

•

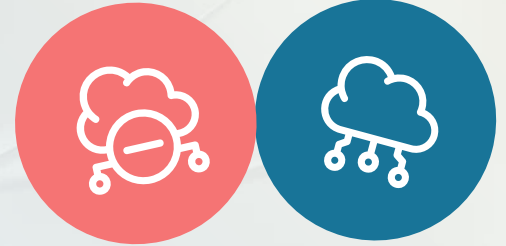
# Topic :

## Filesystem Project for Embedded Systems

---

Marwa Awil

**Student number: M00755729**



# Introduction

## Project Overview

The filesystem project aims to develop a foundational component for a lightweight operating system tailored specifically for embedded systems. Sponsored by Embedded Solutions Inc., this initiative focuses on optimizing data management efficiency and performance within resource-constrained environments.

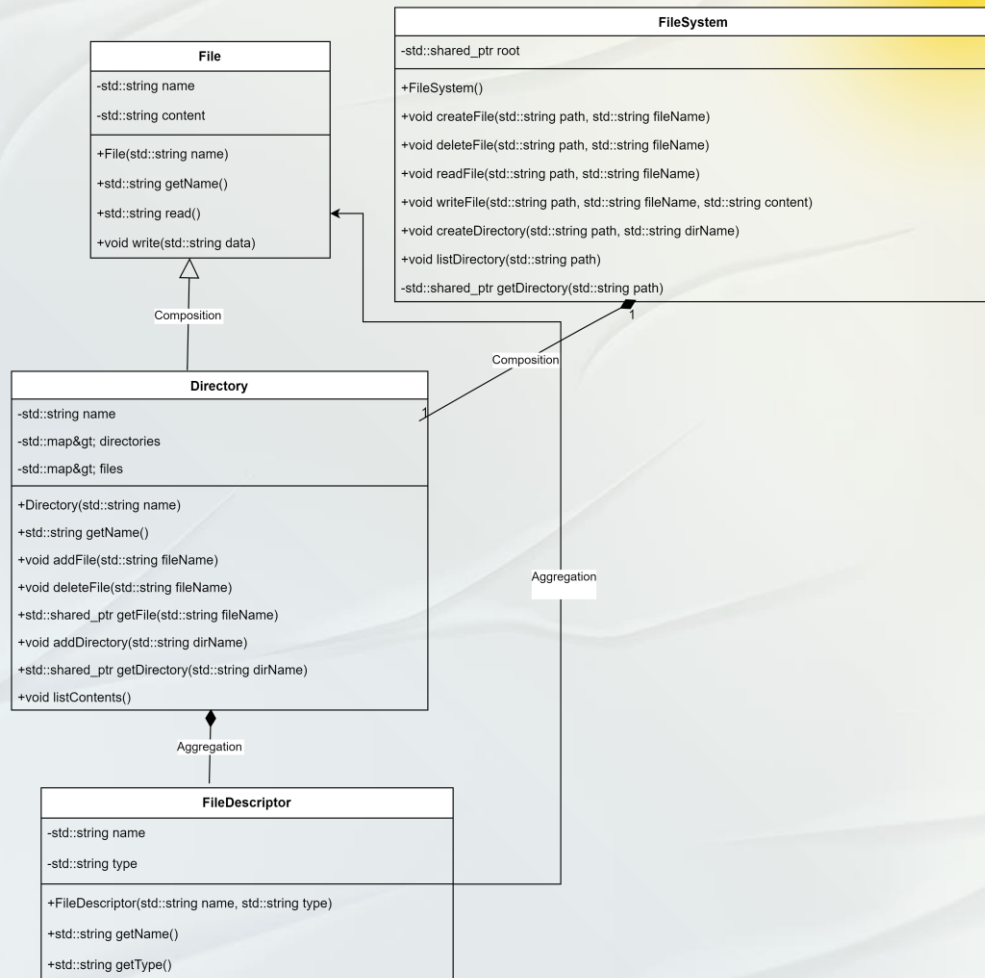
- **Goal:** Develop a lightweight filesystem for an embedded operating system.
- **Significance:** Critical for managing limited resources and optimizing performance in embedded systems.
- **Key Objectives:** Modular architecture, compatibility with diverse hardware, rigorous testing.

# Design Phase

## UML Class Diagram

The UML class diagram provided represents the structure and relationships within a simple filesystem.

- **Description:** Shows the structure and relationships within the filesystem.
- **Key Components:** File, Directory, FileSystem, FileDescriptor



# Use Case Diagram

The Use Case Diagram provided represents the structure and relationships within a simple filesystem.

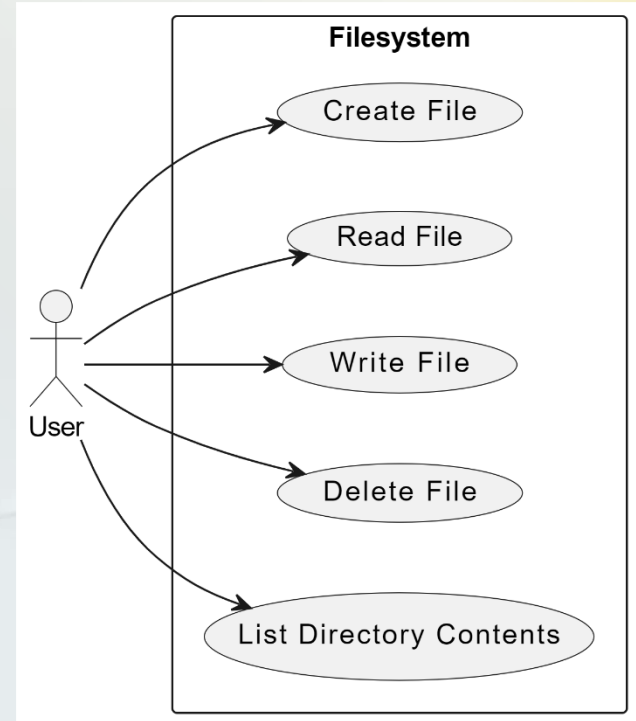
- **Description:** Illustrates user interactions with the filesystem.
- **Key Components:** Create File, Read File, Write File, Delete File, List Directory Contents

**Creating Files:** Users can create new files within directories.

**Reading Files:** Accessing and reading file contents.

**Writing to Files:** Updating or adding content to existing files.

**Deleting Files:** Removing files from the filesystem.



## Key Features

- **File operations:** Create, read, write, delete files.
- **Directory management:** Create, list, manage nested directories.
- **Error handling:** Gracefully manage invalid operations.
- **Performance optimization:** Minimize overhead and maximize efficiency.

# Implementation Phase

## Filesystem.hpp

```
1 #ifndef FILESYSTEM_HPP
2 #define FILESYSTEM_HPP
3
4 #include <string>
5 #include <unordered_map>
6
7 class File {
8 public:
9     std::string name;
10    std::string content;
11
12    File(const std::string &name) : name(name), content("") {}
13 };
14
15 class Directory {
16 public:
17     std::unordered_map<std::string, File> files;
18     std::unordered_map<std::string, Directory> directories;
19 };
20
21 class FileSystem {
22 private:
23     Directory root;
24
25     Directory* navigateToDirectory(const std::string &path);
26 public:
27     bool createFile(const std::string &path);
28     bool writeFile(const std::string &path, const std::string &content);
29     std::string readFile(const std::string &path);
30     bool deleteFile(const std::string &path);
31 };
32
33 #endif // FILESYSTEM_HPP
34
```

## Filesystem.cpp

```
1 #include "Filesystem.hpp"
2 #include <sstream>
3 #include <iostream>
4
5 Directory* FileSystem::navigateToDirectory(const std::string &path) {
6     std::stringstream iss(path);
7     std::string token;
8     Directory *current = &root;
9     while (std::getline(iss, token, '/')) {
10         if (current->directories.find(token) == current->directories.end()) {
11             current->directories[token] = Directory();
12         }
13         current = &current->directories[token];
14     }
15     return current;
16 }
17
18 bool FileSystem::createFile(const std::string &path) {
19     std::string directoryPath = path.substr(0, path.find_last_of('/'));
20     std::string fileName = path.substr(path.find_last_of('/') + 1);
21     Directory *dir = navigateToDirectory(directoryPath);
22     if (dir->files.find(fileName) != dir->files.end()) {
23         return false; // File already exists
24     }
25     dir->files[fileName] = File(fileName);
26     return true;
27 }
28
29 bool FileSystem::writeFile(const std::string &path, const std::string &content) {
30     std::string directoryPath = path.substr(0, path.find_last_of('/'));
31     std::string fileName = path.substr(path.find_last_of('/') + 1);
32     Directory *dir = navigateToDirectory(directoryPath);
33     if (dir->files.find(fileName) == dir->files.end()) {
34         return false; // File does not exist
35     }
36     dir->files[fileName].content = content;
37     return true;
38 }
39
40 std::string FileSystem::readFile(const std::string &path) {
41     std::string directoryPath = path.substr(0, path.find_last_of('/'));
42     std::string fileName = path.substr(path.find_last_of('/') + 1);
43     Directory *dir = navigateToDirectory(directoryPath);
44     if (dir->files.find(fileName) == dir->files.end()) {
45         return ""; // File does not exist
46     }
47     return dir->files[fileName].content;
48 }
49
50 bool FileSystem::deleteFile(const std::string &path) {
51     std::string directoryPath = path.substr(0, path.find_last_of('/'));
52     std::string fileName = path.substr(path.find_last_of('/') + 1);
53     Directory *dir = navigateToDirectory(directoryPath);
54     if (dir->files.find(fileName) == dir->files.end()) {
55         return false; // File does not exist
56     }
57     dir->files.erase(fileName);
58     return true;
59 }
60
```

# main.cpp

```
1 #include "FileSystem.hpp"
2 #include <iostream>
3
4 int main() {
5     FileSystem fs;
6
7     // Creating a file
8     if (fs.createFile("/home/user/test.txt")) {
9         std::cout << "File created successfully.\n";
10     } else {
11         std::cout << "Failed to create file.\n";
12     }
13
14     // Writing to a file
15     if (fs.writeFile("/home/user/test.txt", "Hello, World!")) {
16         std::cout << "File written successfully.\n";
17     } else {
18         std::cout << "Failed to write file.\n";
19     }
20
21     // Reading a file
22     std::string content = fs.readFile("/home/user/test.txt");
23     if (!content.empty()) {
24         std::cout << "File content: " << content << "\n";
25     } else {
26         std::cout << "Failed to read file.\n";
27     }
28
29     // Deleting a file
30     if (fs.deleteFile("/home/user/test.txt")) {
31         std::cout << "File deleted successfully.\n";
32     } else {
33         std::cout << "Failed to delete file.\n";
34     }
35
36     return 0;
37 }
38
```

# Make File

```
1 CXX = g++
2 CXXFLAGS = -std=c++11 -Wall
3
4 SRC = src/FileSystem.cpp src/main.cpp
5 TEST_SRC = tests/test_FileSystem.cpp src/FileSystem.cpp
6 OBJ = $(SRC:.cpp=.o)
7 EXEC = filesystem
8 TEST_EXEC = test_filesystem
9
10 all: $(EXEC)
11
12 $(EXEC): $(OBJ)
13     $(CXX) $(OBJ) -o $@ $(CXXFLAGS)
14
15 %.o: %.cpp
16     $(CXX) $(CXXFLAGS) -c $< -o $@
17
18 clean:
19     rm -f $(OBJ) $(EXEC) $(TEST_EXEC)
20
21 test: $(TEST_EXEC)
22
23 $(TEST_EXEC): $(TEST_SRC)
24     $(CXX) $(TEST_SRC) -o $@ $(CXXFLAGS) -I$(CATCH_INCLUDE_DIR)
25
26 .PHONY: clean all test
27
```

## Output :



1 File created successfully.

2 File written successfully.

3 File content: Hello, World!

4 File deleted successfully.



5






6 =====

7 All tests passed (4 assertions in 1 test case)








# Version Control


 marwaawil / SimpleFileSystem



[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#) [Insights](#) [Settings](#)

 **SimpleFileSystem** Public


 Pin  Unwatch **1**  Fork **0**  Star **0**



### Set up GitHub Copilot

Use GitHub's AI pair programmer to autocomplete suggestions as you code.

[Get started with GitHub Copilot](#)






### Add collaborators to this repository

Search for people using their GitHub username or email address.

[Invite collaborators](#)

### Quick setup — if you've done this kind of thing before

 Set up in Desktop or  HTTPS  SSH

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

# Testing Approach

**Description:** Catch2 for comprehensive unit testing.

## Testing Types:

- **Unit Testing:** Validate individual components (File, Directory, FileSystem) for correctness.
- **Integration Testing:** Ensure seamless interaction between filesystem components.
- **Edge Case Testing:** Handle unusual scenarios (e.g., large files, nested directories) to verify robustness.

## Testing cases

**Description:** Comprehensive coverage confirming functionality and robustness.

### Evidence

- **File Operations:** Successful creation, reading, writing, and deletion.
- **Directory Operations:** Accurate directory creation and listing.
- **Error Handling:** Appropriate responses to invalid operations.
- **Edge Case Testing:** Handles unexpected inputs gracefully.

## Challenges Faced

- **Optimizing performance for large files:** Implemented efficient algorithms and data structures to handle large files.
- **Enhancing error handling mechanisms:** Developed robust error-handling strategies to improve system reliability.
- **Managing complex directory structures:** Applied modular design to simplify directory management and improve scalability.

## Conclusion

The filesystem project successfully implemented essential file and directory management functionalities for embedded systems, demonstrating robust performance and reliability. Future improvements will focus on optimizing for larger files, enhancing error handling, and refining modular design for scalability.

## Future Approach

- Prioritize performance optimizations.
- Develop robust error handling mechanisms.
- Emphasize modular design for scalability.

# Questions & Answers

**Thank You**