Student Octavian-Mihai Matei

Group 30431

PC Benchmark application in C#

Table of contents

QR to the github repository

Introduction

Goal

The goal of this project is to design, implement and test a benchmark application that runs on a machine and can determine the following statistics:

- For the CPU: Type, MIPS and speed of simple operations
- For the RAM: Dimension, Health, Speed of RAM
- For the Storage: Dimension, Health, Speed of Storage

These tests should be comparable with the system status and online information provided with the hardware, but could also be ranked on a service against other machines, in order to determine the efficiency of the current system.

Specifications

For the local application, the program should apply specific algorithms, in order to determine in the most efficient way the status of the system, as well as work out any major flaw in the RAM configuration. It will run on the machine directly, to be able to access the hardware components more easily. After the tests are done, the results will be displayed in a user-friendly way to the screen and a rank fetched from the service will be displayed. If the user wants to share any information with the service, they will can do it.

Bibliographic Study

For this kind of application, there are lots of information in different categories, but the main sites were the Intel website, the Microsoft documentation, but also univeristy studies and niche-websites. All of them can be consulted in the bibliography.

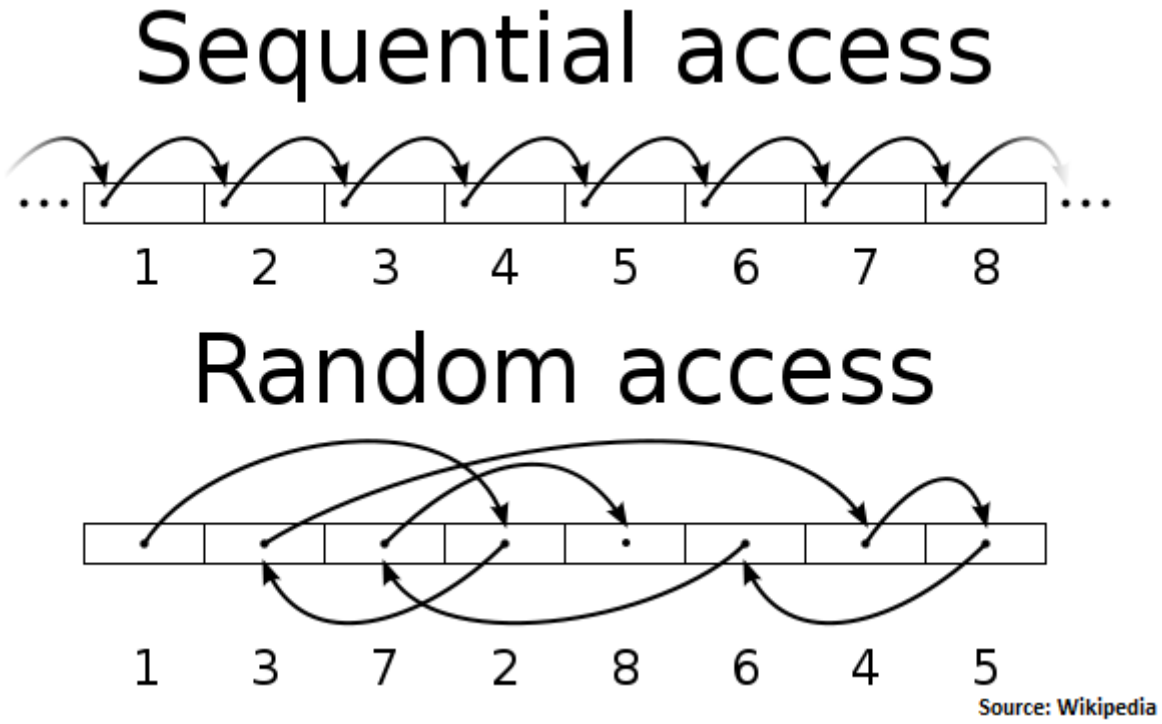
CPU Benchmarking

Modern CPUs have a lot of fine tunning in order to maximise the efficiency of the hardware, so a lot of work has to be done in order to ensure consistent metrics and repeatable measurements. In order to do that, we have the ability to set the afinity of the processor, manage the transition of the compiler from readable code to machine code, but also a way to asses if the measurements are correct, by checking the internal library managed by Microsoft. We also have to bind each iteration of the run to one physical processor with the command export `OMP_PROC_BIND=true`^1 and we have to take into account the hyperthreading of the processor by dividing by 2 the physical number of processors.

For the simple operations, a simple sum from 1 to 1.000.000 should suffice. The same would be true for the substraction operation.

RAM Benchmarking

There are a lot of algorithms to asses the health of the RAM, from MSCAN algorithms to GALPAT or WALPAT^6^7. All of these algorithms test the read-write ability of the system of specific RAM cells zones so they are all good benchmarks for the health consideration. We will use the sequential vs random write-read modes, ways explained in the photo bellow:



That means data, as bytes, will be declared as a vector/array in order to test the sequential access health and as linked lists for the random access. The data used will either be 0xFF or 0x00 in chunks of 32768 addresses (2^16). In that way, we can assure the user of the efficiency of the algorithm and the real health of the RAM.

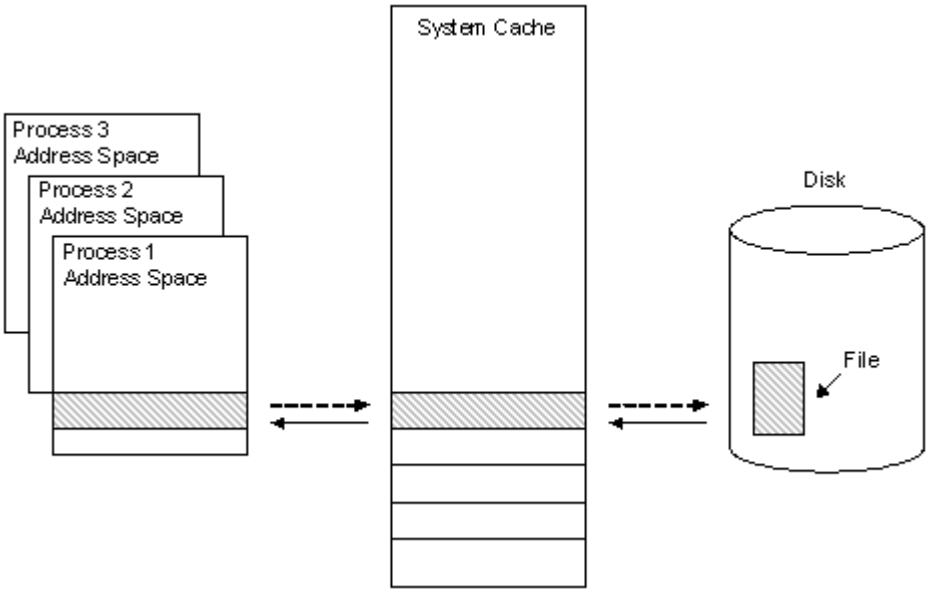
The speed of the RAM would be the time the algorithm described above takes to write-read into the system memory.

Storage Benchmarking

The user should be allowed to choose a drive on which the test takes place. For example, `Choose Drive C` will make the application to run on that drive, in order to test the speed and health of the partition.

The storage benchmarking will be similar to the one of the RAM, with the exception that the data must be written on the actual storage, not in the RAM. There is also the problem where the storage will write in the cache memory[^9] as well as in the actual file, so the test will be nullified. Windows creates a 256 KB buffer that is read into a 256 KB cache "slot" in system address space when it is first requested by the cache manager during a file read operation which can be explained with the photo below. This can be turned off by setting the flag FILE_FLAG_NO_BUFFERING on off.

The speed of the Storage would be the time the algorithm described above takes to write-read into the system storage.

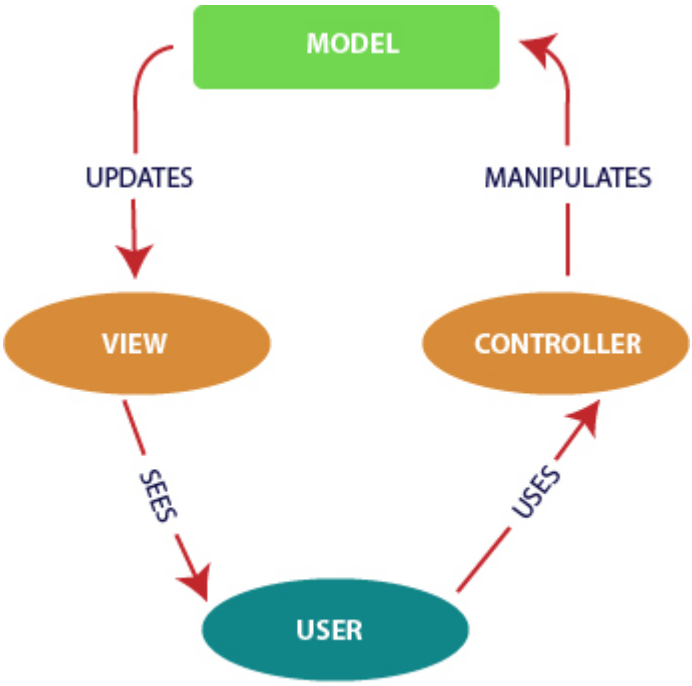


Microsoft Defined Data Structures Information

Microsoft provides with a lot of information about the system, information which can be used to validate and display as "Expected values" to the user. The same information can include the type of the CPU, the battery status, RAM type/frequency etc. These data structures can be used with a simple method call to obtain a lot of information about the current system, without the need to implement complex algorithms and manual testing of the performance.

Service Side

The application will have the ability to display similar systems or better ones, in order to tell the user what is wrong and what can be changed. Moreover, the system will have the ability to upload the system specifications on the service to be used by other people. From our experience, the most facil way for us to implement a Saas[^10] is to use ASP.NET with a MVC methodology.



MVC Architecture

Planning

- Until 19th October:
  - Research the subject and make the introduction, bibliographic study and planning part of the documentation
- Until 2nd November:
  - Write CPU analysis chapter in documentation
  - Write the design of the CPU benchmark classes in documentation
  - Implement the CPU benchmark classes
  - Write the implementation in documentation
- Until 16th November:
  - Write RAM and Storage analysis chapter in documentation
  - Write the design of the RAM and Storage benchmark classes in documentation
  - Implement the RAM and Storage benchmark classes
  - Write the implementation in documentation
- Until 30th November:
  - Write the design of the Microsoft Defined Data Structures Integration in documentation
  - Implement the Microsoft Defined Data Structures Integration classes
  - Write the implementation in documentation
  - Write the design of the GUI in documentation
  - Implement the GUI classes
  - Write the implementation in documentation
- Until 14th December:
  - Finalize the documentation
- Present the project on the 14th December

Analysis

CPU analysis

MIPS

By analysing the algorithms for the computation of the MIPS of the CPU, three choices were selected for the ease of coding and their clarity. All the results from the algorithms are averaged and the CPU MIPS is resulted.

First algorithm - Sha256 hashing

The SHA-256 algorithm is one flavor of Secure Hash Algorithm 2, which was created by the NSA in 2001 as a successor to SHA-1. SHA-256 is a patented cryptographic hash function that outputs a value that is 256 bits long. What is the difference between encryption and hashing? In encryption, data is transformed into a secure format that is unreadable unless the recipient has a key. In its encrypted form, the data may be of unlimited size, often just as long as when unencrypted. In hashing, by contrast, data of arbitrary size is mapped to data of fixed size. For example, a 512-bit string of data would be transformed into a 256-bit string through SHA-256 hashing.[^15]

```
Mathematical behaviour of SHA256
Ch(x, y,z) = (x ∧ y) ⊕ (¬x ∧ z) (4.2)
Maj(x, y,z) = (x ∧ y) ⊕ (x ∧ z) ⊕ ( y ∧ z) (4.3)
∑{256}0 (x) = ROTR 2(x) ⊕ ROTR 13(x) ⊕ ROTR 22(x)
∑{256}1 (x) = ROTR 6(x) ⊕ ROTR 11(x) ⊕ ROTR 25(x)
σ{256}0 x = ROTR 7(x) ⊕ ROTR 18(x) ⊕ SHR 3(x)
σ{256}1 x = ROTR 17(x) ⊕ ROTR 19(x) ⊕ SHR 10(x)
```

As can be seen from the mathematical behaviour, this benchmark was designed to test the logical speed of the ALU of the CPU.

Second algorithm - Exponentiation function

Exponentiation is a mathematical operation, written as b^n, involving two numbers, the base b and the exponent or power n, and pronounced as "b raised to the power of n". When n is a positive integer, exponentiation corresponds to repeated multiplication of the base, that is: `b^n = b*b*b*b*...*b*b*b`

As can be seen from the mathematical behaviour, this benchmark was designed to test the multiplication speed of the ALU of the CPU

Third algorithm - For loop

A for-loop is a computer-science concept that has two parts: a header specifying the iteration, and a body which is executed once per iteration. The header explicitly tells the compiler how the loop should behave, by storing the current iteration index, a condition for stop and an increment of the index. On the other hand, the body has implemented the actual code that hav to be run multiple times, but also breaking statements that exit the loop. A simple example in assembly:

```
MOV CL, 10
L1:
<LOOP-BODY>
DEC CL
JNZ L1
```

As can be seen from the mathematical behaviour, this benchmark was designed to test the speed of the entire CPU.

Speed of simple operations

By analysing the algorithms for the computation of speed of simple operations done by the CPU, a simple algorithm was considered, because it needs only the basic ALU behaviour to be executed (addition and substraction). For this, a loop that adds and substracts a given int was considered because it needs only around 3-4 clock cycles to be completed. A simple example in assembly of the behaviour:

```
// Addition
mov     eax, 14
mov     ebx, 10
add     eax, ebx
// Substraction
mov     eax, 14
mov     ebx, 10
sub     eax, ebx
```

RAM analysis

Random-access memory (RAM) is a form of computer memory that can be read and changed in any order, typically used to store working data and machine code. A random-access memory device allows data items to be read or written in almost the same amount of time irrespective of the physical location of data inside the memory, in contrast with other direct-access data storage media (such as hard disks, CD-RWs, DVD-RWs and the older magnetic tapes and drum memory), where the time required to read and write data items varies significantly depending on their physical locations on the recording medium, due to mechanical limitations such as media rotation speeds and arm movement.

RAM contains multiplexing and demultiplexing circuitry, to connect the data lines to the addressed storage for reading or writing the entry. Usually more than one bit of storage is accessed by the same address, and RAM devices often have multiple data lines and are said to be "8-bit" or "16-bit", etc. devices.

RAM Health

In order to test the RAM Health, a bit per bit check will be required. Unfortunately, this is impossible in the modern programming languages, so a 4/8 bytes chunk will be used at a time in the form of an int. These values will be x5555 (0101010101010101) and xAAAA (1010101010101010).

These values will be written and read from RAM multiple times and discrepancys between write and read values will be added to an errorCheck. In the end, the fraction of the errorCheck and total values will be computed and a procentage will be displayed.

Storage analysis

A hard disk drive (HDD), hard disk, hard drive, or fixed disk is an electro-mechanical data storage device that stores and retrieves digital data using magnetic storage and one or more rigid rapidly rotating platters coated with magnetic material. The platters are paired with magnetic heads, usually arranged on a moving actuator arm, which read and write data to the platter surfaces. Data is accessed in a random-access manner, meaning that individual blocks of data can be stored and retrieved in any order. HDDs are a type of non-volatile storage, retaining stored data even when powered off. Modern HDDs are typically in the form of a small rectangular box.

Introduced by IBM in 1956, HDDs were the dominant secondary storage device for general-purpose computers beginning in the early 1960s. HDDs maintained this position into the modern era of servers and personal computers, though personal computing devices produced in large volume, like cell phones and tablets, rely on flash memory storage devices. More than 224 companies have produced HDDs historically, though after extensive industry consolidation most units are manufactured by Seagate, Toshiba, and Western Digital. HDDs dominate the volume of storage produced (exabytes per year) for servers. Though production is growing slowly (by exabytes shipped), sales revenues and unit shipments are declining because solid-state drives (SSDs) have higher data-transfer rates, higher areal storage density, somewhat better reliability, and much lower latency and access times.

The revenues for SSDs, most of which use NAND flash memory, slightly exceed those for HDDs. Flash storage products had more than twice the revenue of hard disk drives as of 2017. Though SSDs have four to nine times higher cost per bit, they are replacing HDDs in applications where speed, power consumption, small size, high capacity and durability are important. Cost per bit for SSDs is falling, and the price premium over HDDs has narrowed.

Storage Health

In order to test the Storage Health, a bit per bit check will be required. Unfortunately, this is impossible in the modern programming languages, so a 4/8 bytes chunk will be used at a time in the form of an int. These values will be x5555 (0101010101010101) and xAAAA (1010101010101010).

These values will be written and read from RAM multiple times and discrepancys between write and read values will be added to an errorCheck. In the end, the fraction of the errorCheck and total values will be computed and a procentage will be displayed.

Microsoft Defined Data Structures Information

The Computer System Hardware category groups classes together that represent hardware related objects. Examples include input devices, hard disks, expansion cards, video devices, networking devices, and system power. These classes provide information about the system without any computation, only using ManagementObjectSearcher object, which retrieves a collection of management objects based on a specified query. This class is one of the more commonly used entry points to retrieving management information.

For example, it can be used to enumerate all disk drives, network adapters, processes and many more management objects on a system, or to query for all network connections that are up, services that are paused, and so on. When instantiated, an instance of this class takes as input a WMI query represented in an ObjectQuery or its derivatives, and optionally a ManagementScope representing the WMI namespace to execute the query in. It can also take additional advanced options in an EnumerationOptions.

When the Get() method on this object is invoked, the ManagementObjectSearcher executes the given query in the specified scope and returns a collection of management objects that match the query in a ManagementObjectCollection.

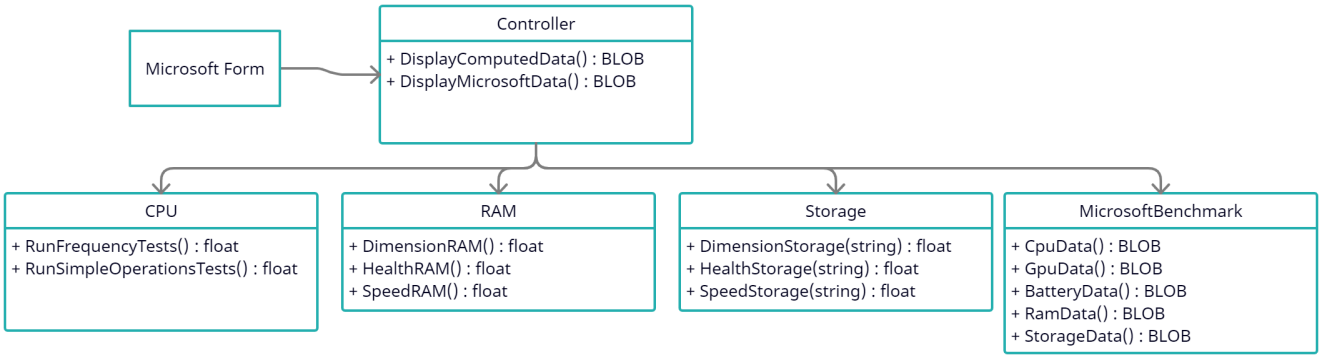
Design

Local application design

Class Diagram

The Unified Modeling Language (UML) is a general-purpose, developmental, modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system.

A class diagram is a diagram used in designing and modeling software to describe classes and their relationships. Class diagrams enable us to model software in a high level of abstraction and without having to look at the source code. Classes in a class diagram correspond with classes in the source code. The diagram shows the names and attributes of the classes, connections between the classes, and sometimes also the methods of the classes.



Class explanation

Controller

The controller class should have 2 public classes:

1. DisplayComputedData -> has no parameters and returns a complex object with all the data that was computed by the algorithms implemented in the application
2. DisplayMicrosoftData -> has no parameters and returns a complex object with all the data from the Microsoft Library

UI

The UI is a cluster of Microsoft forms that display the information to the user. These are composed of two parts:

1. The actual UI that draws the information
2. An internal controller that manages the data

CPU

The CPU class should have 2 public classes:

1. RunMIPSTests -> has no parameters and returns a float value representing the MIPS of the machine
2. RunSImpleOperationTests -> has no parameters and returns a float value representing the number of clock cycles the CPU needs to perform simple operations

RAM

The RAM class should have 3 public classes:

1. DimensionRAM -> has no parameters and returns a float value that represents the space available on the RAM chip in Mb
2. HealthRAM -> has no parameters and returns a float value representing the health of the RAM stick from 0 to 100% healthy
3. SpeedRAM -> has no parameters and returns a float value representing the MIPS of the RAM stick in MHz

Storage

The Storage class should have 3 public classes:

1. DimensionStorage -> has a string parameter that represents the path/drive for analysis and returns a float value that represents the space available on the storage path in Mb
2. HealthStorage -> has a string parameter that represents the path/drive for analysis and returns a float value representing the health of the storage path from 0 to 100% healthy
3. SpeedStorage -> has a string parameter that represents the path/drive for analysis and returns a float value representing the MIPS of the storage in MHz

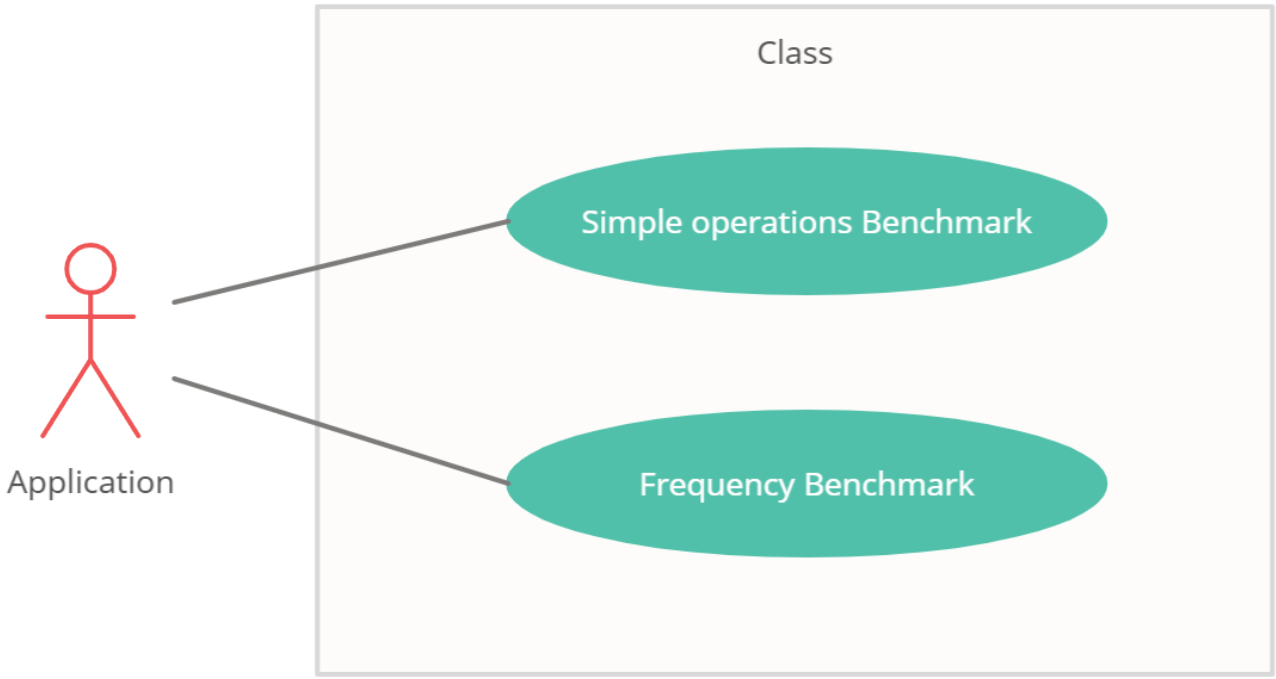
MicrosoftBenchmark

The MicrosoftBenchmark class should have 5 public classes:

1. CpuData -> has no parameters and returns a complex object representing CPU information obtained by querring the Microsoft Library, from Win32_Processor^17 (ProcName, Manufacturer, Version, NrCores, NrLogicalProcessors, NrThreads, CurrentClockSpeed, MaximumClockSpeed)
2. BatteryData -> has no parameters and returns a complex object representing battery information obtained by querring the Microsoft Library, from Win32_Battery^17 (Name, Status, EstimatedChargeRemaining, FullChargeCapacity, DesignCapacity, MaxRechargeTime, DesignVoltage)
3. RamData -> has no parameters and returns a complex object representing RAM information obtained by querring the Microsoft Library, from Win32_PhysicalMemory^17 (Name, Speed, MinVoltage, MaxVoltage, Status, Capacity)
4. StorageData -> has no parameters and returns a complex object representing storage information obtained by querring the Microsoft Library, from Win32_DiskDrive^17 (Name, Model, BytesPerSector, Size, Status, TotalSectors, Partitions, Manufacturer)
5. GpuData -> has no parameters and returns a complex object representing gpu information obtained by querring the Microsoft Library, from Win32_VideoController^17 (Name, DriverVersion, LastErrorCode, MinRefreshRate, MaxRefreshRate, Status, VideoArchitecture, VideoMemoryType, VideoProcessor)

CPU Benchmark Design

The user will not interact directly with these algorithms, but the application will, so bellow it is described the use case diagram:



As seen from the use case, the class should provide only three public methods:

1. Constructor -> Basic Constructor
2. RunMIPSTests -> Run all the MIPS tests and average their MIPS results
3. RunSimpleOperationsTests -> Run all the simple operation tests and add their execution time

The two public custom methods should force the CPU to run only on one thread and to prioritize the benchmark execution. This should be done on all threads, in order to average the entire CPU speed and characteristics. The following code will be used:

```
foreach (ProcessThread pt in Process.GetCurrentProcess().Threads)
{
    pt.IdealProcessor = threadIndex;
    pt.ProcessorAffinity = (IntPtr)(1 << threadIndex);
}
```

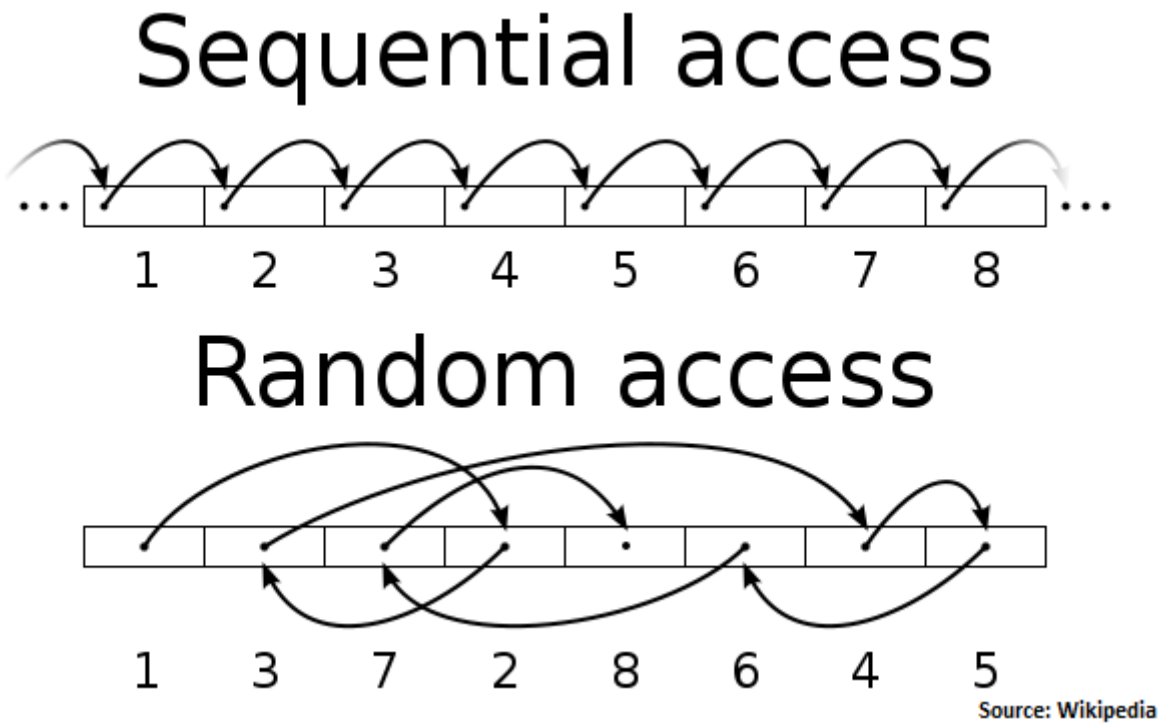
All the other private methods should be:

1. BenchmarkSHA256 -> Run a benchmark which runs a simple string to sha256 algorithm and measure its execution MIPS
2. BenchmarkPower -> Run a benchmark which runs the power function and measure its execution MIPS
3. BenchmarkForLoops -> Run a benchmark which runs a big loop and measure its execution MIPS
4. BenchmarkSimpleOperations -> Run a benchmark which runs a simple addition and substraction and measure its clock cycles

All the Benchmark named methods will run multiple times and average the results, in order to obtain a more consistent measurement.

RAM Benchmark Design

As seen from the UML, this benchmark will be composed from only one class, with two public methods:



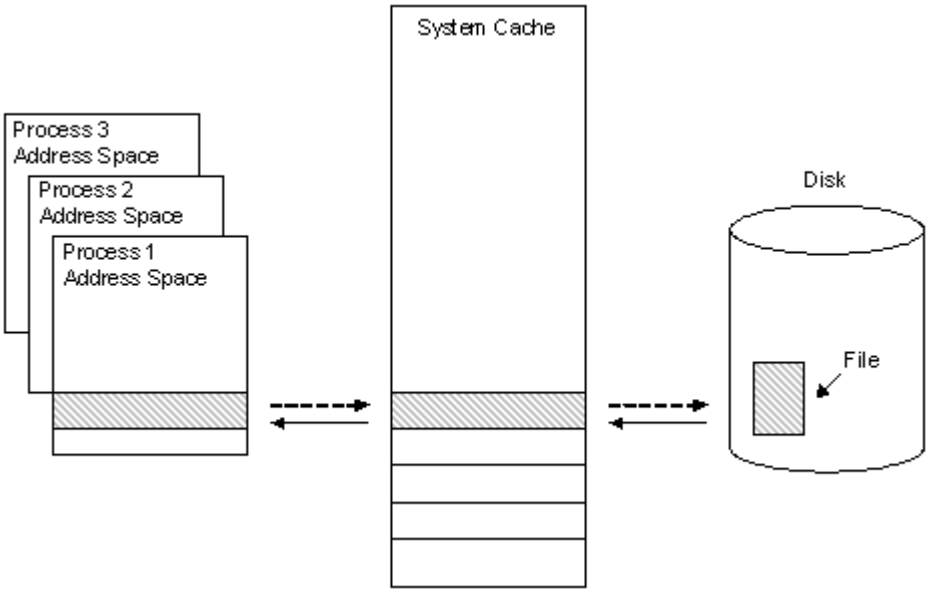
1. SequentialAccess with size of data type and number of repetitions as parameters -> Runs a benchmark for the sequential read/write capabilities of the RAM.
2. RandomAccess with size of data type and number of repetitions- as parameters> Runs a benchmark for the random rear/write capabilities of the RAM.

Random access in ram will be made with the help of the library `using System.Collections.Generic`, using lists in the form of the `List<uint>` data type. Sequential access in ram will be made using arrays in the form of the `uint[]` data type.

The test will firstly write `0x5555` equivalent integer to RAM for a number of times equivalent to the size parameter, then it will read from RAM and count the errors. The same algorithm will be applied for the `0xAAAA` equivalent integer on RAM and append to the error variable.

In order to bypass the CPU cache used for small variables, byte size wise, one needs to run the benchmark for bigger sizes of lists and arrays, using a lot of data to detect any diffect. INTEL manual recomands around 1024 * size of RAM of data ran for around one week.

Storage Benchmark Design



As seen from the UML, this benchmark will be composed from only one class, with only one public method:

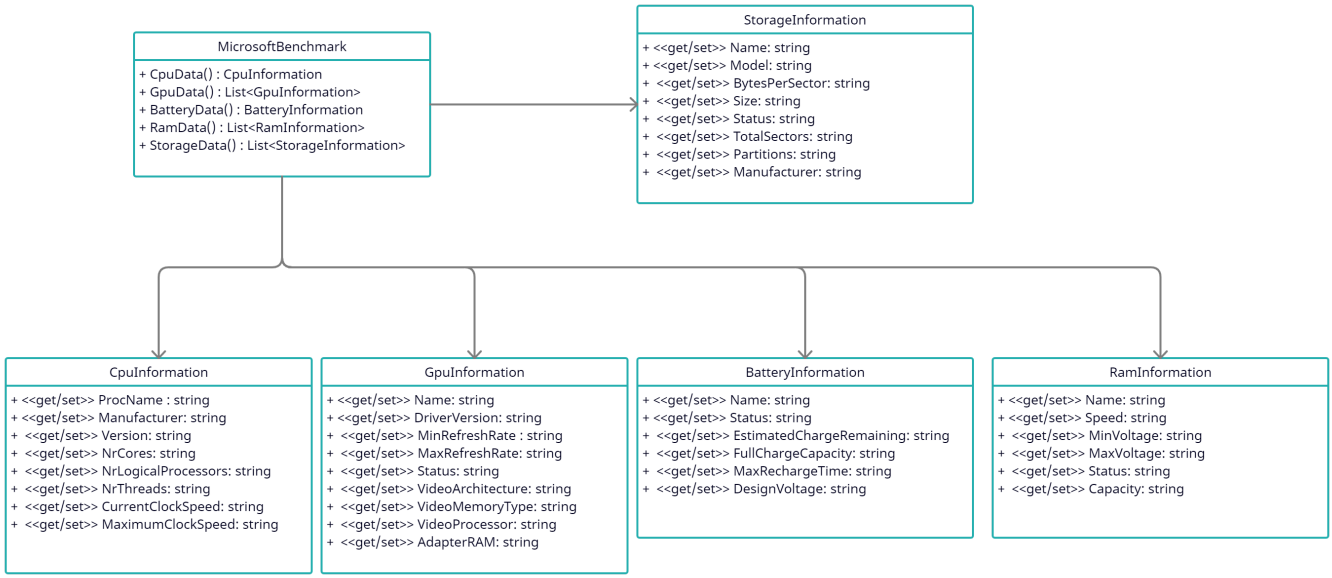
1. FileAccess with file path, size of file and number of repetitions as parameters-> Runs a benchmark for the write/read capabilities of the Storage solutions.

The user should be able to select from the UI the file path in which the test runs. The test will firstly write `0x5555` equivalent integer to the file for a number of lines equivalent to the size parameter, then it will read the file and count the errors. The same algorithm will be applied for the `0xAAAA` equivalent integer to the file and append to the error variable.

In order to bypass the Storage cache used for small variables, byte size wise, one needs to run the benchmark for bigger sizes of files, using a lot of data to detect any diffect.

Microsoft Defined Data Structures Integration Design

The structure of the Microsoft Defined Structures Integration will be explained using the following Class Diagram:



The usage of Lists is necessary in the case of:

- GpuInformation - because a system can have one or more Gpus (in the most cases, a integrated and dedicated one)
- RamInformation - because a system can have one or more Ram sticks inserted into the motherboard
- StorageInformation - because a system can have one or more storage solutions

For every information class, the same sequence of lines of codes will be respeceted:

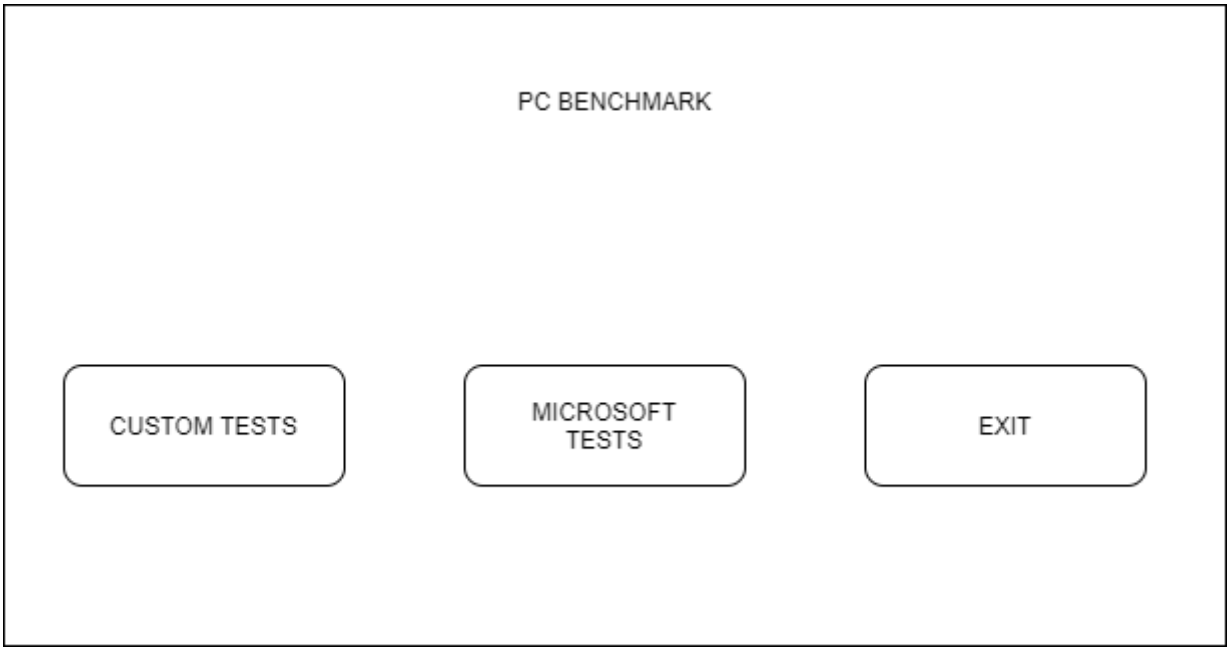
```
ManagementObjectCollection moc;
// Instantiate the information class required
try
{
    moc = new ManagementObjectSearcher("select * from Win32_[Category]").Get();
}
catch
{
    return ramInformationList;
}
foreach (ManagementObject obj in moc)
{
    // Get the information needed from the obj["What to get"].ToString();
}
return [Instantiated class with information];
```

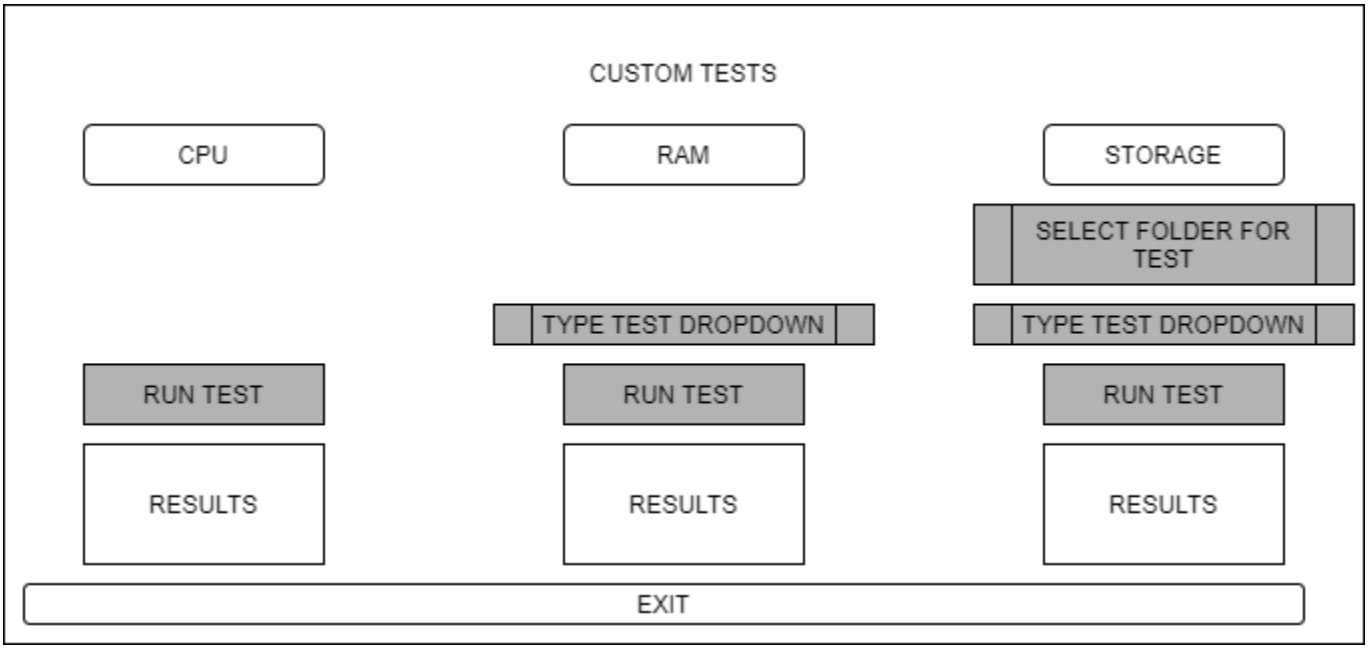
For the GpuInformation, the VideoMemoryType and VideoArchitecture will be encoded, so a decoder in the form of a switch will be required.

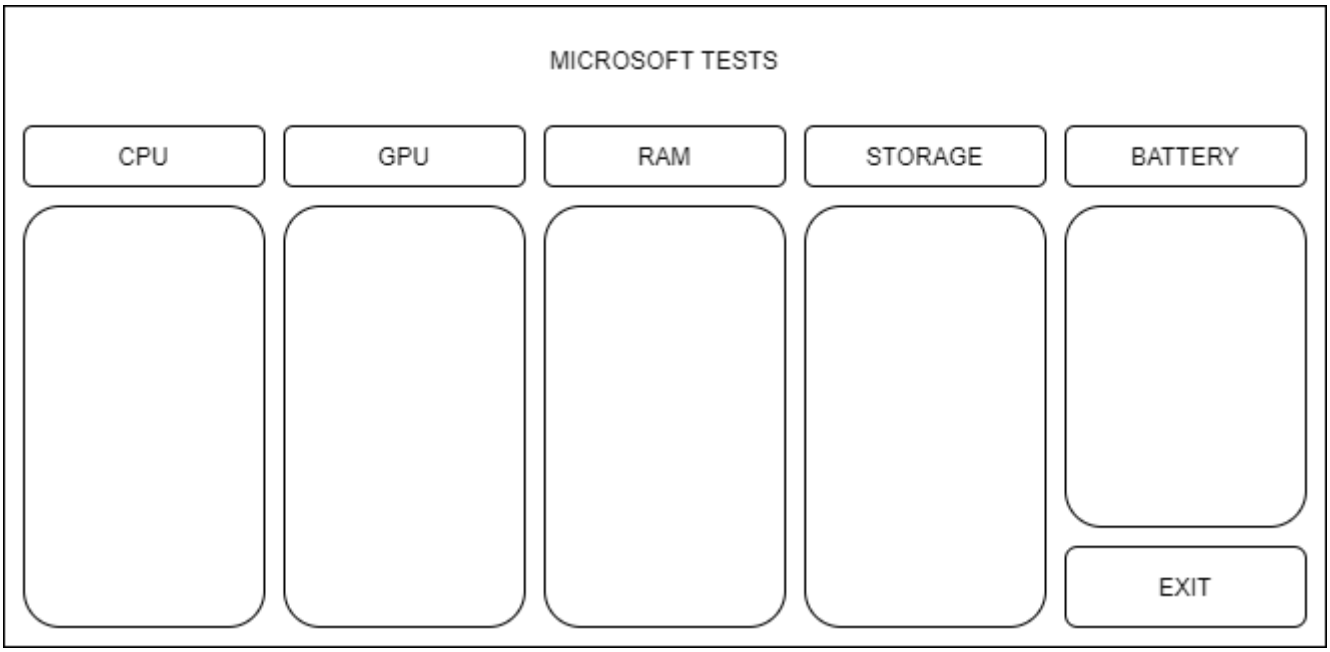
GUI Design

In order to use the application, a GUI (Graphical user interface) is needed. The graphical user interface is a form of user interface that allows users to interact with electronic devices through graphical icons and audio indicator such as primary notation, instead of text-based user interfaces, typed command labels or text navigation. GUIs were introduced in reaction to the perceived steep learning curve of command-line interfaces (CLIs), which require commands to be typed on a computer keyboard.

For this purpose, the following mockups were created:







In order to create and manage the GUI, we will use the Windows Forms classes and library. Windows Forms is a Graphical User Interface(GUI) class library which is bundled in .Net Framework. Its main purpose is to provide an easier interface to develop the applications for desktop, tablet, PCs. It is also termed as the WinForms.

The applications which are developed by using Windows Forms or WinForms are known as the Windows Forms Applications that runs on the desktop computer. WinForms can be used only to develop the Windows Forms Applications not web applications. WinForms applications can contain the different type of controls like labels, list boxes, tooltip etc.[^18]

Implementation

Local application

CPU Benchmark

The implementation needed some libraries that should be mentioned here:

1. System.Security.Cryptography -> Library that creates and outputs the SHA of a given raw data string
2. System.Environment -> Library that detects the environment and outputs the number of physical cores
3. System.Diagnostics.ProcessThread -> Library that enables the programmer to use the threads and process of the computer
4. System.Numerics -> Library that enables the use of large numbers for the MIPS calculation

The implementation of the public methods is the following:

RunMIPSTests

```
public float RunMIPSTests()
{
    int nrTests = 3;
    float averageCPU = 0;
    int processorCount = Environment.ProcessorCount;
    // For every physical processor
    for (int i=0;i< processorCount; i++)
    {
        // Enable the current processor and set the affinity to it
        foreach (ProcessThread pt in Process.GetCurrentProcess().Threads)
        {
            pt.IdealProcessor = i;
            pt.ProcessorAffinity = (IntPtr)(1 << i);
        }
        // measure the time passed for each of the available algorithms
        float currentCPUMIPS = BenchmarkSHA256(10000) + BenchmarkPower(10000) +
BenchmarkForLoops(1000);
        averageCPU += currentCPUMIPS;
    }
    averageCPU /= (processorCount * nrTests);
    return averageCPU;
}
```

As it can be observed, for every physical processor, the application sets the the CPU affinity to that processor and then it computes the MIPS, returning the actual value.

RunSimpleOperationsTests

```
public float RunSimpleOperationsTests()
{
    float averageCPU = 0;
    int processorCount = Environment.ProcessorCount;
    // For every physical processor
    int nrTests = 100;
    for (int i = 0; i < processorCount; i++)
    {
        // Enable the current processor and set the affinity to it
        foreach (ProcessThread pt in Process.GetCurrentProcess().Threads)
        {
            pt.IdealProcessor = i;
            pt.ProcessorAffinity = (IntPtr)(1 << i);
        }
        // measure the time passed for each of the available algorithms
        averageCPU += BenchmarkSimpleOperations(nrTests);
    }
    averageCPU /= (nrTests * nrTests * 2);
    averageCPU /= processorCount;
    return averageCPU;
}
```

As it can be observed, for every physical processor, the application sets the the CPU affinity to that processor and then it computes the clock cycles, returning the actual value in clock cycles.

As for the actual tests, the private methods have the following implementation:

BenchmarkSHA256

```
private float BenchmarkSHA256(int nrTests)
{
    Stopwatch swTotal = new Stopwatch();
    BigInteger averageTimeSha256 = new BigInteger(0);
    for(int i=0;i< nrTests; i++)
    {
        Stopwatch sw = new Stopwatch();
        string data = RandomString(256);
        sw.Start();
        swTotal.Start();
        string returnSha256 = ComputeSha256Hash(data);
        sw.Stop();
        swTotal.Stop();
        // add to the average the partial execution time
        averageTimeSha256 = BigInteger.Add(averageTimeSha256, new
BigInteger(sw.Elapsed.Ticks));
    }
    averageTimeSha256 = BigInteger.Divide(averageTimeSha256, new
BigInteger(nrTests));
    return ((float)((float)averageTimeSha256 /
```

```
(swTotal.Elapsed.TotalMilliseconds)));
}
```

This method has two clock, one for the actual number of cycles of execution of the algorithm, the other for the total miliseconds passed. It will compute the sha256 hash of a random string and count the clock cycles.

BenchmarkPower

```
private float BenchmarkPower(int nrTests)
{
    Stopwatch swTotal = new Stopwatch();
    BigInteger averageTimePower = new BigInteger(0);
    for (int i = 0; i < nrTests; i++)
    {
        Stopwatch sw = new Stopwatch();
        BigInteger result = new BigInteger(0);
        BigInteger a = new BigInteger(3);
        int b = 128;
        sw.Start();
        result = BigInteger.Pow(a, b);
        sw.Stop();
        // add to the average the partial execution time
        averageTimePower = BigInteger.Add(averageTimePower, new
BigInteger(sw.Elapsed.Ticks));
    }
}
```

This method has two clock, one for the actual number of cycles of execution of the algorithm, the other for the total miliseconds passed. It will compute the power 128 of 3 and count the clock cycles.

BenchmarkForLoops

```
private float BenchmarkForLoops(int nrTests)
{
    Stopwatch swTotal = new Stopwatch();
    BigInteger averageTimeForLoops = new BigInteger(0);
    for (int i = 0; i < nrTests; i++)
    {
        Stopwatch sw = new Stopwatch();
        sw.Start();
        bool a = true;
        for(int j=0;j< nrTests; j++)
        {
            for(int k=0;k< nrTests; k++)
            {
                a = !a;
            }
        }
```

```
        sw.Stop();
        // add to the average the partial execution time
        averageTimeForLoops = BigInteger.Add(averageTimeForLoops, new
BigInteger(sw.Elapsed.Ticks));
    }
}
```

This method has two clock, one for the actual number of cycles of execution of the algorithm, the other for the total miliseconds passed. It will loop in loop for nrTests times and inverse the variable a and count the clock cycles.

BenchmarkSimpleOperations

```
private int BenchmarkSimpleOperations(int nrTests)
{
    long averageSimpleOperations = 0;
    do
    {
        averageSimpleOperations = 0;
        for (int times = 0; times < nrTests; times++)
        {
            long a = 0;
            for (int i = 0; i < nrTests; i++)
            {
                Stopwatch sw = new Stopwatch();
                sw.Start();
                a = a + 1;
                sw.Stop();
                averageSimpleOperations += sw.ElapsedTicks;
            }
            for (int i = 0; i < nrTests; i++)
            {
                Stopwatch sw = new Stopwatch();
                sw.Start();
                a = a - 1;
                sw.Stop();
                averageSimpleOperations += sw.ElapsedTicks;
            }
        }

    } while (averageSimpleOperations <= (nrTests * nrTests * 2)); // to make sure
that a valid number is returned
}
```

This method has two clock, one for the actual number of cycles of execution of the algorithm, the other for the total miliseconds passed. It will count the add operation, then the substraction operation to return the number of clock cycles needed to perform such operation.

RAM Benchmark

RAM Benchmark consists of two tests, one sequential, one random. As it was explained in the design part. In the RAM will be written 0x5555 and 0xAAAA in order to test every bit of the bytes involved.

The two are implemented as followed:

```
[Type according to sequential or random] setOfNumbers = new type;
int errorCount = 0;

for (int repeat = 0; repeat < repeating; repeat++)
{
    setOfNumbers.Clear();
    for (int i = 0; i < size; i++)
    {
        setOfNumbers.Add(_firstValue);
    }

    for (int i = 0; i < size; i++)
    {
        if (setOfNumbers[i] != _firstValue)
        {
            errorCount++;
        }
    }

    setOfNumbers.Clear();
    for (int i = 0; i < size; i++)
    {
        setOfNumbers.Add(_secondValue);
    }

    for (int i = 0; i < size; i++)
    {
        if (setOfNumbers[i] != _secondValue)
        {
            errorCount++;
        }
    }
}
return 100 * ((float)errorCount / (float)size);
```

Storage Benchmark

For the Storage Benchmark, the user chooses a path to which the file will be written, the size of the file and the number of repetitions. In the file will be written 0x5555 and 0xAAAA in order to test every bit of the bytes involved.

The implementation is the following:

```
if(File.Exists(path))
{
    File.Delete(path);
}

for (int i = 0; i < size; i++)
{
    File.AppendAllText(path, _firstValue.ToString()+"\n");
}

foreach (string line in File.ReadLines(path))
{
    if(_firstValue.ToString() != line)
    {
        errorCount++;
    }
}
File.Delete(path);
for (int i = 0; i < size; i++)
{

    File.AppendAllText(path, _secondValue.ToString() + "\n");
}

foreach (string line in File.ReadLines(path))
{
    if (_secondValue.ToString() != line)
    {
        errorCount++;
    }
}
File.Delete(path);
}

return 100 * ((float)errorCount / (float)size);
```

Microsoft Defined Data Structures Integration

The implementation needed some libraries that should be mentioned here:

1. System.Management -> Library that is necessary for the call of the ManagementObjectSearcher method
2. System.Collections.Generic -> Library that contains the `List` data structure

Single instance of ManagementObject

In a system, there can be only one battery, so the following implementation was required:

```
public static BatteryInformation BatteryData()
{
    ManagementObjectCollection moc;
    BatteryInformation batteryInformation = new BatteryInformation();
    try
    {
        moc = new ManagementObjectSearcher(new ObjectQuery("select * from
Win32_Battery")).Get();
    }
    catch
    {
        return batteryInformation;
    }
    foreach (ManagementObject obj in moc)
    {
        if (obj["Name"] != null)  batteryInformation.Name =
obj["Name"].ToString();
        if (obj["Status"] != null)  batteryInformation.Status =
obj["Status"].ToString();
        if (obj["EstimatedChargeRemaining"] != null)
batteryInformation.EstimatedChargeRemaining =
obj["EstimatedChargeRemaining"].ToString();
        if (obj["FullChargeCapacity"] != null)
batteryInformation.FullChargeCapacity = obj["FullChargeCapacity"].ToString();
        if (obj["DesignCapacity"] != null)  batteryInformation.DesignCapacity =
obj["DesignCapacity"].ToString();
        if (obj["DesignVoltage"] != null)  batteryInformation.DesignVoltage =
obj["DesignVoltage"].ToString();
        if (obj["MaxRechargeTime"] != null)  batteryInformation.MaxRechargeTime =
obj["MaxRechargeTime"].ToString();
    }
    return batteryInformation;
}
```

Multiple instances of ManagementObject

In a system, there can be one or more storage units, so the following implementation was required

```
public static List<StorageInformation> StorageData()
{
    ManagementObjectCollection moc;
    List<StorageInformation> storageInformationList = new List<StorageInformation>
();
    try
    {
        moc = new ManagementObjectSearcher("select * from Win32_DiskDrive").Get();
    }
    catch
    {
        return storageInformationList;
    }
    foreach (ManagementObject obj in moc)
    {
        StorageInformation storageInformation = new StorageInformation();

        if (obj["Name"] != null) storageInformation.Name = obj["Name"].ToString();
        if (obj["Model"] != null) storageInformation.Model =
obj["Model"].ToString();
        if (obj["BytesPerSector"] != null) storageInformation.BytesPerSector =
obj["BytesPerSector"].ToString();
        if (obj["Size"] != null) storageInformation.Size = obj["Size"].ToString();
        if (obj["Status"] != null) storageInformation.Status =
obj["Status"].ToString();
        if (obj["TotalSectors"] != null) storageInformation.TotalSectors =
obj["TotalSectors"].ToString();
        if (obj["Partitions"] != null) storageInformation.TotalSectors =
obj["Partitions"].ToString();
        if (obj["Manufacturer"] != null) storageInformation.TotalSectors =
obj["Manufacturer"].ToString();

        storageInformationList.Add(storageInformation);
    }
    return storageInformationList;
}
```

Decoding video card special properties

For the VideoMemoryType, the following method was required to transform the codes of the memory types into useful string data:

```
private static string GetVideoMemoryType(ManagementObject obj)
{
    string videoMemoryType = obj["VideoMemoryType"].ToString();
    switch (videoMemoryType)
    {
        case "1":
            return "Other";
        case "2":
            return "Unknown";
        case "3":
            return "VRAM";
        case "4":
            return "DRAM";
        case "5":
            return "SRAM";
        case "6":
            return "WRAM";
        case "7":
            return "EDO RAM";
        case "8":
            return "Burst Synchronous DRAM";
        case "9":
            return "Pipelined Burst SRAM";
        case "10":
            return "CDRAM";
        case "11":
            return "3DRAM";
        case "12":
            return "SDRAM";
        case "13":
            return "SGRAM";
        default:
            return "Error";
    }
}
```

For the VideoArchitecture, the following method was required to transform the codes of the Architecture into useful string data:

```
private static string GetVideoArchitecture(ManagementObject obj)
{
    string videoArchitecture = obj["VideoArchitecture"].ToString();
    switch (videoArchitecture)
    {
```

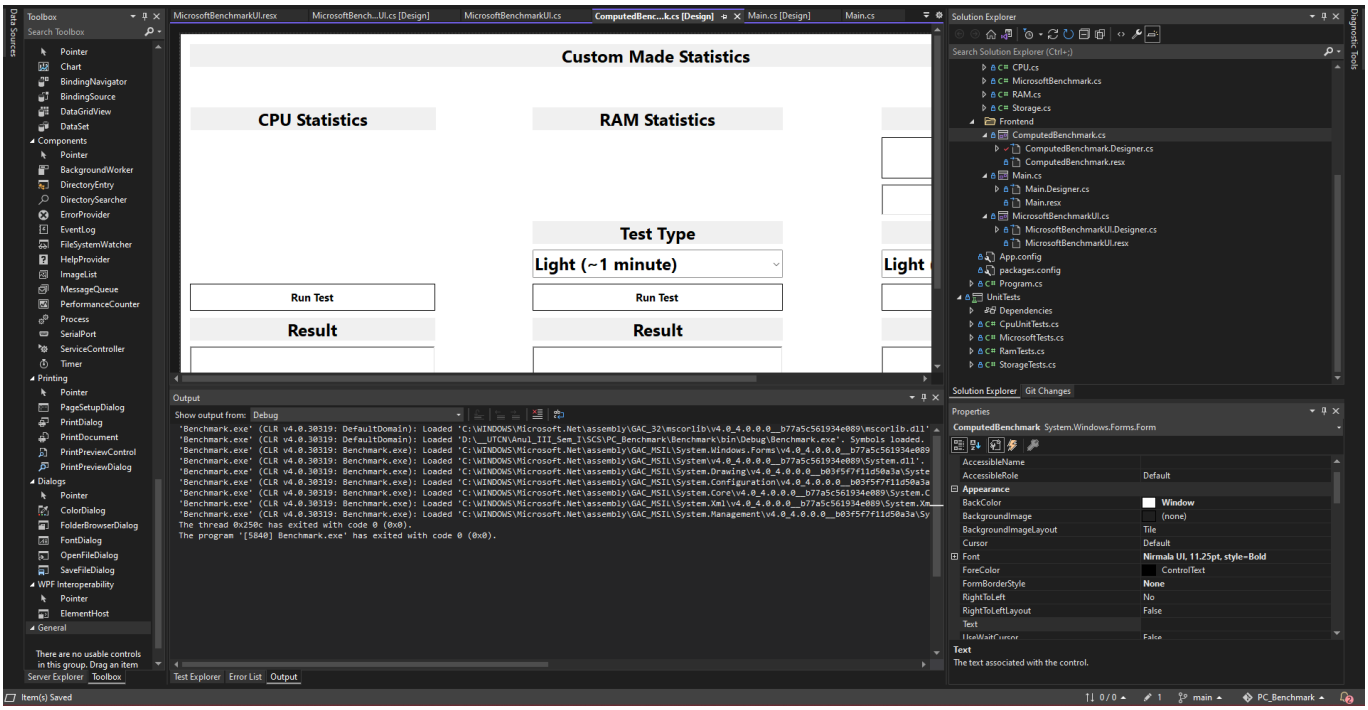
```
            case "1":
                return "Other";
            case "2":
                return "Unknown";
            case "3":
                return "CGA";
            case "4":
                return "EGA";
            case "5":
                return "VGA";
            case "6":
                return "SVGA";
            case "7":
                return "MDA";
            case "8":
                return "HGC";
            case "9":
                return "MCGA";
            case "10":
                return "8514A";
            case "11":
                return "XGA";
            case "12":
                return "Linear Frame Buffer";
            case "160":
                return "PC-98";
            default:
                return "Error";
        }
    }
```

GUI

Actual implementation of the GUI requires two steps:

The Form.Designer.cs

This file which is modified using the ToolBox feature available in the Visual Studio C# template. This feature is simply Drag&Drop methodology with complex objects that have a lot of useful accessible public properties, like the name of the object, font, size, is ReadOnly etc. Such a window looks something like:



The Form.cs

This is the file where all the useful code is written. This part is specific for each form and has the Constructor and the auxiliary methods that are bound to graphic elements

Main Menu UI

Here, the buttons 1 and 2 (Microsoft Benchmark and Custom Benchmark buttons) create a new instance of the MicrosoftBenchmarkUI, respectively ComputedBenchmark classes, whereas button 3 exits the program.

```
private void button1_Click(object sender, EventArgs e)
{
    MicrosoftBenchmarkUI microsoftBenchmarkUI = new MicrosoftBenchmarkUI();
    microsoftBenchmarkUI.ShowDialog();
}

private void button2_Click(object sender, EventArgs e)
{
    ComputedBenchmark computedBenchmark = new ComputedBenchmark();
    computedBenchmark.ShowDialog();
}
```

```
private void button3_Click(object sender, EventArgs e)
{
    this.Close();
}
```

MicrosoftBenchmarkUI

In order to save computing time, the application creates a new thread to fetch the microsoft information for each of the desired components (CPU, GPU, RAM, Storage, Battery), and then it prints said information in richTextBoxes. These are locations where the formated text should be written.

```
Parallel.Invoke(
    () => {
        cpuData = Backend.MicrosoftBenchmark.CpuData();
        cpuTextBox.Text = cpuData.ToString();
    },
    () => {
        gpuData = Backend.MicrosoftBenchmark.GpuData();
        gpuTextBox.Text = "";
        gpuData.ForEach(gpu => gpuTextBox.Text += gpu.ToString() + "\n");
    },
    () => {
        ramData = Backend.MicrosoftBenchmark.RamData();
        ramTextBox.Text = "";
        ramData.ForEach(ram => ramTextBox.Text += ram.ToString() + "\n");
    },
    () => {
        storageData = Backend.MicrosoftBenchmark.StorageData();
        storageTextBox.Text = "";
        storageData.ForEach(storageSolution => storageTextBox.Text +=
storageSolution.ToString() + "\n");
    },
    () => {
        batteryData = Backend.MicrosoftBenchmark.BatteryData();
        batteryTextBox.Text = batteryData.ToString();
    }
);
```

ComputedBenchmark

This form is more complicated than the rest, because it has dropdown elements and select path element. These are important, because they enable the user to select the preffered kind of test (Light, Medium, Stress or Extreme).

Path Selection

This element uses the `FolderBrowserDialog` class, in order to open a dialog window, select the desired path of the storage test and then store this information in a string variable:

```
using (var fbd = new FolderBrowserDialog())
{
    DialogResult result = fbd.ShowDialog();

    if (result == DialogResult.OK && !string.IsNullOrWhiteSpace(fbd.SelectedPath))
    {
        selectionFileTextBox.Text = "Folder Selected:\n"+fbd.SelectedPath;
        pathSelected = fbd.SelectedPath;

        if (fbd.SelectedPath.Contains("C:\\"))
        {
            MessageBox.Show("The Benchmark will fail if not started in admin
mode", "Message");
        }
    }
}
```

DropDown

This element is activate upon index change of the respective comboBox element and it was implemented. These methods were crucial in the user choice parsing procedure and it was implemented (for example is the storage ComboBox) as follows:

```
private void storageComboBox_SelectedIndexChanged(object sender, EventArgs e)
{
    ComboBox cmb = (ComboBox)sender;
    string selectedValue = cmb.SelectedItem.ToString();
    selectedValue = selectedValue.Substring(0, selectedValue.IndexOf(" "));
    storageChoice = selectedValue;
}
```

Actual `Start Test` Button

These methods gather all the information from the GUI that is necessary for the required test and then block all the user interaction with the GUI and run said tests.

1. For the CPU it is run the `CPU.RunMIPSTests()` and `CPU.RunSimpleOperationsTests()`
2. For the RAM, according to the test selected, it is written/read from memory 8Mb with the following repetition behaviour (16 repetitions, 32*16 repetitions,128*16 repetitions,1024*16 repetitions)
3. For the Storage, according to the test selected, it is written/read from the selected path 500Mb with the following repetition behaviour (1 repetitions, 32 repetitions,128 repetitions,1024 repetitions)

A choice type method has the following implementation, returning a string to the richTextBox:

```
switch (choice)
{
    case "Light":
        returnValue = Test();
        break;
    case "Medium":
        returnValue = Test();
        break;
    case "Stress":
        returnValue = Test();
        break;
    case "Extreme":
        returnValue = Test();
        break;
}
return "The health of the [component] is at" + returnValue.ToString();
```

Testing and validation

CPU Benchmark

In order to test the efficiency of the algorithm, a batch of unit tests were created. They test the results of the two public methods from the CPU class in the following manner:

```
[Test]
public void OneMIPSTest()
{

    float cpuMIPSValue = cpu.RunMIPSTests();
    Assert.IsTrue(cpuMIPSValue > 3); // This value was chosen looking at the Task
Manager of the developing machine
}
[Test]
public void OneSimpleOperationsTest()
{
    float cpuSimpleOperationsValue = cpu.RunSimpleOperationsTests();
    Assert.IsTrue(cpuSimpleOperationsValue < 5);
}
```

RAM

In order to test the efficiency of the algorithm, a batch of unit tests were created. They test the results of the two public methods from the RAM class in the following manner:

```
[Test]
public void RAMTest()
{
    float sequentialRAM = ram.SequentialAccess(size, repetitions);
    float randomRAM = ram.RandomAccess(size, repetitions);

    Assert.IsTrue(sequentialRAM == 0);
    Assert.IsTrue(randomRAM == 0);
}
```

Storage

In order to test the efficiency of the algorithm, a batch of unit tests were created. They test the result of the only public methods from the Storage class in the following manner:

```
[Test]
public void SystemDriveTest()
{
    float storageTest = storage.FileAccess(@"C:\\Program Files\\BenchmarkPC\\",
size, repetitions);

    Assert.IsTrue(storageTest == 0);
}

[Test]
public void OtherDriveTest()
{
    float storageTest = storage.FileAccess(@"D:\\", size, repetitions);

    Assert.IsTrue(storageTest == 0);

    storageTest = storage.FileAccess(@"E:\\", size, repetitions);

    Assert.IsTrue(storageTest == 0);
}
```

Microsoft Defined Data Structures Integration

In order to test the usefulness of the Microsoft library, a batch of unit tests were created. They test the presence of certain elements that are specific to the test machine, as well as, test the good implementation of the MicrosoftBenchmark class.

The tests respect the following model, where `[Component]` is the component that needs to be tested (i.e. Cpu):

```
[Test]
public void [Component]DataTest()
{
    [ComponentClass] [Component]Information =
Benchmark.Backend.MicrosoftBenchmark.[Component]Data();
    Assert.True([Component]Information != null);
    // custom tests tailored for each component
}
```

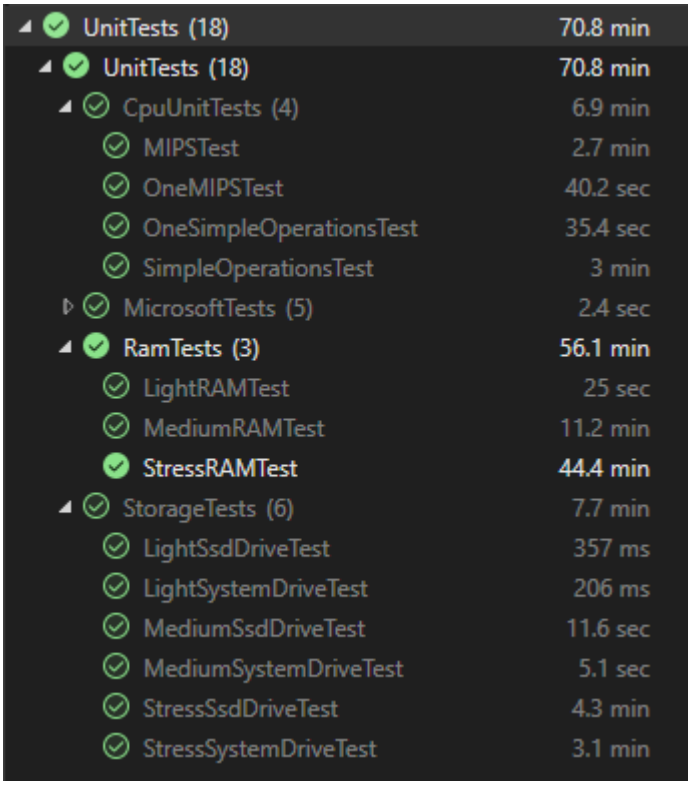
A specific example for this behavior is the `CpuDataTest()`which has the following implementation:

```
[Test]
public void CpuDataTest()
{
    CpuInformation cpuInformation;
    cpuInformation = Benchmark.Backend.MicrosoftBenchmark.CpuData();

    Assert.True(cpuInformation != null);
    Assert.True(cpuInformation.Manufacturer.ContainsLower("Intel"));
    Assert.True(cpuInformation.NrCores.EqualsLower("4"));
    Assert.True(cpuInformation.NrLogicalProcessors.EqualsLower("8"));
}
```

Whole Test Unit package

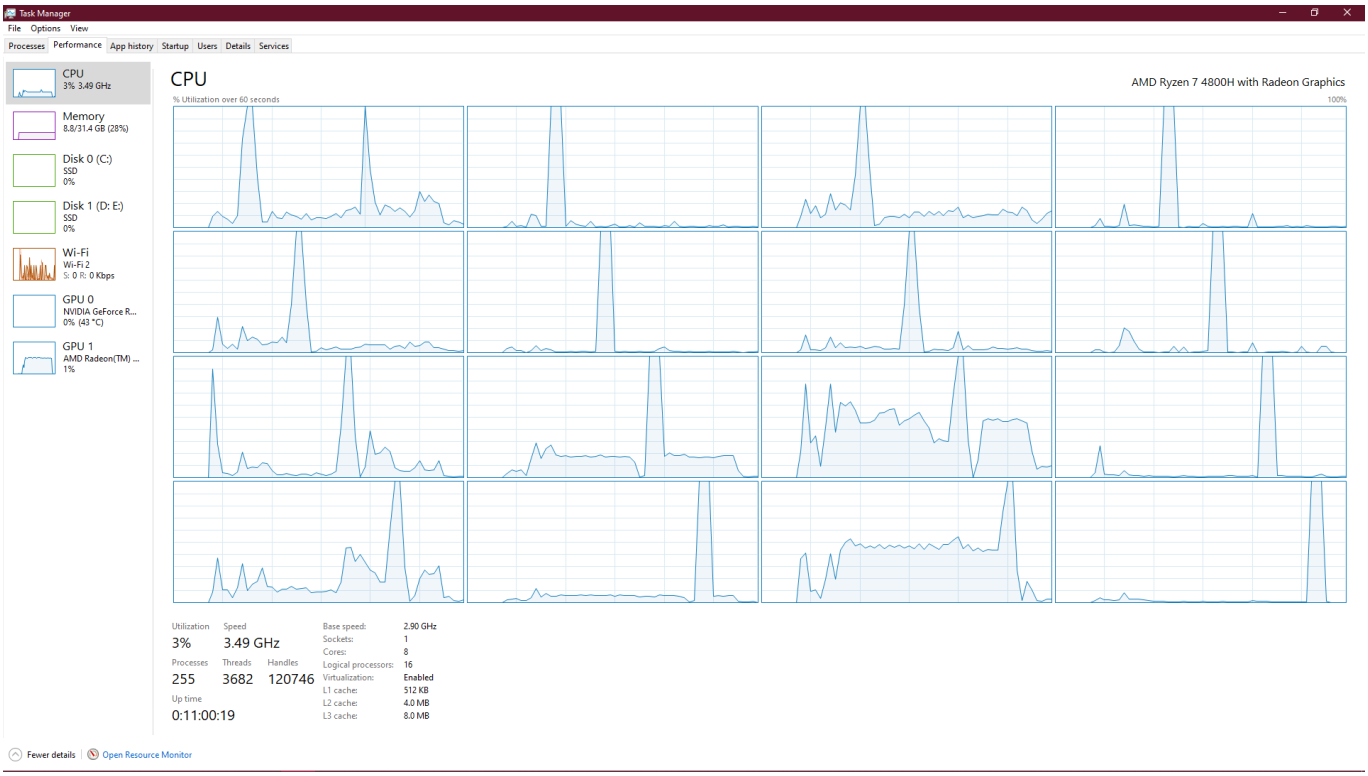
As can be seen from the Test Explorer provided by VisualStudio, all the tests pass:



Conclusions

CPU Benchmark

Looking at the Task Manager on the machine, it can be seen that when it runs on a specific CPU processor, there is a 100% spike in its activity.



RAM

Contrary to expectations, running the health analysis on RAM did not output any bit-wise error. The tests were made using the following data volume:

- 1Gb * sizeof(int) - taking 25 seconds to complete
- 1Gb * sizeof(int) * 32 - taking 11.2 minutes to complete
- 1Gb * sizeof(int) * 128 - taking 44.4 minutes to complete

That is around 512Gb written and read from RAM, which would yield minimum one error. This behaviour correlates to the power of the bit&byte correction available at the hardware and sotfware level, corrections that cannot be bypassed with modern programming languages.

Storage

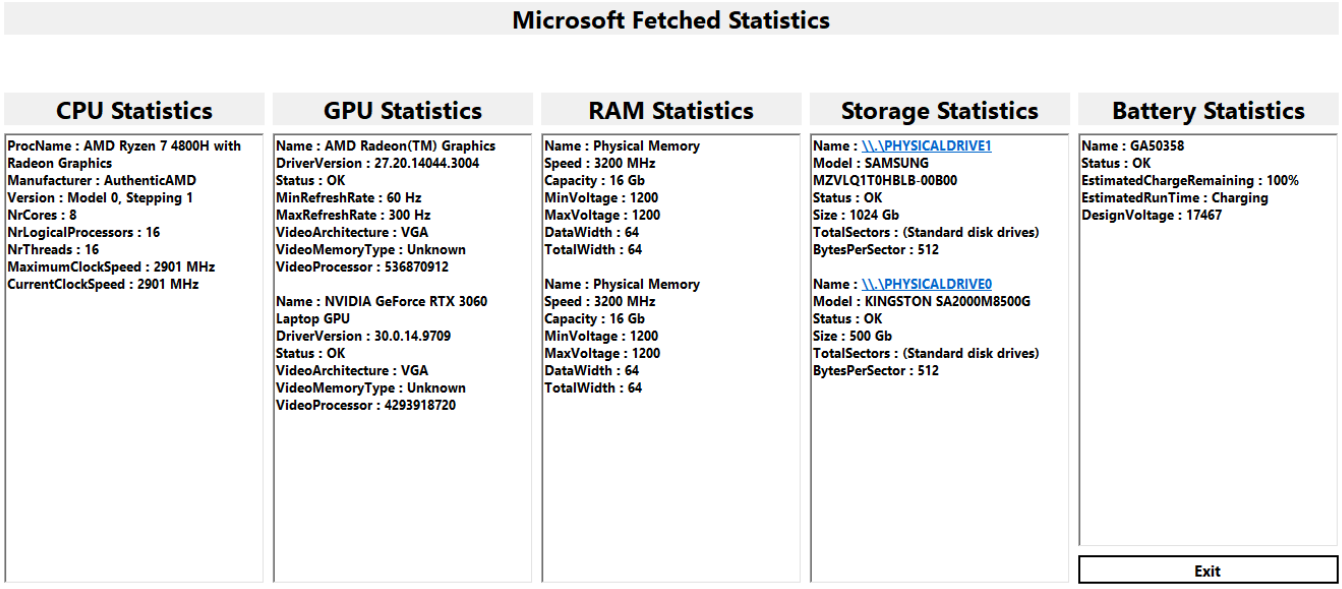
Contrary to expectations, running the health analysis on RAM did not output any bit-wise error. The tests were made using the following data volume:

- 8Kb * sizeof(int) - taking 0.5 seconds to complete
- 8Kb * sizeof(int) * 32 - taking 17 seconds to complete
- 8Kb * sizeof(int) * 1024 - taking 8 minutes to complete

That is around 32Gb written and read from the whole system storage, which would yield minimum one error. This behaviour correlates to the power of the bit&byte correction available at the hardware and sotfware level, corrections that cannot be bypassed with modern programming languages.

Microsoft Defined Data Structures Integration

The calls to the Computer System Hardware Classes provide the user with a wide range of available information that can be used. In this application, information about the CPU, RAM, Storage, Battery and GPU were fetched and used with success. As can be seen from a screenshot from the application, the results are:



These results were validated by the CPU-Z application, as well as, the Steam Hardware & Software Survey application.

Bibliography

1. Intel Architecture information
2. CPUID type
3. CPUID wikipage
4. Stackoverflow question with a lot of useful links about CPU
5. RAM information
6. Memory Testing
7. On-the-fly RAM tests
8. Storage Benchmark
9. Storage cache problem
10. Software as a service
11. ASP.NET
12. Microsoft battery Information
13. Microsoft CPU Information
14. Microsoft RAM Information
15. General Microsoft Information
16. Sha 256
17. Microsoft Computer System Hardware Classes
18. Hard Disk Information
19. Windows Forms