Examensarbete
Kandidat-nivå
Forensisk undersökning av Solid State Drive

Forensic investigation of Solid State Drive

Författare: Richard Ritola
Handledare: Pascal Rebreyend
Examiner: Pascal Rebreyend
Ämne/huvudområde: Datateknik
Poäng: 15 hp
Betygsdatum:

Examensarbete nr: E4144D

Högskolan Dalarna
791 88 Falun
Sweden
Tel 023-77 80 00
Solid State Diskar (SSD) är relativt nya och mycket om dem är okänt. Detta examensarbete fokuserar på att läsa forensiskt viktig information från både lagringsutrymmet och reservutrymmet. Detta har försöks genom att ett program har byggts i C++, detta program använder ATA kommandon för att läsa information från disken. Även om programmet aldrig blev färdigt kunde det skicka och ta emot data från en SSD, dock inte reservutrymmet vilket var fokus i detta examensarbete.
Abstract

Solid State Drives (SSD) are relatively new and not much is known about them. This thesis focuses on retrieving forensically important information, not only from the storage area of the SSD but also from the spare area. This was attempted by writing a program in C++ that, using ATA commands, could read information from the SSD. Although the program was not finished within the given time, it could read some information from the SSD, but not the spare area which was the main focus.
Table of contents
1. Introduction ...................................................................................................................................... 1
  1.1. Purpose and goal ....................................................................................................................... 1
  1.2. Background ............................................................................................................................... 1
  1.3. Methodology ............................................................................................................................. 1
  1.4. Delimitation .............................................................................................................................. 1
  1.5. Resources .................................................................................................................................. 2
  1.6. Basic description of the program .............................................................................................. 2
2. The SSD ........................................................................................................................................... 3
  2.1. SSD basics ................................................................................................................................ 3
    2.1.1 Storage ............................................................................................................................... 3
    2.1.2 Wear leveling ..................................................................................................................... 4
    2.1.3 Addressing ......................................................................................................................... 4
    2.1.4 Spare area .......................................................................................................................... 4
    2.1.5 Garbage collection ............................................................................................................. 4
    2.1.6 Garbage collection and forensics ....................................................................................... 5
  2.2. Choice of SSD ........................................................................................................................... 5
  2.3. The SATA interface ................................................................................................................... 5
3. Program Development ..................................................................................................................... 6
  3.1. Program Structure ..................................................................................................................... 6
  3.2. Basic functions .......................................................................................................................... 8
  3.3. The ATA commands .................................................................................................................. 9
4. Testing ............................................................................................................................................ 10
  4.1 Test stages ................................................................................................................................. 10
    4.2 First test – retrieving device information ................................................................................. 10
    4.3 Reading the maximum address .............................................................................................. 11
    4.4 Changing the maximum address .......................................................................................... 12
    4.5 Reading the entire SSD ........................................................................................................... 13
5. Results and recommendation for future research ........................................................................... 14
  5.1 Results ...................................................................................................................................... 14
  5.2 Recommendation for future research ....................................................................................... 14
  5.3. Problems in the development .................................................................................................. 14
References .......................................................................................................................................... 15

Appendices
Appendix A, Program code for the Main function ............................................................................... 1
Appendix B, Program code for the DevIoCtrl class ............................................................................ 2
Appendix C, Program code for the Vector class .................................................................................. 5
1. Introduction

1.1. Purpose and goal

Since Solid State Drives are relatively new and not much is known about how to retrieve information during forensic investigations in a good way, I aim to investigate this using my own software. My main objective is to try to extract as much information as possible from a Solid State Drive and document which method provides the best results.

1.2. Background

Since a SSD (Solid State Drive) does not provide addressing to the entire storage area but saves some as spare area a normal file carving or imaging tool might not work due to the fact that a part of the drive is hidden outside the address range. This is due to the simple explanation that current imaging tools work within the address range provided by the system.

1.3. Methodology

During the report, I will use my own software written in C++ to connect to the SSD and to send ATA commands in order to try to control it.

Focus will be on using IOCTL_ATA_PASS_THROUGH requests on a computer running Windows OS.

The IOCTL_ATA_PASS_THROUGH control code will be used to send the ATA commands and is available on Windows Server 2003 and later versions of the Windows OS.

The ATA commands that are recognized by the SSD are listed in the documentation I used to determine which SSD was best suited for my thesis.

1.4. Delimitation

The focus of this thesis will be to gain access to the spare area of an SSD.

In order to avoid permanent damage to the SSD, the hardware shall not be opened and the firmware shall not be modified in an attempt to gain access to more data.
1.5. Resources

The SSD used in this research is an Intel X25-E 32GB SSD. Most of the data and information in this report is collected through use of the internet.

1.6. Basic description of the program

The program was built in C++ using Visual Studio 2010 and divided into three main parts: the device interface part, the storage part and the control part. The device interface part of the program refers to all functions and structures related to controlling and retrieving information from the SSD. The storage part of the program refers to the vectors that temporarily store information that is to be used by the program, both the information that is defined during startup and information that is retrieved using the ATA commands. The storage part also contains the file interface functions. The control part of the program is the part that handles controls and initializes the program.

The program uses a set of commands for retrieving data from the drive. Although the program is built for use with the Intel X25-E 32GB SSD, a simple change in the code can adapt the program to other SSDs. The changes could involve changing some ATA commands, since some ATA commands are not included in the general feature set and does not have to be implemented on all devices.
2. The SSD

A SSD is similar to a large flash drive. The absence of any moving parts makes the SSD silent and less sensitive to shaking compared to a traditional hard disk drive (later called HDD) (Moulton, 2008).

Although a SSD may seem like a HDD both to the user and the system, there are still a number of differences.

One difference is that a SSD uses algorithms to determine when a block is damaged and a possible loss of data is likely. When this happens the SSD marks the block as bad and the block will no longer be used to store data. Another difference is that data cannot be overwritten but has to be written to a new location on the disk.

2.1. SSD basics

When comparing a SSD to a HDD more and more differences, not only in behavior but also in data storage, emerge. A HDD stores the data as magnetic charges while a SSD uses flash memory to store the data as electric charges. As for behavior, a SSD moves the data to different places on the device to keep the wear level of each data page even, while a HDD leaves the data in the same place until it is overwritten by new data.

2.1.1 Storage

SSDs use nand flash to store data. Nand flash works by trapping an electric charge between 2 gates in a nand transistor. Typical for nand flash used in solid state drives is that one page is the smallest programmable area, however one block is the smallest erasable area. This is because the drive is physically set up to trap charges to erase a full block at once but can release charges for writing one page. (Moulton, 2008)

As data is written to the SSD, the circuits used to store data is worn down and will eventually be less reliable. When a block has been written to a certain number of times\(^1\), it will be marked as bad and will no longer be included in the user accessible area or the spare area.

\(^1\) The number of times a block can be written to before it is marked as bad and how this marking is done differs between different SSDs.
2.1.2 Wear leveling

Wear leveling is the process of keeping the number of times a page has been written even throughout the device. This is accomplished by moving data to a page that has a higher number of writes than the current page. The page with the lower number of writes is now free and will be used by the SSD for storing new data. Wear leveling also includes blocks within the spare area.

2.1.3 Addressing

Since SSDs move data on the device to new locations for wear leveling purposes, the physical addresses for that data keeps changing. The operating system is not aware of these changes in the addresses and will, therefore, not be able to find the data. To solve this problem, SSDs use a mapping table consisting of LBAs (Logical Block Addresses) and PBAs (Physical Block Addresses) for stored data. As files are moved across the device, the device updates the PBA of data so that the operating system can use the LBA to find data (Wei et al, 2010).

2.1.4 Spare area

Depending on the model of the SSD, the size of the spare area differs. Some SSD only have about 7% of the total area reserved as spare area, while others use up to 28%. These percentages are not included in the user storage space but referred to as extra area. The spare area is used for read-modify-writes, wear leveling and bad block replacement.

Since a file has to be written to a new location when it is updated, it will sometimes leave one or more pages of obsolete data behind. Depending on the SSD and the amount of valid data currently on the SSD, this can leave a block full of obsolete data behind. This block will be rotated into the spare area as a new block is rotated in from the spare area as a part of the wear leveling and garbage collection. This block of obsolete data can, depending on the SSD, be erased upon entering the spare area, upon leaving the spare area or while inside the spare area.

2.1.5 Garbage collection

Because a write operation is only able to release a charge (changing a 1 to a 0), pages containing obsolete data have to be erased before that page can be written to. This means that any time a file is updated, the page (or pages) that are to be updated is written to a new location and the old pages are marked as obsolete to later be erased. This means that even correcting a spelling error in a small text file will result in a new version of the file being written to a new location, leaving the old version virtually untouched in the old location(Wei et al, 2010; Moulton, 2008).
Garbage collection can work in a number of ways, for example a page can be erased directly before it is written to, or when a page is marked to be erased, it is merely added to a queue of pages and will be erased at a later point.

2.1.6 Garbage collection and forensics
Different types of garbage collection will affect forensics in different ways. The garbage collection that implements a queue will slowly clean the SSD of any obsolete data that might sometimes be of forensic significance, without any instructions being sent from the operating system. This will cause the data on the drive to slowly change. This change might even happen as an image is being captured from the drive. The result is that the hash sum of the image and the drive might no longer match and the evidence can be invalidated.

As long as the garbage collection is started on the drive after an image has been captured or while it is being captured, the hash sums of the drive and the image will not match, which can lead to the evidence being invalidated.²

2.2. Choice of SSD
Since the main goal of this thesis is to use ATA commands to retrieve information from an SSD, the documentation, of which commands that were available, was the main point when choosing the SSD. The Intel X25-E 32 GB SSD was chosen because of the extensive documentation of the controller. The documentation listed all the ATA commands that the SSD were to acknowledge and process.

2.3. The SATA interface
SATA stands for Serial Advanced Technology Attachment. SATA is the newer standard for connecting storage devices such as hard drives and CD-ROM. The older standard is called Parallel ATA but it is commonly known as IDE (Wilson, 2011).

The SATA interface has a set of commands where some commands are mandatory to implement on a device. Depending on the type of the device the mandatory commands will change. The SATA interface also has what is called a “General feature set”. This feature set contains all commands that are mandatory for devices that are able to both read and write data, unless they apply the “PACKET feature set” (T13, 2006).

² Of course an exception would be that there is no data for the garbage collection to erase.
The commands included in the general feature set are:

- EXECUTE DEVICE DIAGNOSTIC
- FLUSH CACHE
- IDENTIFY DEVICE
- READ DMA
- READ MULTIPLE
- READ SECTOR(S)
- READ VERIFY SECTOR(S)
- SET FEATURES
- SET MULTIPLE MODE
- WRITE DMA
- WRITE MULTIPLE
- WRITE SECTOR(S)

By focusing on these mandatory commands, the program will be as general as possible without becoming a commercial size project.

3. Program Development

This section describes the development of the program and the commands that were used.

3.1. Program Structure

The program sends one command at a time to the SSD and then waits for the result to be written to a file before sending the next command. By using a vector to store the commands, the number of commands in each set does not need to be specified in advance. Each set of commands have one specific purpose.

The program uses two classes outside of the main function. These classes are DevIoCtrl and Vector. DevIoCtrl contains all code used to communicate with the device, while the Vector class is used to build and contain a set of commands that is to be used by the DevIoCtrl class.

Figure 1 - Describing the Vector class
An object of the Vector class is created by the DevIoCtrl class constructor to handle the storage of ATA commands that are to be sent to the device. The commands are stored in an UCHAR array. When the array is full, it is replaced with a bigger array to which the commands are copied before the old array is deleted. The Vector class contains 6 functions; get, getCapacity, getSize, pushback, incCap and clear. The get function reads a command from the storage array at the index that is received by the functions parameter and returns that command. The getSize function returns the current number of commands. This function is later used by the CallAPT function in the DevIoClass, this to ensure that the program does not try to use more commands than what is saved in the vector. The pushBack function compares the number of commands to the current capacity to see whether the array is full or not before saving the command. If the array is full, the function calls the incCap function to increase the size of the array to make room for the new command before saving it. The clear function deletes the storage of the vector and creates a new storage.

When an object is created from this class it is tied to a specific device, this is due to the fact that a valid handle needs to be sent to the constructor of the class and that handle cannot be changed. The DevIoCtrl class contains 3 functions and one destructor. The functions are addCommand, CallAPT and Ata_Pass_Through. The addCommand function is used to add a command to the vector. This is done by calling the vectors pushback function. The CallAPT function contains a for loop that uses an index starting at 0 and ends at the returned value of the vectors getSize function. This index is later used to retrieve the current command from the vector, when the command is retrieved the function calls the Ata_Pass_Through function using the current command and the handle for the device the class is tied to. The Ata_Pass_Through function builds the structure and sends the command to the device using the DeviceIoControl function described in part 3.2.
### 3.2. Basic functions

The program uses Windows DeviceIoControl (MSDN, 2011) function and ATA_PASS_THROUGH IOCTL (IO Control) to pass control codes to the Solid State Drive. By embedding an ATA command into an ATA_PASS_THROUGH_EX structure, the command can be passed to any ATA device using the DeviceIoControl function.

![Figure 3 - Describing the ATA_PASS_THROUGH_EX structure](Source: MSDN, 2011)

```c
typedef struct _ATA_PASS_THROUGH_EX {
    USHORT          Length;
    USHORT          AtaFlags;
    UCHAR           PathId;
    UCHAR           TargetId;
    UCHAR           Lun;
    UCHAR           ReservedAsUchar;
    ULONG           DataTransferLength;
    ULONG           TimeoutValue;
    ULONG           ReservedAsUlong;
    ULONG_PTR       DataBufferOffset;
    UCHAR           PreviousTaskFile[8];
    UCHAR           CurrentTaskFile[8];
} ATA_PASS_THROUGH_EX, *PATA_PASS_THROUGH_EX;
```

The ATA_PASS_THROUGH_EX structure contains several parameters as can be seen in figure 3. Except for the five parameters PathId, TargetId, Lun, ReservedAsUchar and ReservedAsUlong all parameters are set by the user. The CurrentTaskFile is where the ATA commands, among other things, are stored when they are sent to the device. The CurrentTaskFile is also where some commands return their results, READ NATIVE MAX ADDRESS (0xF8) is one of those commands.

![Figure 4 - Describing the DeviceIoControl function](Source: MSDN, 2011)

```c
BOOL WINAPI DeviceIoControl(
    __in  HANDLE hDevice,
    __in  DWORD dwIoControlCode,
    __in_opt LPVOID lpInBuffer,
    __in  DWORD nInBufferSize,
    __out_opt LPVOID lpOutBuffer,
    __in_opt DWORD nOutBufferSize,
    __out_opt LPDWORD lpdwBytesReturned,
    __inout_opt LPOVERLAPPED lpOverlapped
);
```

To be able to send a command to a device, the hDevice parameter of the DeviceIoControl function needs to contain a valid handle that describes what device that is to receive the command. This handle is created using the CreateFile function which is described below.
As can be seen in the figure before each parameter is declared there is either a __in, __in_opt, __out, __out_opt or a __inout_opt description. The __in and __out parameters are mandatory and describe whether the parameter contains or points to data that is to be sent to the device (__in) or a pointer to a buffer that is to receive data from the device (__out). __inout means that the parameter is a pointer to a buffer that is both read from and written to. If a parameter is optional _opt is added to the description. Although it is important to remember that it is the command being sent to the device or the mode in which the handle was created that decides whether or not a parameter is optional.

Figure 5 - Describing the CreateFile function
(Source: MSDN, 2011)

HANDLE WINAPI CreateFile(
    __in LPCTSTR lpFileName,
    __in DWORD dwDesiredAccess,
    __in DWORD dwShareMode,
    __in_opt LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    __in DWORD dwCreationDisposition,
    __in DWORD dwFlagsAndAttributes,
    __in_opt HANDLE hTemplateFile
);

As shown in figure 5 the CreateFile function returns a handle for the device specified by the lpFileName parameter. The way used to describe the name of the device when calling the function is “Text(“\\\.\PhysicalDrive1” )”, although the number of the drive was one in my case this can change based on the setup of each individual computer.

3.3. The ATA commands

Following is a list of all ATA commands used by the program.

IDENTIFY DEVICE (0xEC) – this command returns a 512 byte block of data that contains basic configuration information about the device as well as serial number and much more. The output parameters are usually defined as 16 bit values (words) although some are defined as 32 bit values (double words).

READ SECTOR(S) (0x20) – By issuing this command the device is told to read a specified number of sectors. The numbers of sectors to be read is specified in the sector count register of the ATA_PASS_THROUGH_EX structure and must be within the range of 0 – 255 where 0 is read as 256 sectors. This command also requires a LBA (Logical Block Address) which consists of 48 bits where bit 47-28 is zero and bit 27-0 contains the address.

SET MAX ADDRESS (0xF9) – this command allows me to change the maximum value of the address range for the storage area in the device. By issuing this command to the device the size
of the user addressable space can be changed.

READ NATIVE MAX ADDRESS (0xF8) – This command returns the maximum address that can be used to address space on the device. The returned result is saved in the CurrentTaskFile array of the ATA_PASS_THROUGH_EX structure.

4. Testing

This section will explain the different tests that were done and which commands were included in each particular test. Each test is done during a specific part of the development phase.

4.1 Test stages

The different stages of testing were used to confirm that the development had been successful up to the point of the test.

The first test determined that the program actually were able to connect to the SSD and retrieve information using the commands and functions described earlier.

Initially the second test was to mainly be focused on preparing for reading small amounts of data while the third test was supposed to focus more on reading larger parts of the SSD. However during the development it was realized that it was more important to get access to the spare area before reading sectors from the SSD. This lead to a change in the plan for the second and third tests so that they focused on reading the maximum address as the second test and changing the maximum address as the third test.

The fourth test was the main focus in this project. This test focuses on reading the entire SSD including the spare area. To do this the maximum address within the device needs to have been changed to force it to use the spare area as user accessible storage area to accommodate for the increase in logical sectors and blocks.

4.2 First test – retrieving device information

The purpose of the first test was to send only one command to see if it was recognized by the SSD. This was done by issuing the ATA command “IDENTIFY DEVICE” (0xEC) and comparing the model and serial number provided by the command to that of the device.

The first test after changing the operating system was successful. After receiving data from the device, and then comparing serial and model number provided by the program to the serial and model number provided by the operating system, it was clear that the program was now connected to the correct device.
As shown in figure 6, it is possible to see both Serial number and Model number which can be used to confirm that the correct device is connected to the program. It is also possible to see that the command copied 552 bytes to the buffer. This includes the 40 bytes ATA_PASS_THROUGH_EX structure that was sent and then returned from the device.

### 4.3 Reading the maximum address

To read the maximum address of the SSD, I used the READ NATIVE MAX ADDRESS (0xF8) command. To prove that the data was read from the SSD, an IDENTIFY DEVICE command was issued in the beginning of the command set. This provided both model and serial number of the device currently connected to the program. The user is prompted to confirm that the program is connected to the correct device. This is done so that the user can abort the program to ensure that the commands are not sent to a device other than the SSD.
Figure 7 shows the result from a successful READ MAX ADDRESS command. As can be seen in figure 7, the result is 40 bytes (bytes copied) which means that all information is contained within the returned ATA_PASS_THROUGH_EX structure. The CurrentTaskFile contains the address and shall be interpreted as [0] error register, [1] sector count register, [2] sector number register, [3] cylinder low register [4] cylinder high register and [5] device/head register, [6] status register, [7] reserved (MSDN, 2011).

The sector count register ([1]) is not used by this command and is therefore set to 0.

4.4 Changing the maximum address

This stage was dedicated to changing the maximum address of the device by using the SET MAX ADDRESS (0xF9) command. Due to a number of unknown reasons, the SSD did not agree to a change of the maximum address, both when trying to increase and decrease it. One of many results that I got during these tests is shown in the figure 8 below.
Figure 8 - Output from SET MAX ADDRESS and READ NATIVE MAX ADDRESS

As can be seen in figure 8, the program first reads the maximum address, then tries to set a new maximum address and finally reads it once more to determine whether the set command was successful.

Although the program reports that the 0xf9 (SET MAX ADDRESS) command was sent properly, the maximum address does not change, as seen in the second 0xf8 (READ NATIVE MAX ADDRESS) command output.

4.5 Reading the entire SSD

My initial plan for this stage was to read the entire SSD. This should have been done by increasing the maximum address so that it included both the user accessible storage area and the spare area. When this was done, I would have read the entire drive and created an image of the SSD.

Since I did not manage to set a new maximum address, this test could not been performed within the given time.
5. Results and recommendation for future research

This section will describe the result and provide recommendations for future research.

5.1 Results

The program created in this paper is able to send ATA commands to a SSD to control it in a way that is similar to the way it would be controlled by an operating system.

Although my program cannot change the maximum address of a SSD, it is still able to connect to and retrieve information from a SSD by using ATA commands. Since commands to increase and decrease the user accessible address range is implemented on the device I do believe that the ideas and methods I have used and was planning on using is one of the possible ways of retrieving forensically significant data from the spare area of a SSD.

5.2 Recommendation for future research

The SSD is a product that most likely will be more common in the future due to its advantages compared to HDDs. This will lead to an increase in the demand for forensic software and investigative techniques focusing on SSDs.

A software or technique that is able to preserve deleted data by disabling the garbage collection and wear leveling, or a way of accessing the controller to extend or control the LBA will most probably impact the forensic investigations of SSDs in a significant way.

5.3. Problems in the development

The biggest problem I encountered while writing my program was Windows XP. According to Microsoft, Windows 2000 and later do support the DeviceIoControl functions and in one of their support articles (Microsoft, 2007) implies that Windows XP SP2 and later do support the ATA_PASS_THROUGH structure, although there is a problem with the 48bit command flag. This however was not the case for me.

In the beginning of the development phase I was working with Windows XP SP 3 and for a long time I was convinced that the fault was somewhere in my code. But after testing my code in both Windows Vista and Windows 7 it was clear to me that the fault was not in my code but in the operating system. This problem was solved after Windows 7 was installed on my computer. However, the program still requires administrative privileges to be able to use the ATA commands.
References

Articles

Internet
Moulton, S. Solid State Drives will Ruin Forensics <http://www.youtube.com/watch?v=WcO7xn0wJ2I> (August 2008)
Appendices

Appendix A, Program code for the Main function

```cpp
#include <Windows.h>
#include <iostream>
#include <cstring>
#include <ntddscsi.h>
#include "DevIoCtrl.h"

using namespace std;

void main()
{
    char answer;
    //open a device handle for physicaldrive 1
    HANDLE device = CreateFile(TEXT("\\\PhysicalDrive1"), GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING, NULL, NULL);
    if(device != INVALID_HANDLE_VALUE)
    {
        cout << "Handle Created!  " << device << endl;
        DevIoCtrl Ctrler(device); // Initializes the DevIoCtrl class

        // adds an identify device command to the current command set
        Ctrler.addCommand(0xEC);
        Ctrler.callAPT(); // calls the current commandset

        // checks that the program connected to the intended device
        cout << endl << "Is this the correct device?(y/n): ", cin >> answer;
        cout << endl;
        if (answer == 'y' || answer == 'Y')
        {
            Ctrler.addCommand(0xF8); //read native max address
            Ctrler.addCommand(0xF9); //set max address
            Ctrler.addCommand(0xF8); //read native max address
            Ctrler.callAPT(); // start sending commands
        }
    }
    else // called if the program could not open a device handle
    {
        cout << "Could not open the device" << endl;
        cout << "device " << endl;
        cout << dec << "Error: " << (int)GetLastError() << endl;
    }

    /*Closes the device handle and checks wheter the device closed successfully*/
    int closed = CloseHandle(device);
    if (closed != 0)
    {
        cout << endl << "Device handle closed successfully" << endl;
    }
    cout << "DONE" << endl;
}
```
Appendix B, Program code for the DevIoCtrl class

```cpp
#ifdef DevIoCtrl_h
#define DevIoCtrl_h

#include <iostream>
#include <string>
#include "vector.h"
#include <fstream>
#include <winioctl.h>
#include <Winbase.h>
using namespace std;

class DevIoCtrl
{
public:
    DevIoCtrl(HANDLE Device) // constructor that initializes the variables of the class.
    {
        device = Device;
        readIndex = 0x00;
        firstSector = 0x00;
        OFsectorRead.open("Sectors.txt");
        cylinderLow = 0x00;
        cylinderHigh = 0x00;
        maxRead = false;
    }

    void addCommand(unsigned char command) // adds a command to the vector
    {
        commands.pushBack(command);
    }

    void callAPT()
    {
        UCHAR command;
        for (int i = 0; i < commands.getSize(); i++) // loop that goes from 0 to the size of the current
            command set - 1
        {
            command = commands.get(i); // gets the current command to be sent
            bool result = Ata_Pass_Through(device, command); // calls the Ata_Pass_Through
                function which returns the result as true or false
                if(result) // notifies the user that the command was sent successfully
                {
                    cout << "Command sent successfully ";
                    cout << "(Command " << i+1 << " of " << commands.getSize() << ")" << endl;
                }
                else // notifies the user that the command was not sent successfully
                {
                    cout << "command 0x" << hex << (int)command << " Failed" << endl;
                    cout << "call APT Error: " << GetLastError() << endl;
                }
                commands.clear(); // clears the current command set.
            }

    BOOL Ata_Pass_Through(HANDLE hDevice, UCHAR cmd) // function that prepares and sends a
        command to the device
    {
```
BOOL status = FALSE;
PATA_PASS_THROUGH_EX pATAData;
DWORD dataSize = sizeof(ATA_PASS_THROUGH_EX) + 512;
BYTE Buffer[sizeof(ATA_PASS_THROUGH_EX) + 512];
DWORD bytescopied = 0;
//UCHAR DriveNum = 0;

//Allocate memory for ATA_PASS_THROUGH_EX and clear the contents
pATAData = (ATA_PASS_THROUGH_EX*)Buffer; //allocates the Buffer variable as storage space to simplify the data extraction
ZeroMemory(pATAData,dataSize); // clears the buffer

// Fills in the required data
pATAData->Length = sizeof(ATA_PASS_THROUGH_EX);
pATAData->DataBufferOffset = sizeof(ATA_PASS_THROUGH_EX);
pATAData->DataTransferLength = 512;
pATAData->TimeOutValue = 2; //Seconds

if(cmd == 0xF9 && maxRead == true) // if the current command is SET MAX ADDRESS
{
    // Values for the new maximum address
    pATAData->CurrentTaskFile[1] = 0x01;
pATAData->CurrentTaskFile[2] = 0x9F;
pATAData->CurrentTaskFile[3] = 0xAC;
pATAData->CurrentTaskFile[4] = 0xBA;
pATAData->CurrentTaskFile[5] = 0xA3;
pATAData->AtaFlags = ATA_FLAGS_DATA_OUT;
}
else
{
    // Values for reading a sector on the device
    pATAData->CurrentTaskFile[1] = 0x00;
pATAData->CurrentTaskFile[2] = readIndex;
pATAData->AtaFlags = ATA_FLAGS_DATA_IN;
}
pATAData->CurrentTaskFile[6] = cmd; // stores the command

/* sends the command to the device*/
status = DeviceIoControl(hDevice, IOCTL_ATA_PASS_THROUGH, pATAData, dataSize, 
pATAData, dataSize, &bytescopied, NULL);

if(!status) // if the command could not be sent
{
    cout << endl << "Error in DevIoCtrl.DevIO.Ata_Pass_Through function!" << endl;
    cout << "Error: " << GetLastError() << endl;
}
else // if the command was send successfully
{
    cout << hex << endl << "Current command: 0x" << (int)cmd << endl;
    if(cmd == 0xec) // if the command was IDENTIFY DEVICE
    {
        toFile(Buffer, pATAData->DataBufferOffset);
        BYTE *p;
        for (int j = 0; j <= sizeof(Buffer) - 0x28; j++)
        {
            p = Buffer + 0x28 + j*2;
            OFsectorRead << *(p+1) << *p;
        }
    }
cout << endl << "Device read successfully" << endl;  
cout << "Result saved in Devinfo.txt" << endl;
}
else if (cmd == 0x20) // if the command was READ SECTOR(S)
{
BYTE *p;
cout << (int)cylinderHigh << (int)cylinderLow << (int)readIndex <<
endl; // prints the current address

if (readIndex == 0xff) // increasing the address properly
{
    readIndex = 0x00;
    if (cylinderLow == 0x0f)
    {
        cylinderHigh++;
        cylinderLow = 0x00;
    }
    else
    {
        cylinderLow++;
    }
}
else
{
    readIndex++;
}
// writes the bytes that were read to a file
for (int i = 0; i < ((bytescopied-0x28) / 2); i++)
{
    p = Buffer + 0x28 + i*2;
    OFsectorRead << *p << *(p+1);
}
cout << endl << "Device read successfully" << endl;
OFsectorRead.flush();
}
//if the command was a READ NATIVE MAX ADDRESS
// or SET MAX ADDRESS
else if(cmd == 0xF8 || cmd == 0xF9)
{
// writes the content of the PreviousTaskFile and CurrentTaskFile
    cout << "PreviousTaskFile:";
    for (int i = 0; i < 8 ; i++)
    {
        cout << " ["<< i <<"]": " << (int)pATAData->PreviousTaskFile[i];
    }
    cout << endl << "CurrentTaskFile:";
    for (int i = 0; i < 8 ; i++)
    {
        cout << " ["<< i <<"]": " << (int)pATAData->CurrentTaskFile[i];
        setmax[i] = pATAData->CurrentTaskFile[i];
    }
    cout << endl;
    maxRead = true;
}
}

cout << "bytes copied: " << dec << (int)bytescopied<< endl; // writes the number of copies bytes

VirtualFree(pATAData, dataSize, MEM_RELEASE); // releases the memory
return status; // returns true if the command was successfully sent, otherwise returns false
// writes the IDENTIFY DEVICE data to the console and to a file
void toFile(BYTE * Buffer, ULONG buffOffset)
{
    Outfile.open("Devinfo.txt");
    BYTE *p;

    // writes the Serialnumber, Microcode version and modelnumber to the console and a file
    for (int i = 0; i <= 2; i++)
    {
        cout << text[i];
        Outfile << text[i];
        for (int j = start[i]; j <= stop[i]; j++)
        {
            p = Buffer + 0x28 + j*2;
            cout << *(p+1) << *p;
            Outfile << *(p+1) << *p;
        }
        Outfile << endl;
        cout << endl;
    }
    Outfile.close();
}
~DevIoCtrl() // class destructor
{
    OFsectorRead.close();
}

private:
    Vector commands;
    ofstream Outfile, OFsectorRead;
    HANDLE device;
    UCHAR readIndex, firstSector, cylinderLow, cylinderHigh, setmax[8];
    bool maxRead;
};
#endif

**Appendix C, Program code for the Vector class**

ifndef vector_h
#define vector_h
#include <Windows.h>

class Vector
{
public:
    Vector(int Cap):capacity(Cap) // constructor that can be used to set a starting size of the vector
    {
        firstEmpty = 0;
        size = 0;
        storage = new unsigned char[capacity];
    }
    Vector():capacity(1)
    {
        firstEmpty = 0;
        size = 0;
        storage = new unsigned char[capacity];
    }

    ~Vector() // destructor
    {
        delete[] storage;
    }
    // ...
UCHAR get(int Index) //Gets a value from a given index of the vector
{
    if (Index > (size - 1))
    {} else
        return *(storage+Index);
}

int getCapacity() //returns the capacity of the vector
{
    return capacity;
}

void clear() //empties the vector
{
    UCHAR *old;
    old = storage;
    storage = new unsigned char[1];
    delete[] old;
    firstEmpty = 0;
    size = 0;
}

int getSize() //shows the number of commands currently in the vector
{
    return size;
}

void pushBack(UCHAR command) //adds a new command at the end of the vector
{
    if (size < capacity)
    {
        *(storage+firstEmpty) = command;
        firstEmpty++;
        size++;
    } else
    {
        incCap();
        *(storage+firstEmpty) = command;
        firstEmpty++;
        size++;
    }
}

void incCap() //increases the size of the vector
{
    unsigned char *old;
    int oldCap;
    old = storage;
    oldCap = capacity;
    capacity = capacity * 2;
    storage = new unsigned char[capacity];

    for (int i = 0; i < size; i++)
    {
        *(storage + i) = *(old + i);
    }
    delete[] old;
}
~ Vector()
{
    delete[] storage;
}

private:
    int size;
    int firstEmpty;
    int capacity;
    unsigned char *storage;
};
#endif