МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютерної інженерії

Курсовий проект

з дисципліни "Системне програмування"

на тему: "Розробка програмного забезпечення   
файлового менеджера"

Виконав:

студент IV курсу групи ІПЗ-13

напряму підготовки ПІ

спеціальності ІПЗ

Кривенко О. М.

Перевірила:

старший викладач кафедри КІ

Шевченко О. Г.

|  |  |  |  |
| --- | --- | --- | --- |
| Члени комісії |  |  |  |
| (підпис) | (прізвище та ініціали) |
|  |  |
| (підпис) | (прізвище та ініціали) |
|  |  |
| (підпис) | (прізвище та ініціали) |

Покровськ 2016

Затверджую

Зав. кафедри КІ \_\_\_\_\_\_ Святний В.А.

\_\_\_\_\_\_\_\_\_\_\_ 2016 року.

# **Технічне завдання**

**до курсового проекту з дисципліни «Системне програмування»**

Тема роботи: Розробка програмного забезпечення файлового менеджера.

Студента групи ІПЗ-13 Кривенко Олександра Миколайовича

Термін здачі роботи на перевірку: згідно графіку виконання.

Варіант - №3

***Мета*** - дослідження схем сегментації пристроїв зовнішньої пам’яті та фізичних моделей файлових систем ОС Windows.

***Практичний результат*** - розробка програмного забезпечення файлового менеджера (ФМ), що реалізує наступні функції:

* пошук і найменування **всіх** логічних дисків у межах даної конфігурації технічних засобів;
* визначення характеристик логічних дисків;
* навігація по дереву файлової системи;
* пошук файлів імена, яких задовольняють шаблону.

***Вимоги до виконання:***

* 1. Програмне забезпечення ФМ розробляється як графічний WIN -додаток.
  2. Мова програмування – будь-яка мова класу С, інструментальне середовище – згідно обраної мови
  3. Кожна з зазначених вище функцій ФМ повинна ініціюватися тільки за вимогою користувача.
  4. Вихідний інтерфейс розробляється в традиційному стилі відомих файлових менеджерів (FAR, WC, провідник, тощо).
  5. Після запуску ФМ на екрані відображається «список» об'єктів файлової системи (ФС) кореневого каталогу завантажувального диску.
  6. ФМ підтримує логічні диски з файловою системою FAT32, NTFS.
  7. Доступ до об'єктів файлової системи FAT32 виконується тільки через аналіз службової інформації диску.
  8. Доступ до об'єктів файлової системи NTFS виконується з використанням засобів мови програмування. Індивідуальне. завдання для диску з NTFS- не реалізується.
  9. Імена логічних дисків повинні збігатися з іменами відповідних дисків у рамках ОС, де функціонує ФМ.
  10. Ідентифікація типів об'єктів ФС у вихідному інтерфейсі виконується або по графічному відображенню їхніх імен (колір, шрифт, іконки, тощо), або явною вказівкою типу (файл, директорія).
  11. Повнота інформації про об'єкт ФС визначається індивідуальним завданням і повинна бути достатньою для перевірки його виконання.
  12. Характеристики логічних дисків відображаються в форматі таблиці.
  13. У вихідному інтерфейсі зобов'язаний відображатися поточний шлях дерева ФС.
  14. Обов'язкова підтримка довгих імен і кирилиці.
  15. Довге ім'я об'єкта повинно бути представлене по всій довжині.
  16. У вихідному інтерфейсі інформація індивідуального завдання і характеристики диску не повинна затирати список об'єктів ФС поточної директорії.
  17. У ФМ повинні бути передбачені індикатори виконання тривалих за часом операцій.
  18. Колірна палітра вхідного/вихідного інтерфейсу повинна бути витримана в спокійних тонах.

***Примітки***.

1. Робота ФМ повинна бути передбачувана.
2. Керуючі компоненти вихідного інтерфейсу повинні бути інформативні, повідомлення про можливі помилки - лаконічними

***Зміст пояснювальної записки:***

Титульний лист.

Технічне завдання.

Реферат.

Вступ.

1. Структура програмного забезпечення файлового менеджеру.

2. Структура даних.

3. Опис алгоритмів ПЗ ФМ.

4. Опис програмних модулів.

5. Методика роботи.

6. Дослідження результатів.

Висновок.

Список використаної літератури.

Додатки.

***Склад графічної частини***

1. Структура програмного забезпечення (А2 чи А3).
2. Схеми-алгоритмів (А2 чи А3).
3. Плакат з вихідними результатами.

***Графік виконання***

1. Розробка структури програмного забезпечення (1-тиждень)
2. Розробка алгоритмів інтерфейсу файлового менеджера. (1-тиждень)
3. Програмування та автономне налагодження вхідного та

вихідного інтерфейсу. (2-тижні)

1. Розробка алгоритмів доступу до об'єктів файлової системи. (1-тиждень)
2. Програмування та автономне налагодження модулів доступу (2-тижні)
3. Комплексне налагодження програмного забезпечення (3-тижні)
4. Оформлення пояснювальної записки та графічної частини (1-тиждень)
5. Захист курсової роботи.

Керівник роботи \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Завдання прийняв до виконання \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Реферат

*Змн.*

*Арк.*

*№ докум.*

*Підпис*

*Дата*

КП 6.050102.136404.ПЗ

*Розроб.*

Кривенко О.М.

*Перевір.*

Шевченко О. Г.

*Н. Контр.*

Програмне забезпечення файлового менеджера

Пояснювальна записка

*Літ.*

*ДонНТУ, гр.ІПЗ-13*

*Лист*

*Листів*

стор. 50, рис. 21, табл. 13, додатків 2.

Метою курсового проекту є розробка файлового менеджера, що реалізує наступні функції: формування списку фізичних та логічних дисків з вказанням загальних параметрів для них; навігація за деревом файлової системи; пошук файлів, імена яких задовольняють шаблону.

Об’єктом дослідження є структура збереження інформації у дискових накопичувачах, а предметом дослідження виступають розмітки диску типу MBR та GPT, а також файлові системи FAT16, FAT32, exFAT і NTFS.

Методи дослідження – аналітичний (ознайомлення з документацією та специфікаціями), емпіричний (ознайомлення з системними структурами даних дисків) та моделювання (написання файлового менеджеру).

Результатами роботи є програмне забезпечення та документація до нього, що реалізує перегляд інформації про жорсткі диски комп’ютера, а також навігацію по дереву файлової системи.

ФАЙЛОВИЙ МЕНЕДЖЕР, ЖОРСТКИЙ ДИСК, ЛОГІЧНИЙ ДИСК, ФАЙЛОВА СИСТЕМА, ФАЙЛ, ДИРЕКТОРІЯ, ФАЙЛОВИЙ ДЕСКРИПТОР, КЛАСТЕР, СЕКТОР, ВООТ‑СЕКТОР, PARTITION TABLE, ТАБЛИЦЯ FAT.

# Зміст

[Вступ 4](#_Toc470229313)

[1 Програмна структура файлового менеджеру 6](#_Toc470229314)

[2 Структури даних 8](#_Toc470229315)

[2.1 Структура MBR розмітки диску 8](#_Toc470229316)

[2.1.1 Legacy MBR та структура EBR 8](#_Toc470229317)

[2.1.2 Protective MBR 9](#_Toc470229318)

[2.2 Структура GPT розмітки диску 9](#_Toc470229319)

[2.2.1 Заголовок таблиці розділів 9](#_Toc470229320)

[2.2.2 Записи розділів 11](#_Toc470229321)

[2.3 Структури ФС сімейства FAT (FAT12/FAT16/FAT32) 12](#_Toc470229322)

[2.3.1 Структура завантажувального сектору 12](#_Toc470229323)

[2.3.2 Структура інформаційного сектору ФС 15](#_Toc470229324)

[2.3.3 Структура директорії 16](#_Toc470229325)

[2.4 Структури інших підтримуваних файлових систем 20](#_Toc470229326)

[2.4.1 Файлова система exFAT 20](#_Toc470229327)

[2.4.2 Файлова система NTFS 22](#_Toc470229328)

[3 Опис алгоритмів 25](#_Toc470229329)

[3.1 Отримання списку фізичних дисків 25](#_Toc470229330)

[3.2 Отримання списку логічних дисків 26](#_Toc470229331)

[3.3 Отримання системного імені диску 29](#_Toc470229332)

[3.4 Визначення файлової системи 30](#_Toc470229333)

[3.5 Отримання записів директорії засобами WinAPI 33](#_Toc470229334)

[3.6 Отримання списку послідовних кластерів запису ФС FAT32 34](#_Toc470229335)

[3.7 Отримання записів директорії для ФС FAT32 36](#_Toc470229336)

[3.8 Пошук за шаблоном 38](#_Toc470229337)

[4 Опис програмних модулів 39](#_Toc470229338)

[4.1 Точка входу до програми 39](#_Toc470229339)

[4.2 Інтерфейс користувача 39](#_Toc470229340)

[4.3 Модулі структур 40](#_Toc470229341)

[4.4 Логіка програми 40](#_Toc470229342)

[5 Методика роботи 44](#_Toc470229343)

[6 Дослідження результатів 45](#_Toc470229344)

[Висновки 49](#_Toc470229345)

[Список використаної літератури 50](#_Toc470229346)

[Додаток А Вихідний код](#_Toc470229347)

[Додаток Б Графічна частина](#_Toc470229357)

# Вступ

Питання логічної структури носіїв цифрової інформації є одним із базисних питань комп’ютерної архітектури. Зазвичай кожен сучасний жорсткий диск, який підтримується MS Windows, має на собі структуру MBR (Master Boot Record), яка зберігає параметри логічної розмітки жорсткого диску. Структура MBR буває двох базових типів Legacy MBR та Protective MBR.

Структура Legacy MBR зберігає у собі чотири первині розділи жорсткого диску. Первинний розділ жорсткого диску може у собі зберігати логічний розділ, чи розширений розділ жорсткого диску, що зберігає запис, який називається EBR (Extended Boot Record). Цей запис за структурою відповідний запису MBR.

Структура Protective MBR позначає, що диск розмічений за допомогою розмітки GPT (GUID Partition Table), яка є частиною інтерфейсу EFI (Extensible Firmware Interface), який прийшов на заміну застарілому BIOS. У такому випадку у структурі MBR диску збережена інформація про параметри GPT та вона призначена для захисту носія. Структура GPT включає в себе Header, в якому зберігається системна інформація, а також дві структури Partition Table, у якій зберігається інформація про логічні розділи [1, 2].

Одну з цих двох розміток можна знайти на більшості сучасних комп’ютерів. Незважаючи на різницю цих двох розміток вони виконують однакову роль, а саме призначенні для збереження інформації про логічні розділи диску.

Логічний розділ диску представляє собою частину довготривалої пам'яті комп'ютера, що розглядається як єдине ціле для зручності роботи. Логічні розділи зберігають у собі файлову систему. Файлова система (ФС) - це засіб для організації зберігання файлів на будь-якому носії. Сучасна MS Windows з коробки підтримує ФС сімейства FAT (File Allocation Table) – (FAT12, FAT16, FAT32, exFAT), а також журнальовану ФС NTFS (New technology file system).

ФС сімейства FAT є найбільш простими для вивчення, оскільки мають більш просту структуру ніж журнальовані файлові системи.

Одним з найкращих виборів для томів невеликого розміру є ФС FAT12/16 у порівнянні з NTFS, а ФС FAT32 є логічним продовженням ФС FAT12/16, а її можливості набагато перевищують можливості попередніх. Так, ця ФС підтримує більшу кількість кластерів, за рахунок чого може підтримувати жорсткі диски більшого розміру.

При розробці ФС exFAT не ставилась задача сумісності коду з попередніми версіями ФС, а ставилась задача зменшення не рівномірного зносу пам’яті флеш накопичувачів. Також ФС на відміну від попередників має біт карту вільних кластерів, призвану зменшити фрагментацію диску, а також додано підтримку списку прав доступу та підтримка транзакції.

ФС NTFS забезпечує таке поєднання продуктивності, надійності і ефективності, яке неможливо надати за допомогою будь-якої з реалізацій FAT. Основними цілями розробки NTFS були забезпечення швидкісного виконання стандартних операцій над файлами (включаючи читання, запис, пошук) і надання додаткових можливостей, включаючи відновлення пошкодженої ФС на надзвичайно великих дисках.

# Програмна структура файлового менеджеру

Програму, що розробляється, тобто файловий менеджер структурно можна розділити на декілька частин, а саме:

* модуль структур записів фізичних дисків;
* модуль логічних структур файлових систем;
* модуль програмної логіки для роботи з жорстким диском;
* модуль програмної логіки для роботи з логічними дисками;
* модуль інтерфейсу користувача.

Детальна структура кожного модуля описана у відповідному розділі нижче та представлена на рис. Рисунок 1.1.



Рисунок . – Програмна структура файлового менеджеру

Модуль структур записів фізичних дисків включає до себе структури даних, які описують записи MBR та GPT розміток дисків, що будуть описані в пунктах 2.1 та 2.2 відповідно. Цей модуль використовуються у модулі програмної логіки для роботи з жорсткими дисками.

Модуль структур записів файлових систем відповідає за структури даних, що описують записи ФС, що підтримуються програмою. А саме файлових систем: FAT16, FAT32, exFAT, NTFS. Цей модуль використовується у модулі програмної логіки для роботи з жорсткими дисками. Структури даного модулю описані в пунктах 2.3 та 2.4.

Модуль програмної логіки для роботи з жорсткими дисками зберігає у собі логіку для пошуку фізичних дисків, підключених до комп’ютера, а також логіку для отримання параметрів даних дисків і інформацію про розділи диску. Алгоритми, описані у даному модулі, наведені далі.

Модуль програмної логіки для роботи містить у собі програмну логіку для отримання інформації про логічний диск та отримання елементів директорії.

Модуль інтерфейсу користувача слугує для відображення отриманих за допомогою описаних вище модулів даних та для обробки дії користувача.

# Структури даних

## Структура MBR розмітки диску

### Legacy MBR та структура EBR

Структура MBR та EBR є окремими випадками структури DBR (Drive Boot Record). Ця структура містить у собі код і дані, необхідні для подальшого завантаження ОС та інформацію про логічні розділи диску. Структура MBR розташована в перших фізичних секторах (найчастіше в найпершому) на пристроях для зберігання інформації [1].

Структура DBR переставлена у програмі структурою note, що знаходиться у просторі імен dbr. Опис даної структури наведений у таблиці Таблиця 2.1. Дана структура містить у собі інформацію про логічні диски, що представлені структурою part, що знаходиться у просторі імен dbr. Опис даної структури можна знайти у таблиці Таблиця 2.2.

Таблиця . – Класична структура DBR

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Зміщення | Розмір, байт | Тип даних | Поле структури | Опис |
| 0x0000 | 446 | BYTE[446] | boot | Код завантажувача |
| 0x01BE | 16 | dbr::part[4] | part | Таблиця розділів диску |
| 0x01FE | 2 | WORD | sign | Сигнатура (0x55AA) |

Таблиця . – Структура запису розділів

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Зміщення | Розмір, байт | Тип даних | Поле структури | Опис |
| 1 | 2 | 3 | 4 | 5 |
| 0x00 | 1 | BYTE | boot\_fl | Признак активності розділу, якщо 0x00 – розділ звичайний, а якщо 0x80 – активний |

Продовження Таблиця 2.2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| 0x01 | 3 | \_chs | beg\_chs | Початок розділу CHS (зараз не використовується) |
| 0x04 | 1 | BYTE | os\_type | Код типу розділу |
| 0x05 | 3 | \_chs | end\_chs | Кінець розділу CHS (зараз не використовується) |
| 0x08 | 4 | DWORD | beg\_lba | Зсув першого сектору розділу |
| 0x0C | 4 | DWORD | lba\_sct | Кількість секторів у розділі |

### Protective MBR

Структура запису Protective MBR ні яким чином не відрізняється від структури запису Legacy MBR. Основна мета структури Protective MBR є захист від переписування диску з не MBR розміткою при застосуванні MBR-орієнтованих дискових утиліт [2].

У таблиці розділів такого диску зберігається інформація про всього один розділ, що охоплює весь диск. Код типу для цього розділу встановлюється в значення 0xEE, яке вказує, що застосовується не MBR диск. А зсув першого кластеру встановлюється на початок розмітки диску.

Деякі операційні системи та утиліти не пристосовані для читання дисків, що містять захищений MBR, проте розпізнають код типу розділу і представляють його як недоступний логічний диск. Таким чином запобігається випадкове стирання вмісту не MBR диску.

## Структура GPT розмітки диску

### Заголовок таблиці розділів

GPT − це стандарт формату розміщення таблиць розділів на фізичному жорсткому диску. Він є частиною Extensible Firmware Interface від Intel.

Структура заголовку таблиці розділів GPT у програмі представлена структурою head, що знаходиться у просторі імен gpt та наведена у таблиці Таблиця 2.3.

Таблиця . – Структура заголовку GPT

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Зміщення | Розмір, байт | Тип даних | Поле структури | Опис |
| 0x00 | 8 | CHAR[8] | head\_sign | Рядок сигнатури, що дорівнює "EFI PART" |
| 0x08 | 4 | DWORD | gpt\_vers | Версія розмітки (актуальна 1.0) |
| 0x0C | 4 | DWORD | head\_size | Розмір структури заголовку |
| 0x10 | 4 | DWORD | check\_sum | CRC32 хеш сума заголовку |
| 0x14 | 4 | DWORD | nt\_reserve | Завжди 0 |
| 0x18 | 8 | QWORD | this\_lba | Зсув цього заголовку |
| 0x20 | 8 | QWORD | copy\_lba | Зсув копії заголовку |
| 0x28 | 8 | QWORD | use\_beg\_lba | Зсув першого кластеру даних |
| 0x30 | 8 | QWORD | use\_end\_lba | Зсув останнього кластеру даних |
| 0x38 | 16 | \_guid | guid | GUID даного диску |
| 0x48 | 8 | QWORD | parts\_lba | Позиція розділів диску цієї копії |
| 0x50 | 4 | DWORD | parts\_num | Кількість розділів |
| 0x54 | 4 | DWORD | part\_size | Розмір інформації про розділ |
| 0x58 | 4 | DWORD | parts\_sum | CRC32 хеш сума розділів |

Заголовок таблиці розділів GPT визначає використовувані блоки на диску, а також містить кількість і розмір записів розділів, які складають таблицю розділів. EFI передбачає як мінімум 16384 байт зарезервованих для масиву таблиці розділів, таким чином, є 128 записів розділів, кожен з яких в 128 байтів в довжину.

Заголовок диску містить GUID розділу. Також у заголовку записано свій власний розмір і місце розташування цього заголовку на диску, а також розмір і місце розташування вторинного GPT заголовка і таблиці.

Важливо відзначити, що він також містить контрольну суму CRC32 для себе і для таблиці розділів, які можуть бути перевірені за допомогою програмно-апаратних засобів, завантажувача операційної системи або при завантаженні системи. Через це, HEX редактори не можуть бути використані для модифікації вмісту GPT. Така модифікація зробить контрольну суму недійсною. В цьому випадку первинна GPT може бути перезаписана вторинною за допомогою програмного забезпечення для відновлення даних.

Якщо обидва GPT містять неприпустимі контрольні суми, то багато завантажувачів (зокрема ті, що регулюють моделі цілісності) і операційних систем будуть відмовлятися працювати з диском, поки пошкоджені таблиці не будуть відремонтовані або видалені.

### Записи розділів

Структура запису таблиці розділів GPT у програмі представлена структурою part, що знаходиться у просторі імен gpt та наведена у таблиці Таблиця 2.4.

Після заголовку знаходиться таблиця розділів GPT. Записи розділів розміщуються послідовно заповнюючи сектори таблиці розділів. Один запис розділу займає 128 байтів. Тобто, для дисків з класичним розміром сектору в 512 байт на сектор приходиться 4 записи, а на дисках Advanced Format з більшим розміром фізичного та логічного секторів знаходиться більше записів.

Перші 16 байтів запису розділу позначають тип розділу за допомогою глобально унікального ідентифікатора (GUID). Інші 16 байт містять GUID для ідентифікації розділу. За ними йдуть початкова і кінцева 64-розрядні адреси, атрибути і ім’я розділу.

Таблиця . – Структура запису

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Зміщення | Розмір, байт | Тип даних | Поле структури | Опис |
| 0x00 | 16 | \_guid | type\_guid | GUID типу розділу |
| 0x10 | 16 | \_guid | part\_guid | Унікальний GUID ідентифікатор розділу |
| 0x20 | 8 | QWORD | beg\_lba | Зсув початку розділу |
| 0x28 | 8 | QWORD | end\_lba | Зсув кінця розділу |
| 0x30 | 8 | QWORD | options | Атрибути розділу |
| 0x38 | 72 | WCHAR[36] | name | Назва розділу |

## Структури ФС сімейства FAT (FAT12/FAT16/FAT32)

### Структура завантажувального сектору

Перша важлива структура на FAT диску називається BPB (BIOS Parameter Block), яка розташована в першому секторі диска, в Reserved Region. Цей сектор ще іноді називають "boot сектор", "reserved sector" або "0th sector," але головне лише те, що це перший сектор диска.

FAT32 BPB цілком відповідає FAT12 / FAT16 BPB, і доповнений полем BPB\_TotSec32. Вони відрізняються, починаючи зі зсуву 36, де вже залежить від типу FAT12/FAT16 або FAT32 (опис визначення типу FAT буде нижче). Головне, що BPB в boot секторі FAT диска завжди містить всі поля для FAT12/FAT16 або FAT32 BPB типів. Таким чином, забезпечена максимальна сумісність FAT дисків, драйвери файлових систем FAT будуть правильно визначати і підтримувати структуру, тому що вона містить зумовлені поля [3, 4].

Структура boot сектора FAT дисків наведена у таблицах Таблиця 2.5 – Таблиця 2.7.

Таблиця . – Початок boot сектору FAT дисків

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Зміщення | Розмір, байт | Тип даних | Поле структури | Опис |
| 0x0000 | 3 | BYTE[3] | BS\_JmpBoot | Jump інструкція на boot code |
| 0x0003 | 8 | CHAR[8] | BS\_OEMName | OEM ім'я |
| 0x000B | 2 | WORD | BPB\_BytsPerSec | Ступінь 2. Мінімум 9 (512 байт на сектор), максимум 12 (4096 байт на сектор) |
| 0x000D | 1 | BYTE | BPB\_SecPerClus | Ступінь 2. Мінімум 0 (1 сектора в кластері), а максимум 25, максимальний розмір кластера становить 32 МБ |
| 0x000E | 2 | WORD | BPB\_RsvdSecCnt | Кількість секторів в Reserved region (починається з першого сектора диска). Повинно бути більше 0. |
| 0x0010 | 1 | BYTE | BPB\_NumFATs | Кількість таблиць FAT на диску. Повинно бути 2 для будь-якої FAT. Хоча і допустимі і інші значення. |
| 0x0011 | 2 | WORD | BPB\_RootEntCnt | Для FAT12 і FAT16 дисків, це поле містить число 32-байтних елементів кореневій директорії. Для FAT32 дисків, це поле повинно бути 0. |
| 0x0013 | 2 | WORD | BPB\_TotSecSm | Старе 16-бітове поле: загальна кількість секторів на диску. |
| 0x0015 | 1 | BYTE | BPB\_MediaDesc | Медіа дескриптор диску |
| 0x0016 | 2 | WORD | BPB\_FATSizeSm | Для FAT12 / FAT16 це кількість секторів однієї FAT. Для FAT32 це значення дорівнює 0, а кількість секторів однієї FAT міститься в BPB\_FATSzLr. |
| 0x0018 | 2 | WORD | BPB\_SecPerTrk | Секторів на доріжці. Це поле має відношення до дисків з геометрію. |
| 0x001A | 2 | WORD | BPB\_NumOfHead | Кількість головок. Це поле має таке ж значення, як і BPB\_SecPerTrk. |
| 0x001C | 4 | DWORD | BPB\_HiddenSec | Кількість прихованих секторів, перед початком даного розділу диска. |
| 0x0020 | 4 | DWORD | BPB\_TotSecLr | Нове 32-бітове поле: загальна кількість секторів на диску. |

Таблиця . – Кінець boot сектору для FAT12/FAT16 дисків

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Зміщення | Розмір, байт | Тип даних | Поле структури | Опис |
| 0x0024 | 1 | BYTE | BPB\_DrvNum | Номер диску. |
| 0x0025 | 1 | BYTE | BPB\_CurrHead | Поточна головка. Це поле має відношення до дисків з геометрію. |
| 0x0026 | 1 | BYTE | BPB\_BootSig | Додаткова сигнатура (0x29). Байт є індикатором того, що нижченаведені 3 поля присутні. |
| 0x0027 | 4 | DWORD | BPB\_VolSerial | Серійний номер диска. |
| 0x002B | 11 | CHAR[11] | BPB\_VolLabel | Ім'я диска. Це ім'я збігається з 11-байтним ім'ям, прописаним в кореневій директорії. |
| 0x0036 | 8 | CHAR[8] | BPB\_SystemID | Назва ФС |
| 0x003D | 420 | BYTE[420] | BS\_Boot | Код завантажувача |
| 0x01FE | 2 | WORD | BS\_Sign | Сигнатура (0x55AA) |

Таблиця . – Кінець boot сектору для FAT32 дисків

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Зміщення | Розмір, байт | Тип даних | Поле структури | Опис |
| 1 | 2 | 3 | 4 | 5 |
| 0x0024 | 4 | DWORD | BPB\_FATSzLr | Це поле є тільки в FAT32, і відсутня в FAT12 і FAT16. Містить кількість секторів однієї FAT. |
| 0x0028 | 2 | WORD | BPB\_ExtFlags | Параметри FAT32 |
| 0x002A | 2 | WORD | BPB\_FSVer | Це поле є тільки в FAT32, і відсутня в FAT12 і FAT16. У старшому байті: номер версії. Молодший байт: номер проміжної версії. Це версія FAT32. |
| 0x002C | 4 | DWORD | BPB\_RootClus | Це поле є тільки в FAT32, і відсутня в FAT12 і FAT16. Номер першого кластера кореневої директорії. Зазвичай 2, але може бути і іншим. |

Продовження Таблиця 2.7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| 0x0030 | 2 | WORD | BPB\_FSInfo | Це поле є тільки в FAT32, і відсутня в FAT12 і FAT16. Номер сектора зі структурою FSINFO в зарезервованої частини FAT32. Зазвичай 1. |
| 0x0032 | 2 | WORD | BPB\_BkBootSec | Це поле є тільки в FAT32, і відсутня в FAT12 і FAT16. Якщо не нуль, то це номер сектора в резервної області диска, де зберігається копія boot сектора. Зазвичай 6. Інші значення не рекомендуються. |
| 0x0034 | 12 | BYTE[12] | BPB\_Reserved | Це поле є тільки в FAT32, і відсутня в FAT12 і FAT16. Зарезервовано для майбутнього розширення. |
| 0x0040 | 1 | BYTE | BPB\_DrvNum | Ці поля такі ж, як в FAT12 і FAT16. Відмінність тільки в тому, що у них інший зсув. |
| 0x0041 | 1 | BYTE | BPB\_CurrHead |
| 0x0042 | 1 | BYTE | BPB\_BootSig |
| 0x0043 | 4 | DWORD | BPB\_VolSerial |
| 0x0047 | 11 | CHAR[11] | BPB\_VolLabel |
| 0x0052 | 8 | CHAR[8] | BPB\_SystemID |
| 0x0090 | 448 | BYTE[420] | BS\_Boot |
| 0x01FE | 2 | WORD | BS\_Sign |

### Структура інформаційного сектору ФС

На диску FAT32, таблиця FAT може мати значний розмір, на відміну від FAT16, обмеженого розміром 128K, і FAT12, обмеженого розміром 6K. Тому є резон зберігати "останнє відому" кількість вільних кластерів диска FAT32, щоб не обчислювати його кожного разу, коли функції API запитують розмір вільного місця на диску (наприклад, в кінці виведення вмісту директорії). Номер сектора FSInfo зберігається в поле BPB\_FSInfo; в Microsoft ОС воно завжди дорівнює 1.

Таблиця . – Структура сектору FSInfo

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Зміщення | Розмір, байт | Тип даних | Поле структури | Опис |
| 0x0000 | 4 | DWORD | FSI\_LeadSig | Значення 0x41615252. Це початкова сигнатура для визначення сектора. |
| 0x0004 | 480 | BYTE[480] | FSI\_Reserved1 | Зарезервовано на майбутнє. |
| 0x01E4 | 4 | DWORD | FSI\_StructSign | Значення 0x61417272. Ця сигнатура для визначення положення наступних полів. |
| 0x01E8 | 4 | DWORD | FSI\_FreeCount | Зберігає "останнє відоме" кількість вільних кластерів диска. Якщо одно 0xFFFFFFFF, то кількість невідомо, і має бути обчислено. |
| 0x01EC | 4 | DWORD | FSI\_FirstFree | Містить номер кластера, починаючи з якого треба шукати вільний кластер. Зазвичай тут номер останнього виділеного кластера. Якщо тут значення 0xFFFFFFFF, то немає допоміжного значення, і пошук повинен проводитися з номера 2. |
| 0x01F0 | 12 | BYTE[12] | FSI\_Reserved2 | Зарезервовано на майбутнє. |
| 0x01FC | 4 | DWORD | FSI\_TrailSign | Значення 0xAA550000. Це кінцева сигнатура для визначення сектора. |

### Структура директорії

Директорія FAT − ні що інше як "файл", що складається зі списку 32-байтних структур. Лише одна спеціальна директорія, яка завжди має бути присутня, це коренева директорія. На дисках FAT12 і FAT16, коренева директорія розташована в фіксованому місці − безпосередньо після останньої таблиці FAT, і складається з фіксованої кількості секторів, що обчислюється з BPB\_RootEntCnt (дивись обчислення RootDirSectors вище). Для дисків FAT12 і FAT16, номер першого сектора кореневої директорії залежить від номера першого сектора таблиці FAT:

FirstRootDirSecNum = BPB\_ResvdSecCnt + (BPB\_NumFATs \* BPB\_FATSz16);

Для FAT32, коренева директорія може бути довільного розміру з послідовності кластерів, так само як будь-яка інша директорія. Номер першого кластера кореневої директорії FAT32 зберігається в BPB\_RootClus. На відміну від інших директорій, коренева директорія не має штампа дати і часу, не має імені (крім неявного імені "\"), і не містить "." І ".." в перших двох записах.

Таблиця . – Структура файлового запису

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Зміщення | Розмір, байт | Тип даних | Поле структури | Опис |
| 0x0000 | 11 | CHAR | DI\_ShortName | Коротке ім'я |
| 0x000B | 1 | BYTE | DI\_Attributes | Атрибути файлу. |
| 0x000C | 1 | BYTE | DI\_NTReserved | Зарезервовано. |
| 0x000D | 1 | BYTE | DI\_CreateMS | Штамп мілі секунд поточного часу. Фактично це поле містить десяті частки секунди. Діапазон допустимих значень 0-199 включно. |
| 0x000E | 4 | \_timestump | DI\_Create | Час та дата створення файлу |
| 0x0012 | 2 | \_date | DI\_Access | Дата останнього доступу до файлу |
| 0x0014 | 2 | WORD | DI\_NumClusHI | Старше слово номера першого кластера (завжди 0 для FAT12 і FAT16). |
| 0x0016 | 4 | \_timestump | DI\_Modify | Час останньої зміни файлу |
| 0x001A | 2 | WORD | DI\_NumClusLO | Молодше слово номера першого кластера. |
| 0x001C | 4 | DWORD | DI\_FileSize | Розмір файлу в байтах. |

Перший символ DI\_ShortName має спеціальне призначення. Якщо перший символ 0xE5, то запис директорії вільна (в цьому записі немає ні файлу, ні директорії). Якщо перший символ 0x00, то запис директорії вільна (як і 0xE5), і немає існуючих записів після цієї (всі DI\_ShortName[0] байти після цього запису також рівні 0). Якщо перший символ 0x05, то справжній байт імені файлу дорівнює 0xE5. 0xE5 відповідає старшому байту японського кодування KANJI. Спеціальне значення 0x05 використано тому, що дане значення можна коректно інтерпретувати і в японському кодуванні, і драйвер FAT вважатиме запис вільним. Поле DI\_ShortName складається з двох частин, 8-символьна основна частина імені, і 3-символьне розширення. Вільні кінці обох частин заповнені байтами 0x20.

При додаванні довгих імен до ФС FAT, була задача компактного розташування на диску, поруч з асоційованим коротким ім'ям, а також задача сумісності зі старою архітектурою, що дозволило б використання старих дискових утиліт без остраху пошкодження цілісності структури ФС.

Пошкодження структури ФС можливо оскільки, дискові утиліти зазвичай не використовують API системи для доступу до структур диска, а зчитують фізичні або логічні сектора з диску, і самі аналізують їх вміст. Евристичним методом, вони можуть вжити заходів щодо «виправлення» структур, які вони вважатимуть «пошкодженими». Тому записи довгих імен були додані в систему FAT таким чином, що дані не втрачаються при «виправленні» їх утилітами, ще не сумісними з Windows 95 на старих версіях MS-DOS / Windows.

Результатом вирішення обох задач є структура довгого імені, наведена у таблиці 2.10. Запис довгого імені знаходиться поруч до записів директорії та безпосередньо перед записом файлу чи директорії до якого належить. Дана структура може бути однозначно ідентифікована, як структура довгого імені файлу.

Набір записів довгих імен завжди асоційований до запису елементу директорії, якій вони належать. Кожна складова запису довгого імені послідовно пронумерована, і останній запис в поле номера має прапор індикатор останнього запису списку. Для цього використовується поле LN\_Ordinal. Зауважимо, що поле LN\_Ordinal не може містити значення 0xE5 і 0x00. Ці значення завжди використовуються ФС для системних потреб.

Таблиця . – Структура довгого імені

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Зміщення | Розмір, байт | Тип даних | Поле структури | Опис |
| 0x0000 | 1 | BYTE | LN\_Ordinal | Порядковий номер цього запису в послідовності записів довгих імен, асоційованих із записом короткого імені в кінці списку. |
| 0x0001 | 10 | WCHAR[5] | LN\_Part1 | Символи 1-5 довгого імені в даному компоненті. |
| 0x000B | 1 | BYTE | DI\_Attribute | Атрибут довгого імені |
| 0x0C | 1 | BYTE | LN\_Type | Якщо 0, то запис є компонентом довгого імені. ПРИМІТКА: Інші значення зарезервовані на майбутнє. |
| 0x0D | 1 | BYTE | LN\_Hashsum | Контрольна сума |
| 0x0E | 12 | WCHAR[6] | LN\_Part2 | Символи 6-11 довгого імені в даному компоненті. |
| 0x1A | 2 | WORD | LN\_FstClus | Повинно бути НУЛЬ. Це помилкове значення FAT "перший кластер", і має бути нульовим для сумісності з дисковими утилітами. Для довгого імені значення не використовується. |
| 0x1C | 4 | WCHAR[2] | LN\_Part3 | Символи 12-13 довгого імені в даному компоненті. |

Також використовується 8-бітна контрольна сума, обчислена з короткого імені при створенні записів довгого і короткого імені. В обчисленні використовуються всі 11 символів короткого імені. Контрольна сума присутня в кожному записі довгого імені. Якщо в будь-якому елементі списку записів довгого імені контрольна сума не відповідає короткому імені, то довге ім'я вважається сирітським [5].

## Структури інших підтримуваних файлових систем

### Файлова система exFAT

#### Загальний опис файлової системи

Extended FAT чи exFAT – пропріетарна ФС, призначена головним чином для флеш-накопичувачів. Вперше представлена фірмою Microsoft. Розмір кластера за замовчуванням для файлової системи exFAT становить від 4 КБ до 128 КБ залежно від розміру тому.

Основними перевагами exFAT перед попередніми версіями FAT є:

* Зменшення кількості перезаписів одного і того ж сектора, це сильно пом'якшується вирівнюванням зносу (wear leveling). Це було основною причиною розробки exFAT.
* Теоретичний ліміт на розмір файлу 264 байт (16 ексабайт).
* Максимальний розмір кластера збільшений до 225 байт (32 мегабайта).
* Поліпшення розподілу вільного місця за рахунок введення біт-карти вільного місця, що може зменшувати фрагментацію диска.
* Введена підтримка списку прав доступу.
* Підтримка транзакцій (опціональна можливість, повинна підтримуватися пристроєм).

#### Структура завантажувального сектору

Boot сектор ФС exFAT зазнав кардинальних змін в порівнянні з попередніми версіями ФС сімейства FAT. Були прибрані поля, які містилися для сумісності коду роботи з ФС. Також було додано параметри структур, які були введені в даній ФС [5]. Структура boot сектору для exFAT представлена у таблиці Таблиця 2.11.

Таблиця . – Структура boot сектору для exFAT

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Зміщення | Розмір, байт | Тип даних | Поле структури | Опис |
| 1 | 2 | 3 | 4 | 5 |
| 0x0000 | 3 | BYTE[3] | BS\_JmpBoot | Jump інструкція на boot code |
| 0x0003 | 8 | CHAR[8] | BS\_OEMName | OEM ім'я |
| 0x000B | 53 | BYTE[53] | PBP\_AlwaysNull | Нульові байти |
| 0x0040 | 8 | QWORD | BPB\_PrtOffset | Початковий сектор кластерів даних. Повинен бути 0 для змінних носіїв. |
| 0x0048 | 8 | QWORD | BPB\_LengthVol | Кількість секторів у розділі |
| 0x0050 | 4 | DWORD | BPB\_FATOffset | Адреса сектору первинної FAT таблиці |
| 0x0054 | 4 | DWORD | BPB\_FATLength | Довжина FAT таблиці у секторах. Може перевищувати необхідний простір для того, щоб вирівняти другий FAT |
| 0x0058 | 4 | DWORD | BPB\_ClusHepOff | Фізичний сектор знаходження таблиці $BitMap |
| 0x005C | 4 | DWORD | BPB\_ClusCnt | Кількість кластерів. Максимальна кількість кластерів дорівнює 232 - 11 |
| 0x0060 | 4 | DWORD | BPB\_RtDirClus | Номер кластеру кореневого каталогу в кластерах |
| 0x0064 | 4 | DWORD | BPB\_VolSerial | Серійний номер диска. |
| 0x0068 | 2 | WORD | PBP\_FlSysVer | Номер версії. У старшому байті: номер версії. Молодший байт: номер проміжної версії. |
| 0x006A | 2 | WORD | PBP\_VolFlags | Параметри розділу |
| 0x006C | 1 | BYTE | PBP\_BytsPerSec | Ступінь 2. Мінімум 9 (512 байт на сектор), максимум 12 (4096 байт на сектор) |
| 0x006D | 1 | BYTE | PBP\_SecPerClus | Ступінь 2. Мінімум 0 (1 сектора в кластері), а максимум 25, максимальний розмір кластера становить 32 МБ |
| 0x006E | 1 | BYTE | BPB\_NumFATs | Кількість копії FAT |

Продовження Таблиця 2.11

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| 0x006F | 1 | BYTE | BPB\_DrvNum | Номер диску. |
| 0x0070 | 1 | BYTE | BPB\_PrcntInUse | 0..100 – відсоток кластерів, що були виділені. Округляється до цілого числа 0xFF - відсоток не доступний |
| 0x0071 | 7 | BYTE[7] | BPB\_Reserved | Зарезервовано для майбутнього розширення |
| 0x0078 | 390 | BYTE[390] | BS\_Boot | Код завантажувача |
| 0x01FE | 2 | WORD | BS\_Sign | Сигнатура (0x55AA) |

### Файлова система NTFS

#### Загальний опис файлової системи

NTFS − пропріетарна ФС розроблена корпорацією Microsoft. ФС NTFS має ряд технічних удосконалень у порівнянні з FAT і HPFS (High Performance File System), ФС отримала ряд змін, таких як поліпшення підтримки метаданих та використання складних структур даних для підвищення продуктивності, надійності та використання дискового простору, а також додаткові розширення, такі як списки контролю доступу (ACL безпеки) і журналювання [6].

#### Структура завантажувального сектору

Структура завантажувального сектору ФС представлена у таблиці Таблиця 2.12.

Таблиця . – Структура boot сектору для NTFS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Зміщення | Розмір, байт | Тип даних | Поле структури | Опис |
| 1 | 2 | 3 | 4 | 5 |
| 0x0000 | 3 | BYTE[3] | BS\_JmpBoot | Jump інструкція на boot code |
| 0x0003 | 8 | CHAR[8] | BS\_OEMName | OEM ім'я |
| 0x000B | 2 | WORD | BPB\_BytsPerSec | Ступінь 2. Мінімум 9 (512 байт на сектор), максимум 12 (4096 байт на сектор) |
| 0x000D | 1 | BYTE | BPB\_SecPerClus | Ступінь 2. Мінімум 0 (1 сектора в кластері), а максимум 25, максимальний розмір кластера становить 32 МБ |
| 0x000E | 2 | WORD | BPB\_RsvdSecCnt | Кількість секторів в Reserved region (починається з першого сектора диска). Повинно бути більше 0. |
| 0x0010 | 1 | BYTE | BPB\_Unused0 | Не використанні поля. Завжди дорівнюють 0. |
| 0x0011 | 2 | WORD | PBP\_Unused1 |
| 0x0013 | 2 | WORD | PBP\_Unused2 |
| 0x0015 | 1 | BYTE | BPB\_MediaDesc | Медіа дескриптор диску |
| 0x0016 | 2 | WORD | PBP\_Unused3 | Не використане поле. Завжди дорівнює 0. |
| 0x0018 | 2 | WORD | BPB\_SecPerTrk | Секторів на доріжці. Це поле має відношення до дисків з геометрію. |
| 0x0018 | 2 | WORD | BPB\_NumOfHead | Кількість головок. Це поле має таке ж значення, як і BPB\_SecPerTrk. |
| 0x001A | 4 | DWORD | BPB\_HiddenSec | Кількість прихованих секторів, перед початком даного розділу диска. |
| 0x001C | 4 | DWORD | PBP\_Unused4 | Не використане поле. |
| 0x0020 | 4 | DWORD | PBP\_Unused5 | Не використане поле. |

Продовження Таблиця 2.12

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| 0x0024 | 8 | QWORD | BPB\_TotalSec | Кількість секторів тому |
| 0x0028 | 8 | QWORD | PBP\_MFTPrimSecNum | Логічний номер кластера для файлу $MFT |
| 0x0030 | 8 | QWORD | PBP\_MFTMirrSecNum | Логічний номер кластера для файлу $MftMirr |
| 0x0040 | 1 | BYTE | PBP\_Unused6 | Не використане поле. |
| 0x0044 | 1 | BYTE | PBP\_ClusPerIdxBuf | Кластерів для кожного файлу запису сегмента |
| 0x0045 | 1 | BYTE | PBP\_Unused7 | Не використане поле. |
| 0x0048 | 8 | QWORD | BPB\_VolSerial | Серійний номер диска. |
| 0x0050 | 4 | DWORD | PBP\_VolHashSum | Контрольна сума |
| 0x0054 | 426 | BYTE[426] | BS\_Boot | Код завантажувача |
| 0x01FE | 2 | WORD | BS\_Sign | Сигнатура (0x55AA) |

# Опис алгоритмів

## Отримання списку фізичних дисків

Пошук логічних дисків здійснюється в циклі по всім можливим системним дескрипторам дисків. Якщо відкриття диску за дескриптором можливе, то виконується отримання параметрів заданого фізичного диску, а також пошук логічних розділів диску. Блок схема даного алгоритму наведена на рис. 3.1.



Рисунок . – Алгоритм отримання списку фізичних дисків

## Отримання списку логічних дисків

Загальний алгоритм отримання списку логічних дисків гуртований на початковому визначенні типу розмітки диска (рис. Рисунок 3.2). Для цього зчитується головний MBR запису диску. Якщо перший розділ MBR запису диску має код 0xFF, то цей диск має розмітку GPT. У інакшому випадку диск має розмітку MBR. Після визначення типу розмітки диска відбувається пошук відповідних розділів диску.



Рисунок . – Загальний алгоритм отримання списку логічних дисків

У випадку MBR розбивки диску для отримання розділів обходимо усі записи первинних розділів. Якщо первинний розділ не є контейнером, тоді додаємо його до списку розділів диску. Інакше, читаємо запис EBR контейнерного розділу та додаємо його до стеку та обходимо його розширені розділи. У випадку, якщо розділ не є контейнером додаємо його до списку розділів. Інакше, читаємо ще один запис EBR, до стеку та аналогічно обходимо його розділи. Коли ми обійшли усі розділи поточного EBR запису, то забираємо його зі стеку та продовжуємо обходити розділи попереднього EBR запису. Блок схема представлена на рис. Рисунок 3.3.



Рисунок . – Алгоритм отримання списку логічних дисків MBR

Для отримання розділів GPT диску зчитуємо заголовок GPT диску, з якого отримуємо параметри таблиці розділів диску. За допомогою цих параметрів зчитуємо таблицю розділів GPT. Після чого обходимо її не пусті записи. Якщо GUID запису розділу відповідає MBR підрозділу, то отримуємо MBR розділи за допомогою алгоритму описаного раніше. Інакше, якщо розділ не пустий, то додаємо його до списку розділів. Блок схема представлена на рис. Рисунок 1.1.



Рисунок . – Алгоритм отримання списку логічних дисків GPT

## Отримання системного імені диску

Отримання системного імені диску здійснюється за допомогою функцій WinAPI. Здійснюється пошук диску з відповідним серійним номером чи за допомогою GUID диску. Для цього обходимо усі іменовані диски за допомогою бітової маски імен дисків, яку повертає функція GetLogicalDrives. Після цього отримуємо серійний номер за допомогою GetVolumeInformation або GUID диску за допомогою функції GetVolumeNameForVolumeMountPoint. Після цього отримане значення звіряємо з відомою інформацією про логічний диск, ім’я якого ми шукаємо. Якщо значення пошуку дорівнює отриманому значенню, то ми знайшли букву диску. Блок схема алгоритму пошуку диску за серійним номером наведена на рис. Рисунок 3.5, а алгоритм пошуку за GUID не відрізняється.



Рисунок . – Алгоритм отримання системного імені диску

## Визначення файлової системи

Алгоритму визначення файлової системи по завантажувальному сектору ґрунтується на послідовній перевірці відповідності завантажувального сектору логічного диску до певної ФС. Блок схема алгоритму наведена на рис. Рисунок 3.6.



Рисунок . – Алгоритм визначення файлової системи

Перевірка завантажувального сектору ФС сімейства FAT здійснюється за допомогою перевірки вірності структури завантажувального сектору і потім за допомогою перевірки відповідності кількості кластерів даних до обмежень ФС. Блок схема алгоритму наведена на рис. Рисунок 3.7.

Перевірка на відповідність ФС exFAT та NTFS здійснюється за допомогою перевірки структури завантажувального сектору та її відповідності певним знаковим ознакам. Блок схеми алгоритмів наведені на рис Рисунок 3.8 та Рисунок 3.9 відповідно.



Рисунок . – Перевірка завантажувального сектору ФС сімейства FAT



Рисунок . – Перевірка завантажувального сектору ФС exFAT



Рисунок . – Перевірка завантажувального сектору ФС NTFS

## Отримання записів директорії засобами WinAPI

Отримання списку файлів директорії засобами WinAPI засновано на функціях FindFirstFile та FindNextFIle. Ці функції призначені для отримання файлів у певній директорії ФС.

За допомогою FindFirstFile ми знаходимо перший елемент директорії, якщо функція FindFirstFile повертає недійсний дескриптор, то директорія не існує. Інакше, через другий параметр вона повертає структуру WIN32\_FIND\_DATA. Ця структура зберігає у собі інформацію про елемент директорії, який ми зберігаємо до списку елементів директорії. Отримання інших елементів директорії здійснюється за допомогою використання функції FindNextFIle, яка повертає признак кінцевого файлу або структуру з інформацією про елемент директорії.

Блок схема алгоритму наведена на рис. Рисунок 3.10.



Рисунок . – Алгоритм отримання записів директорії засобами WinAPI

## Отримання списку послідовних кластерів запису ФС FAT32

Основна концепція файлової системи FAT полягає в тім, що кожному файлу й каталогу виділяється структура даних, що називається дескриптором. У цій структурі зберігається ім'я файлу, його розмір, початкова адреса вмісту файлу й інші метадані. Зміст файлів і каталогів зберігається в блоках даних, що називаються кластерами. Якщо файлу або каталогу виділяється більш одного кластера, інші кластери знаходять за допомогою структури даних, називаної FAT (File Allocation Table). Структура FAT використовується як для ідентифікації наступних кластерів у файлах, так і для визначення стану кластерів. Існує три версії FAT: FAT12, FAT16 і FAT32. Вони відрізняються друг від друга насамперед розміром запису у структурі FAT. Приклад зв’язків між кластерами даних наведено на рис. Рисунок 3.11.



Рисунок . – Зв’язки між кластерами даних

Тобто отримання списку послідовних кластерів запису ФС FAT32 здійснюється за допомогою послідовного читання дескриптору з FAT таблиці та запису його до списку. Формування цього списку здійснюється, доки дескриптор відповідного кластеру не буде рівним сервісному коду. У випадку, якщо цей сервісний код не дорівнює коду кінцевого кластеру, то вважаємо файл битим та повертаємо негативний флаг формування списку кластерів. Інакше, позитивний флаг формування кластерів та список кластерів файлу. Блок схема даного алгоритму наведена на рис. Рисунок 3.12.



Рисунок . – Отримання списку послідовних кластерів запису ФС FAT32

## Отримання записів директорії для ФС FAT32

Отримання списку файлів для ФС FAT32 здійснюється за допомогою читання та розбору кластерів логічного диску.

Після отримання першого кластеру директорії використовуємо описану раніше функцію для отримання послідовності кластерів, що відповідає за дану директорію. Після цього обходимо записи елементів директорії, які знаходяться в даних кластерах, доки не зустрінемо елемент з ознакою закінчення списку, або не пройдемо усі записи в кластерах директорії.

У випадку зустрічі з останнім записом довгого імені очищаємо стек записів довгого імені та додаємо до нього цей запис. Якщо ми зустріли проміжний запис довгого імені, тоді просто додаємо його до стеку. Зазначимо, що записи довгого імені розташовуються над записом файлу чи директорії у зворотньому порядку (спочатку – останній).

При зустрічі з файловим записом формуємо назву файлу або папки, а також додаємо елемент до списку записів елементів директорії з відповідним ім’ям.

Блок схема відповідного алгоритму наведена на рис. Рисунок 3.13.



Рисунок . – Отримання записів директорії для ФС FAT32

## Пошук за шаблоном

Пошук за шаблоном було реалізовано за допомогою стандартних функцій для роботи з регулярними виразами. Послідовно перевіряється значення пошуку на відповідність до вказаного регулярного виразу. Якщо значення пошуку відповідно до вказаного шаблону, то додаємо його до результатів пошуку. Блок схему алгоритму пошуку наведено на рис. Рисунок 3.14.



Рисунок . – Пошук за шаблоном

# Опис програмних модулів

## Точка входу до програми

Точка входу знаходиться у модулі MainForm.cpp. Перед ініціалізацією інтерфейсу програми перевіряється, чи належить поточна ОС до сімейства Windows NT. Якщо не належить, програма викидає помилку. Інакше завантажується головна форма програми.

## Інтерфейс користувача

Інтерфейс користувача реалізується класом MainForm, що знаходиться у просторі імен Drive. Цей клас знаходиться у модулі MainForm.h. Клас форми користувача включає до себе ряд обробників подій, які представлені як методи даного класу.

На завантаження форми та натисканні кнопки updateButton встановлено обробник loadDrive. Даний обробник події відповідає за очищення форми, а також за отримання списку фізичних та логічних дисків.

На вибір фізичного диску у елементі driveCBox встановлено обробник driveCBox\_selectAction. Цей обробник прибирає встановлені на елемент itemGView обробники, виводить до нього список дисків, після цього встановлює для нього обробники itemGView\_selectDrive та itemGView\_chooseDrive для виведення інформації про активний диск та обирання диску відповідно, а до volumeGView виводить список логічних дисків та встановлює обробник на вибір volumeGView\_selectAction.

Обробник volumeGView\_selectAction виконує майже аналогічні дії до обробників driveCBox\_selectAction, але з лише не великою різницею, яка полягає у виведенні списку логічних дисків замість фізичних, а також у встановленні обробників itemGView\_selectVolume та itemGView\_chooseDrive для itemGView, які відповідають за майже ті самі дії, що їх попередники, а замість встановлення обробника на volumeGView встановлює обробник pathTBox\_ChangePath на змінення тексту у полі pathTBox.

Обробник pathTBox\_ChangePath виконує більш відмінні дії від попередніх обробників. Він виводить список файлів та папок, які знаходяться у директорії, вказаній у pathTBox, а також знімає з itemGView всі обробники та встановлює свій обробник на itemGView\_chooseDrive, який дозволяє переміщуватися по каталогам даної директорії.

На натискання кнопки upButton встановлено обробник levelUP, що відповідає за повернення на рівень вище, ніж зараз відображується на формі. Тобто в випадку з не кореневою директорією на директорію вище, з кореневою до списку логічних дисків, у випадку з логічними дисками до списку фізичних дисків.

На змінення значення на поля searchTBox встановлено обробник itemsSearch, цей обробник виконує пошук за відповідним полем строк елементу itemGView та виводить результат до поля infoTBox.

## Модулі структур

Ці модулі представлені файлами partition.h, partition.cpp, filesystem.h та filesystem.cpp. У цих модулях описуються структури записів розмітки диску та файлових систем.

## Логіка програми

Уся базова логіка програми знаходиться у модулях drive.h та drive.cpp. Вона представлена 4 класами та набором функції. Перший клас − це клас DriveAccess. Він відповідає за доступ та роботу з фізичними дисками. Він має у собі властивості, з яких можна отримати загальні характерники фізичного диску, а також функцію ReadSectors, що призначена для читання секторів з диску. Інші класи знаходяться у просторі імен volume. Першим з них є клас part. Цей клас відповідає за роботу з логічним розділом диску. У класі доступний ряд опцій логічного розділу, а також функція ReadSectors, призначення якої відповідає аналогічній функції у класі DriveAccess, але з тією умовою, що вона читає тільки в рамках логічного диску. Також у класі доступна функція getDirItems, яка відповідає за виведення елементів відповідного каталогу логічного диску.

Також у програмі присутній ряд функції для роботи з ФС FAT32. Вони знаходяться у файлі drive.cpp. А їх опис наведено у таблиці Таблиця 4.1.

Таблиця . – Призначення функції для роботи з ФС FAT32

|  |  |
| --- | --- |
| Інтерфейс функції | Опис функції |
| 1 | 2 |
| DWORD getFAT32DataClusCout  ( const filesystem::fat32::boot &Boot ); | Призначення: Визначення кількості кластерів даних у логічному розділі з ФС FAT32  Параметри:  Boot – структура завантажувального сектору логічного диску |
| DWORD getFAT32RecsPerFATSec  ( const filesystem::fat32::boot &Boot ); | Призначення: Визначення кількості записів в одному секторі ФС FAT32  Параметри:  Boot – структура завантажувального сектору логічного диску |
| DWORD getFAT32DirRecPerClus ( const filesystem::fat32::boot &Boot ); | Призначення: Визначення кількості записів елементів директорії на один кластер елементу директорії  Параметри:  Boot – структура завантажувального сектору логічного диску |
| BOOL readFAT32BootSect ( volume::part ^Volume, filesystem::fat32::boot &Boot ); | Призначення: Зчитування завантажувального сектору логічного диску  Параметри:  Volume – клас логічного диску  Boot – структура завантажувального сектору логічного диску |

Продовження Таблиця 4.1

|  |  |
| --- | --- |
| 1 | 2 |
| BOOL readFAT32DataClus ( volume::part ^Volume, const filesystem::fat32::boot &Boot, DWORD NumClus,  LPVOID Buff ); | Призначення: Зчитування кластеру даних  Параметри:  Volume – клас логічного диску  Boot – структура завантажувального сектору логічного диску  NumClus – номер кластеру  Buff – посилання на пам'ять для записування отриманих даних |
| BOOL readFAT32ClusMark ( volume::part ^Volume,  const filesystem::fat32::boot &Boot,  DWORD NumClus,  DWORD &Mark ); | Призначення: Зчитування відмітки кластеру з FAT таблиці  Параметри:  Volume – клас логічного диску  Boot – структура завантажувального сектору логічного диску  NumClus – номер кластеру  Buff – посилання на пам'ять для записування отриманих даних |
| BOOL readFAT32ClusList ( volume::part ^Volume,  const filesystem::fat32::boot &Boot, DWORD NumClus,  Generic::List<DWORD> ^ClusList ); | Призначення: Отримання ланцюга послідовних кластерів  Параметри:  Volume – клас логічного диску  Boot – структура завантажувального сектору логічного диску  NumClus – номер кластеру  Buff – посилання на пам'ять для записування отриманих даних |
| BOOL readFAT32InfoSec ( volume::part ^Volume,  const filesystem::fat32::boot &Boot,  filesystem::fat32::info &Info ); | Призначення: Зчитування інформаційного сектору логічного диску  Параметри:  Volume – клас логічного диску  Boot – структура завантажувального сектору логічного диску  Info – структура boot сектору диску |

Продовження Таблиця 4.1

|  |  |
| --- | --- |
| 1 | 2 |
| QWORD getFAT32NumOfFreeByte ( volume::part ^Volume ); | Призначення: Отримання кількості вільних байтів на диску  Параметри:  Volume – клас логічного диску |
| String ^SmallNameString ( const CHAR SmallName[11] ); | Призначення: Формування назви файлу з короткого імені  Параметри:  SmallName – коротке ім’я |
| BYTE SmallNameHashsum ( const CHAR SmallName[11] ); | Призначення: Обчислення хеш суми короткого імені  Параметри:  SmallName – коротке ім’я |
| String ^DirItemNameString ( CHAR SmallName[11], stack<filesystem::fat32::dir\_name> FullName ); | Призначення: Формування назви файлу  Параметри:  SmallName – коротке ім’я  FullName – розділи довгого імені файлу |
| Generic::List<filesystem::object^> ^getFAT32DirIrems ( volume::part ^ Volume,  String ^ Path ); | Призначення: Отримання списку файлів та директорії за визначеним шляхом  Параметри:  Volume – клас логічного диску  Path – путь до визначеної директорії |

# Методика роботи

При розробці даного програмного забезпечення було використане наступне програмне оточення:

* операційна система – MS Windows 10;
* середа програмування – MS Visual Studio 2015.

Обмежень з боку апаратного забезпечення не встановлюється. Тому вимоги на апаратне забезпечення встановлює ОС. Слід зазначити, що програма розрахована на використання та може бути виконана в інших операційних системах Windows, які підтримують файли, компільовані за допомогою MSVS2015 64bit.

Проект MS Visual Studio представлений, як C++/CLI проект. Також у налаштуваннях проекту було змінено опцію маніфест файлу, яка відноситься до прав користувача, які потрібні для запуску програми. Тобто було змінено пункт UAC Execution Level (рис. Рисунок 5.1) до значення "highestAvalible (/level='highestAvalible')". Тобто програма вимагає максимального рівня прав, що доступні користувачу в операційній системі. Цей пункт знаходиться в під розділі Manifest file розділу Linker.

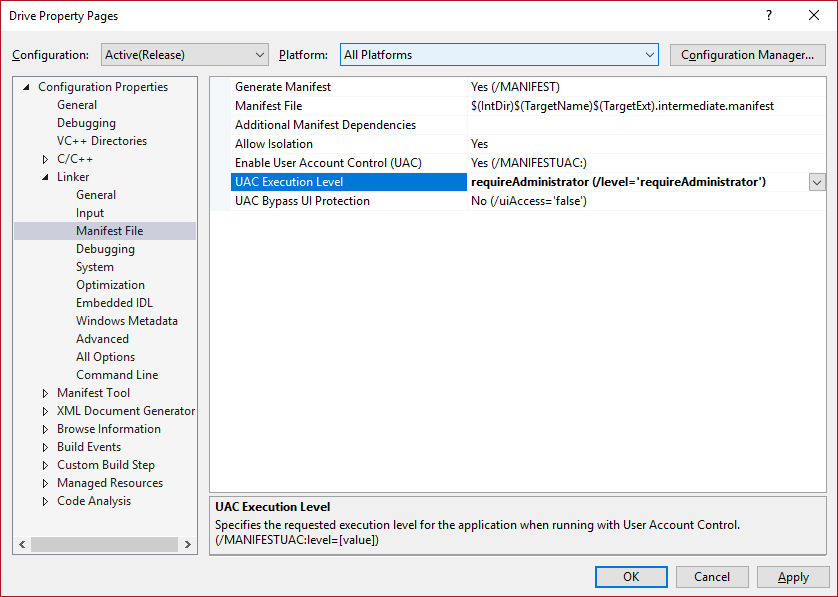


Рисунок . – Параметри маніфест файлу проекту MS Visual Studio

# Дослідження результатів

Дослідження результатів роботи програми здійснюється за двома напрямками: реалізація функцій файлового менеджеру та реалізація функцій пошуку за файлами в обраній директорії.

Для файлового менеджеру програмою для порівняння обирається Windows Explorer зі стандартного набору програм Windows. Слід зазначити, що Explorer − це програма, що має набагато більші можливості. Однак, за інтерфейсом та іншими параметрами програмне забезпечення було максимально наближено до Explorer. Суттєвою відмінністю Explorer є виклик системного контекстного меню при натисканні правої кнопки миші на об’єкті ФС, яке надає широкі можливості для роботи з даним об’єктом.

Хоча надане програмне забезпечення не має широких можливостей, але з функцією файлового менеджера як програми для перегляду вмісту дисків та каталогів воно справляється. Також слід зазначити, що відображення об’єктів ФС відбувається у тому ж обсязі, що і Explorer.

На рисунках Рисунок 6.1 − Рисунок 6.2 представлені результати роботи програми Explorer і власного менеджера.

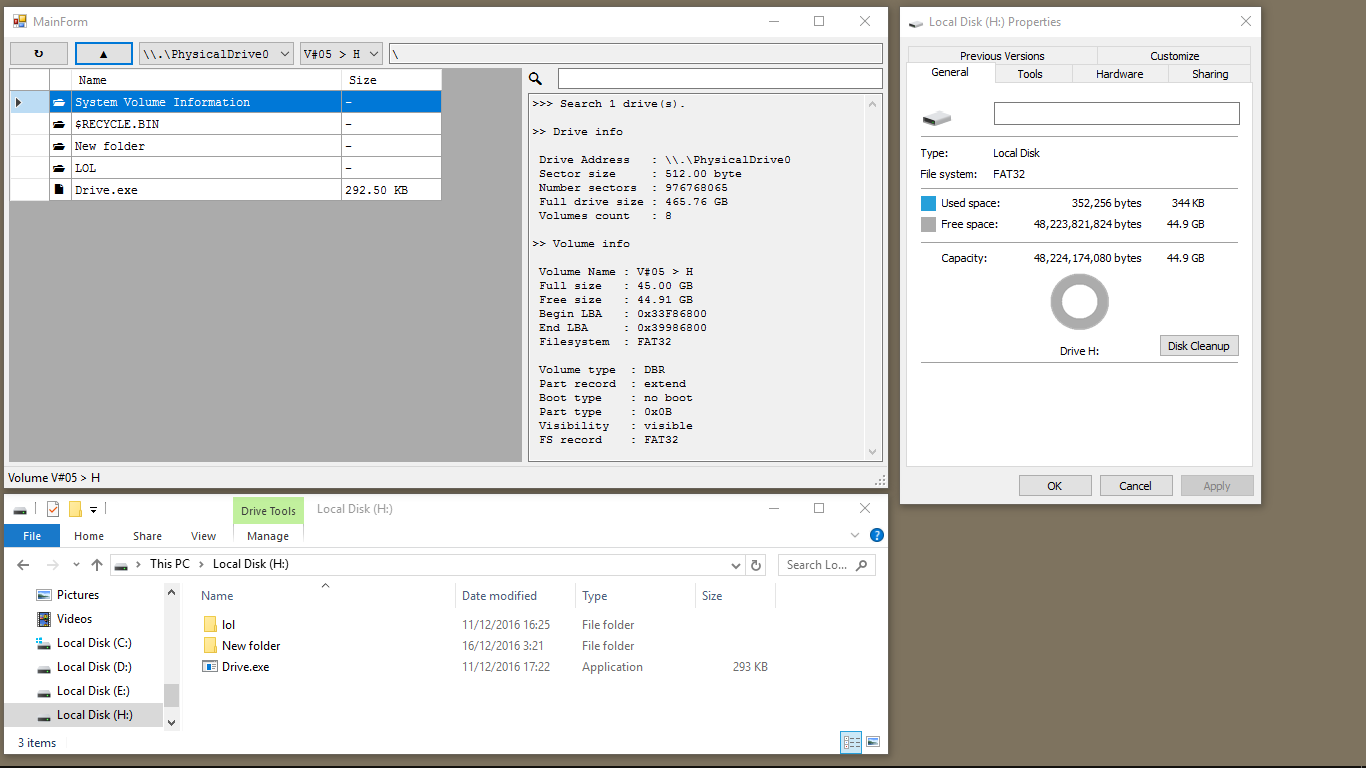


Рисунок . – Перегляд однакової папки розділу FAT32

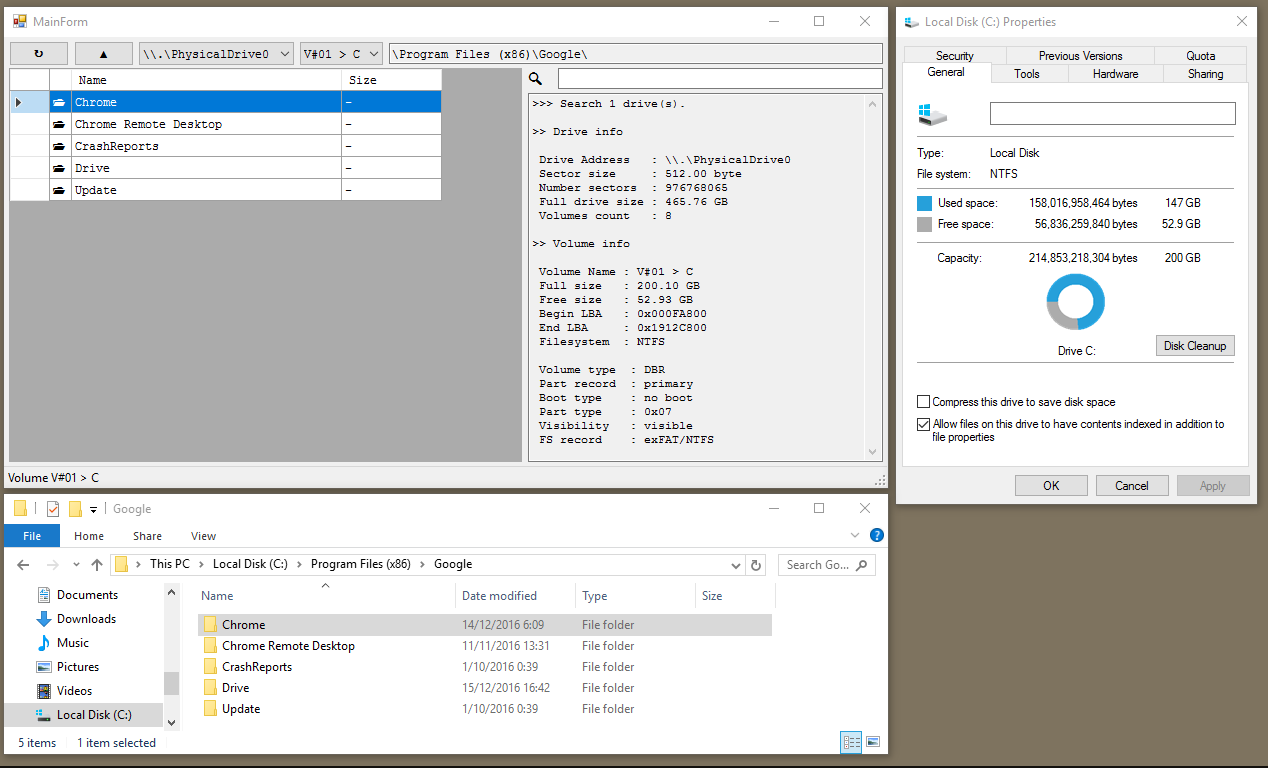


Рисунок . – Перегляд однакової папки розділу NTFS

Пошук файлів, імена яких задовольняють шаблону. Дана частина програмного забезпечення повністю реалізує покладені на неї функції. Щоб перевірити результати роботи цієї функції необхідно зіставити знайдені результати до шаблону пошуку, а також переглянути файли директорії у пошуках додаткових файлів, які задовольняють даному шаблону.

Розроблене програмне забезпечення результат пошуку виводить до інформаційної області форми. Активація функції пошуку відбувається при зміненні тексту поля пошуку.

Приклади результатів пошуку за шаблонами наведені на рис. Рисунок 6.3 - Рисунок 6.5.

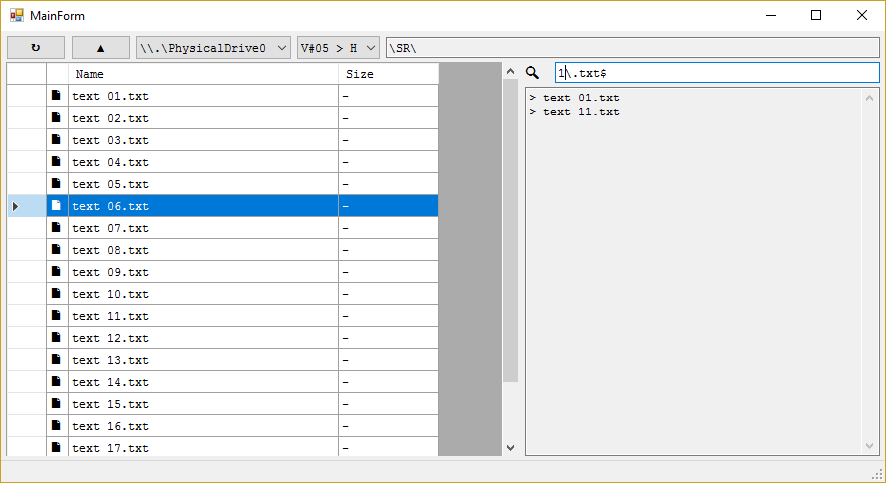


Рисунок . – Результати пошуку за шаблоном (Варіант 1)

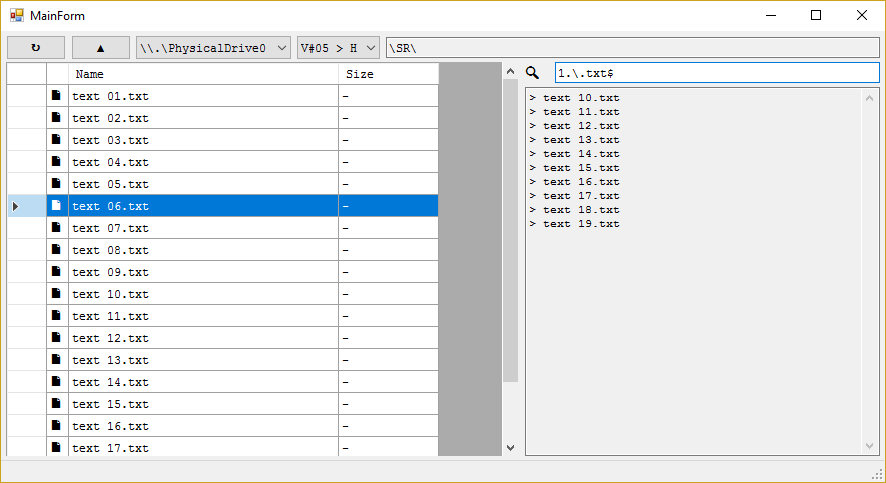


Рисунок . – Результати пошуку за шаблоном (Варіант 2)

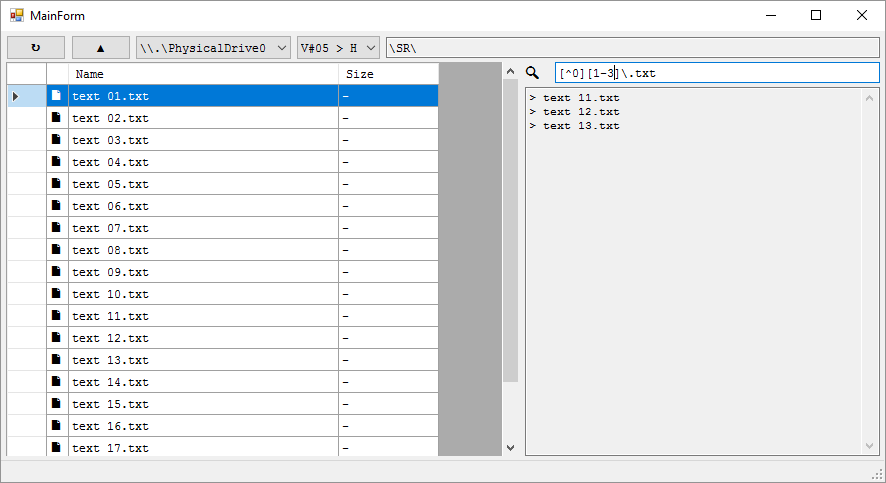


Рисунок . – Результати пошуку за шаблоном (Варіант 3)

# Висновки

У курсовому проекті була реалізована можливість пошуку та відображення об’єктів ФС, які підтримуються операційною системою MS Windows, а також відображення списку логічних дисків з інформацією про фізичне місцезнаходження диску.

Програму було розроблено за допомогою мови програмування C++/CLI. C++/CLI − це мова для середовища програмування Microsoft .NET. Вона інтегрує С++ ISO стандарту з Об'єднаною Системою Типів (Unified Type System, UTS), що розглядається як частина загальномовної інфраструктури (Common Language Infrastructure, CLI). Реалізація проекту за допомогою даної мови програмування дозволила використовувати графічний інтерфейс Windows Forms, який є частиною .NET Framework.

В рамках роботи було розроблено функції для роботи з ФС FAT32. Робота даних функцій здійснюється за допомогою аналізу секторів та кластерів, характерних для даної ФС. Реалізація роботи з іншими ФС (відмінними від FAT32) здійснюється за допомогою функції WinAPI.

Досвід, отриманий при написанні, може знадобитися для програм, які здійснюють роботу з ФС (файлові менеджери, програми для дефрагментації дисків, програми відновлення інформації та тощо).

Програма, реалізована в рамках даної роботи, відповідає обумовленому ТЗ, а також реалізує розширений функціонал, не обумовлений в ТЗ.

# Список використаної літератури

1. Unified Extensible Firmware Interface Specification. Version 2.6 January, 2016 [Електронний ресурс] − Режим доступу до ресурсу: [http://www.uefi.org/sites/ default/files/resources/UEFI%20Spec%202\_6.pdf](http://www.uefi.org/sites/%20default/files/resources/UEFI%20Spec%202_6.pdf) (Дата звернення: 10/12/2016)
2. BIOS Boot Specification. Version 1.01 January 11, 1996 [Електронний ресурс] − − Режим доступу до ресурсу: [https://acpica.org/sites/acpica/files/ specsbbs101.pdf](https://acpica.org/sites/acpica/files/specsbbs101.pdf) (Дата звернення: 10/12/2016)
3. Microsoft Extensible Firmware Initiative FAT32 File System Specification. FAT: General Overview of On-Disk Format. Version 1.03, December 6, 2000. Microsoft Corporation [Електронний ресурс] − Режим доступу до ресурсу: [http://download .microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen1 03.doc](http://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc) (Дата звернення: 10/12/2016)
4. Брайан Кэрриэ. Криминалистический анализ файловых систем / Брайан Кэрриэ, СПб.: Питер, 2007. — 480 с.: ил. — (Для профессионалов). — ISBN: 5-469-01311-1
5. Reverse Engineering the Microsoft exFAT File System: SANS Institute InfoSec Reading Room, December, 2016 [Електронний ресурс] − Режим доступу до ресурсу: [https://www.sans.org/reading-room/whitepapers/forensics/reverse-engine ering-microsoft-exfat-file-system-33274](https://www.sans.org/reading-room/whitepapers/forensics/reverse-engineering-microsoft-exfat-file-system-33274) (Дата звернення: 10/12/2016)
6. Arpaci-Dusseau, Remzi H.; Arpaci-Dusseau, Andrea C. (2014). [Operating Systems: Three Easy Pieces](http://www.ostep.org/). Arpaci-Dusseau Books.

# Додаток А Вихідний код

## Зміст

[А.1 Вихідний код модулю MainForm.cpp 1](#_Toc469741041)

[А.2 Вихідний код модулю MainForm.h 2](#_Toc469741042)

[А.3 Вихідний код модулю drive.h 8](#_Toc469741043)

[А.4 Вихідний код модулю drive.cpp 11](#_Toc469741044)

[А.5 Вихідний код модулю partition.h 20](#_Toc469741045)

[А.6 Вихідний код модулю partition.cpp](#_Toc469741046) 22

[А.7 Вихідний код модулю filesystem.h 24](#_Toc469741047)

[А.8 Вихідний код модулю filesystem.cpp 28](#_Toc469741048)

## А.1 Вихідний код модулю MainForm.cpp

#include "MainForm.h"

#include <Windows.h>

using namespace Drive;

// The entry point to the program

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {

OSVERSIONINFO ver;

ver.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);

GetVersionEx(&ver);

if (ver.dwPlatformId != VER\_PLATFORM\_WIN32\_NT) {

MessageBox::Show("Starting is not possible. Requires Windows NT family!", L"Error");

return 1;

}

try {

Application::EnableVisualStyles();

Application::SetCompatibleTextRenderingDefault(false);

Application::Run(gcnew MainForm);

}

catch (Exception ^ex) {

MessageBox::Show(ex->Message, L"Unknown error!");

}

return 0;

}

## А.2 Вихідний код модулю MainForm.h

#pragma once

#include"drive.h"

namespace Drive {

using namespace System;

using namespace System::ComponentModel;

using namespace System::Collections;

using namespace System::Windows::Forms;

using namespace System::Data;

using namespace System::Drawing;

using namespace System::Text::RegularExpressions;

/\* Тут повинен бути автоматично генерований код \*/

// Drives list

Generic::List<DriveAccess ^> ^drives = nullptr;

// Current drive

DriveAccess ^cDrive = nullptr;

// Current volume

volume::part ^cVolume = nullptr;

// FS elements

Generic::List<filesystem::object ^> ^cFSObjects = nullptr;

// ---------------------------------------

// Set drives list

void driveCBox\_set(Generic::List<DriveAccess ^> ^Drives) {

this->driveCBox->Items->Clear();

this->driveCBox->Items->Add(L"");

this->volumeCBox\_set(nullptr);

if (Drives != nullptr) {

this->driveCBox->Items->AddRange(Drives->ToArray());

this->driveCBox->SelectedIndex = 0;

}

}

// Get select drive

DriveAccess ^driveCBox\_selected() {

System::Object ^object = this->driveCBox->SelectedItem;

if (object != nullptr && DriveAccess::typeid->IsAssignableFrom(object->GetType())) {

return (DriveAccess ^)object;

}

return nullptr;

}

// Set select drive

bool driveCBox\_select(String ^Address) {

volumeCBox\_select(nullptr);

if (Address == nullptr) {

this->driveCBox->SelectedIndex = 0;

return true;

}

for (size\_t i = 1; i < this->driveCBox->Items->Count; i++) {

if (DriveAccess::typeid->IsAssignableFrom(this->driveCBox->Items[i]->GetType())

&& Address == ((DriveAccess ^)this->driveCBox->Items[i])->Address) {

this->driveCBox->SelectedIndex = i;

return true;

}

}

return false;

}

// ---------------------------------------

// Set drive volumes list

void volumeCBox\_set(Generic::List<volume::part ^> ^Volumes) {

this->volumeCBox->Items->Clear();

this->volumeCBox->Items->Add(L"");

this->pathTBox\_set(nullptr);

if (Volumes != nullptr) {

this->volumeCBox->Items->AddRange(Volumes->ToArray());

}

this->volumeCBox->SelectedIndex = 0;

}

// Get select volume

volume::part ^volumeCBox\_selected() {

System::Object ^object = this->volumeCBox->SelectedItem;

if (volume::part::typeid->IsAssignableFrom(object->GetType())) {

return (volume::part ^)object;

}

return nullptr;

}

// Set select volume

bool volumeCBox\_select(String ^Name) {

pathTBox\_set(nullptr);

if (Name == nullptr) {

this->volumeCBox->SelectedIndex = 0;

return true;

}

for (size\_t i = 1; i < this->volumeCBox->Items->Count; i++) {

if (volume::part::typeid->IsAssignableFrom(this->volumeCBox->Items[i]->GetType())

&& Name == ((volume::part ^)this->volumeCBox->Items[i])->Name) {

this->volumeCBox->SelectedIndex = i;

return true;

}

}

return false;

}

// ---------------------------------------

// Set path string

void pathTBox\_set(String ^Path) {

this->pathTBox->Text = Path ? Path : L"";

}

// Get path string

String ^pathTBox\_get() {

return this->pathTBox->Text;

}

// Path level up

bool pathTBox\_up() {

String ^path = this->pathTBox->Text;

if (path != L"" && path != L"\\") {

size\_t newAddress = path->Remove(path->Length - 1)->LastIndexOf(L'\\');

this->pathTBox->Text = !newAddress ? L"\\" : path->Remove(newAddress + 1);

return true;

}

else if (this->cVolume) {

this->volumeCBox\_select(nullptr);

return true;

}

return false;

}

// ---------------------------------------

// Set status label text

void statusLabelSet(String ^Label) {

this->statusLabel->Text = Label ? Label : L"";

}

// ---------------------------------------

// Show info to info text box

void infoTBox\_showInfo(DriveAccess ^ drive, ::volume::part ^ volume) {

this->infoTBox->Text = L">>> Search " + this->drives->Count + L" drive(s).";

if (drive) {

this->infoTBox->Text += L"\r\n\r\n>> Drive info";

this->infoTBox->Text += L"\r\n";

this->infoTBox->Text += L"\r\n Drive Address : " + drive->Address;

this->infoTBox->Text += L"\r\n Sector size : " + getFormatSize(drive->BytesPerSector);

this->infoTBox->Text += L"\r\n Number sectors : " + drive->NumberOfSectors;

this->infoTBox->Text += L"\r\n Full drive size : " + getFormatSize(drive->NumberOfBytes);

this->infoTBox->Text += L"\r\n Volumes count : " + drive->Volumes->Count;

Generic::List<String ^> ^errors = drive->Errors;

if (errors->Count) {

this->infoTBox->Text += L"\r\n\r\n>> Warnings";

for each (String ^error in errors) {

this->infoTBox->Text += L"\r\n " + error;

}

}

}

if (volume) {

this->infoTBox->Text += L"\r\n\r\n>> Volume info";

this->infoTBox->Text += L"\r\n";

this->infoTBox->Text += L"\r\n Volume Name : " + volume->Name;

this->infoTBox->Text += L"\r\n Full size : " + getFormatSize(volume->NumberOfBytes);

QWORD free = volume->NumberOfFreeBytes;

this->infoTBox->Text += L"\r\n Free size : " + (free ? getFormatSize(free) : L"-");

this->infoTBox->Text += L"\r\n Begin LBA : " + String::Format(L"0x{0:X8}", volume->BegLBA);

this->infoTBox->Text += L"\r\n End LBA : " + String::Format(L"0x{0:X8}", volume->EndLBA);

this->infoTBox->Text += L"\r\n Filesystem : " + volume->FSNameAuto;

this->infoTBox->Text += L"\r\n";

Type ^type = volume->GetType();

if (type == ::volume::part\_dbr::typeid) {

::volume::part\_dbr ^dbr\_volume = (::volume::part\_dbr ^)volume;

this->infoTBox->Text += L"\r\n Volume type : DBR";

this->infoTBox->Text += L"\r\n Part record : " + (dbr\_volume->IsPrime ? L"primary" : L"extend");

this->infoTBox->Text += L"\r\n Boot type : " + (dbr\_volume->IsBoot ? L"bootable" : L"no boot");

this->infoTBox->Text += L"\r\n Part type : " + String::Format(L"0x{0:X2}", dbr\_volume->OSType);

this->infoTBox->Text += L"\r\n Visibility : " + (dbr\_volume->IsHide ? L"hidden" : L"visible");

this->infoTBox->Text += L"\r\n FS record : " + dbr\_volume->FSNameRec;

}

else if (type == ::volume::part\_gpt::typeid) {

::volume::part\_gpt ^gpt\_volume = (::volume::part\_gpt ^)volume;

this->infoTBox->Text += L"\r\n Volume type : GPT";

this->infoTBox->Text += L"\r\n Drive : " + gpt\_volume->DriveGUID.ToString();

this->infoTBox->Text += L"\r\n GUID : " + gpt\_volume->PartGUID.ToString();

this->infoTBox->Text += L"\r\n Type : " + gpt\_volume->TypeGUID.ToString() +

L"\r\n " + gpt\_volume->Descript;

}

else {

this->infoTBox->Text += L"\r\n Volume type : Unknown[" + type + L"]";

}

}

}

// ---------------------------------------

System::EventHandler ^ itemGView\_handler\_select = nullptr;

System::Windows::Forms::DataGridViewCellEventHandler ^ itemGView\_handler\_choose = nullptr;

String ^searchColume = nullptr;

// ---------------------------------------

// Display drives and set actions

void itemGView\_displayDrives(Generic::List<DriveAccess ^> ^ drives) {

this->itemGView->SelectionChanged -= this->itemGView\_handler\_select;

this->itemGView->CellContentDoubleClick -= this->itemGView\_handler\_choose;

this->volumeCBox\_select(nullptr);

this->cDrive = nullptr;

this->itemGView->Rows->Clear();

this->itemGView->Columns->Clear();

System::Windows::Forms::DataGridViewTextBoxColumn ^ address, ^ size, ^ sectors, ^ bytesInSector;

address = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());

address->HeaderText = L"Address";

address->Name = L"Address";

address->ReadOnly = true;

address->Width = 160;

size = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());

size->HeaderText = L"Size";

size->Name = L"Size";

size->ReadOnly = true;

size->SortMode = DataGridViewColumnSortMode::NotSortable;

size->Width = 100;

sectors = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());

sectors->HeaderText = L"Sectors";

sectors->Name = L"Sectors";

sectors->ReadOnly = true;

sectors->Width = 100;

bytesInSector = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());

bytesInSector->HeaderText = L"Sector size";

bytesInSector->Name = L"SectorSize";

bytesInSector->ReadOnly = true;

bytesInSector->Width = 110;

this->itemGView->Columns->AddRange(gcnew cli::array< System::Windows::Forms::DataGridViewColumn^ >(4) { address, size, sectors, bytesInSector });

this->searchColume = L"Address";

this->itemGView\_handler\_select = gcnew System::EventHandler(this, &MainForm::itemGView\_selectDrive);

this->itemGView->SelectionChanged += this->itemGView\_handler\_select;

this->itemGView\_handler\_choose = gcnew System::Windows::Forms::DataGridViewCellEventHandler(this, &MainForm::itemGView\_chooseDrive);

this->itemGView->CellContentDoubleClick += this->itemGView\_handler\_choose;

for each (auto drive in drives) {

array<String ^> ^ driveItem = { drive->Address, getFormatSize(drive->NumberOfBytes), Convert::ToString(drive->NumberOfSectors), getFormatSize(drive->BytesPerSector) };

this->itemGView->Rows->Add(driveItem);

}

this->infoTBox\_showInfo(drives->Count ? drives[0] : nullptr, nullptr);

}

// Action : Double item click - Select drive

System::Void itemGView\_selectDrive(System::Object ^ sender, System::EventArgs ^ e) {

if (this->itemGView->Rows->Count) {

String ^address = (String ^)this->itemGView->SelectedRows[0]->Cells[0]->Value;

; if (address != nullptr && address != L"") {

auto drives = this->drives;

for each (auto drive in drives) {

if (drive->Address == address) {

this->infoTBox\_showInfo(drive, nullptr);

break;

}

}

}

}

}

// Action : Double item click - Choose drive

System::Void itemGView\_chooseDrive(System::Object ^ sender, System::Windows::Forms::DataGridViewCellEventArgs ^ e) {

if (this->itemGView->Rows->Count) {

String ^address = (String ^)this->itemGView->SelectedRows[0]->Cells[L"Address"]->Value;

driveCBox\_select(address);

}

}

// ---------------------------------------

// Display volumes and set actions

void itemGView\_displayVolumes(DriveAccess ^ drive) {

this->itemGView->SelectionChanged -= this->itemGView\_handler\_select;

this->itemGView->CellContentDoubleClick -= this->itemGView\_handler\_choose;

this->cVolume = nullptr;

this->volumeCBox\_set(drive->Volumes);

this->itemGView->Rows->Clear();

this->itemGView->Columns->Clear();

System::Windows::Forms::DataGridViewTextBoxColumn ^ name, ^ size, ^ beg, ^ end, ^ sectors;

name = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());

name->HeaderText = L"Name";

name->Name = L"Name";

name->ReadOnly = true;

name->Width = 80;

size = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());

size->HeaderText = L"Size";

size->Name = L"Size";

size->ReadOnly = true;

size->SortMode = DataGridViewColumnSortMode::NotSortable;

size->Width = 85;

beg = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());

beg->HeaderText = L"Begin LBA";

beg->Name = L"BegLBA";

beg->ReadOnly = true;

beg->Width = 100;

end = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());

end->HeaderText = L"End LBA";

end->Name = L"EndLBA";

end->ReadOnly = true;

end->Width = 100;

sectors = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());

sectors->HeaderText = L"Sectors";

sectors->Name = L"Sectors";

sectors->ReadOnly = true;

sectors->Width = 100;

this->itemGView->Columns->AddRange(gcnew cli::array< System::Windows::Forms::DataGridViewColumn^ >(5) { name, size, beg, end, sectors });

this->searchColume = L"Name";

this->itemGView\_handler\_select = gcnew System::EventHandler(this, &MainForm::itemGView\_selectVolume);

this->itemGView->SelectionChanged += this->itemGView\_handler\_select;

this->itemGView\_handler\_choose = gcnew System::Windows::Forms::DataGridViewCellEventHandler(this, &MainForm::itemGView\_chooseVolume);

this->itemGView->CellContentDoubleClick += this->itemGView\_handler\_choose;

auto volumes = drive->Volumes;

for each (auto volume in volumes) {

array<String ^> ^ driveItem = { volume->Name, getFormatSize(volume->NumberOfBytes), String::Format(L"0x{0:X8}", volume->BegLBA), String::Format(L"0x{0:X8}", volume->EndLBA), Convert::ToString(volume->NumberOfSectors) };

this->itemGView->Rows->Add(driveItem);

}

this->infoTBox\_showInfo(this->cDrive, volumes->Count ? volumes[0] : nullptr);

}

// Action : Double item click - Select volume

System::Void itemGView\_selectVolume(System::Object ^ sender, System::EventArgs ^ e) {

if (this->cDrive && this->itemGView->SelectedRows->Count) {

String ^name = (String ^)this->itemGView->SelectedRows[0]->Cells[L"Name"]->Value;

if (name != nullptr && name != L"") {

auto volumes = this->cDrive->Volumes;

for each (auto volume in volumes) {

if (volume->Name == name) {

this->infoTBox\_showInfo(this->cDrive, volume);

break;

}

}

}

}

}

// Action : Double item click - Choose volume

System::Void itemGView\_chooseVolume(System::Object ^ sender, System::Windows::Forms::DataGridViewCellEventArgs ^ e) {

if (this->cDrive && this->itemGView->Rows->Count) {

String ^name = (String ^)this->itemGView->SelectedRows[0]->Cells[L"Name"]->Value;

volumeCBox\_select(name);

}

}

// ---------------------------------------

// Display path records and set actions

void itemGView\_displayPath(volume::part ^ volume, String ^ path) {

try {

this->cFSObjects = volume->getDirItems(path);

this->itemGView->SelectionChanged -= this->itemGView\_handler\_select;

this->itemGView->CellContentDoubleClick -= this->itemGView\_handler\_choose;

this->itemGView->Rows->Clear();

this->itemGView->Columns->Clear();

System::Windows::Forms::DataGridViewTextBoxColumn ^ icon, ^ name, ^ size;

icon = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());

icon->HeaderText = L"";

icon->Name = L"Icon";

icon->ReadOnly = true;

icon->Width = 22;

icon->Resizable = DataGridViewTriState::False;

name = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());

name->HeaderText = L"Name";

name->Name = L"Name";

name->ReadOnly = true;

name->Width = 270;

size = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());

size->HeaderText = L"Size";

size->Name = L"Size";

size->ReadOnly = true;

size->SortMode = DataGridViewColumnSortMode::NotSortable;

size->Width = 100;

this->itemGView->Columns->AddRange(gcnew cli::array< System::Windows::Forms::DataGridViewColumn^ >(3) { icon, name, size });

this->searchColume = L"Name";

this->itemGView\_handler\_choose = gcnew System::Windows::Forms::DataGridViewCellEventHandler(this, &MainForm::itemGView\_choosePath);

this->itemGView->CellContentDoubleClick += this->itemGView\_handler\_choose;

// 📄 📂

for each (filesystem::object ^ object in cFSObjects) {

array<String ^> ^ driveItem = { object->attr & FILE\_ATTRIBUTE\_DIRECTORY ? L"📂" : L"📄", object->name, object->size ? getFormatSize(object->size) : L"-" };

this->itemGView->Rows->Add(driveItem);

}

this->infoTBox\_showInfo(this->cVolume->Drive, this->cVolume);

statusLabelSet(L"Volume " + this->cVolume->Name);

}

catch (NotSupportedException ^ex) {

this->volumeCBox\_select(nullptr);

statusLabelSet(gcnew String(ex->Message));

}

catch (Exception ^ex) {

MessageBox::Show(gcnew String(ex->Message));

this->pathTBox\_up();

}

}

// System::Void itemGView\_selectPath(System::Object ^ sender, System::EventArgs ^ e) {}

// Action : Double item click - Choose path

System::Void itemGView\_choosePath(System::Object ^ sender, System::Windows::Forms::DataGridViewCellEventArgs ^ e) {

statusLabelSet(nullptr);

if (this->cDrive && this->itemGView->Rows->Count) {

auto select = this->itemGView->SelectedRows[0];

if (select->Cells[L"Icon"]->Value->Equals(L"📂")) {

String ^name = (String ^)select->Cells[L"Name"]->Value;

this->pathTBox->Text += name + L"\\";

}

}

}

// ---------------------------------------

// Action : Load form - Get drives

System::Void loadDrive(System::Object ^ sender, System::EventArgs ^ e) {

statusLabelSet(nullptr);

this->Enabled = false;

this->drives = DriveAccess::OpenAll();

this->driveCBox\_set(this->drives);

this->searchTBox->Text = "";

this->Enabled = true;

}

// ---------------------------------------

// Action : Click UP button - Up level

System::Void levelUP(System::Object ^ sender, System::EventArgs ^ e) {

if (!this->pathTBox\_up()) {

if (this->cDrive) {

this->driveCBox\_select(nullptr);

}

}

}

// ---------------------------------------

// Action : Change searche value - Searche in items

System::Void itemsSearch(System::Object ^ sender, System::EventArgs ^ e) {

statusLabelSet(nullptr);

this->searchErrorLabel->Visible = false;

try {

infoTBox->Text = L"";

if (this->searchTBox->Text != L"") {

Regex ^ search = gcnew Regex(this->searchTBox->Text);

if (this->searchColume != nullptr) {

for (size\_t i = 0; i < this->itemGView->Rows->Count; i++) {

Object ^ object = this->itemGView->Rows[i]->Cells[this->searchColume]->Value;

String ^ value = (String::typeid->IsAssignableFrom(object->GetType())) ? (String ^)object : object->ToString();

//this->itemGView->Rows[i]->Visible = search->IsMatch(value);

if (search->IsMatch(value)) {

infoTBox->Text += L"> " + value + L"\r\n";

}

}

}

}

else this->infoTBox\_showInfo(this->cDrive, this->cVolume);

}

catch (const ArgumentException ^ e) {

this->searchErrorLabel->Visible = true;

statusLabelSet(L"Invalid expression format!");

}

}

// ---------------------------------------

System::EventHandler ^volumeSelect = gcnew System::EventHandler(this, &MainForm::volumeCBox\_SelectAction);

// Action : Select drive - Display volumes

System::Void driveCBox\_SelectAction(System::Object ^ sender, System::EventArgs ^ e) {

statusLabelSet(nullptr);

this->volumeCBox->SelectedIndexChanged -= this->volumeSelect;

this->cDrive = this->driveCBox\_selected();

if (this->cDrive != nullptr) {

statusLabelSet(L"Drive " + this->cDrive->Address);

this->itemGView\_displayVolumes(this->cDrive);

this->volumeCBox->SelectedIndexChanged += this->volumeSelect;

}

else {

this->itemGView\_displayDrives(this->drives);

}

}

// ---------------------------------------

System::EventHandler ^pathChange = gcnew System::EventHandler(this, &MainForm::pathTBox\_ChangePath);

// Action : Select volume - Display root items

System::Void volumeCBox\_SelectAction(System::Object ^ sender, System::EventArgs ^ e) {

statusLabelSet(nullptr);

this->pathTBox->TextChanged -= pathChange;

this->cVolume = this->volumeCBox\_selected();

if (this->cVolume != nullptr) {

statusLabelSet(L"Drive " + this->cDrive->Address);

this->pathTBox->TextChanged += pathChange;

pathTBox\_set(L"\\");

}

else this->driveCBox\_SelectAction(sender, e);

}

// ---------------------------------------

// Action : Change Label Value - Display path items

System::Void pathTBox\_ChangePath(System::Object ^ sender, System::EventArgs ^ e) {

this->cFSObjects = nullptr;

String ^path = this->pathTBox->Text;

if (path != nullptr) {

this->itemGView\_displayPath(this->cVolume, path);

}

}

};

}

## А.3 Вихідний код модулю drive.h

#pragma once

// ---------------------------------------

#include"partition.h"

#include"filesystem.h"

// ---------------------------------------

using namespace System;

using namespace System::Collections;

using namespace System::Data;

// ---------------------------------------

namespace volume {

ref class part;

ref class part\_dbr;

ref class part\_gpt;

}

// ---------------------------------------

typedef ref class DriveAccess {

// ---------------------------------------

protected:

// ---------------------------------------

bool ReadHandle (HANDLE &Handle);

bool ReadSectors (HANDLE &Handle, QWORD LBAPos, LPVOID Buffer, DWORD ToRead, LPDWORD Read);

void ReadVolumes (HANDLE &Handle, QWORD Sectors);

bool ReadDBRVolumes (HANDLE &Handle, QWORD Sectors, ::dbr::note &mbr);

bool ReadGPTVolumes (HANDLE &Handle, QWORD Sectors, ::dbr::note &mbr);

DriveAccess (size\_t DriveID);

// ---------------------------------------

size\_t driveID;

String ^ address;

DWORD bytesPerSector;

QWORD numberOfSectors;

Generic::List<String ^> ^ errors = gcnew Generic::List<String ^>;

Generic::List<volume::part ^> ^ volumes = gcnew Generic::List<volume::part ^>;

// ---------------------------------------

public:

// ---------------------------------------

static DriveAccess ^Open(size\_t DriveID);

// ---------------------------------------

static Generic::List<DriveAccess^> ^OpenAll();

// ---------------------------------------

virtual String ^ToString() override;

// ---------------------------------------

bool ReadSectors(QWORD LBAPos, LPVOID Buffer, DWORD ToRead, LPDWORD Read);

// ---------------------------------------

static property size\_t MaxNumber {

size\_t get() {

return 32;

}

}

property size\_t DriveID {

size\_t get() {

return this->driveID;

}

}

property String ^Address {

String ^ get() {

return gcnew String(this->address);

}

}

property DWORD BytesPerSector {

DWORD get() {

return this->bytesPerSector;

}

}

property QWORD NumberOfSectors {

QWORD get() {

return this->numberOfSectors;

}

}

property QWORD NumberOfBytes {

QWORD get() {

return this->numberOfSectors \* this->bytesPerSector;

}

}

property Generic::List<String ^> ^Errors {

Generic::List<String ^> ^ get() {

return this->errors;

}

}

property Generic::List<volume::part ^> ^Volumes {

Generic::List<volume::part ^> ^ get() {

return this->volumes;

}

}

// ---------------------------------------

} DriveAccess;

// ---------------------------------------

typedef ref class volume::part {

// ---------------------------------------

protected:

// ---------------------------------------

DriveAccess ^drive;

String ^name;

WCHAR letter;

QWORD beg\_lba;

QWORD end\_lba;

String ^fs\_auto;

// ---------------------------------------

QWORD getNumOfFreeBytes();

// ---------------------------------------

public:

// ---------------------------------------

part(DriveAccess ^Drive, size\_t PartID, QWORD BegLBA, QWORD EndLBA);

// ---------------------------------------

virtual String ^ToString() override;

// ---------------------------------------

bool ReadSectors(QWORD LBAPos, LPVOID Buffer, DWORD ToRead, LPDWORD Read);

// ---------------------------------------

Generic::List<filesystem::object^> ^getDirItems(String ^Path);

// ---------------------------------------

property DriveAccess ^Drive {

DriveAccess ^get() {

return this->drive;

}

}

property String ^Name {

String ^get() {

return this->name + L" > " + this->letter;

}

}

property WCHAR Letter {

WCHAR get() {

return this->letter;

}

}

property QWORD BegLBA {

QWORD get() {

return this->beg\_lba;

}

};

property QWORD EndLBA {

QWORD get() {

return this->end\_lba;

}

};

property String ^FSNameAuto {

String ^get() {

return gcnew String(this->fs\_auto);

}

}

property QWORD NumberOfSectors {

QWORD get() {

return this->end\_lba - this->beg\_lba;

}

}

property DWORD BytesPerSector {

DWORD get() {

return this->drive->BytesPerSector;

}

}

property QWORD NumberOfBytes {

QWORD get() {

return this->NumberOfSectors \* this->BytesPerSector;

}

}

property QWORD NumberOfFreeBytes {

QWORD get() {

return this->getNumOfFreeBytes();

}

}

// ---------------------------------------

} part;

// ---------------------------------------

typedef ref class volume::part\_dbr : public part {

// ---------------------------------------

protected:

// ---------------------------------------

bool is\_primary;

bool is\_boot;

BYTE os\_type;

bool is\_hide;

String ^fs\_rec;

// ---------------------------------------

public:

// ---------------------------------------

part\_dbr(DriveAccess ^Drive, size\_t PartID, const ::dbr::part &Part, QWORD Offset);

// ---------------------------------------

property bool IsPrime {

bool get() {

return this->is\_primary;

}

}

property bool IsBoot {

bool get() {

return this->is\_boot;

}

}

property BYTE OSType {

BYTE get() {

return this->os\_type;

}

}

property bool IsHide {

bool get() {

return this->is\_hide;

}

}

property String ^ FSNameRec {

String ^get() {

return gcnew String(this->fs\_rec);

}

}

// ---------------------------------------

} part\_dbr;

// --------------------------------------- ---

typedef ref class volume::part\_gpt : public part {

// ---------------------------------------

protected:

// ---------------------------------------

\_guid \*drive\_guid;

\_guid \*type\_guid;

\_guid \*part\_guid;

String ^desc;

// ---------------------------------------

public:

// ---------------------------------------

part\_gpt(DriveAccess ^Drive, size\_t PartID, const \_guid &DriveGUID, const ::gpt::part &Part);

~part\_gpt();

// ---------------------------------------

property \_guid DriveGUID {

\_guid get() {

return \*this->drive\_guid;

}

}

property \_guid TypeGUID {

\_guid get() {

return \*this->type\_guid;

}

};

property \_guid PartGUID {

\_guid get() {

return \*this->part\_guid;

}

}

property String ^Descript {

String ^ get() {

return gcnew String(this->desc);

}

}

// ---------------------------------------

} part\_gpt;

// ---------------------------------------

String ^getFormatSize(QWORD Size);

// ---------------------------------------

## А.4 Вихідний код модулю drive.cpp

#include"Drive.h"

// ---------------------------------------

#include<iomanip>

using std::fixed;

using std::setw;

using std::hex;

#include<stack>

using std::stack;

#include<vcclr.h>

// ---------------------------------------

using namespace System;

using namespace System::Collections;

using namespace System::Data;

// ---------------------------------------

WCHAR getSysLetter(DWORD Serial);

WCHAR getSysLetter(\_guid GUID);

QWORD getPartNumOfFreeByte(WCHAR Letter);

QWORD getFAT32NumOfFreeByte(volume::part ^Volume);

Generic::List<filesystem::object^> ^getDirItems(String ^ Path);

Generic::List<filesystem::object^> ^getFAT32DirIrems(volume::part ^Volume, String ^Path);

// ---------------------------------------

DriveAccess ^DriveAccess::Open(size\_t DriveID) {

try {

return gcnew DriveAccess(DriveID);

} catch (const System::Exception ^) {

return nullptr;

}

}

Generic::List<DriveAccess^>^ DriveAccess::OpenAll() {

Generic::List<DriveAccess^>^ drives = gcnew Generic::List<DriveAccess^>;

for (size\_t i = 0; i < DriveAccess::MaxNumber; i++) {

DriveAccess ^drive = DriveAccess::Open(i);

if (drive) {

drives->Add(drive);

}

}

return drives;

}

// ---------------------------------------

String ^ DriveAccess::ToString() {

return this->Address;

}

// ---------------------------------------

bool DriveAccess::ReadSectors(QWORD LBAPos, LPVOID Buffer, DWORD ToRead, LPDWORD Read) {

HANDLE handle;

bool result = this->ReadHandle(handle)

&& this->ReadSectors(handle, LBAPos, Buffer, ToRead, Read);

CloseHandle(handle);

return result;

}

// ---------------------------------------

DriveAccess::DriveAccess(size\_t DriveID) :

driveID(DriveID), address(L"\\\\.\\PhysicalDrive" + DriveID) {

HANDLE handle;

// Reading Drive geometry

DISK\_GEOMETRY geometry;

if (!this->ReadHandle(handle) ||

!DeviceIoControl(handle, IOCTL\_DISK\_GET\_DRIVE\_GEOMETRY, NULL, 0, &geometry, sizeof(DISK\_GEOMETRY), NULL, NULL)) {

CloseHandle(handle);

throw gcnew Exception();

}

this->bytesPerSector = geometry.BytesPerSector;

this->numberOfSectors = geometry.Cylinders.QuadPart \* geometry.TracksPerCylinder \* geometry.SectorsPerTrack;

// Reading Drive volumes

this->ReadVolumes(handle, 0x00);

CloseHandle(handle);

}

1// ---------------------------------------

bool DriveAccess::ReadHandle(HANDLE &Handle) {

pin\_ptr<const wchar\_t> address = PtrToStringChars(this->address);

Handle = CreateFile(address, GENERIC\_READ, FILE\_SHARE\_READ, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

return Handle != INVALID\_HANDLE\_VALUE;

}

// ---------------------------------------

bool DriveAccess::ReadSectors(HANDLE &Handle, QWORD LBAPos, LPVOID Buffer, DWORD ToRead, LPDWORD Read) {

LBAPos \*= this->bytesPerSector;

PLONG overflow = reinterpret\_cast<PLONG>(&LBAPos) + 1;

return SetFilePointer(Handle, (LONG)LBAPos, overflow, FILE\_BEGIN) != INVALID\_SET\_FILE\_POINTER

&& ReadFile(Handle, Buffer, ToRead \* this->bytesPerSector, Read, NULL);

}

// ---------------------------------------

typedef struct dbr\_view {

QWORD shift;

DWORD index;

::dbr::note note;

} dbr\_view;

// ---------------------------------------

bool DriveAccess::ReadDBRVolumes(HANDLE &Handle, QWORD LBAPos, ::dbr::note &mbr) {

if (mbr.part[0].os\_type == 0xEE) {

return false;

}

for (size\_t i = 0; i < 4; i++) {

const ::dbr::info \*master\_i = ::dbr::getPartInfo(mbr.part[i]);

if (master\_i == nullptr || master\_i->is\_part) {

if (mbr.part[i].lba\_sct > 0) {

this->volumes->Add(gcnew volume::part\_dbr(this, this->volumes->Count, mbr.part[i], LBAPos));

}

}

// Partition is a container

else {

stack<dbr\_view \*> analysis\_stack;

QWORD shift = mbr.part[i].beg\_lba;

do {

::dbr::note ebr;

if (!this->ReadSectors(Handle, LBAPos + shift, &ebr, 1, NULL)) {

this->errors->Add(L"EBR element Read error!");

}

else if (ebr.sign != 0xAA55) {

this->errors->Add(L"EBR signature error!");

}

else {

dbr\_view \* view = new dbr\_view;

view->shift = shift;

view->index = 0;

view->note = ebr;

analysis\_stack.push(view);

}

// EBR stack analysis

while (!analysis\_stack.empty()) {

dbr\_view \* current = analysis\_stack.top();

while (current->index < 4) {

const dbr::info \*extend\_i = dbr::getPartInfo(current->note.part[current->index]);

if (extend\_i != nullptr && !extend\_i->is\_part) {

shift = (QWORD)mbr.part[i].beg\_lba + current->note.part[current->index].beg\_lba;

current->index++;

break;

}

if (current->note.part[current->index].lba\_sct) {

this->volumes->Add(gcnew volume::part\_dbr(this, this->volumes->Count, current->note.part[current->index], LBAPos + shift));

}

current->index++;

}

if (current->index < 4) break;

analysis\_stack.pop();

delete current;

}

} while (!analysis\_stack.empty());

}

}

return true;

}

bool DriveAccess::ReadGPTVolumes(HANDLE &Handle, QWORD LBAPos, ::dbr::note &mbr) {

if (mbr.part[0].os\_type != 0xEE) {

return false;

}

::gpt::head gpt\_head;

if (!this->ReadSectors(Handle, LBAPos + mbr.part[0].beg\_lba, &gpt\_head, 1, NULL)) {

this->errors->Add(L"GPT header Read error!");

}

else if (strcmp("EFI PART", gpt\_head.head\_sign)) {

this->errors->Add(L"GPT header signature error!");

}

else if (0x00010000 != gpt\_head.gpt\_vers) {

this->errors->Add(L"GPT version not supported!");

}

else if (sizeof(::gpt::head) != gpt\_head.head\_size) {

this->errors->Add(L"GPT header Size invalid!");

}

else if (sizeof(::gpt::part) != gpt\_head.part\_size) {

this->errors->Add(L"GPT NamePart Size not standard!");

}

else {

::gpt::part \*gpt\_parts = new ::gpt::part[gpt\_head.parts\_num];

DWORD sectors = sizeof(gpt\_parts) / this->bytesPerSector + (sizeof(gpt\_parts) % this->bytesPerSector ? 1 : 0);

if (!this->ReadSectors(Handle, LBAPos + gpt\_head.parts\_lba, gpt\_parts, sectors, NULL)) {

this->errors->Add(L"GPT parts table Read error!");

}

else for (size\_t i = 0; i < gpt\_head.parts\_num; i++) {

if (gpt\_parts[i].part\_guid == \_guid(0x024DEE41, 0x33E7, 0x11D3, 0x9D69, 0x0008C781F39F)) {

this->ReadVolumes(Handle, gpt\_parts[i].beg\_lba);

}

else if (gpt\_parts[i].part\_guid) {

this->volumes->Add(gcnew volume::part\_gpt(this, this->volumes->Count, gpt\_head.guid, gpt\_parts[i]));

}

}

delete[] gpt\_parts;

}

return true;

}

void DriveAccess::ReadVolumes(HANDLE &Handle, QWORD LBAPos) {

::dbr::note mbr;

if (!this->ReadSectors(Handle, LBAPos, &mbr, 1, NULL)) {

this->errors->Add(L"MBR element Read error!");

}

// MBR not found

else if (mbr.sign != 0xAA55) {

this->errors->Add(L"MBR signature not found!");

}

// Protect MBR

if (mbr.part[0].os\_type == 0xEE) {

ReadGPTVolumes(Handle, LBAPos, mbr);

}

// Legacy MBR

else {

ReadDBRVolumes(Handle, LBAPos, mbr);

}

}

// ---------------------------------------

QWORD volume::part::getNumOfFreeBytes() {

return this->fs\_auto == L"FAT32"

? getFAT32NumOfFreeByte(this)

: this->letter != L'?'

? getPartNumOfFreeByte(this->letter)

: 0;

}

// ---------------------------------------

volume::part::part(DriveAccess ^Drive, size\_t PartID, QWORD BegLBA, QWORD EndLBA) :

drive(Drive), beg\_lba(BegLBA), end\_lba(EndLBA) {

this->name = String::Format(L"V#{0:X2}", PartID);

filesystem::boot \*boot = new filesystem::boot;

this->ReadSectors(0, boot, 1, NULL);

this->fs\_auto = filesystem::fs\_auto(boot);

if ((gcnew String(L"NTFS"))->Equals(this->fs\_auto)) {

filesystem::ntfs::boot \*ntfs\_boot = (filesystem::ntfs::boot \*)boot;

this->letter = getSysLetter((DWORD)ntfs\_boot->BPB\_VolSerial);

}

else if ((gcnew String(L"exFAT"))->Equals(this->fs\_auto)) {

filesystem::exfat::boot \*exfat\_boot = (filesystem::exfat::boot \*)boot;

this->letter = getSysLetter(exfat\_boot->BPB\_VolSerial);

}

else if ((gcnew String(L"FAT32"))->Equals(this->fs\_auto)) {

filesystem::fat32::boot \*fat32\_boot = (filesystem::fat32::boot \*)boot;

this->letter = getSysLetter(fat32\_boot->BPB\_VolSerial);

}

else if ((gcnew String(L"FAT16"))->Equals(this->fs\_auto)) {

filesystem::fat16::boot \*fat16\_boot = (filesystem::fat16::boot \*)boot;

this->letter = getSysLetter(fat16\_boot->BPB\_VolSerial);

}

else this->letter = L'?';

delete[] boot;

}

// ---------------------------------------

String ^ volume::part::ToString() {

return this->Name;

}

// ---------------------------------------

bool volume::part::ReadSectors(QWORD LBAPos, LPVOID Buffer, DWORD ToRead, LPDWORD Read) {

LBAPos += this->beg\_lba;

if (this->end\_lba <= LBAPos + ToRead) {

ToRead = (DWORD)(this->end\_lba - LBAPos);

if (this->end\_lba <= LBAPos) {

return false;

}

}

return this->drive->ReadSectors(LBAPos, Buffer, ToRead, Read);

}

// ---------------------------------------

Generic::List<filesystem::object^>^ volume::part::getDirItems(String ^Path) {

if (this->fs\_auto == L"FAT32") {

return getFAT32DirIrems(this, Path);

}

else if (this->letter != L'?') {

return ::getDirItems(this->letter + L":\\" + Path);

}

else {

throw gcnew NotSupportedException(L"Unsupported file system!");

}

}

// ---------------------------------------

volume::part\_dbr::part\_dbr(DriveAccess ^Drive, size\_t PartID, const ::dbr::part &Part, QWORD Offset) :

volume::part(Drive, PartID, Offset + Part.beg\_lba, Offset + Part.beg\_lba + Part.lba\_sct) {

this->is\_primary = !Offset;

this->is\_boot = Part.boot\_fl == 0x80;

this->os\_type = Part.os\_type;

auto info = ::dbr::getPartInfo(Part);

this->is\_hide = info->is\_hide;

this->fs\_rec = gcnew String(info->fs\_name);

}

// ---------------------------------------

volume::part\_gpt::part\_gpt(DriveAccess ^Drive, size\_t PartID, const \_guid &DriveGUID, const ::gpt::part &Part) :

volume::part(Drive, PartID, Part.beg\_lba, Part.end\_lba) {

this->drive\_guid = new \_guid(DriveGUID);

this->part\_guid = new \_guid(Part.part\_guid);

this->type\_guid = new \_guid(Part.type\_guid);

auto info = ::gpt::getPartInfo(Part);

this->desc = gcnew String(info->type\_desc);

this->letter = getSysLetter(Part.part\_guid);

}

// ---------------------------------------

volume::part\_gpt::~part\_gpt() {

delete this->type\_guid;

delete this->part\_guid;

}

// ---------------------------------------

String ^getFormatSize(QWORD Size) {

static LPWSTR sign[] = { L"byte", L"KB", L"MB", L"GB", L"TB", L"PB", L"EB", L"ZB", L"YB" };

static size\_t sign\_s = sizeof(sign) / sizeof(LPWSTR);

double \_\_size = (double)Size;

size\_t i = 0;

while (i < sign\_s && \_\_size / 1024 >= 0.8) {

\_\_size = \_\_size / 1024; i++;

}

return String::Format(L"{0:0.00} ", \_\_size) + gcnew String(sign[i]);

}

// ---------------------------------------

WCHAR getSysLetter(DWORD Serial) {

WCHAR address[] = L"A:\\";

for (DWORD disks = GetLogicalDrives(), vSerial; disks; disks = disks >> 1) {

if (disks & 0x00000001

&& GetVolumeInformation(address, NULL, NULL, &vSerial, NULL, NULL, NULL, NULL)

&& Serial == vSerial) {

return address[0];

}

address[0]++;

}

return L'?';

}

WCHAR getSysLetter(\_guid GUID) {

WCHAR address[] = L"A:\\";

WCHAR guidStr[50];

for (DWORD disks = GetLogicalDrives(); disks; disks = disks >> 1) {

if (disks & 0x00000001

&& GetVolumeNameForVolumeMountPoint(address, guidStr, 50)

&& (gcnew String(guidStr))->Contains(GUID.ToString())) {

return address[0];

}

address[0]++;

}

return L'?';

}

// ---------------------------------------

// Get a partition free bytes count

QWORD getPartNumOfFreeByte(WCHAR Letter) {

WCHAR Name[] = L" :\\\\";

Name[0] = Letter;

DWORD SecPerClus = 0, BytsPerSec = 0, NumOfFreeClus = 0;

return GetDiskFreeSpace(Name, &SecPerClus, &BytsPerSec, &NumOfFreeClus, NULL)

? (QWORD)NumOfFreeClus \* BytsPerSec \* SecPerClus : 0;

}

// ---------------------------------------

// Get a directory items using WinAPI

Generic::List<filesystem::object^> ^getDirItems(String ^Path) {

Generic::List<filesystem::object^>^ objects = gcnew Generic::List<filesystem::object^>;

Path += "\\\*.\*";

pin\_ptr<const wchar\_t> address = PtrToStringChars(Path);

WIN32\_FIND\_DATA data;

HANDLE handle = FindFirstFile(address, &data);

if (handle == INVALID\_HANDLE\_VALUE) {

FindClose(handle);

throw gcnew ArgumentException(gcnew String(L"Access error."));

}

do {

filesystem::object^ object = gcnew filesystem::object;

if (!(data.dwFileAttributes & FILE\_ATTRIBUTE\_REPARSE\_POINT)) {

object->name = gcnew String(data.cFileName);

if (!object->name->Equals(L".") && !object->name->Equals(L"..")) {

object->size = data.nFileSizeHigh \* ((QWORD)MAXDWORD + 1) + data.nFileSizeLow;

object->attr = data.dwFileAttributes;

objects->Add(object);

}

}

} while (FindNextFile(handle, &data));

FindClose(handle);

return objects;

}

// ---------------------------------------

// Get FAT32 data clusters cout

DWORD getFAT32DataClusCout(const filesystem::fat32::boot &Boot) {

DWORD TotSec = Boot.BPB\_TotSecSm ? Boot.BPB\_TotSecSm : Boot.BPB\_TotSecLr;

DWORD FATSize = Boot.BPB\_FATSizeSm ? Boot.BPB\_FATSizeSm : Boot.BPB\_FATSizeLr;

return (TotSec - Boot.BPB\_RsvdSecCnt - (Boot.BPB\_NumFATs \* FATSize)) / Boot.BPB\_SecPerClus;

}

// Get FAT record per FAT32 cluster

DWORD getFAT32RecsPerFATSec(const filesystem::fat32::boot &Boot) {

return Boot.BPB\_BytsPerSec / sizeof(DWORD);

}

// Get FAT directory per cluster

DWORD getFAT32DirRecPerClus(const filesystem::fat32::boot &Boot) {

return (DWORD)Boot.BPB\_BytsPerSec \* Boot.BPB\_SecPerClus / sizeof(filesystem::fat32::dir\_rec);

}

// ---------------------------------------

// Read FAT32 Boot Sector

BOOL readFAT32BootSect(volume::part ^Volume, filesystem::fat32::boot &Boot) {

DWORD Read;

return Volume->ReadSectors(0x00, &Boot, 1, &Read) && Read == sizeof(Boot);

}

// Read FAT32 data cluster

BOOL readFAT32DataClus(volume::part ^Volume, const filesystem::fat32::boot &Boot, DWORD NumClus, LPVOID Buff) {

if (NumClus < 2 || NumClus > getFAT32DataClusCout(Boot)) {

return false;

}

DWORD FATSize = Boot.BPB\_FATSizeSm ? Boot.BPB\_FATSizeSm : Boot.BPB\_FATSizeLr;

QWORD ClusPos = Boot.BPB\_RsvdSecCnt + (QWORD)Boot.BPB\_NumFATs \* FATSize + (NumClus - 2) \* Boot.BPB\_SecPerClus;

return Volume->ReadSectors(ClusPos, Buff, Boot.BPB\_SecPerClus, NULL);

}

// Read FAT32 FAT Value

BOOL readFAT32ClusMark(volume::part ^Volume, const filesystem::fat32::boot &Boot, DWORD NumClus, DWORD &Mark) {

if (NumClus > getFAT32DataClusCout(Boot)) {

return false;

}

DWORD FATsPerSect = getFAT32RecsPerFATSec(Boot);

DWORD \*FATsSect = new DWORD[FATsPerSect];

DWORD NumClusFAT = Boot.BPB\_RsvdSecCnt + NumClus / FATsPerSect;

DWORD NumMarkPos = NumClus % FATsPerSect;

DWORD Read;

BOOL result = Volume->ReadSectors(NumClusFAT, FATsSect, 1, &Read) && Read == sizeof(Boot);

Mark = result ? FATsSect[NumMarkPos] : 0;

delete[] FATsSect;

return result;

}

// Read FAT32 element clusters number list

BOOL readFAT32ClusList(volume::part ^Volume, const filesystem::fat32::boot &Boot, DWORD NumClus, Generic::List<DWORD> ^ClusList) {

ClusList->Clear();

static CONST DWORD SYS = 0x0FFFFFF5;

static CONST DWORD EOC = 0x0FFFFFFF;

DWORD FATsPerSect = getFAT32RecsPerFATSec(Boot);

DWORD \*FATsSect = new DWORD[FATsPerSect];

DWORD Read;

DWORD ClusCount = getFAT32DataClusCout(Boot);

DWORD OldNumClus = EOC;

DWORD NumSectFAT;

DWORD NumMarkPos;

while (NumClus > 1 && NumClus < ClusCount) {

ClusList->Add(NumClus);

NumSectFAT = Boot.BPB\_RsvdSecCnt + NumClus / FATsPerSect;

NumMarkPos = NumClus % FATsPerSect;

if (NumSectFAT == OldNumClus) {

NumClus = FATsSect[NumMarkPos];

}

else if (Volume->ReadSectors(NumSectFAT, FATsSect, 1, &Read) && Read == sizeof(Boot)) {

NumClus = FATsSect[NumMarkPos];

OldNumClus = NumSectFAT;

}

else {

ClusList->Clear();

delete[] FATsSect;

return false;

}

}

delete[] FATsSect;

if (NumClus > SYS && NumClus != EOC) {

ClusList->Clear();

return false;

}

return true;

}

// Read FAT32 info sector

BOOL readFAT32InfoSec(volume::part ^Volume, const filesystem::fat32::boot &Boot, filesystem::fat32::info &Info) {

DWORD Read;

return Volume->ReadSectors((QWORD)Boot.BPB\_SecNumInfo, &Info, 1, &Read) && Read == sizeof(filesystem::fat32::info);

}

// ---------------------------------------

// Get fat32 partition free bytes count

QWORD getFAT32NumOfFreeByte(volume::part ^Volume) {

filesystem::fat32::boot Boot;

if (!readFAT32BootSect(Volume, Boot)) {

return 0;

}

filesystem::fat32::info Info;

if (!readFAT32InfoSec(Volume, Boot, Info)) {

return 0;

}

DWORD ClusPerFAT = getFAT32DataClusCout(Boot);

if (Info.FSI\_FreeCount != 0xFFFFFFFF || Info.FSI\_FreeCount > ClusPerFAT) {

return (QWORD)Info.FSI\_FreeCount \* Boot.BPB\_SecPerClus \* Boot.BPB\_BytsPerSec;

}

DWORD NumClusFAT = Boot.BPB\_RsvdSecCnt;

DWORD RecsPerSec = getFAT32RecsPerFATSec(Boot);

DWORD NumFATSec = ClusPerFAT / RecsPerSec;

DWORD \*FATSec = new DWORD[ClusPerFAT];

QWORD FreeClust = 0;

if (!Volume->ReadSectors(NumClusFAT, FATSec, NumFATSec, NULL)) {

return 0;

}

for (size\_t i = 0; i < ClusPerFAT; i++) {

if (FATSec[i] == 0) {

++FreeClust;

}

}

delete[] FATSec;

return FreeClust \* Boot.BPB\_SecPerClus \* Boot.BPB\_BytsPerSec;

}

// ---------------------------------------

// Get format small name string

String ^SmallNameString(const CHAR SmallName[11]) {

CHAR FormatName[13];

size\_t NamePos = 0;

for (size\_t i = 0; i < 8 && SmallName[i] != 0x20; NamePos++, i++) {

FormatName[NamePos] = SmallName[i];

}

if (SmallName[8] != 0x20) {

FormatName[NamePos] = '.'; NamePos++;

}

for (size\_t i = 8; i < 11 && SmallName[i] != 0x20; NamePos++, i++) {

FormatName[NamePos] = SmallName[i];

}

FormatName[NamePos] = '\0';

return gcnew String(FormatName);

}

// Get format small name hashsum

BYTE SmallNameHashsum(const CHAR SmallName[11]) {

BYTE Sum = 0;

for (size\_t i = 11; i != 0; i--) {

Sum = ((Sum & 1) ? 0x80 : 0) + (Sum >> 1) + (BYTE)\*SmallName++;

}

return (Sum);

}

// Get format directory item name

String ^DirItemNameString(CHAR SmallName[11], stack<filesystem::fat32::dir\_name> FullName) {

if (FullName.empty()) {

return SmallNameString(SmallName);

}

BYTE Hashsum = SmallNameHashsum(SmallName);

String ^LongName = gcnew String(L"");

for (size\_t i = 1; !FullName.empty(); i++, FullName.pop()) {

filesystem::fat32::dir\_name NameItem = FullName.top();

NameItem.LN\_Ordinal == 0x40;

if ((FullName.size() == 1 ? NameItem.LN\_Ordinal & 0x40 != 0x40 : NameItem.LN\_Ordinal != i) || NameItem.LN\_Hashsum != Hashsum) {

return SmallNameString(SmallName);

}

WCHAR NamePart[14];

size\_t PartPos = 0;

for (size\_t i = 0; i < 5; i++, PartPos++) {

NamePart[PartPos] = NameItem.LN\_Part1[i];

}

for (size\_t i = 0; i < 6; i++, PartPos++) {

NamePart[PartPos] = NameItem.LN\_Part2[i];

}

for (size\_t i = 0; i < 2; i++, PartPos++) {

NamePart[PartPos] = NameItem.LN\_Part3[i];

}

NamePart[PartPos] = L'\0';

LongName += gcnew String(NamePart);

}

return LongName;

}

// Get FAT32 directory items

Generic::List<filesystem::object^> ^getFAT32DirIrems(volume::part ^ Volume, String ^ Path) {

Generic::List<filesystem::object^> ^objects = gcnew Generic::List<filesystem::object^>;

filesystem::fat32::boot Boot;

if (!readFAT32BootSect(Volume, Boot)) {

throw gcnew ArgumentException(L"Error reading boot sector.");

}

if (!filesystem::fat32::verify(&Boot)) {

throw gcnew ArgumentException(L"Error boot sector corrupt.");

}

Generic::List<DWORD> ^DirClusList = gcnew Generic::List<DWORD>;

if (!readFAT32ClusList(Volume, Boot, Boot.BPB\_ClusNumRt, DirClusList)) {

throw gcnew Exception(L"Error in directory fat segmentation.");

}

DWORD DirRecCount = getFAT32DirRecPerClus(Boot);

filesystem::fat32::dir\_rec \*FATClus = new filesystem::fat32::dir\_rec[DirRecCount];

stack<filesystem::fat32::dir\_name> \*DirLongNameStack = new stack<filesystem::fat32::dir\_name>;

String ^ClearPAth = (Path->Length ? Path->Remove(0, 1) : Path);

Generic::List<String^> ^Dirs = gcnew Generic::List<String^>(ClearPAth->Split(L'\\'));

Diagnostics::Debug::WriteLine(Dirs->Count);

do {

String ^DirName = Dirs[0];

BOOL IsExstDir = false;

BOOL IsStopped = false;

for each (DWORD DirClusNum in DirClusList) {

if (!readFAT32DataClus(Volume, Boot, DirClusNum, FATClus)) {

throw gcnew Exception(L"Error reading exeption.");

}

for (size\_t i = 0; i < DirRecCount; i++) {

BYTE Flag = (BYTE)FATClus[i].item.DI\_ShortName[0];

if (Flag == 0x00) {

IsStopped = true;

break;

}

bool isEmpty = Flag == (BYTE)0x05;

if (isEmpty) {

isEmpty = true;

for (size\_t i = 1; isEmpty && i < 11; i++) {

if ((BYTE)FATClus[i].item.DI\_ShortName[i] != 0x00) {

isEmpty == false;

}

}

}

if (!isEmpty && Flag == (BYTE)0xE5) {

isEmpty = true;

}

if (!isEmpty) {

if (FATClus[i].name.LN\_Type == 0 && FATClus[i].name.DI\_Attribute == 0x0F && FATClus[i].name.LN\_FstClus == 0) {

if (DirLongNameStack->size()) {

byte LN\_Ordinal = DirLongNameStack->top().LN\_Ordinal;

if (LN\_Ordinal != 0x42 && LN\_Ordinal - 1 != FATClus[i].name.LN\_Ordinal) {

delete DirLongNameStack;

DirLongNameStack = new stack<filesystem::fat32::dir\_name>;

}

}

DirLongNameStack->push(FATClus[i].name);

}

else {

if (DirName == L"") {

String ^ItemName = DirItemNameString(FATClus[i].item.DI\_ShortName, \*DirLongNameStack);

if (ItemName != L"." && ItemName != L"..") {

filesystem::object ^object = gcnew filesystem::object;

object->name = ItemName;

object->attr = FATClus[i].item.DI\_Attributes;

object->size = FATClus[i].item.DI\_FileSize;

objects->Add(object);

}

}

else if (DirName == DirItemNameString(FATClus[i].item.DI\_ShortName, \*DirLongNameStack)) {

DWORD DirClus = FATClus[i].item.DI\_NumClusHI \* ((QWORD)MAXWORD + 1) + FATClus[i].item.DI\_NumClusLO;

if (!readFAT32ClusList(Volume, Boot, DirClus, DirClusList)) {

throw gcnew Exception(L"Error in directory fat segmentation.");

}

IsStopped = true;

IsExstDir = true;

}

delete DirLongNameStack;

DirLongNameStack = new stack<filesystem::fat32::dir\_name>;

if (IsStopped) {

break;

}

}

}

}

if (IsStopped) {

break;

}

}

if (DirName != L"" && !IsExstDir) {

throw gcnew Exception(L"Invalid path.");

}

Dirs->RemoveAt(0);

} while (Dirs->Count);

delete[] FATClus;

delete DirLongNameStack;

return objects;

}

## А.5 Вихідний код модулю partition.h

#pragma once

// ---------------------------------------

#include<Windows.h>

// ---------------------------------------

using namespace System;

// ---------------------------------------

typedef unsigned long long QWORD;

typedef QWORD \*LPQWORD;

// ---------------------------------------

#pragma pack (push, 1)

// ---------------------------------------

// Cylinder-Head-Sector address

typedef struct \_chs {

public:

\_chs();

WORD cyl();

BYTE head();

BYTE sect();

protected:

BYTE bin[3];

} \_chs;

// ---------------------------------------

// Globally Unique Identifier

typedef struct \_guid {

public:

\_guid();

\_guid(DWORD part\_a, WORD part\_b, WORD part\_c, WORD part\_d, QWORD part5);

// convert guid\_t

operator bool() const;

String ^ToString() const;

// guid\_t compare

friend bool operator==(const \_guid &left, const \_guid &right);

protected:

DWORD part\_a; // GUID part 1

WORD part\_b; // GUID part 2

WORD part\_c; // GUID part 3

BYTE part\_d[8]; // GUID part 4

} \_guid;

// ---------------------------------------

#pragma pack (pop)

// ---------------------------------------

// Drive boot record

namespace dbr {

// ---------------------------------------

#pragma pack (push, 1)

// ---------------------------------------

// DBR partition record

typedef struct part {

public:

BYTE boot\_fl; // Boot flag: 0x80 is boot, 0x00 is no boot, else is incorrect

\_chs beg\_chs; // Begin CHS address

BYTE os\_type; // Partition type id

\_chs end\_chs; // End CHS address

DWORD beg\_lba; // Begin LBA address

DWORD lba\_sct; // LBA sectors count

} part;

// ---------------------------------------

// Drive boot record

typedef struct note {

BYTE boot[446]; // Boot sector bytes

part part[4]; // Partition table

WORD sign; // Magic number 0xAA55

} note;

// ---------------------------------------

#pragma pack (pop)

// ---------------------------------------

// DBR partition information

typedef struct info {

BYTE os\_type;

bool is\_part;

bool is\_hide;

LPCWSTR fs\_name;

} info;

// ---------------------------------------

// get part info

const info \*getPartInfo(part part);

// ---------------------------------------

};

// ---------------------------------------

// GUID Partition Table

namespace gpt {

// ---------------------------------------

#pragma pack (push, 1)

// ---------------------------------------

// GPT header record

typedef struct head {

CHAR head\_sign[8]; // Gpt header signature = "EFI PART"

DWORD gpt\_vers; // Version drive GPT

DWORD head\_size; // GPT header size = 92 byte

DWORD check\_sum; // Checksum of header with this \* field zeroed during calculation

DWORD nt\_reserve; // Reserved value = 0

QWORD this\_lba; // Location of this header

QWORD copy\_lba; // Location of other header

QWORD use\_beg\_lba; // First usable LBA for partitions

QWORD use\_end\_lba; // Last usable LBA for partitions

\_guid guid; // Drive GUID indicator

QWORD parts\_lba; // GPT partition table

DWORD parts\_num; // Reserved partition number

DWORD part\_size; // Partition size

DWORD parts\_sum; // Partition checksum

} head;

// ---------------------------------------

// GPT part record

typedef struct part {

\_guid type\_guid; // GUID type for GPT

\_guid part\_guid; // GUID id

QWORD beg\_lba; // Partition begin

QWORD end\_lba; // Partition end

QWORD options; // Options

WCHAR name[36]; // System drive name

} part;

// ---------------------------------------

#pragma pack (pop)

// ---------------------------------------

// GPT part table info struct

typedef struct info {

\_guid type\_guid;

LPCWSTR type\_desc;

} info;

// ---------------------------------------

// get part info

const info \*getPartInfo(part part);

// ---------------------------------------

};

## А.6 Вихідний код модулю partition.cpp

#include "partition.h"

// ---------------------------------------

#include<iomanip>

using std::setw;

using std::setfill;

using std::hex;

using std::dec;

// ---------------------------------------

using namespace System;

// ---------------------------------------

\_chs::\_chs() {

for (size\_t i = 0; i < 3; ++i) {

this->bin[i] = 0;

}

}

WORD \_chs::cyl() {

return ((this->bin[1] >> 6) << 8) | this->bin[2];

}

BYTE \_chs::head() {

return this->bin[0];

}

BYTE \_chs::sect() {

return this->bin[1] & 0x3F;

}

// ---------------------------------------

// Type partition information map

const dbr::info \_dbr\_info\_map[] = {

{ 0x00, true, false, L"EMPTY" },

{ 0x01, true, false, L"FAT12" },

{ 0x8D, true, false, L"FAT12" },

{ 0x04, true, false, L"FAT16" },

{ 0x06, true, false, L"FAT16" },

{ 0x16, true, true, L"FAT16" },

{ 0x0E, true, false, L"FAT16" },

{ 0x1E, true, true, L"FAT16" },

{ 0x0B, true, false, L"FAT32" },

{ 0x1B, true, true, L"FAT32" },

{ 0x0C, true, false, L"FAT32" },

{ 0x1C, true, true, L"FAT32" },

{ 0x07, true, false, L"exFAT/NTFS" },

{ 0x82, true, true, L"LINUX SWAP" },

{ 0x83, true, false, L"LINUX EXT\*" },

{ 0x05, false, false, L"Container" },

{ 0x15, false, true, L"Container", },

{ 0x0F, false, false, L"Container", },

{ 0x0F, false, true, L"Container", },

{ 0x85, false, false, L"Container", }

};

// ---------------------------------------

const dbr::info \*dbr::getPartInfo(dbr::part part) {

size\_t length = sizeof(\_dbr\_info\_map) / sizeof(dbr::info);

for (size\_t i = 0; i < length; i++) {

if (\_dbr\_info\_map[i].os\_type == part.os\_type) {

return \_dbr\_info\_map + i;

}

}

return nullptr;

}

// ---------------------------------------

\_guid::\_guid() :

part\_a(0),

part\_b(0),

part\_c(0) {

for (size\_t i = 0; i < 8; ++i) {

part\_d[i] = 0;

}

}

\_guid::\_guid(DWORD part1, WORD part2, WORD part3, WORD part4, QWORD part5) :

part\_a(part1),

part\_b(part2),

part\_c(part3) {

BYTE \*ptr;

ptr = reinterpret\_cast<BYTE\*>(&part4);

for (size\_t i = 0; i < 2; i++) {

this->part\_d[i] = ptr[1 - i];

}

ptr = reinterpret\_cast<BYTE\*>(&part5);

for (size\_t i = 2; i < 8; i++) {

this->part\_d[i] = ptr[7 - i];

}

}

\_guid::operator bool() const {

if (!this->part\_a || !this->part\_b || !this->part\_c) {

return false;

}

for (size\_t i = 0; i < 8; ++i) {

if (!this->part\_d[i]) {

return false;

}

}

return true;

}

String ^\_guid::ToString() const {

String ^ result = String::Format(L"{0:x8}-{1:x4}-{2:x4}-", this->part\_a, this->part\_b, this->part\_c);

for (size\_t i = 0; i < 2; ++i) {

result += String::Format(L"{0:x2}", (WORD)this->part\_d[i]);

}

result += L'-';

for (size\_t i = 2; i < 8; ++i) {

result += String::Format(L"{0:x2}", this->part\_d[i]);

}

return L"{" + result + L"}";

}

bool operator==(const \_guid &left, const \_guid &right) {

if (left.part\_a != right.part\_a || left.part\_b != right.part\_b || left.part\_c != right.part\_c) {

return false;

}

for (size\_t i = 0; i < 8; ++i) {

if (left.part\_d[i] != left.part\_d[i]) {

return false;

}

}

return true;

}

// ---------------------------------------

const gpt::info \_gpt\_info\_map[] = {

{ { 0x00000000, 0x0000, 0x0000, 0x0000, 0x000000000000 }, L"EMPTY\0 " },

{ { 0x024DEE41, 0x33E7, 0x11D3, 0x9D69, 0x0008C781F39F }, L"MBR Partitions Scheme\0 " },

{ { 0xC12A7328, 0xF81F, 0x11D2, 0xBA4B, 0x00A0C93EC93B }, L"EFI System\0 " },

{ { 0x21686148, 0x6449, 0x6E6F, 0x744E, 0x656564454649 }, L"BIOS Boot\0 " },

{ { 0xD3BFE2DE, 0x3DAF, 0x11DF, 0xBA40, 0xE3A556D89593 }, L"Intel Fast Flash (iFFS)\0 " },

{ { 0xE3C9E316, 0x0B5C, 0x4DB8, 0x817D, 0xF92DF00215AE }, L"Microsoft Reserved Partition (MSR)\0 " },

{ { 0xEBD0A0A2, 0xB9E5, 0x4433, 0x87C0, 0x68B6B72699C7 }, L"Basic data Manager (LDM) metadata\0 " },

{ { 0xAF9B60A0, 0x1431, 0x4F62, 0xBC68, 0x3311714A69AD }, L"Logical Disk Manager data\0 " },

{ { 0xDE94BBA4, 0x06D1, 0x4D40, 0xA16A, 0xBFD50179D6AC }, L"Windows Recovery Environment\0 " },

{ { 0x37AFFC90, 0xEF7D, 0x4E96, 0x91C3, 0x2D7AE055B174 }, L"IBM General Parallel File System (GPFS)\0 " },

{ { 0xE75CAF8F, 0xF680, 0x4CEE, 0xAFA3, 0xB001E56EFC2D }, L"Storage Spaces partition\0 " },

{ { 0x0FC63DAF, 0x8483, 0x4772, 0x8E79, 0x3D69D8477DE4 }, L"Linux filesystem data\0 " },

{ { 0xA19D880F, 0x05FC, 0x4D3B, 0xA006, 0x743F0F84911E }, L"RAID partition\0 " },

{ { 0x0657FD6D, 0xA4AB, 0x43C4, 0x84E5, 0x0933C84B4F4F }, L"/swap\0 " },

{ { 0x44479540, 0xF297, 0x41B2, 0x9AF7, 0xD131D5F0458A }, L"/root (x86)\0 " },

{ { 0x4F68BCE3, 0xE8CD, 0x4DB1, 0x96E7, 0xFBCAF984B709 }, L"/root (x86-64)\0 " },

{ { 0x69DAD710, 0x2CE4, 0x4E3C, 0xB16C, 0x21A1D49ABED3 }, L"/root (ARM x86)\0 " },

{ { 0xB921B045, 0x1DF0, 0x41C3, 0xAF44, 0x4C6F280D3FAE }, L"/root (ARM x64)\0 " },

{ { 0xE6D6D379, 0xF507, 0x44C2, 0xA23C, 0x238F2A3DF928 }, L"Logical Volume Manager (LVM)\0 " },

{ { 0x933AC7E1, 0x2EB4, 0x4F13, 0xB844, 0x0E14E2AEF915 }, L"/home\0 " },

{ { 0x3B8F8425, 0x20E0, 0x4F3B, 0x907F, 0x1A25A76F98E8 }, L"/srv (server data)\0 " },

{ { 0x7FFEC5C9, 0x2D00, 0x49B7, 0x8941, 0x3EA10A5586B7 }, L"Plain dm-crypt partition\0 " },

{ { 0xCA7D7CCB, 0x63ED, 0x4C53, 0x861C, 0x1742536059CC }, L"LUKS partition\0 " },

{ { 0x8DA63339, 0x0007, 0x60C0, 0xC436, 0x083AC8230908 }, L"RESERVE\0 " }

};

// ---------------------------------------

const gpt::info \*gpt::getPartInfo(gpt::part part) {

size\_t length = sizeof(\_gpt\_info\_map) / sizeof(gpt::info);

for (size\_t i = 0; i < length; i++) {

if (\_gpt\_info\_map[i].type\_guid == part.type\_guid) {

return \_gpt\_info\_map + i;

}

}

return nullptr;

}

// ---------------------------------------

## А.7 Вихідний код модулю filesystem.h

#pragma once

// ---------------------------------------

#include<Windows.h>

// ---------------------------------------

using namespace System;

using namespace System::Collections;

// ---------------------------------------

typedef unsigned long long QWORD;

typedef QWORD \*LPQWORD;

// ---------------------------------------

#pragma pack (push, 1)

// ---------------------------------------

// Time

typedef struct \_time {

BYTE second() const;

BYTE minute() const;

BYTE hours() const;

String ^ToString() const;

protected:

WORD time;

} \_time;

// ---------------------------------------

// Date

typedef struct \_date {

BYTE day() const;

BYTE month() const;

DWORD year() const;

String ^ToString() const;

protected:

WORD data;

} \_date;

// ---------------------------------------

// Data & Time

typedef struct \_timestump {

\_time time;

\_date date;

String ^ToString() const;

} \_timestump;

// ---------------------------------------

#pragma pack (pop)

// ---------------------------------------

// Filesystem

namespace filesystem {

// ---------------------------------------

// Directory object

typedef ref struct object {

String ^name; // Object name

DWORD attr; // Object attributes

QWORD size; // Object size

} object;

// ---------------------------------------

// exFAT

namespace exfat {

// ---------------------------------------

#pragma pack (push, 1)

// ---------------------------------------

// exFAT boot record

typedef struct boot {

BYTE BS\_JmpBoot[3]; // JMP instruction

CHAR BS\_OEMName[8]; // OEM ID

BYTE PBP\_AlwaysNull[53]; // Must be zero

QWORD BPB\_PrtOffset; // Partition offset

QWORD BPB\_LengthVol; // Volume length

DWORD BPB\_FATOffset; // Fat offset

DWORD BPB\_FATLength; // Fat length

DWORD BPB\_ClusHepOff; // Cluster heap offset

DWORD BPB\_ClusCnt; // Cluster count

DWORD BPB\_RtDirClus; // Root directory cluster

DWORD BPB\_VolSerial; // Volume serial number

WORD PBP\_FlSysVer; // File system revision

WORD PBP\_VolFlags; // Volume flags

BYTE PBP\_BytsPerSec; // Bytes per sector shift

BYTE PBP\_SecPerClus; // Sectors per cluster shift

BYTE BPB\_NumFATs; // Number ofFats

BYTE BPB\_DrvNum; // Drive select

BYTE BPB\_PrcntInUse; // Percent in use

BYTE BPB\_Reserved[7]; // Reserved

BYTE BS\_Boot[390]; // Boot sector bytes

WORD BS\_Sign; // Magic number 0xAA55

} boot;

// ---------------------------------------

#pragma pack (pop)

// ---------------------------------------

// check boot record

bool verify(const filesystem::exfat::boot \*boot);

// ---------------------------------------

};

// ---------------------------------------

namespace fat32 {

// ---------------------------------------

#pragma pack (push, 1)

// ---------------------------------------

// FAT32 boot record

typedef struct boot {

BYTE BS\_JmpBoot[3]; // JMP instruction

CHAR BS\_OEMName[8]; // OEM ID

WORD BPB\_BytsPerSec; // Bytes per sector

BYTE BPB\_SecPerClus; // Sectors Per Cluster

WORD BPB\_RsvdSecCnt; // Reserved Sectors

BYTE BPB\_NumFATs; // Number of Copies of FAT

WORD BPB\_RootEntCnt; // Maximum number of files in the root directory

WORD BPB\_TotSecSm; // 16-bit count of sectors

BYTE BPB\_MediaDesc; // Media descriptor

WORD BPB\_FATSizeSm; // Sectors per file allocation table (FAT)

WORD BPB\_SecPerTrk; // Sectors per Track

WORD BPB\_NumOfHead; // Number of Head

DWORD BPB\_HiddenSec; // Hidden Sectors

DWORD BPB\_TotSecLr; // Sectors Count in Volume

DWORD BPB\_FATSizeLr; // Sectors per file allocation table (FAT)

WORD BPB\_ExtFlags; // Flags

WORD BPB\_FlSysVers; // Version of FAT32 Drive

DWORD BPB\_ClusNumRt; // Root cluster number

WORD BPB\_SecNumInfo; // FSInfo struct cluster

WORD BPB\_BkBootSec; // Sector Number of the BackupBoot Sector

BYTE BPB\_Reserved[12]; // Not used by FAT

BYTE BPB\_DriveNum; // Logical Drive Number of Partition

BYTE BPB\_CurrHead; // Not used by FAT

BYTE BPB\_BootSign; // Extended Signature (Must be either 0x28 or 0x29)

DWORD BPB\_VolSerial; // Serial Number of Partition

CHAR BPB\_VolLabel[11]; // Partition Volume Name ANSI String

CHAR BPB\_SystemID[8]; // System ID ANSI String

BYTE BS\_Boot[420]; // Boot sector bytes

WORD BS\_Sign; // Magic number 0xAA55

} boot;

// ---------------------------------------

// FAT32 info record

typedef struct info {

DWORD FSI\_LeadSign; // Always = 0x41615252

BYTE FSI\_Reserved1[480]; // Reserved area

DWORD FSI\_StructSign; // Always = 0x61417272

DWORD FSI\_FreeCount; // Free sectors count

DWORD FSI\_FirstFree; // First free cluster

BYTE FSI\_Reserved2[12]; // Reserved area

DWORD FSI\_TrailSign; // Always = 0xAA550000

} info;

// ---------------------------------------

// Directory item struct

typedef struct dir\_item {

CHAR DI\_ShortName[11];

BYTE DI\_Attributes;

BYTE DI\_NTReserved;

BYTE DI\_CreateMS;

\_timestump DI\_Create;

\_date DI\_Access;

WORD DI\_NumClusHI;

\_timestump DI\_Modify;

WORD DI\_NumClusLO;

DWORD DI\_FileSize;

} dir\_item;

// ---------------------------------------

// Directory name struct

typedef struct dir\_name {

BYTE LN\_Ordinal;

WCHAR LN\_Part1[5];

BYTE DI\_Attribute;

BYTE LN\_Type;

BYTE LN\_Hashsum;

WCHAR LN\_Part2[6];

WORD LN\_FstClus;

WCHAR LN\_Part3[2];

} dir\_name;

// ---------------------------------------

// Directory note

typedef union dir\_rec {

dir\_item item;

dir\_name name;

} dir\_rec;

// ---------------------------------------

#pragma pack (pop)

// ---------------------------------------

// check boot record

bool verify(const filesystem::fat32::boot \*Boot);

// ---------------------------------------

};

// ---------------------------------------

namespace fat16 {

// ---------------------------------------

#pragma pack (push, 1)

// ---------------------------------------

// FAT16 boot record

typedef struct boot {

BYTE BS\_JmpBoot[3]; // JMP instruction

CHAR BS\_OEMName[8]; // OEM ID

WORD BPB\_BytsPerSec; // Bytes per sector

BYTE BPB\_SecPerClus; // Sectors Per Cluster

WORD BPB\_RsvdSecCnt; // Reserved Sectors

BYTE BPB\_NumFATs; // Number of Copies of FAT

WORD BPB\_RootEntCnt; // Maximum number of files in the root directory

WORD BPB\_TotSecSm; // 16-bit count of sectors

BYTE BPB\_MediaDesc; // Media descriptor

WORD BPB\_FATSizeSm; // Sectors per file allocation table (FAT)

WORD BPB\_SecPerTrk; // Sectors per Track

WORD BPB\_NumOfHead; // Number of Head

DWORD BPB\_HiddenSec; // Hidden Sectors

DWORD BPB\_TotSecLr; // Sectors Count in Volume

BYTE BPB\_DriveNum; // Logical Drive Number of Partition

BYTE BPB\_CurrHead; // Not used by FAT

BYTE BPB\_BootSign; // Extended Signature (Must be either 0x28 or 0x29)

DWORD BPB\_VolSerial; // Serial Number of Partition

CHAR BPB\_VolLabel[11]; // Partition Volume Name ANSI String

CHAR BPB\_SystemID[8]; // System ID ANSI String

BYTE BS\_Boot[448]; // Boot sector bytes

WORD BS\_Sign; // Magic number 0xAA55

} boot;

// ---------------------------------------

#pragma pack (pop)

// ---------------------------------------

// check boot record

bool verify(const filesystem::fat16::boot \*Boot);

// ---------------------------------------

};

// ---------------------------------------

namespace ntfs {

// ---------------------------------------

#pragma pack (push, 1)

// ---------------------------------------

// NTFS boot record

typedef struct boot {

BYTE BS\_JmpBoot[3]; // JMP instruction

CHAR BS\_OEMName[8]; // OEM ID ANSI String

WORD BPB\_BytsPerSec; // Bytes per sector

BYTE BPB\_SecPerClus; // Sectors Per Cluster

WORD BPB\_RsvdSecCnt; // Reserved Sectors

BYTE BPB\_Unused0; // Always = 0

WORD PBP\_Unused1; // Always = 0

WORD PBP\_Unused2; // Always = 0

BYTE BPB\_MediaDesc; // Media descriptor

WORD PBP\_Unused3; // Always = 0

WORD BPB\_SecPerTrk; // Sectors per Track

WORD BPB\_NumOfHead; // Number of Head

DWORD BPB\_HiddenSec; // Hidden Sectors

DWORD PBP\_Unused4; // Not used by NTFS

DWORD PBP\_Unused5; // Not used by NTFS

QWORD BPB\_TotalSec; // Total sectors in Volume

QWORD PBP\_MFTPrimSecNum; // Logical Cluster Number for the file $MFT

QWORD PBP\_MFTMirrSecNum; // Logical Cluster Number for the file $MFTMirr

BYTE PBP\_BytsPerMTFRec; // Clusters Per MFT Record

BYTE PBP\_Unused6[3]; // Not used by NTFS

BYTE PBP\_ClusPerIdxBuf; // Clusters Per Index Buffer

BYTE PBP\_Unused7[3]; // Not used by NTFS

QWORD BPB\_VolSerial; // Serial Number of Partition

DWORD PBP\_VolHashSum; // Volume Boot Sector Checksum

BYTE BS\_Boot[426]; // Boot sector bytes

WORD BS\_Sign; // Magic number 0xAA55

} boot;

// ---------------------------------------

#pragma pack (pop)

// ---------------------------------------

// check boot record

bool verify(const filesystem::ntfs::boot \*Boot);

// ---------------------------------------

};

// ---------------------------------------

typedef union boot {

filesystem::exfat::boot exfat;

filesystem::fat32::boot fat32;

filesystem::fat16::boot fat16;

filesystem::ntfs::boot ntfs;

} boot;

// ---------------------------------------

String ^fs\_auto(const filesystem::boot \*Boot);

// ---------------------------------------

};

// ---------------------------------------

## А.8 Вихідний код модулю filesystem.cpp

#include "filesystem.h"

// ---------------------------------------

#include<vcclr.h>

// ---------------------------------------

BYTE \_time::second() const {

return (time & 0x0E) \* 2;

}

BYTE \_time::minute() const {

return (time >> 0x05) & 0x1F;

}

BYTE \_time::hours() const {

return (time >> 0x0B) & 0x0F;

}

String ^ \_time::ToString() const {

return String::Format("{0:00}:{1:00}:{2:00}", this->hours(), this->minute(), this->second());

}

// ---------------------------------------

BYTE \_date::day() const {

return data & 0x0F;

}

BYTE \_date::month() const {

return (data >> 0x05) & 0x0F;

}

DWORD \_date::year() const {

return 1980 + (data >> 0x09);

}

String ^\_date::ToString() const {

return String::Format(L"{0}/{1:00}/{2:00}", this->year(), this->month(), this->day());

}

// ---------------------------------------

String ^\_timestump::ToString() const {

return this->date.ToString() + L" " + this->time.ToString();

}

// ---------------------------------------

// ---------------------------------------

String ^ filesystem::fs\_auto(const filesystem::boot \*Boot) {

if (filesystem::exfat::verify(&(Boot->exfat))) {

return L"exFAT";

}

if (filesystem::fat32::verify(&(Boot->fat32))) {

return L"FAT32";

}

if (filesystem::fat16::verify(&(Boot->fat16))) {

return L"FAT16";

}

if (filesystem::ntfs::verify(&(Boot->ntfs))) {

return L"NTFS";

}

return L"UNKNOWN";

}

// ---------------------------------------

bool filesystem::fat16::verify(const filesystem::fat16::boot \*Boot) {

if (Boot->BS\_Sign != 0xAA55 /\*|| !(gcnew String(Boot->BPB\_SystemID))->ToUpper()->Contains(L"FAT16")\*/

|| !Boot->BPB\_BytsPerSec || !Boot->BPB\_RsvdSecCnt || Boot->BPB\_CurrHead) {

return false;

}

DWORD RootDirSect = (Boot->BPB\_RootEntCnt \* 32) / Boot->BPB\_BytsPerSec;

DWORD TotSec = Boot->BPB\_TotSecSm ? Boot->BPB\_TotSecSm : Boot->BPB\_TotSecLr;

DWORD FATSize = Boot->BPB\_FATSizeSm;

if (!TotSec || !FATSize) {

return false;

}

DWORD DataSec = TotSec - Boot->BPB\_RsvdSecCnt - ((DWORD)Boot->BPB\_NumFATs \* FATSize) - RootDirSect;

DWORD CountOfClusters = DataSec / Boot->BPB\_SecPerClus;

return CountOfClusters <= 65524;

}

// ---------------------------------------

bool filesystem::fat32::verify(const filesystem::fat32::boot \*Boot) {

if (Boot->BS\_Sign != 0xAA55 /\*|| !(gcnew String(Boot->BPB\_SystemID))->ToUpper()->Contains(L"FAT32")\*/

|| !Boot->BPB\_BytsPerSec || !Boot->BPB\_RsvdSecCnt || Boot->BPB\_FlSysVers) {

return false;

}

DWORD RootDirSect = (Boot->BPB\_RootEntCnt \* 32) / Boot->BPB\_BytsPerSec;

DWORD TotSec = Boot->BPB\_TotSecSm ? Boot->BPB\_TotSecSm : Boot->BPB\_TotSecLr;

DWORD FATSize = Boot->BPB\_FATSizeSm ? Boot->BPB\_FATSizeSm : Boot->BPB\_FATSizeLr;

if (!TotSec || !FATSize) {

return false;

}

DWORD DataSec = TotSec - Boot->BPB\_RsvdSecCnt - ((DWORD)Boot->BPB\_NumFATs \* FATSize) - RootDirSect;

DWORD CountOfClusters = DataSec / Boot->BPB\_SecPerClus;

return CountOfClusters > 65524;

}

// ---------------------------------------

bool filesystem::exfat::verify(const filesystem::exfat::boot \*Boot) {

if (Boot->BS\_Sign != 0xAA55 || !(gcnew String(Boot->BS\_OEMName))->Contains(L"EXFAT")) {

return false;

}

size\_t AlwaysNullCount = sizeof(Boot->PBP\_AlwaysNull) / sizeof(\*Boot->PBP\_AlwaysNull);

for (size\_t i = 0; i < AlwaysNullCount; i++) {

if (Boot->PBP\_AlwaysNull[i]) {

return false;

}

}

return true;

}

// ---------------------------------------

bool filesystem::ntfs::verify(const filesystem::ntfs::boot \*Boot) {

if (Boot->BS\_Sign != 0xAA55 || !(gcnew String(Boot->BS\_OEMName))->Contains(L"NTFS")

|| Boot->BPB\_Unused0 || Boot->PBP\_Unused1 || Boot->PBP\_Unused2 || Boot->PBP\_Unused3) {

return false;

}

return true;

}

// ---------------------------------------

# Додаток Б Графічна частина