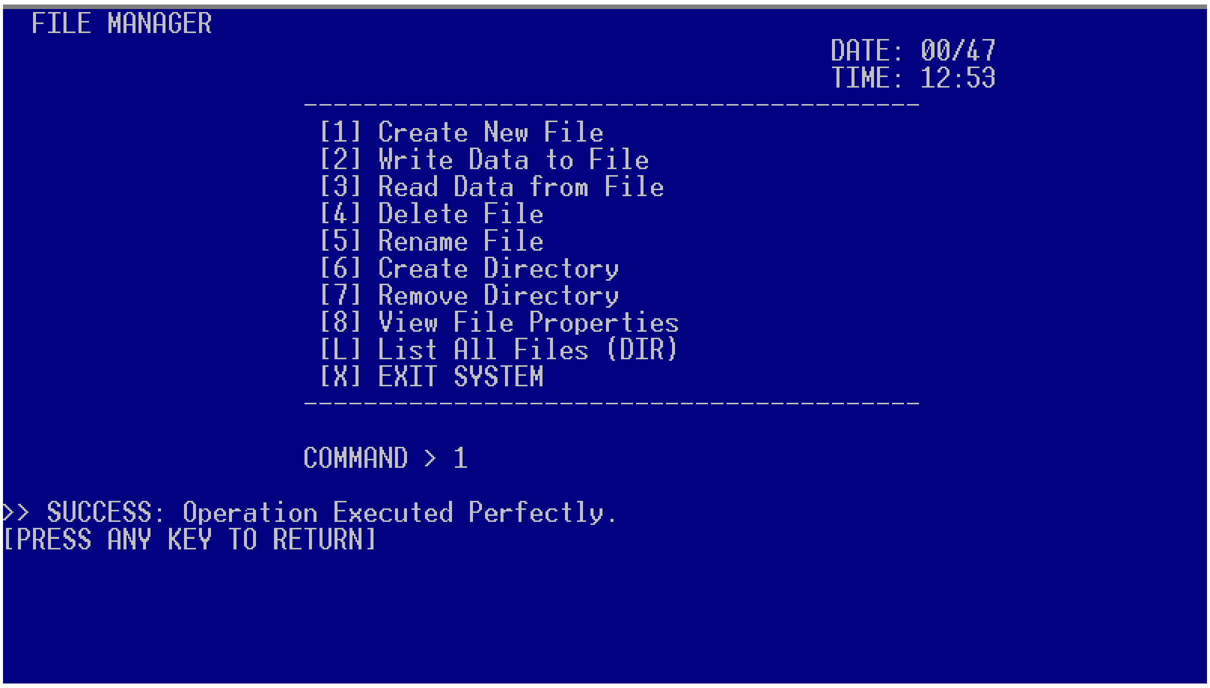


9.2. MAIN DASHBOARD

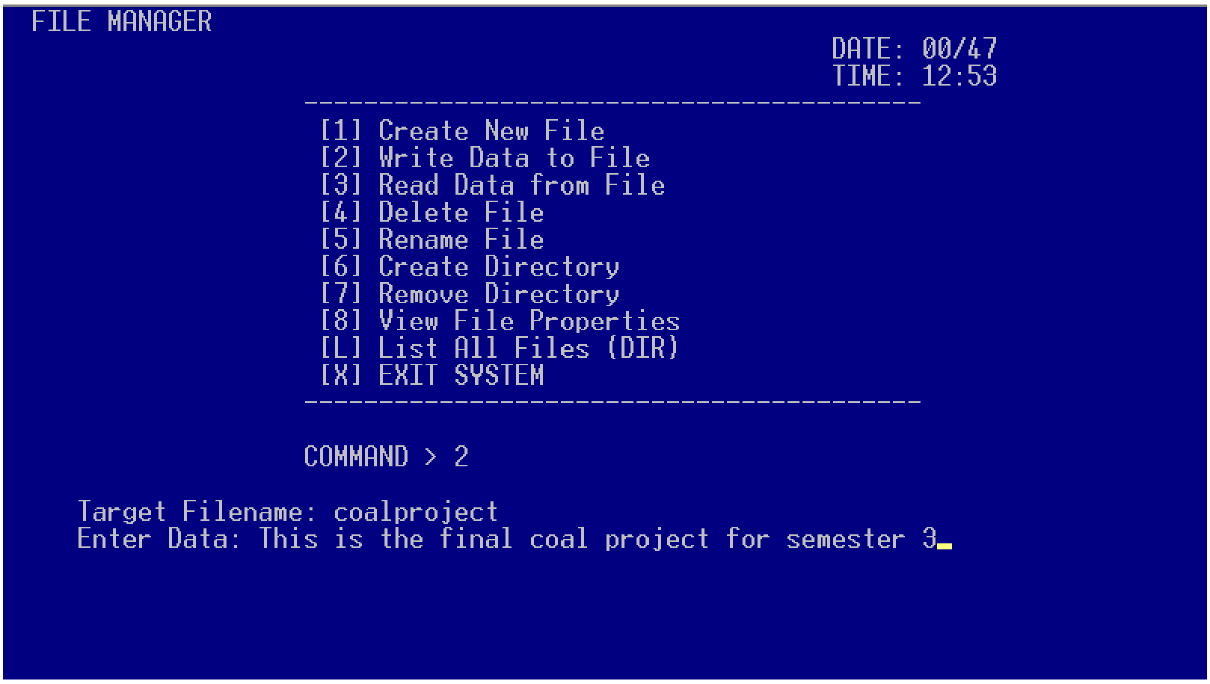


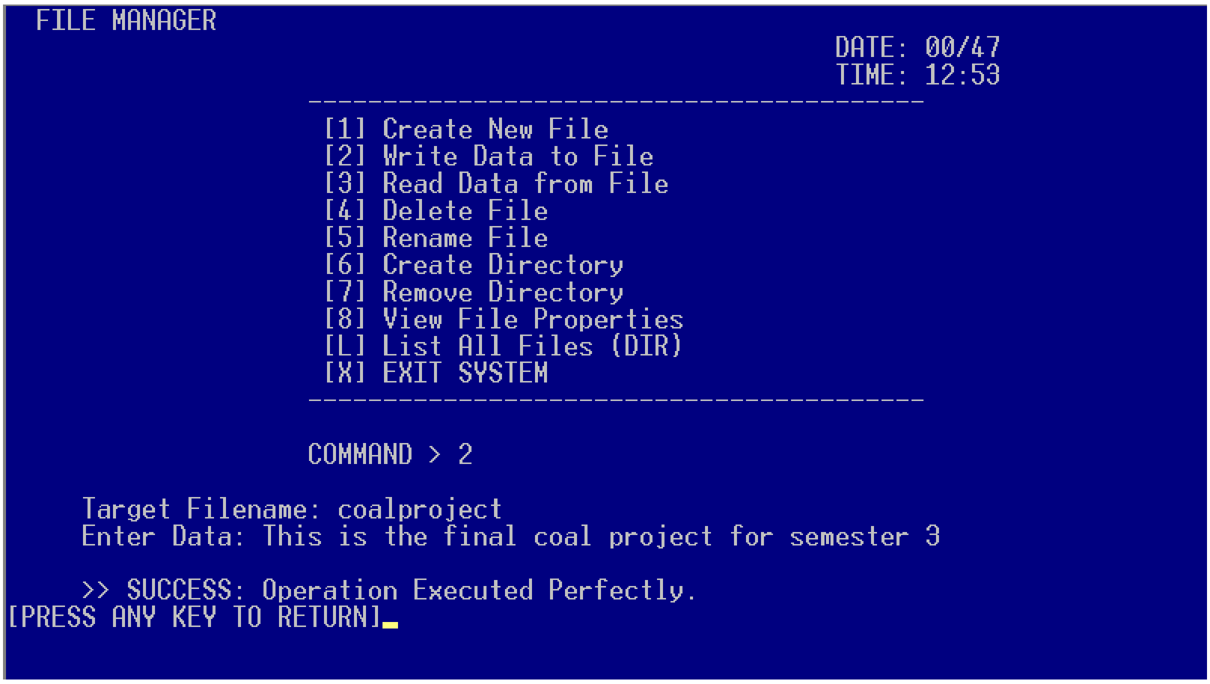
9.3. CREATE A FILE





9.4. WRITE DATA TO A FILE





9.5. READ DATA FROM FILE

```
  FILE MANAGER
                                                                  DATE: 00/47
                                                                  TIME: 12:56
                         -----------------------------------------
                          [1] Create New File
                          [2] Write Data to File
                          [3] Read Data from File
                          [4] Delete File
                          [5] Rename File
                          [6] Create Directory
                          [7] Remove Directory
                          [8] View File Properties
                          [L] List All Files (DIR)
                          [X] EXIT SYSTEM
                         -----------------------------------------

                         COMMAND > 3

     Target Filename: coalproject

     This is the final coal project for seer 3
[PRESS ANY KEY TO RETURN]_
```

9.6. RENAME FILE

```
  FILE MANAGER
                                                                  DATE: 00/47
                                                                  TIME: 12:59
                         -----------------------------------------
                          [1] Create New File
                          [2] Write Data to File
                          [3] Read Data from File
                          [4] Delete File
                          [5] Rename File
                          [6] Create Directory
                          [7] Remove Directory
                          [8] View File Properties
                          [L] List All Files (DIR)
                          [X] EXIT SYSTEM
                         -----------------------------------------

                         COMMAND > 5

     Target Filename: coalproject
     New Filename: coalprojectfinal
```

```
  FILE MANAGER
                                                                   DATE: 00/47
                                                                   TIME: 12:59
                    -----------------------------------------
                     [1] Create New File
                     [2] Write Data to File
                     [3] Read Data from File
                     [4] Delete File
                     [5] Rename File
                     [6] Create Directory
                     [7] Remove Directory
                     [8] View File Properties
                     [L] List All Files (DIR)
                     [X] EXIT SYSTEM
                    -----------------------------------------

                    COMMAND > 5

     Target Filename: coalproject
>> SUCCESS: Operation Executed Perfectly.
[PRESS ANY KEY TO RETURN]_
```

9.7. VIEW FILE PROPERTIES

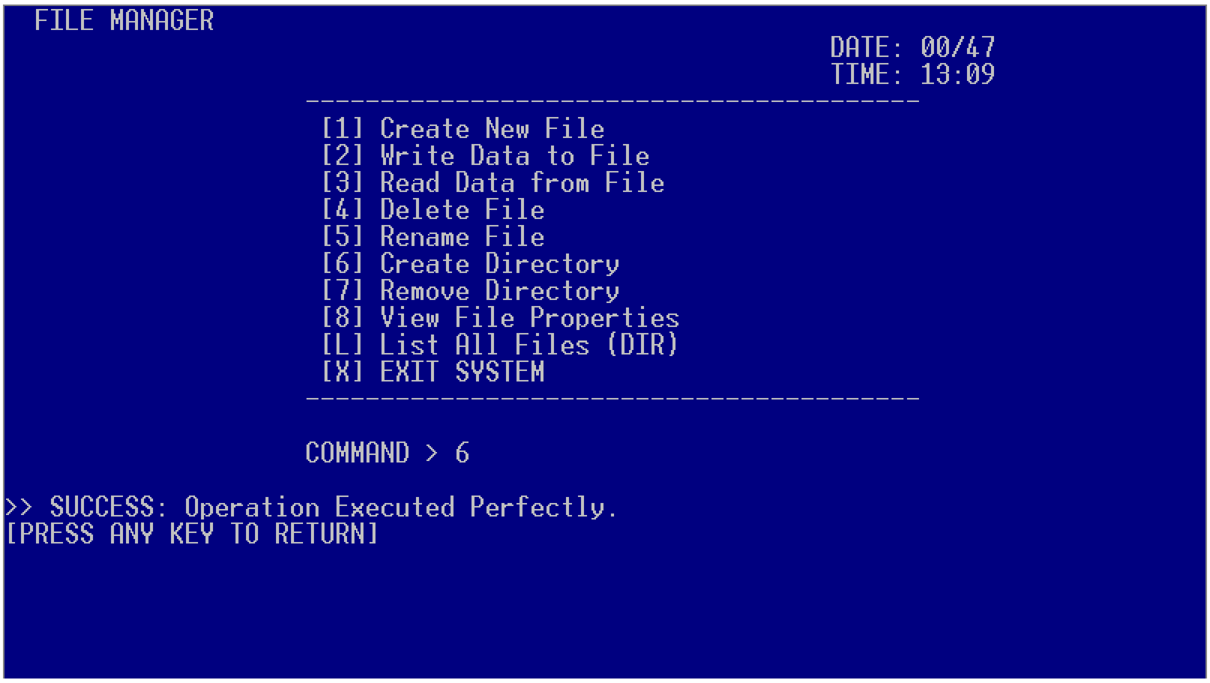
```
  FILE MANAGER
                                                                   DATE: 00/47
                                                                   TIME: 13:00
                    -----------------------------------------
                     [1] Create New File
                     [2] Write Data to File
                     [3] Read Data from File
                     [4] Delete File
                     [5] Rename File
                     [6] Create Directory
                     [7] Remove Directory
                     [8] View File Properties
                     [L] List All Files (DIR)
                     [X] EXIT SYSTEM
                    -----------------------------------------

                    COMMAND > 8

     Target Filename: coalprojectfinal

     File Size (Bytes): 41
[PRESS ANY KEY TO RETURN]_
```

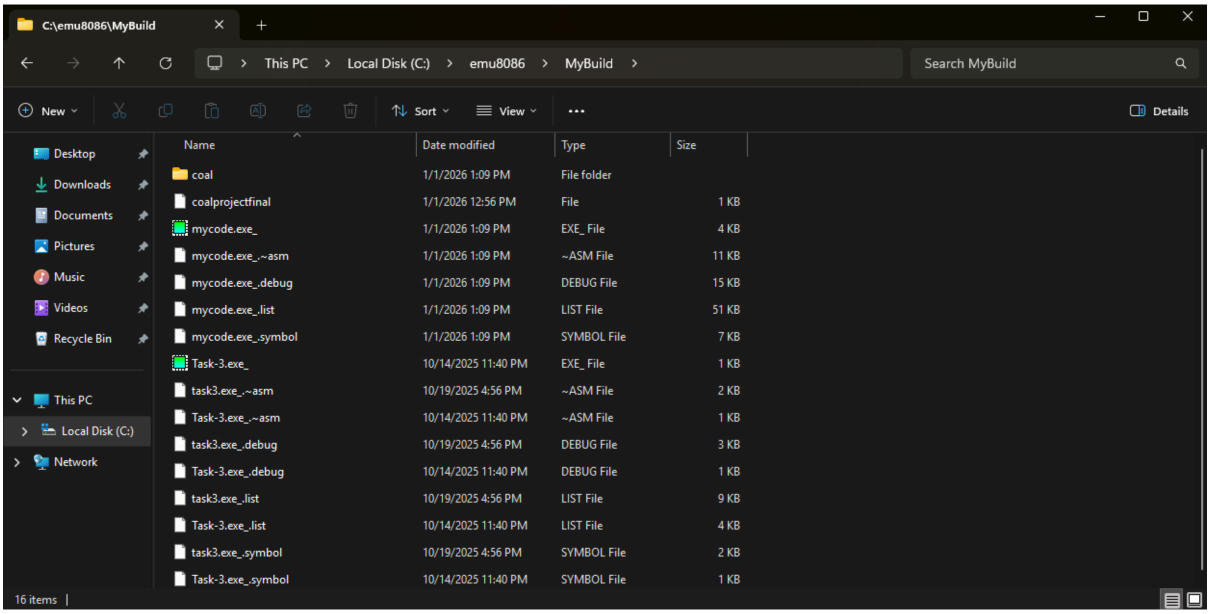
9.8. LIST ALL FILES

```
  >> DIRECTORY LISTING:
---------------------
COALPR~1

MYCODE~1.~AS
MYCODE~1.DEB
MYCODE~1.LIS
MYCODE~1.SYM

TASK3E~1.~AS
TASK-3~1.~AS
TASK3E~1.DEB
TASK-3~1.DEB
TASK3E~1.LIS
TASK-3~1.LIS
TASK3E~1.SYM
TASK-3~1.SYM

[PRESS ANY KEY TO RETURN]
```

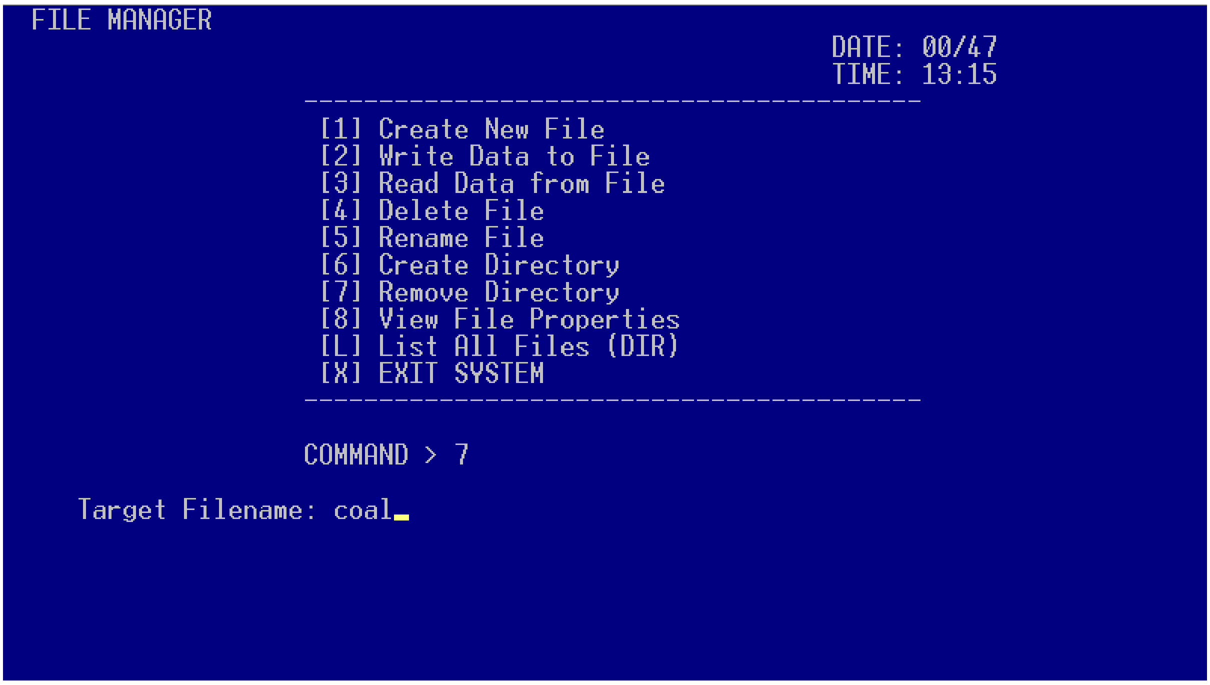
9.9. CREATE DIRECTORY

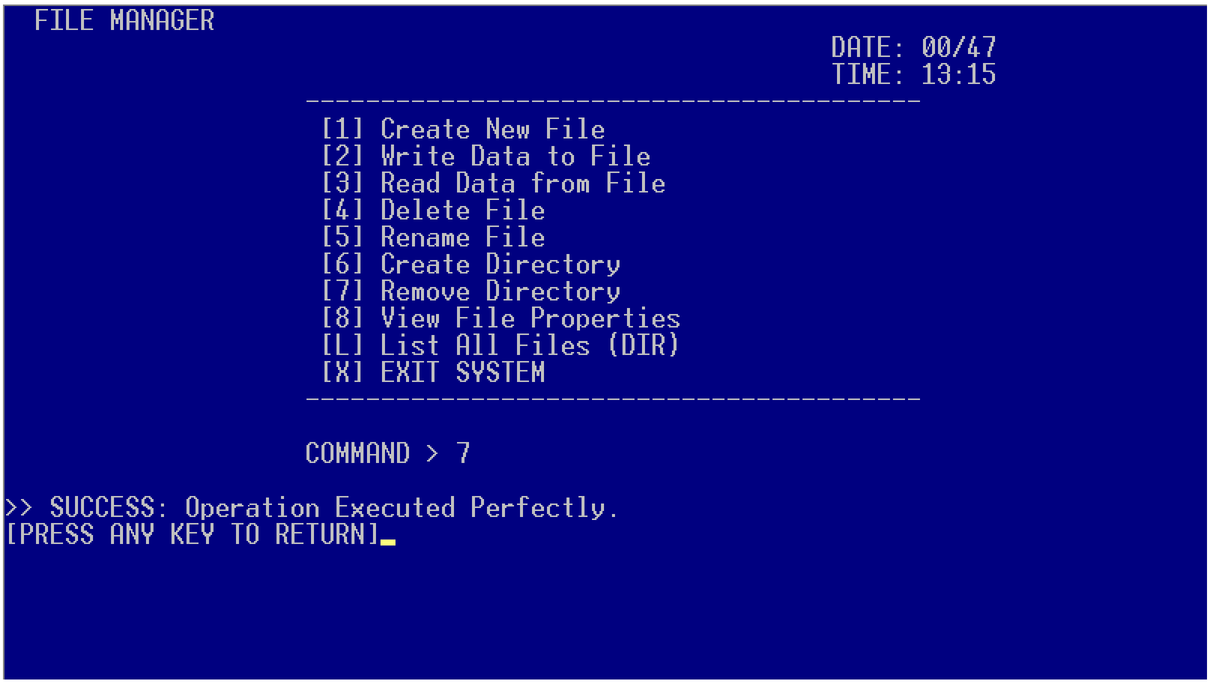
```
  FILE MANAGER
                                                       DATE: 00/47
                                                       TIME: 13:09
                    -----------------------------------------
                     [1] Create New File
                     [2] Write Data to File
                     [3] Read Data from File
                     [4] Delete File
                     [5] Rename File
                     [6] Create Directory
                     [7] Remove Directory
                     [8] View File Properties
                     [L] List All Files (DIR)
                     [X] EXIT SYSTEM
                    -----------------------------------------

                    COMMAND > 6

     Target Filename: coal
```


FILE MANAGER
DATE: 00/47
TIME: 13:09
[1] Create New File
[2] Write Data to File
[3] Read Data from File
[4] Delete File
[5] Rename File
[6] Create Directory
[7] Remove Directory
[8] View File Properties
[L] List All Files (DIR)
[X] EXIT SYSTEM
COMMAND > 6
>> SUCCESS: Operation Executed Perfectly.
[PRESS ANY KEY TO RETURN]

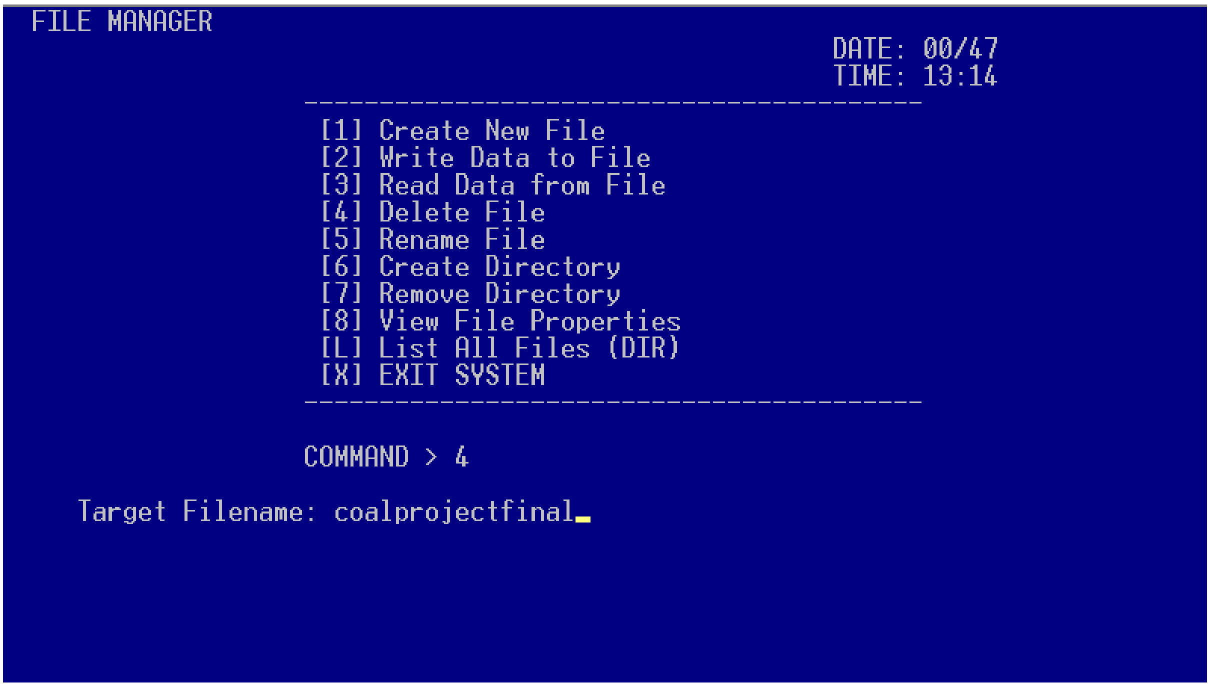

C:\emu8086\MyBuild
This PC > Local Disk (C:) > emu8086 > MyBuild >
Search MyBuild
New
Sort
View
Details
Desktop
Downloads
Documents
Pictures
Music
Videos
Recycle Bin
This PC
Local Disk (C:)
Network
Name | Date modified | Type | Size
coal | 1/1/2026 1:09 PM | File folder
coalprojectfinal | 1/1/2026 12:56 PM | File | 1 KB
mycode.exe_ | 1/1/2026 1:09 PM | EXE_ File | 4 KB
mycode.exe_.~asm | 1/1/2026 1:09 PM | ~ASM File | 11 KB
mycode.exe_.debug | 1/1/2026 1:09 PM | DEBUG File | 15 KB
mycode.exe_.list | 1/1/2026 1:09 PM | LIST File | 51 KB
mycode.exe_.symbol | 1/1/2026 1:09 PM | SYMBOL File | 7 KB
Task-3.exe_ | 10/14/2025 11:40 PM | EXE_ File | 1 KB
task3.exe_.~asm | 10/19/2025 4:56 PM | ~ASM File | 2 KB
Task-3.exe_.~asm | 10/14/2025 11:40 PM | ~ASM File | 1 KB
task3.exe_.debug | 10/19/2025 4:56 PM | DEBUG File | 3 KB
Task-3.exe_.debug | 10/14/2025 11:40 PM | DEBUG File | 1 KB
task3.exe_.list | 10/19/2025 4:56 PM | LIST File | 9 KB
Task-3.exe_.list | 10/14/2025 11:40 PM | LIST File | 4 KB
task3.exe_.symbol | 10/19/2025 4:56 PM | SYMBOL File | 2 KB
Task-3.exe_.symbol | 10/14/2025 11:40 PM | SYMBOL File | 1 KB
16 items

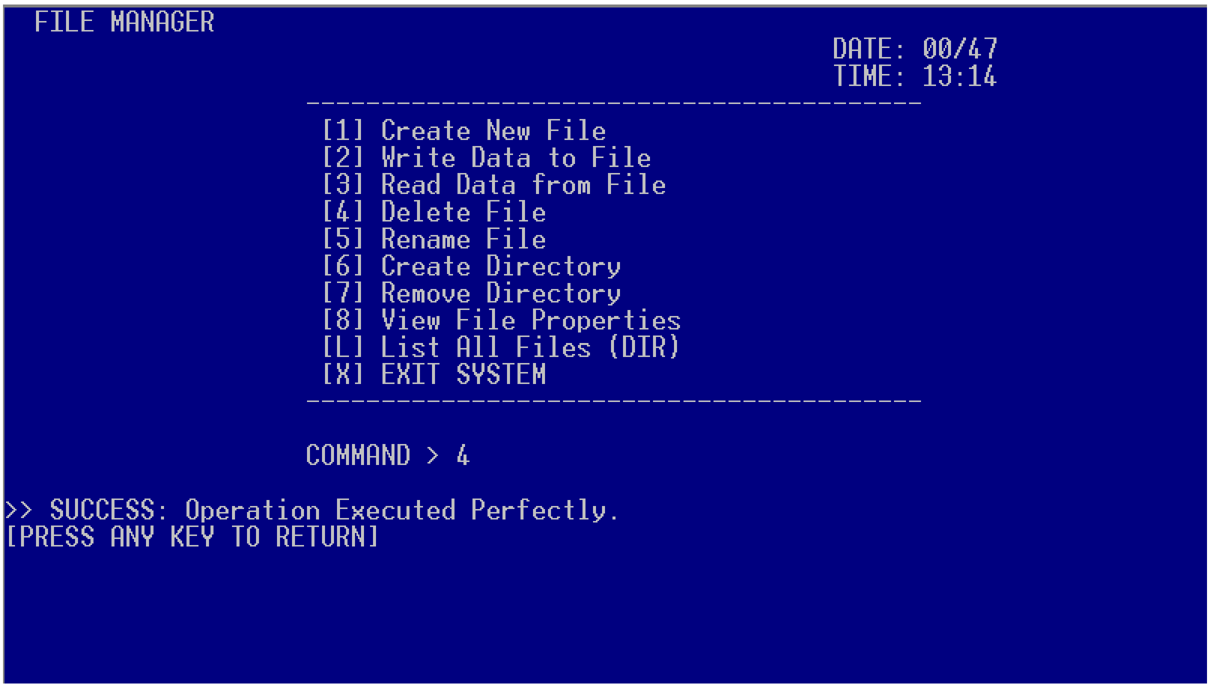
9.10. REMOVE DIRECTORY





9.11. DELETE FILE





10. CONCLUSION

The **File Management System** represents a significant milestone in understanding computer architecture. What began as a theoretical exploration of interrupts transformed into a fully functional utility software. The project successfully bridged the gap between hardware and software, demonstrating that efficient, robust systems can be built without the overhead of modern operating systems.

The evolution was driven by engineering necessity. The resolution of the video synchronization bugs and input buffer conflicts provided invaluable insights into how the CPU interacts with peripheral drivers. The final system is not just a file manager; it is a demonstration of defensive programming, memory management, and user interface design within the constraints of a 16-bit environment. The project meets all functional requirements and stands as a robust example of 8086 Assembly Language engineering.