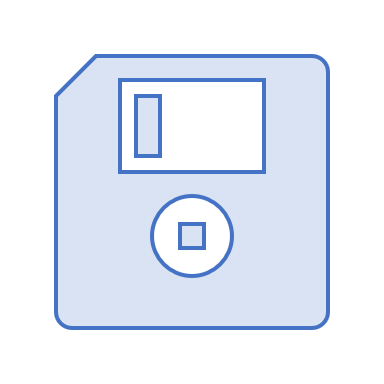
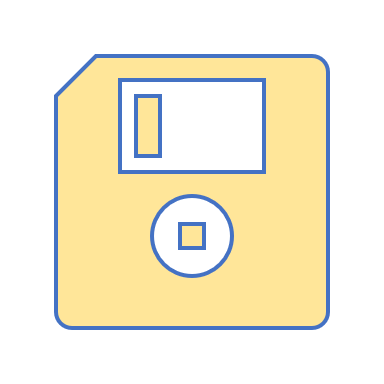
Windows Backup

March 2018



https://github.com/louisyang2015/windows\_backup

# Overview

## References

**NuGet**

|  |  |
| --- | --- |
| AWSSDK.S3 | AWS S3 |
| Google.Cloud.Storage.V1 | Google Cloud Storage |
| WindowsAzure.Storage | Microsoft Azure Storage |

**.Net**

using System.Windows.Forms; // NotifyIcon

using System.Drawing; // Icon

## Projects

**WindowsBackup** - GUI application

**WindowsBackup\_Console** - Testing

## Source Files

**WindowsBackup\src\**

|  |  |
| --- | --- |
| CloudBackupService.cs | Uploading and downloading from AWS, GCP, and Azure storage services. |
| EncryptStream.cs | Encryption and decryption implemented as a stream. |
| FileNameRegistration.cs | Generate and manage alternative file names for files. Using an alternative name, instead of a file's true name, hides the purpose of a file. |
| BackupManager.cs | Classes for watching specific folders for changes and doing backups. |
| RestoreManager.cs | Classes to support restoration of encrypted backup files. |
| KeyManager.cs | A basic key manager that loads keys from XML. |

**WindowsBackup\_Console**

|  |  |
| --- | --- |
| Test.cs | The tests are NOT fully automatic. |

## Classes

**CloudBackupService.cs**

This file contain classes for uploading to cloud storage providers, including AWS (Amazon), Azure (Microsoft), and GCP (Google).

interface CloudBackupService

List<string> list\_objects(string bucket\_name);  
void upload(Stream stream, string bucket\_name, string key\_name);  
void download(string bucket\_name, string key\_name, Stream stream,   
 long start, long length);  
void delete(string bucket\_name, List<string> key\_names);

Specific implementations:

class AWS\_CloudBackupService : CloudBackupService  
class GCP\_CloudBackupService : CloudBackupService  
class AzureBlob\_CloudBackupService : CloudBackupService

Exactly what AWS, GCP, and Azure does with the "stream" object during "upload" and "download" is determined by experimentation.

I created a basic Stream object to see how the cloud services utilize a stream during the upload and download function calls:

class TestStream : Stream

**EncryptStream.cs**

This file contain classes that does encryption and decryption.

File

class EncryptStream   
: Stream

AWS  
(encrypted file)

unencrypted

File

class DecryptStream   
: Stream

unencrypted

Encryption and decryption is done using System.Security.Cryptography;

Microsoft provides the CryptoStream class, but it does not buffer data.

Data is buffered by

class MyMemoryStream   
: Stream

CryptoStream  
(.Net)

The compression is done using System.IO.Compression;

Again, the Microsoft library class doing the work, GZipStream, does not buffer **output** data. Data is buffered by MyMemoryStream. (Note that GZipStream does buffer ***input*** data, so to learn patterns for its compression algorithm).

Before the data is compressed for real, there's a test run, to determine the post compression size. This test run is done by CountingStream.

class CountingStream   
: Stream

GZipStream  
(.Net)

+1 for every byte written.  
No data is actually stored.

**FileNameRegistration.cs**

When using encryption, this project hides the true file name inside an encrypted file. For example, a file, such as "test.txt", is encrypted and stored under an alternative file name, such as "a123.bin".

This file contains classes that generate and manage alternative file names for files.

class FileNameRegistration

alternative file name

true file path

modified time

The FileNameRegistration class provides an alternative file name for storage on the cloud.

The FileNameRegistration class also returns the file modified time.

**BackupManager.cs**

This file contains classes that utilizes the previously mentioned classes to do backups.

The **BackupRuleList** class defines rules for the backup process. Only files that pass these rules are actually backed up.

var backup\_rule\_list = new BackupRuleList(backup\_rule\_list\_xml);  
if(backup\_rule\_list.accepts(@"E:\temp\temp\whatever.txt"))  
 ...

The **Backup** class watches a single directory for changes. Any change then triggers the "void live\_backup()" function.

The Backup class is actually an abstract class. Exactly how the "live\_backup()" gets carried out is specialized in the "**DiskBackup**" and "**EncryptedBackup**" classes.

The "**BackupManager**" contains lists of "Backup" and "CloudBackupService" objects, and make them easier to use. One major goal is to put the backup and restore operations on a separate thread, so the application GUI can use "BackupManager" directly.

**RestoreManager.cs**

The code supports restoration from encrypted backups. Each encrypted backup destination, which can be a directory on disk, or a bucket in the cloud, appears as a **Restore** object. The **RestoreManager** is a container around a list of "Restore" objects.

The "RestoreManager" can return information about "Restore" objects using:

RestoreManager :: **RestoreInfo** get\_info(...)

Restoration is done by:

RestoreManager :: void restore(**RestoreSettings** settings)

Both "get\_info(...)" and "restore(...)" are time consuming tasks. So in practice, these two functions are not called directly. Instead they are triggered via:

BackupManager :: void get\_restore\_info(...)  
BackupManager :: void restore(...)

**KeyManager.cs**

This software supports different encryption keys for the same disk directory or cloud bucket.

Each encrypted file has a 2 byte "key\_hint" field. A **KeyManager** is something that maps a 2 byte number to a unique encryption key.

The **BasicKeyManager** loads encryption keys from the configuration XML.

# Encryption

## AES Behavior

**Length of AES Encryption**

|  |  |
| --- | --- |
| Clear Input Length | Encrypted Output Length |
| 96 | 112 |
| 97~111 | 112 |
| 112 | 128 |

When the clear input is divisible by 16, the encrypted output is 16 bytes longer (padding).

All other input is rounded up to a higher value that is divisible by 16.

Note that the last encrypted 32 bytes can be decrypted to 16 through 31 bytes.

|  |  |
| --- | --- |
| Clear Input Length | Encrypted Output Length |
| 16~31 | 32 |

**From the decryption point of view, it's necessary to know the length of the decrypted stream by the time one reach those last 32 bytes.**

If the length information is encrypted, then the whole thing should be at least 48 bytes, with the length stored in the first 16 bytes.

**Issues with AesCryptoServiceProvider** **Examples**

The Microsoft Doc examples for "AesCryptoServiceProvider" uses "MemoryStream", which has two issues when applied to this project.

The first issue is that the MemoryStream internal buffer grows without bound. This project uploads files, so it's important to stream that upload. When uploading a 2GB file, it's bad to actually allocate 2GB of memory.

The second issue is that the CryptoStream writes the final bytes only when it's Disposed. However, this causes the underlying MemoryStream to be disposed as well, and renders it unusable.

**MyMemoryStream**

In my project the CryptoStream is used with MyMemoryStream, which has been modified to address the two issues listed above.

1. limited buffer size - the MyMemoryStream internal buffer is static and will not grow.
   1. In the void advance\_position(int increment), if all data from the stream has been read, the internal pointers will be reset.
   2. In the void repack(), if the buffer is out of data, the function will move all unread data to the start of the internal buffer.
   3. In the override void WriteByte(byte value), when the internal buffer is out of memory, an Exception will be thrown.
2. continue usage after CryptoStream disposed
   1. The override void Flush() does not do anything.
   2. The Dispose(bool) is not overridden, so calling Dispose on this class does not make it unusable.

## Cloud Storage Providers Stream Requirements

It's important to do some testing, so to know how the different cloud storage providers treat the upload and download streams.

This is done by passing the TestStream class into the CloudBackupService upload(...) and download(...) functions.

CloudBackupService backup\_service = new AWS\_CloudBackupService(...);

backup\_service.upload(new TestStream(), bucket\_name, "test.bin");  
backup\_service.download(bucket\_name, "test.bin", new TestStream());

The findings are summarized below.

**AWS**

AWS upload is different from the other use cases:

The AWS upload process will ask for "Length" 4 times and "Position" 3 times in the beginning.

Length = 102401 has been returned.  
Position = 0 has been returned.  
Length = 102401 has been returned.  
Position = 0 has been returned.  
Length = 102401 has been returned.  
Length = 102401 has been returned.  
Position = 0 has been returned.  
8192 bytes returned in response to Read(...)  
8192 bytes returned in response to Read(...)

...

8192 bytes returned in response to Read(...)  
4097 bytes returned in response to Read(...)  
0 bytes returned in response to Read(...)

Uploads occur in 8kB chunks.

After the very beginning, AWS does not ask for length and position again.

Note that despite knowing the "Length" of the upload, the AWS library will still keep asking for bytes until a 0 is returned. The AWS library will actually enforce the "Length" number of bytes reported earlier.

At the end of the upload, .Dispose(True) is called on the stream.

AWS download uses the following write sizes:

(16363, 671) happens once

(16363, 1045) happens a bunch of times

A final write depending on the size of the download.

The 16363 is just below the 16 kB line.

**Azure**

Upon uploading, the Microsoft library will read data in 64kB chunks.

When downloading 1 MB + 1 byte of data, the Microsoft library uses the following write sizes:

(15880, 1) happens once.

A bunch of (16383, 1) writes - these two writes total 16kB.

Yes that's a lot of writes with a size of 1 byte, trying to get people to implement WriteByte(...)??

A final write of 504 bytes.

**Google**

Upon uploading, the Google library will read data in 4kB chunks.

Upon downloading, the Google library will write data in huge (10 MB) chunks. I had to upload a 30 MB file to determine this.

**.Dispose() and .Flush() behavior**

None of the three will flush the stream in the event of a download. Therefore the code inside CloudBackupService download(...) needs to do a manual flush.

Amazon's upload function will call .Dispose().

## Header

Before the encrypted file itself there is a header section.

Encryption is used to hide both the file content and the file name, but the other information is not encrypted.

|  |  |  |  |
| --- | --- | --- | --- |
| Byte # | Field Name | Field Length | Description |
| 0 | Version | 1 | Protocol version. |
| 1 | Flags | 1 | Bit[0] - the LSB - the compression bit. A '1' means compression is active. |
| 2 | Initialization Vector | 16 | Initialization vector for use with the AES algorithm. |
| 18 | Relative Path Length | 2 | Length of the relative path in bytes. There is no terminal zero character. |
| 20 | File Size | 8 | The size of the file right before encryption.  If compression is used, this size is the post compression size. |
| 28 | File Modified Time | 8 | This is the DateTime data type stored as "ticks". |
| 36 | Key Hint | 2 | A memory aid, to automatically determine what key to use to decrypt the file.  0 = No knowledge of what key to use. In this case the user of the software must know which key to use. |
| The following bytes are encrypted with AES. | | | |
| 38 | Relative Path | R | Relative path of the file. This is a string encoded in UTF-8.  The "R" is the length of the relative path in bytes, without the terminal zero. |
| End of the header. The following is the file content. | | | |
| 38+R | File | ? |  |

**Compression Trial Run**

As stated in the "AES Behavior" section, in order to successfully decrypt data, it is necessary to know the length of the unencrypted data. Asking the Microsoft library for too many bytes, or too few bytes, will result in a "Padding is invalid" error, either during the decryption call, or a delayed error when the CryptoStream is disposed.

Compression happens before encryption, so if compression is used, the "length" information that is needed is the post compression length.

So before the real compression using GZipStream, there is a trial run, where the compressed data is written into

class CountingStream : Stream

This Stream class does not record the bytes written into it. It only counts how many bytes are written into it.

## EncryptStream

**Internal Streams Diagram**

AWS upload will call EncryptStream .Read(...)

MyMemoryStream   
mem\_stream2

CryptoStream  
crypto\_stream

.Write(...)

.Read(...)

.Read(...)

FileStream file\_stream

GZipStream zip\_stream

MyMemoryStream   
mem\_stream1

.Read(...)

.Write(...)

The blue arrows represent "automatic" data transfers. For example, writing to zip\_stream will eventually result in data put into mem\_stream1. The code doing this transfer in inside the GZipStream, which is from the .Net library.

**The orange arrows represent data transfers done by code.** For example, there's actually code that will read from file\_stream and write into zip\_stream.

**Read call tree**

int Read(byte[] buffer, int offset, int count)  
 |🡪 int fill\_mem\_stream2\_from\_file(int max\_count)  
 |🡪 void transition\_to\_flushed\_stage()

## DecryptStream

**Internal Streams Diagram**

AWS download will call DecryptStream .Write(...)

MyMemoryStream   
mem\_stream1

CryptoStream  
crypto\_stream

.Read(...)

.Write(...)

.Write(...)

FileStream file\_stream

GZipStream zip\_stream

MyMemoryStream   
mem\_stream2

.Write(...)

.Read(...)

As before, the blue arrows represent "automatic" data transfers, while the orange arrows represent data transfers done by code.

**Write call tree**

void Write(byte[] buffer, int offset, int count)  
 |🡪 void process\_data()  
 |🡪 void process\_unenrypted\_header\_bytes()  
 |🡪 void extract\_relative\_path()  
 | |🡪 void decrypt\_to\_mem\_stream2(int bytes\_to\_decrypt, ...)  
 | |🡪 string full\_path\_request(relative\_path, ...)  
 |  
 |🡪 void decrypt\_to\_mem\_stream2(int bytes\_to\_decrypt, ...)  
 |🡪 void copy\_zip\_stream\_to\_file\_stream(bool read\_all\_data = false)  
 |🡪 void copy\_mem\_stream2\_to\_file\_stream()  
  
void Flush()  
 |🡪 void process\_data()  
 |🡪 void process\_data\_final()  
 |🡪 void copy\_zip\_stream\_to\_file\_stream(bool read\_all\_data = false)

# File Name Registration

## File Format

The file is tab delimited text, with one file per line.

**Fields**

|  |  |
| --- | --- |
| file full path | Ex: "E:\temp\temp\text.txt" |
| file prefix | Ex: "a" |
| payload | Base64 encoding   |  |  | | --- | --- | | Offset | Field | | 0 | File ID | | 4 | File Modified Time (8 bytes) | |

**Deleted Node (short term fast delete)**

The payload is set to all zeros.

**Compaction**

The deleted node still takes up space on disk. It's just inactive. So once in a while, the data structure is completely rewritten to disk, which removes the deleted nodes for real.

## Class Composition

class FileNameRegistration  
 DirectoryNode root  
 TreeStatus tree\_status  
  
 void Flush()  
  
 string add\_file(string file\_path)  
  
 DateTime? get\_modified\_time(string file\_path)  
 void set\_modified\_time(string file\_path, DateTime? modified\_time)  
  
 void delete(string file\_or\_dir\_path)

class DirectoryNode : IComparable<DirectoryNode>  
 readonly string name

SortedList<string, DirectoryNode> dir\_children  
 SortedList<string, FileNode> file\_children  
  
 void attach\_new\_file\_node(string file\_path, int start\_index,  
 FileNode file\_node)  
  
 void delete(string path, int start\_index)

class FileNode : IComparable<FileNode>  
  
 readonly string name

readonly string prefix

readonly uint file\_id

DateTime? modified\_time

DateTime? get\_modified\_time()

void set\_modified\_time(DateTime? modified\_time)

class TreeStatus  
  
 uint highest\_file\_id

## Call Trees

**Construction**

FileNameRegistration :: void init(string path)  
 |🡪 void populate\_tree\_from\_file(string path)  
 |🡪 int read\_until\_2nd\_tab(byte[] buffer)  
 |  
 |🡪 ⭯ DirectoryNode :: void attach\_new\_file\_node(string file\_path,   
 | int start\_index, FileNode file\_node) ⭯  
 |  
 |🡪 void rewrite\_tree\_to\_disk(string path)  
 |🡪 ⭯ DirectoryNode :: void rewrite\_children\_to\_disk(  
 | FileStream new\_file\_stream, string path\_so\_far) ⭯  
 |  
 |🡪 FileNode :: void append\_to\_disk(string full\_path)  
 |🡪 void write\_payload\_to\_disk()  
  
The ⭯ denotes a recursive function.

**Add new file**

FileNameRegistration :: string add\_file(string file\_path)  
 |🡪 ⭯ DirectoryNode :: void attach\_new\_file\_node(string file\_path,   
 | int start\_index, FileNode file\_node) ⭯

|🡪 FileNode :: void append\_to\_disk(string full\_path)

|🡪 void write\_payload\_to\_disk()

**Delete a node**

FileNameRegistration :: void delete(string file\_or\_dir\_path)  
 |🡪 ⭯ DirectoryNode :: void delete(string path, int start\_index) ⭯  
 |🡪 FileNode :: void delete\_from\_disk()  
 |  
 |🡪 ⭯ DirectoryNode :: void delete\_all\_children() ⭯  
 |🡪 FileNode :: void delete\_from\_disk()

**Set modified time**

FileNameRegistration :: void set\_modified\_time(string file\_path, DateTime? modified\_time)  
 |🡪 ⭯ DirectoryNode :: FileNode get\_file\_node(string path, int start\_index) ⭯  
 |  
 |🡪 FileNode :: void set\_modified\_time(DateTime? modified\_time)  
 |🡪 void write\_payload\_to\_disk()

# Backup Manager

## Backup Classes

abstract class Backup

abstract class EncryptedBackup

abstract class DiskBackup

The DiskBackup handles unencrypted backups from disk to disk.

The EncryptedBackup handles encrypted backups, both from disk to disk and from disk to cloud.

**BackupClass abstract methods**

abstract protected void backup\_one\_path(string source\_full\_path, FS\_Watcher\_EventType event\_type);  
  
abstract public void check\_all\_backups();  
  
abstract public XElement to\_xml();

These methods represent the kind of specialization DiskBackup and EncryptedBackup has to do.

## Class Compositions

class BackupRuleList  
  
  
  
  
  
  
  
  
  
  
 public List<BackupRule> backup\_rules  
  
 public bool accepts(string file\_path)

public enum BackupRuleType

public class BackupRule  
  
 readonly BackupRuleType rule\_type;

The main function of BackupRuleList is to accept or reject a given "file\_path".

abstract class Backup

FileSystemWatcher file\_system\_watcher;  
  
 public void start\_live\_backup()  
 public void live\_backup()

protected enum FS\_Watcher\_EventType

protected class FileEvent

public readonly FS\_Watcher\_EventType event\_type;

protected class PathStatus

public readonly bool exists, is\_directory, is\_hidden;

Despite being an abstract class, the Backup class does a lot.

The **file\_system\_watcher** object is what allows the class to monitor directories for changes.

The **FileEvent** object encodes the kind of changes seen by the file system wacher.

The **PathStatus** encodes the kind of object a path is. The reason for this class is that file\_system\_watcher treats files and directories equally, because at the OS level, files and directories are both nodes in a tree. The application, on the other hand, must treat files and directories distinctly, as directories require a recursive handling style. Also, empty directories are not backed up, while empty files (of 0 byte length) are backed up.

The constructor instantiates the file\_system\_watcher object, but does not start it. So in the beginning, this object is allocated in memory, but not monitoring directories. The **start\_live\_backup()** is what turns on the file\_system\_watcher object.

The **live\_backup()** reacts to changes reported by the file\_system\_watcher object. This function should not be triggered internally, since backups can take a long time. In this application, all backup and restore functions are triggered externally, from a "**backup\_manager\_thread**".

class DiskBackup : Backup  
  
 readonly BackupRuleList backup\_rule\_list;  
  
 ReportEvent\_Callback report\_event;  
  
  
  
  
 public Parameters future\_params;

public class Parameters

The "**backup\_rule\_list**" determines which files will be backed up and which files will be ignored.

The backup process runs in a separate thread from the main application. It will use the "**report\_event**" delegate to report progress information.

The backup classes, both "DiskBackup" and "EncryptedBackup", cannot be changed on the fly. Instead, the GUI modifies "**future\_params**". To have these future parameters take affect, the entire class is saved to XML and then recreated from scratch.

class EncryptedBackup : Backup  
  
 readonly string key;  
 FileNameRegistration file\_name\_reg;  
 EncryptStream encrypt\_stream;  
 CloudBackupService cloud\_backup;

The EncryptedBackup class has the same "backup\_rule\_list", "report\_event", and "future\_params" as the DiskBackup class. These are just not shown in the picture above. Only the new items related to encryption are shown.

The encryption process hides the file name inside the encrypted file. For faster processing, the "file\_name\_reg" object is necessary. The "encrypt\_stream" encapsulates the encryption process as a stream, so that it can work with "cloud\_backup" objects, which are wrappers around cloud service provider (AWS, Azure, GCP) APIs.

class BackupManager  
  
 public List<Backup> backups;  
 List<CloudBackupService> cloud\_backup\_services;  
  
  
 Thread backup\_manager\_thread;  
  
  
 Queue<BM\_Event> bm\_events\_sv;

public class Parameters

The "BackupManager" is a container that holds all of the "**Backup**" and "**CloudBackupService**" objects.

The backup and restore code runs on a thread separate from the main application, called "**backup\_manager\_thread**". The main application issues requests, which then get turned into "**BM\_Event**" objects that goes into a queue. The "backup\_manager\_thread" works by processing events from the "bm\_events\_sv" queue.

## Threaded Backup and Restores

Anything that takes more than 100 ms should run separate from the GUI thread, and backup and restore operations take a long time in general.

The "BackupManager" class runs a "**backup\_manager\_thread**" that processes backup and restore events.

The general pattern is:

* The "backup\_manager\_thread" is asleep, blocking on the "**backup\_thread\_wait\_obj**" object.
* Some external event happens, and some data structure is set up.
  + File system watcher triggered events, which I call live backup, will set up the Backup :: **List<FileEvent>** file\_events\_sv
  + Application request for full backup and restore will set up BackupManager :: **Queue<BM\_Event>** bm\_events\_sv
* The external event wakes up the sleeping "backup\_manager\_thread" thread using the "backup\_thread\_wait\_obj" object.
* The function "backup\_thread\_run()" processes the events.
* The "backup\_manager\_thread" reports information to the GUI thread using:  
  delegate void ReportEvent\_Callback(...)

**Live backup operation**

abstract class Backup

OS (no name) Thread

Backup Thread

file\_system\_watcher

file\_created(...)  
file\_changed(...)  
file\_deleted(...)  
file\_renamed(...)

class BackupManager  
{  
 void backup\_thread\_run()  
}

void live\_backup()

1. The file\_system\_watcher is an object that monitors a particular directory for file and directory changes. Events are reported to the event handers: file\_create(...), file\_changed(...), file\_deleted(...), and file\_renamed(...).
2. These event handers will wake up the "backup\_thread\_run()" thread using the "backup\_thread\_wait\_obj" object. This thread will wait for the file system to become idle.
3. Once the file system is idle, the live\_backup() function is used to backup the files most recently affected.

There are two different threads here. The file\_system\_watcher is calling from the run time, and is under what I call "OS" thread - actually there is no name. The "backup\_thread\_run()" is spawned by the application and actually has a name, called "BackupManager Thread".

**class BM\_Event**

This class is used for GUI thread 🡪 BackupManager thread communication.

|  |  |
| --- | --- |
| event\_type | param\_array |
| CHECK\_BACKUPS | null |
| GET\_RESTORE\_INFO | {restore\_manager, int[] indices, skip\_file\_names} |
| RESTORE | {restore\_manager, restore\_settings} |

**ReportEvent\_Callback delegate**

delegate void ReportEvent\_Callback(AppEventType event\_type,   
 params object[] param\_array);

This delegate is used for BackupManager thread 🡪 GUI thread communication.

|  |  |  |
| --- | --- | --- |
| event\_type | param\_array | Notes |
| LOG, ERROR | {string} | Messages.  If error, change application icon. |
| CHECK\_BACKUPS\_DONE | null | Switch back to live update. |
| GET\_RESTORE\_INFO\_DONE | {restore\_info} |  |
| FILES\_PROCESSED | {int} |  |
| RESTORE\_DONE | null |  |
| BM\_THREAD\_IDLE | null | Use the "ready" icon. |
| BM\_THREAD\_RUNNING | null | Use the "busy" icon. |

**Restore mode events:**

Most of the above events are for use in backup mode.

The restore mode only produces: FILES\_PROCESSED, GET\_RESTORE\_INFO\_DONE, RESTORE\_DONE.

## Call Trees

**DiskBackup Call Trees**

There are two backup functions. The "live\_backup()" takes a quick look, based on changes reported by the file\_system\_watcher. The "check\_all\_backups()" looks at all files to make sure the backup is up to date.

Backup :: live\_backup()  
 |🡪 DiskBackup :: backup\_one\_path(string source\_full\_path,   
 | FS\_Watcher\_EventType event\_type)  
 |  
 |🡪 delete(string path)  
 |🡪 backup\_file(string source)  
 |🡪 ⭯ backup\_directory(string source) ⭯  
 |🡪 backup\_file(string source)  
  
The ⭯ denotes a recursive function.

Note the process starts at the abstract class "Backup". The specialization starts at "backup\_one\_path(...)".

DiskBackup :: check\_all\_backups()  
 |🡪 ⭯ check\_source(string source) ⭯  
 | |🡪 backup\_file(string source)  
 |  
 |🡪 ⭯ check\_destination(string destination) ⭯

**EncryptedBackup Call Trees**

void backup\_one\_path(string source\_full\_path, FS\_Watcher\_EventType event\_type)  
 |  
 |🡪 void handle\_source\_deleted(string source\_file\_or\_dir\_path)  
 | |  
 | |🡪 void handle\_source\_file\_deleted(string source\_file\_path,   
 | | string alt\_file\_name)  
 | |  
 | |🡪 void handle\_source\_dir\_deleted(string source\_dir\_path)  
 |  
 |🡪 void backup\_file(string source)  
 | |  
 | |🡪 ⭯ void handle\_source\_dir\_deleted(string source\_dir\_path) ⭯  
 |  
 |🡪 ⭯ void backup\_directory(string source) ⭯  
 |  
 |🡪 void backup\_file(string source)

The ⭯ denotes a recursive function.

void check\_all\_backups()  
 |  
 |🡪 ⭯ void check\_source(string source\_dir) ⭯  
 | |🡪 void backup\_file(string source)  
 |  
 |🡪 ⭯ void check\_destination(string destination\_dir) ⭯  
 |  
 |🡪 void handle\_source\_file\_deleted(string source\_file\_path,   
 | string alt\_file\_name)  
 |  
 |🡪 ⭯ void handle\_source\_dir\_deleted(string source\_dir\_path) ⭯

**BackupManager Call Trees**

BackupManager :: void backup\_thread\_run()  
 |🡪 backup\_thread\_wait\_for\_disk\_idle(int idle\_time\_in\_sec)  
 |🡪 Backup :: void live\_backup()  
 |  
 |🡪 void backup\_thread\_handle\_events()  
 |🡪 Backup :: void check\_all\_backups()  
 |🡪 RestoreManager :: RestoreInfo get\_info(int[] indices, ...)  
 |🡪 RestoreManager :: void restore(RestoreSettings settings)

## Minor Notes

**Operation of FileSystemWatcher**

Due to the complexity of the file system watcher events, these events are only treated as hints. The philosophy in the code is to mirror the source directory, rather than blindly follow file system watcher events.

Still, a summary of the events generated might be useful for the future.

|  |  |
| --- | --- |
| User Action | File System Watcher Events |
| Right click in file explorer to create a text file.  Then rename the text file. | File created, file changed, directory changed events.  File rename, directory changed events. |
| Editing a text file and saving it to disk. | Two file changed events. |
| Renaming a file. | One rename event.  Directory changed event. |
| Moving a file. | File deleted, file created, directory changed (2x). |
| Deleting a file. | File deleted, directory changed. |
| Renaming a directory. | Directory renamed, parent directory changed. |
| Moving a directory. (note this is different from renaming a directory). | Directory deleted, directory created, parent directory changed (2x). |
| Deleting a directory. | Directory deleted, parent directory changed. |

**void backup\_one\_path(...) philosophy**

The philosophy in this function is to mirror the source drive on the destination drive, rather than blindly following the file system watcher event type.

**File system watcher messages are only hints**

The backup software should examine the source drive and duplicate the files on the destination drive, taking the file system watcher messages only has hints.

This is due to processing the messages with a delay. Also, the current code processes the messages out of order --- the code uses a dictionary to remove duplicated events. The code then iterates through this dictionary, causing events to be processed out of order.

For example, if the user deletes "temp" directory, and then modifies "temp\test.txt", then these will appear in the software as two different events, due to the different paths. These two paths might not be processed in the right order. So always check the latest situation on the source drive, and then duplicate that situation, rather than blindly following the file system watcher events.

**File and directory non-synchronization**

The concept is that what exist as a file on the source drive can be a directory on the destination drive, and vice versa. The code must not assume what exists as a directory on the source drive is also a directory on the destination drive.

This is due to the backup events not being processed in perfect real time. The software waits for a period of idleness before taking backup action.

As an example, suppose both the source and backup drives start with a directory called "temp". The source drive then deletes the "temp" directory and creates a file called "temp". Both of these actions occur very quickly, so when these actions ended, the source will have a file called "temp", while the backup drive has a directory called "temp".

**Only files are backed up**

The mirroring is for files only. Empty directories are not duplicated on the destination path.

**Modify time checking**

**Summary of current behavior:** There is NO modified time check during live backup, but there is a "!=" modified time check during manual backup.

**Clarification:**

There are two kinds of backup in this program. There is a "live backup", where the user changes a file on the disk, and it gets copied to the backup source. There is a "manual backup", where the user goes to "menu 🡪 check all backups", and the program goes through all the files, and make backups as needed.

During live backup, the call path is:

(DiskBackup or EncryptedBackup) 🡪 backup\_one\_path(...) 🡪 backup\_file(...), there is no checking of the file modified time.

During manual backup, the call path is:

(DiskBackup or EncryptedBackup) 🡪 check\_all\_backups() 🡪 check\_source(...), there is a check on the file modified time inside "check\_source(...)".

The modify time checking code looks like:

if (destination\_time != source\_time)

backup\_file(file\_name);

// The following check would be too lax:

// if (destination\_time < source\_time)

// See documentation --> BackupManager --> Minor Notes --> Modify time checking

**why "if (destination\_time < source\_time)" is insufficient:**

Suppose file Y is older than file X.

The backup program is not running, so there is no "live backup".

User deletes file X, then rename file Y to X.

The user then runs "check all backups". The desired behavior is to delete file Y from backup and upload the new file X to backup.

File Y on the backup side is deleted since it no longer exists on the source side. However, file X on the source side is not backed up because this new X, which has Y's time stamp, is older than the file X on the backup side. As a result, the new file X on the source side, is not backed up.

# RestoreManager

**Call Trees**

Restore :: void get\_info(RestoreInfo restore\_info)  
 |🡪 string get\_info\_handler(...) via DecryptStream :: extract\_relative\_path()  
 |  
 |🡪 string[] get\_alt\_file\_names()  
 |🡪 void break\_down\_alt\_file\_name(...)

Restore :: void restore(...)  
 |🡪 string restore\_handler(...) via DecryptStream :: extract\_relative\_path()  
 |  
 |🡪 string[] get\_alt\_file\_names()  
 |🡪 void break\_down\_alt\_file\_name(...)

The "get\_info\_handler(...)" and "restore\_handler(...)" are callbacks. The true file name is embedded inside the file and is not known when restore(...) starts. When the DecryptStream object decodes sufficient part of the file, it will call the "handler" functions to report the relative path that's found inside a file.

The "get\_info\_handler(...)" records the information found by DecryptStream, and then returns a null string, causing DecryptStream to not save anything to file.

The "restore\_handler(...)" returns a file path and DecryptStream will decrypt to this path.

# GUI

## WindowsBackup\_App

**Application class regions**

* Loading and saving settings
* Handling events from BackupManager thread
* Restore mode
* Utility functions used by multiple classes

**Initialization**

After the software is built, there is no XML nor file registration file. The location of these files are hard coded as:

class WindowsBackup\_App  
 // File locations

const string settings\_path = "data\\settings.xml";

const string fnr\_path = "data\\fnr.tsv";

**Initialization Call Tree**

WindowsBackup\_App :: WindowsBackup\_App()  
 |🡪 void create\_default\_file()  
 | |🡪 void save\_settings()  
 |  
 |🡪 void read\_settings()

**Event Handling**

The event handler is

WindowsBackup\_App::handle\_event(...)

When BackupManager and RestoreManager is constructed, this event handler is passed in:

WindowsBackup\_App::read\_settings()   
{  
 backup\_manager = new BackupManager(..., handle\_event);  
 restore\_manager = new RestoreManager(..., handle\_event);  
}

## Updating Output\_tb

**Background - why writing to output textbox is special**

"Output\_tb" is the output textbox in the "MainWindow.xaml". Updating this textbox is especially complicated due to the need to scale to handle many messages.

The event handler "WindowsBackup\_App::handle\_event(...)" is running on the backup thread and cannot change the GUI directly. Most event handling routines will immediately spawn a WPF dispatcher with code that looks like:

Output\_tb.Dispatcher.BeginInvoke(...)

Updating the textbox can also be done by immediately calling BeginInvoke(...), however this will lock up the GUI if there are a lot of messages, and each message is resulting in a dispatcher call.

If the user backs up a lot of files, say 100,000 files, and this is happening on disk so it's really fast, then the program would be hit with 100,000 messages within a few seconds. Simply throwing this into the WPF dispatcher and textbox results in GUI hang up.

**The solution to handle large amount of messages**

This program puts new messages in a circular buffer, then calls the Dispatcher.BeginInvoke(...) with a 200 ms delay so to not overwhelm the WPF dispatcher system. There are a total of three threads involved - the backup thread that is spawning the message, a delay thread that is creating the 200 ms delay, and the GUI thread that is rendering the text to "Output\_tb".

"buffer\_sv" is the circular buffer holding the text. The "\_sv" means it's a shared variable used for synchronization.

|  |
| --- |
| thread having lock on "buffer\_sv" |

|  |  |  |
| --- | --- | --- |
| Text generation thread | Delay thread | GUI thread |
|  | waiting |  |
| add "line #1" to buffer |  |  |
|  | wakes up |  |
| add "line #2" to buffer  add "line #3" to buffer | 200 ms delay, then invoke "add\_text\_to\_Output\_tb(...)" via GUI thread dispatcher |  |
|  | waiting | Render line #1 through #3 to "Output\_tb" |
| add "line #4" to buffer |  |  |

**Test Program**

All this is sufficiently complex that a special test program is created.

**GUI**

<DockPanel>

<TextBox AcceptsReturn="True" Name="Output\_tb" IsReadOnly="True"

HorizontalScrollBarVisibility="Auto"

VerticalScrollBarVisibility="Auto"/>

</DockPanel>

**Program**

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

delay\_thread = new DelayThread(Output\_tb, new Func\_void(add\_text\_to\_Output\_tb));

}

#region Testing code

private void Window\_Loaded(object sender, RoutedEventArgs e)

{

var thread = new Thread(new ThreadStart(generate\_text));

thread.Name = "Text generation thread";

thread.Start();

}

void generate\_text()

{

// Test settings:

// Small number of lines

/\* int add\_interval = 200; // add text every "add\_interval" ms

int lines\_added\_per\_interval = 1;

int total\_lines = 300; \*/

// Medium number of lines

/\* int add\_interval = 100; // add text every "add\_interval" ms

int lines\_added\_per\_interval = 5;

int total\_lines = 1000; \*/

// Huge number of lines

int add\_interval = 50; // add text every "add\_interval" ms

int lines\_added\_per\_interval = 1000;

int total\_lines = 100000;

int line\_num = 0;

Thread.Sleep(1000);

while (true)

{

for (int i = 0; i < lines\_added\_per\_interval; i++)

{

add\_line("line " + line\_num);

line\_num++;

}

if (line\_num >= total\_lines) break;

Thread.Sleep(add\_interval);

}

Thread.Sleep(1000);

delay\_thread.quit();

}

#endregion

#region TextBuffer data structure

/// <summary>

/// A circular buffer to hold the most recent lines of text added.

/// </summary>

private class TextBuffer

{

public string[] buffer;

int start = 0;

int end = 0;

// The valid strings are at [start, end).

// The string at buffer[end] is not valid, so the length

// of the buffer is end - start, with wrap around adjustment.

/// <summary>

/// Number of valid strings in the circular buffer.

/// </summary>

public int Length

{

get

{

// Length is (end - start) if no wrap around adjustment is needed.

if (end >= start)

return end - start;

else

{

// When "end" < "start", do a wrap around adjustment

return end + buffer.Length - start;

}

}

}

public TextBuffer(int max\_size)

{

buffer = new string[max\_size + 1];

// It's +1 because there is a "wasted spot" at index "end".

// The "start" and "end" are off by 1 even if the size is zero.

}

/// <summary>

/// Store "text" into a local buffer. If the buffer is filled up,

/// oldest strings are removed.

/// </summary>

public void add\_string(string text)

{

// Place text at buffer[end]. Then need to adjust "end"

buffer[end] = text;

end++;

if (end >= buffer.Length) end = 0;

// When the buffer is filled up, "end" will catch up to "start".

// So need to adjust "start".

if (end == start)

{

start++;

if (start >= buffer.Length) start = 0;

}

}

/// <summary>

/// Remove "n" strings from the buffer.

/// </summary>

public void remove\_strings(int n)

{

if (n >= Length)

{

// Remove all strings

start = end;

}

else

{

// Remove only some of the strings. In this case, "start" will

// not be "end" after removal.

// Increment "start" and apply a wrap around.

start += n;

if (start >= buffer.Length) start -= buffer.Length;

}

}

/// <summary>

/// Remove all strings from the buffer.

/// </summary>

public void remove\_all()

{

start = 0; end = 0;

}

public string this[int index]

{

get

{

// Check "index" for valid range.

if (index < 0)

throw new Exception("Software error. Software attempted to access "

+ "a TextBuffer object at index " + index + ", which is less than zero.");

if (index >= Length)

throw new Exception("Software error. Software attempted to access "

+ "a TextBuffer object at index " + index

+ ", which is beyond than the buffer length " + Length);

// The "true\_index" is "start + index", with wrap around.

int true\_index = index + start;

if (true\_index >= buffer.Length) true\_index -= buffer.Length;

return buffer[true\_index];

}

}

}

#endregion

#region Thread to delay WPF dispatcher launch

delegate void Func\_void();

/// <summary>

/// A thread that will call Dispatcher.BeginInvoke(...) after

/// a delay, so to not overwhelm WPF dispatcher system.

/// </summary>

private class DelayThread

{

TextBox Output\_tb;

Func\_void update\_function;

// This thread will call:

// Output\_tb.Dispatcher.BeginInvoke(update\_function)

// after a delay.

bool quit\_flag = false;

AutoResetEvent wait\_flag = new AutoResetEvent(false);

public DelayThread(TextBox Output\_tb, Func\_void update\_function)

{

this.Output\_tb = Output\_tb;

this.update\_function = update\_function;

var thread = new Thread(new ThreadStart(run));

thread.Name = "Delay thread";

thread.Start();

}

void run()

{

while(true)

{

if (quit\_flag == true) return;

wait\_flag.WaitOne();

if (quit\_flag == true) return;

Thread.Sleep(200);

Output\_tb.Dispatcher.BeginInvoke(update\_function);

}

}

public void quit()

{

quit\_flag = true;

wake();

}

public void wake()

{

wait\_flag.Set();

}

}

#endregion

#region Add text code

DelayThread delay\_thread;

// Shared variable: buffer\_sv

// This is shared between the background thread that is

// generating the text, and the GUI thread that is

// displaying the text.

const int buffer\_size = 200;

TextBuffer buffer\_sv = new TextBuffer(buffer\_size);

/// <summary>

/// Adds "text" to "text\_buffer". Uses "delay\_thread" to trigger

/// a delayed update to Output\_tb in the future.

/// </summary>

void add\_line(string text)

{

// This function is on the background thread. The "buffer"

// is being used as a flag variable.

lock (buffer\_sv)

{

buffer\_sv.add\_string(text);

}

delay\_thread.wake();

}

void add\_text\_to\_Output\_tb()

{

// update Output\_tb with what's in the buffer\_sv

lock (buffer\_sv)

{

var sb = new StringBuilder();

for (int i = 0; i < buffer\_sv.Length; i++)

sb.Append(buffer\_sv[i] + "\n");

Output\_tb.Text = sb.ToString();

}

Output\_tb.ScrollToEnd();

}

#endregion

}

## SetupWindow

**Initialization**

The tree view nodes (TreeViewItem objects) on the left hand side has two different kind of "Selected" handlers.

internal SetupWindow(WindowsBackup\_App app)  
 |🡪 void associate\_tv\_item\_to\_page(TreeViewItem tv\_item, Page page)  
 | |- attaches void standard\_tv\_item\_handler(object sender, RoutedEventArgs e)  
 |  
 |🡪 void init\_backup\_nodes()  
 |- attaches void backup\_tv\_item\_handler(object sender, RoutedEventArgs e)

The "standard" version attaches the page object to the tree view node. When that tree view node is selected, the frame navigates to the page.

The "backup" version attaches data object to the tree view node. When that tree view node is selected, the data is loaded into "edit\_backup\_page" and the frame navigates to "edit\_backup\_page".

# Bugs and Issues

## 1. Renaming a directory

Currently, the file / directory "rename" event is handled as a delete event followed by a create event. See

Backup :: void file\_renamed(object source, RenamedEventArgs e)

The worst situation is when the backup is in the cloud. Renaming a directory means a total re-upload if backing up to cloud storage.

The current system stores the file path information inside the encrypted portion of the file. The file name registration (.tsv file) is not used in the restore process. Therefore, handling rename event by modifying the file registration is not sufficient.

In the local backup situation, deleting a directory and then copying a new one, instead of renaming the directory, is sub-optimal, but tolerable.

The main problem, which is backing up to the cloud, has no easy solution. So it's decided to leave the "file\_rename" implementation as it is. Fixing this problem for local backup has a small benefit, but it will introduce a new FS\_Watcher\_EventType and complicate the code slightly.

## 2. No logging, no retry

Currently there is no logging (because there is no support).

There is no network retry, so any network error will stop the whole program. The icon goes red and nothing works.

To implement retry and logging, edit the classes in "CloudBackupService.cs". For example, in the class "AWS\_CloudBackupService":

public void upload(Stream stream, string bucket\_name, string key\_name)

{

try

{

transfer\_utility.Upload(stream, bucket\_name, key\_name);

}

catch (Exception ex)

{

throw new Exception("Error while trying to upload to AWS \""

+ bucket\_name + "\\" + key\_name + "\". AWS error message: "

+ ex.Message);

}

}

My usual practice is to throw something nice looking, so just the

"Error while trying to upload to AWS \"" + bucket\_name + "\\" + key\_name

part. The "ex.Message" part, and any other detail, gets written to a log file.

## 3. Single threaded backup increases idle wait time

**Symptom:**

The expected behavior is the icon turning yellow for 5 seconds during backup.

But having multiple backup objects getting triggered increases this wait time.

Example setup:

Backup object 1 monitors directory X and backup to disk.  
Backup object 2 monitors SAME directory X and backup to cloud.

When a file in X changes, the icon is yellow for 10 seconds, not 5.

**Idle wait time explanation:**

When a file is updated, the "live backup" code does not immediately make a backup, but waits for idleness. As of early 2018, this wait is currently hard coded to be 5 seconds and is located in:

BackupManager::backup\_thread\_run()  
 |🡪 backup\_thread\_wait\_for\_disk\_idle(5); // 5 second wait

If there is a single file backup object, monitoring the directory containing file X, and file X changes, the software waits for 5 seconds without seeing further backup activity, and a back up occurs.

Visually, the icon will turn yellow for slightly longer than 5 seconds. Most of this time is due to the 5 second file system idle wait time.

**Increased wait time explanation:**

All backup objects are operating on the same backup thread. In the situation described in the "symptom" section above, there are two backup objects, monitoring the SAME directory X. When a file in X is changed, these two backup objects wake up and they execute in sequence, on the same thread. So each backup object is waiting for the 5 second idleness, totaling 10 seconds of idleness.

For information on the backup thread, see section: Backup Manager 🡪 Threaded Backup and Restores.

**No fix for now:** single threaded code is easier and this issue is a (minor) nuisance.

## 4. Very low wait time when adding a backup to Google Cloud (GCP)

**Encountered Jan 1, 2019**

When creating a backup to the cloud, this software will try to list objects in the cloud, to be sure that the cloud account is still valid. At most, this software will try to list 10 objects.

Listing from Google Cloud is very slow, like 55 seconds slow. This delay is incurred when listing the very first object - so it takes 55 seconds to list one object. The code is at:

CloudBackupService.cs 🡪 GCP\_CloudBackupService 🡪 list\_objects(...).

foreach (var obj in client.ListObjects(bucket\_name))

{

object\_list.Add(obj.Name);

if (object\_list.Count == max\_objects) return object\_list;

}

Put a breakpoint on "foreach". The very first call to "ListObjects(...)" is very slow

This code is exactly like the code listed at <https://cloud.google.com/storage/docs/listing-objects#storage-list-objects-csharp>

## Other

**Failure at WindowsBackup\_App :: void add\_text\_to\_textbox(string text)**

Date observed: March 16, 2018

Event viewer report:

Application: WindowsBackup.exe

Framework Version: v4.0.30319

Description: The process was terminated due to an unhandled exception.

Exception Info: System.ArgumentOutOfRangeException

at System.String.Remove(Int32, Int32)

at WindowsBackup.WindowsBackup\_App.add\_text\_to\_textbox(System.String)

This seems to be from: text2.Remove(0, chars\_to\_remove);

But the chars\_to\_remove should always be >= 0. Also all calls to "add\_text\_to\_textbox(...)" is done on "Output\_tb.Dispatcher.BeginInvoke(...)", so the calls are being done via GUI thread. Therefore, the error mentioned in the event viewer should not be possible.

I added some range checks on chars\_to\_remove nevertheless.

Also added comment in the function.

# Miscellaneous

## Alternative File Name Registration Format

This software uses a text-based format, so the user can open the file name registration file and feel good about it.

The main downside is that the file paths are highly repetitive. So the file is larger than it should be, and takes longer to load than it should.

An alternative is to use a binary format. Each node in the directory tree is stored just once. Performance should be better, but since the user cannot see the file content, it feels worse. Also, if there is any bug or file corruption, even if it's very minor, the file will be hard to repair by hand.

In the end I went with the text-based route. However, the idea of a binary format is documented below.

### Tree

**A tree backed by a random access file**

The FileNameRegistration class is a tree that can quickly persist changes to a file.

As a file, the tree looks like:

|  |  |  |  |
| --- | --- | --- | --- |
| Header | Node #1000 | Node #1001 | Node #1002 |

Each node contains its parent node's ID.

The in memory tree knows where each node is on disk, can therefore can change its metadata immediately.

New nodes are added at the end of the file.

Deleting a node changes its node ID to an invalid value.

**Compaction**

The deleted node still takes up space on disk. It's just inactive. So once in a while, the data structure is completely rewritten to disk, which removes the deleted nodes for real.

### Binary File Format

**Header**

|  |  |  |  |
| --- | --- | --- | --- |
| Byte # | Field Name | Field Length | Description |
| 0 | Version | 1 | Value = 1 |

**Directory Node**

|  |  |  |  |
| --- | --- | --- | --- |
| Byte # | Field Name | Field Length | Description |
| 0 | Node Type | 1 | Value = 1 for directory nodes |
| 1 | Node ID | 4 |  |
| 5 | Parent ID | 4 |  |
| 9 | Length of Name | 2 | Length of the name when it's UTF8 encoded. |
| 11 | Name | ? | Folder name. |

**File Node**

|  |  |  |  |
| --- | --- | --- | --- |
| Byte # | Field Name | Field Length | Description |
| 0 | Node Type | 1 | Value = 2 for file nodes |
| 1 | Node ID | 4 |  |
| 5 | Parent ID | 4 |  |
| 9 | Prefix | 4 | A single character that should be unique for the computer. Four bytes allocated to allow UTF-8 encoding. |
| 13 | File ID | 4 | The name of a file will be the prefix plus this file ID. |
| 17 | File Modified Time in ticks | 8 | When files are backed up, the file modified time is stored here. Later, this field is used to determine whether the backup is up to date or not.  Value = 0 means null. |
| 25 | Length of Name | 2 | Length of the name when it's UTF8 encoded. |
| 27 | Name | ? | File name. Null terminated. |

**Root Node**

Root node has a node ID of 1000. The next node will have node ID of 1001, and so on.

**Deleted Node (short term fast delete)**

The node ID is set to zero.

The "node type" and "length of name" fields must remain so the software can determine how many bytes to skip over to find the next node.

## Icons

The icon is just the letter "B". I used Windows Paint to create a "png" bit map, and then I used IrfanView to convert to the ".ico" format.

Paint wants to anti-alias the "B", which is fine if the background is actually white. However my taskbar is actually black, so the anti-aliased "B" look poor.

I tried different drawing packages but both GIMP and Paint.Net wants to anti-alias the letter "S". Eventually I wrote an C++ OpenCV program to threshold the image into two colors.

#include <iostream>

#include "opencv2/opencv.hpp"

using namespace cv;

using namespace std;

int main(int argc, char \*argv[]) {

Mat img = imread("Icon1.png");

// determine the intensity range

int highest\_intensity = -1;

int lowest\_intensity = 1000;

for (int i = 0; i < img.rows; i++) {

for (int j = 0; j < img.cols; j++) {

int intensity = (int)img.at<Vec3b>(i, j)[0] + (int)img.at<Vec3b>(i, j)[1]

+ (int)img.at<Vec3b>(i, j)[2];

if (intensity > highest\_intensity) highest\_intensity = intensity;

if (intensity < lowest\_intensity) lowest\_intensity = intensity;

}

}

cout << "highest\_intensity = " << highest\_intensity << endl;

cout << "lowest\_intensity = " << lowest\_intensity << endl;

int cutoff = (highest\_intensity + lowest\_intensity) / 2;

Mat img2 = Mat(img.rows, img.cols, CV\_8UC3, Scalar(0));

for (int i = 0; i < img.rows; i++) {

for (int j = 0; j < img.cols; j++) {

int intensity = (int)img.at<Vec3b>(i, j)[0] + (int)img.at<Vec3b>(i, j)[1]

+ (int)img.at<Vec3b>(i, j)[2];

if (intensity > cutoff) {

img2.at<Vec3b>(i, j)[0] = 255;

img2.at<Vec3b>(i, j)[1] = 255;

img2.at<Vec3b>(i, j)[2] = 255;

}

else {

// new color

img2.at<Vec3b>(i, j)[0] = 0;

img2.at<Vec3b>(i, j)[1] = 209;

img2.at<Vec3b>(i, j)[2] = 50;

}

}

}

imwrite("new.png", img2);

int temp;

cin >> temp;

}