Visual Source Safe File Format

I needed to write an importer that would pull a SourceSafe database into a propritary source control system. Searching around the net, I was surprised to find almost no information about the format of SourceSafe files. There are a few apps that interface with SourceSafe, but do so using the OLE interface. There are fragments of code floating around, but they are uncommented, undocumented, and possibly encumbered by open source licenses. So I sat down and spent a few evenings pulling apart the VSS database in question, trying to identify as much useful information as I possibly could. This document records much of what I figured out, and perhaps will save the next poor sod some time and frustration.

You'll find the test code I used in vsscode.zip. This code doesn't do anything meaningful. It's the experimental code I used to make certain that I could pull apart all of the VSS data files, and figure out what is in them. It's a simple command-line app, with lots of `printf()` statements that dump lots of info. Most of those statements are commented out, so you'll need to rebuild the code, changing what is commented to dump things out. Essentially, I froze the project at the point where I knew what I needed — beyond that, everything I did was proprietary, and cannot be released online.

Most importantly, however, I've commented the code up heavily, noting down all of the observations I made about the contents of the files, along with a few conjectures I was not able to confirm. Keep in mind, the database I was working with had only been in a non-networked environment, had never been subjected to shared check-outs, no merging... In other words, this database had never been touched by some of the operations for which SourceSafe is renowned (perhaps even *infamous*) at corrupting databases.

A warning on the implementation of this code: it scans the database and pre-loads all of the files into memory. Do not try to run this code against a multi-gig database. For reference, the database that was tested contained 3,000 files, taking 50 MB of disk space. The first time the code runs, it requires 30 seconds to read in all of the files -- probably a testament to how fragmented VSS databases can become, more than anything. Running the test a second time requires less than a second once all of the files are loaded into the system file cache. If you're going to experiment with looking inside VSS files, either create a new database, or find a small pre-existing database before testing large ones with years and gigabytes of accumulated data.

The first observation I'll make — to help preserve the sanity of anyone else trying to decipher the contents of VSS files — is that whoever wrote the file-writing code did not believe in initializing data. Unused space in the file typically contains random blocks of data, frequently parts of adjacent data. By way of example, you might be looking at a 32-byte string field. If that field contains a 5-byte string, the other 27 bytes will be filled with random garbage. Worse, that garbage will often look like meaningful data, since it often contains some chunk of memory that was just initialized and written to some other part of the VSS file. Needless to say, this makes it difficult to figure out whether some promising looking data is *really* data, or random junk.

Most of the data in VSS files are contained in RIFF chunks. While most RIFF files use Four Character Code (FourCC) markers, VSS files use Two Character Code (TwoCC) markers. If you were to define the chunk header as a struct, it would look something like this:

```
    DWORD ChunkSize;
    WORD  TwoCC;
    WORD  CRC;
```

The chunk size does not include the 8-byte header. And the 16-bit CRC is mostly the standard algorithm. The one difference is that in my experience, CRCs typically start off by initializing the state to -1 (`0xFFFFFFFF`), accumulating, then returns the logical-NOT of the result. However, the VSS CRC logic initializes state to 0, and does not apply a logical-NOT at the end. Make certain you're using this technique when verifying any CRCs in the file. (Refer to `VssCrc32()` in CRC32.c for a working implementation.)

2CC Markers

Here is a table of all the TwoCCs I've seen in SourceSafe files. Note that I'm using my names for them, since I'm not sure what the real names are supposed to be. The TwoCCs sometimes are a good hint, but other times it's not obvious how they relate to the data they contain.

Chunk Type	TwoCC	Hex
Branch File	BF	`0x4642`
Check Out	CF	`0x4643`
Child	JP	`0x504A`
Comment	MC	`0x434D`
Data Header	DH	`0x4844`
Difference	FD	`0x4446`
Log Entry	EL	`0x4C45`
Name Header	HN	`0x4E48`
Parent Folder	PF	`0x4650`
Short Name	SN	`0x4E53`

One observation from the 2CCs: whoever created them must not have been paying much attention to endian issues when defining them.

Basic Organization

All files and directories (or "projects" as they're called in VSS) are identified by a unique number. When a new database is created, the first file is 0, and each file thereafter increments the number. But VSS never stores these values in number format. They are always encoded as an eight-character string, such that 0 maps to "aaaaaaaa", 1 maps to "baaaaaaa", etc. Use the following chunk of code to map these strings back to useful integer values:

```
int VssNameToNumber(wchar_t name[])
{
    int num = 0;

    // The number is encoded in base-26, using the letters
    // 'a' through 'z'.  Need to scan from left to right,
    // which is the reverse of the conventional symbol
    // ordering used in computing.
    //
    for (int i = 7; i >= 0; --i) {
        if (('a' <= name[i]) && (name[i] <= 'z')) {
            num = (num * 26) + u32(name[i] - 'a');
        }
        else if (('A' <= name[i]) && (name[i] <= 'Z')) {
            num = (num * 26) + u32(name[i] - 'A');
        }
        else {
            return -1;
        }
    }

    return num;
}
```

There are 26 directories where these files are stored, based on the first character of the file name. For each object in the database, there will be two files stored. One is a log file containing a record of all the check-outs, check-ins changes, etc. This is the file that has the simple "aaaaaaaa" name. The other file, which has an extension of ".a" or ".b", will be the data. If this object is a directory, the data file will contain a list of names for every object stored in the directory (and this list appears to always be sorted alphabetically). If the object is a file, this will be the most recent version of the file.

Log File Header

Consult the code in VssScanHeader.cpp for the details of parsing the header of a log file. I'll describe things from a higher-level point-of-view here.

Every log file starts with the 20-byte string `"SourceSafe@Microsoft"`. This is followed by a few other fields, which are always followed by a Data Header ("DH") chunk, which my code considers to be part of the the the file header.

Basically, there are a lot of bytes stored in the header, but much of them are useless. A number of fields tell you how many log entries are stored in the file, where the first and last are located, and how large the file is -- information you'll figure out for yourself when you're pulling the file apart.

Given the way it is organized, many of these values can be used to validate the file, checking for corruption. So you can try to recover information from damaged files by pulling out the values from the header, then testing them against the information you can extract from the individual chunks.

Aside from validation, the only information I use from the header is the `m_Type` field, which indicates whether this is a file or a project, and `m_Name`, which is the name of the directory.

Note that it's not clear if `m_Name` is safe to use for files, since they can be renamed and shared. Therefore, `m_Name` may not reflect the current name of a file as it appears everywhere it is shared. It is sometimes used for logging purposes.

Once you get past the header, everything else that is stored in the log file is a TwoCC chunk. Files and projects tend to store different types of data.

But first, a brief degression from the log file, so that we can go over how to recover the directory structure of the database...

Child Chunk: "JP"

It's not clear what the TwoCC for this chunk is intended to mean: Project? Project Journal? In any case, there is an array of these chunks stored in the ".a" file associated with each project. In code format, this data is arranged like this:

```
    WORD  m_Type;       // 1 = directory, 2 = file
    WORD  m_Flags;      // 0x01 = deleted, 0x02 = binary, 0x08 = shared
    WORD  m_NameFlags;  // 0 = file, 1 = directory
    char  m_Name[34];   // name of file
    DWORD m_NameOffset; // offset into "names.dat"
    WORD  m_Zero;       // apparently unused
    char  m_DBName[10]; // name of database file in "aaaaaaaa" format
```

These fields are commented up in more detail in VssScanChild.h.

These chunks appear to be the only type of data that is stored in a project's data file. Nothing else has been seen in these files, but caution should still be used when scanning the files. Mapping the file as an array of structs may not always be safe. There is a loop at the bottom of `VssTree::AssembleDirectoryLinks()` that shows scanning all of the child chunks in a directory's data file to recurse through the directory tree of the database.

In brief, this is the information you'll want to look at to figure out the directory structure of the database. Start with the file "aaaaaaaa". This will always be the root directory of the database. Its data file (either "aaaaaaaa.a" or "aaaaaaaa.b") will contain an array of child chunks. These chunks appear to always be stored in alphabetical order. Use the `m_Type` field to determine whether the entry is a file or a subdirectory. For any subdirectory, use the `m_DBName` to recurse to that subdirectory and scan all of its child chunks. Continue recursing until all directories have been examined. This allows you to recreate the directory tree stored in the database.

While recursing in this manner, keep in mind that a shared file will be visited multiple times. Every directory that shares the file will have a reference to the same `m_DBName` file. If you're importing these files into your own source control system, avoid processing the same file more than once. My code initialized `m_Type` to zero to mark a file as not having been processed, then sets the type to 1 or 2 when it processes the file the first time.

Note that in the case of shared files, a combination of sharing and renaming may result in the same file having different names in different locations. The name stored in the child chunk is the name that will be used when the file is checked out.

Parent Chunk: "PF"

The companion to the child chunks are parent chunks. Each log file will contain at least on parent chunk. One of these will be created when the file is initially created. A new one will be appended each time the file is shared to another project. However, these chunks are never removed. If a shared link is broken (e.g., "branching" a file), the chunks remain, but they have their name field set to an empty string to indicate that the reference no longer exists.

Parsing a parent chunk is demonstrated in VssScanParent.cpp. There isn't much data there. Each chunk contains the offset of the previous chunk, which allows you to start with the last parent chunk in the log file, then scan backwards to locate all chunks in the file.

```
    DWORD m_PreviousOffset; // offset of previous parent chunk, or 0
    char  m_DBName[10];     // name of parent directory
```

The database name will indicate the name of the database directory that holds a reference to the file. This allows you to determine how many directories share this file.

Note that only files appear to ever have parent chunks. Projects do not have these chunks. This is probably because only files can be shared, not projects -- "sharing" a project will create a new project, then share the files it contains.

Comment Chunk: "CM"

Now that we've roughed out how to traverse the directory tree, we'll go into detail about the other chunks that are found in log files. The first we'll cover is comments. These are found in the log files for both files and projects.

A comment chunk only contains an ASCII string. The string appears to always have a `'\0'` terminator. Comments appear to be mandatory for some types of events, so even if no comment was entered, you will find a reference to a comment chunk that is one byte long, containing only the `'\0'` terminator.

Once a comment chunk has been added to a log file, that chunk will never be deleted or edited. If a user edits a comment, a new comment chunk will be appended to the file, and the existing log entry updated to contain the offset of the new chunk, effectively orphaning the original comment. If you see unused comments sitting in a log file, they are probably the result of someone editing a comment after-the-fact.

Note that log entries for label operations may have two comments, one for the regular comment and one for the label comment. When creating a new label, there is only one edit box for typing in a comment. This comment is stored as the label comment. However, if you edit an existing label, there are two edit boxes, which expose the contents of both the label comment and the regular comment associated with the label operation.

Only log entries for label operations will use the label comment. All other log entries only use the regular comment.

Log Entry: "EL"

Besides comments, the other chunk type that is shared by both files and projects is the log entry. This is the most complex type of chunk to deal with, since there are a couple dozen types of events that can occur, and each type of event stores different data, often in a different format. As I mentioned above, much of the unused data written into a log file is not zeroed out. This makes it difficult to determine whether I've really pulled out all of the data that is found in type of log entry.

The following table lists all of the known log events that are defined. This is based on a number of different sites around the net. For the database that was tested, not all of these events were encountered. Only those marked with '*' were found in the test database. The other events are unconfirmed, and of unknown format.

Code	Event
0 *	Labeled
1 *	Created Project
2 *	Added Project
3 *	Added File
4 *	Destroyed Project
5 *	Destroyed File
6 *	Deleted Project
7 *	Deleted File
8	Recovered Project
9 *	Recovered File
10 *	Renamed Project
11 *	Renamed File
12	Moved Project From
13	Moved Project To
14 *	Shared File
15 *	Branched File
16 *	Created File
17 *	Checked-In File
18	Checked-In Project
19	Rolled Back
20	Archived Version File
21	Restored Version File
22	Archived File
23	Archived Project
24	Restored File
25	Restored Project
26	Pinned File
27	Unpinned File

A log entry is a fixed-size structure, containing exactly 404 bytes. Each of the events stores different data, leaving some amount of unused space at the end of the fixed-size buffer. Some events use the same arrangement of data, while others do not. Consult the VssScanLogEntry.cpp file for explicit details about which fields are defined for each event.

`m_Opcode`: What type of operation was performed? This will be one of the values from the table above.

`m_VersionNumber`: Records the version number associated with the currently-checked in file. The current version number is stored in the log file header. Each check-in operation increments the version number, and the current version number is recorded for log entry.

`m_Timestamp`: When was this operation performed? Like all other timestamps, this is a 32-bit `time_t` value. The one peculiar observation is that `gmtime()` must be used to conver this to the correct local time of the operation. Using `localtime()` will print the wrong timestamp.

`m_PreviousOffset`: Each log entry records the offset of the previous log entry. The log file header stores the offset of the last log entry. Using these two fields, you can start scanning the log file from the last log entry, stepping backwards through the file.

`m_Username`: Name of the user who performed the operation.

`m_DifferenceOffset`: This is only used for check-ins. This is important, since it records the changes between two versions of a file. As detailed elsewhere in this document, if you take the current version of the file and apply the list of difference operations, you will get the previous version of the file. Continue applying each difference operation in reverse order — from newest to oldest — to recreate the original version of the file.

`m_FileReference`: This is the "aaaaaaaa" database file that was effected by the operation. This name is used in project log entries to identify the file that was effected. For example, if file "faaaaaaa" is added to project "daaaaaaa", "daaaaaaa" will contain an "Added File" log entry, with a reference to "faaaaaaa".

`m_BranchReference`: This only exists for a branch operation. When you branch a file, a new file is created in the database. This will contain a copy of the file that was branched, but does not have the history of log entries from the original. The branch reference indicates the original file that was branched. Use this to reference back to the original file so you can pull up the history of the original file from before the point it was branched.

There are quite a few other fields that may occur in a log entry. Refer to the code in VssScanLogEntry.cpp to more details.

Check Out File: "CF"

More details about the format of a checkout chunk can be found in VssScanCheckout.h. If you were to describe the layout of this chunk as a struct, it would look something like this:

```
    char  m_Username[32];
    DWORD m_CheckoutTime;
    char  m_Filename[260];
    char  m_Machine[32];
    char  m_Project[260];
    char  m_Comment[64];
    WORD  m_CheckoutVersion;
    WORD  m_CheckoutFlag;
    DWORD m_NextCheckout;
    WORD  m_Flags1;
    WORD  m_Flags2;
    WORD  m_CheckinVersion;
```

The log file for every file contains a checkout chunk at 0x01A0. When a new file is created (or an existing file is shared), the new checkout chunk is mostly zeroed out. It is not clear how SourceSafe handles files that have been checked out by multiple users. My suspicion is that it follows the pattern used by some other chunks: there is a 32-bit offset that indicates the position of another checkout chunk, with additional chunks being appended to the end of the file when it gets checked out by multiple people. Since there is a 4-byte location in the middle of the struct that is always zero, I've labelled that field as `m_NextCheckout`. This is a only a conjecture, since the test database did not contain any multiple checkouts.

`m_Project`: This indicates the SourceSafe project in which the file was checked out, which is useful for files that are shared across multiple projects. You can use this string to determine the shared location from which the file has been checked out.

`m_CheckoutVersion`: Records the current version of file when it was checked out. This is important when a file has been checked out by more than one person at a time. If the version number has changed when the file is checked in, the changes will need to be merged.

`m_CheckoutFlag`: Indicates whether the file is currently checked out. This field is 0x40 if the file is checked out, or zero if the file is not checked out. If multiple check-outs are possible, there may be other flags in this field that indicate this.

The other fields are commented up in the source code, and most of them are rather straightforward. It's not completely clear what some of the bitfield flags are for. But I have the suspicion that there are more flag bits that appear when files are being checked out across a network, and by multiple users.

Difference: "FD"

The difference chunk contains a set of simple blit commands that are used to roll back changes from the current file to the previous file. Only the current version of the file is kept in the database. If you apply the changes in the most recent difference chunk, you will convert the current version of the file to the previous version of the file. In turn, you can apply the next oldest difference chunk to *that* file to produce the version before it. Apply all of the difference chunks, and you end up with original file as it was when originally added to the database.

Each difference command starts with a 12-byte header:

```
    WORD  Opcode; // 0 = copy, 1 = insert, 2 = stop
    WORD  Junk;   // uninitialized memory?
    DWORD Start;  // starting offset for insert
    DWORD Count;  // byte count for replace or insert
```

Based on observation, I suspect the 16-bit junk word is simply uninitialized memory. It tends to be different for each difference chunk, but the same value gets used for each difference command in the same chunk, as if the same 12-byte block were being used to write each command, and this particular 16-bit word was left uninitialized.

The opcode indicates whether to copy or insert data. The following code shows how to use the difference data to apply changes to the newer version of a file to convert it back to the previous version, before the check-in operation occurred.

```
/////////////////////////////////////////////////////////////////////
//
//  ApplyDiff()
//
//  This will apply a set of differences to a file, converting it
//  into the previous version of the file.  This assumes that a copy
//  of the current file is stored in 'pNewFile'.  It will use the
//  difference data from the byte stream to transform 'pNewFile'
//  into the previous version of the file, which is written out to
//  'pFile'.
//
void ApplyDiff(FILE *pFile, u08 *pNewFile, BinaryReader &reader)
{
    u16 opcode = 0;

    do {
        // Read the opcode that indicates whether to insert,
        // copy, or stop.
        opcode = reader.Read16();

        // Next 16 bits is junk.  Ignore it.
        reader.Read16();

        // Then there are always a pair of values for offset
        // and count, even when they're not needed.
        u32 offset = reader.Read32();
        u32 count  = reader.Read32();
        // Insert 'count' bytes from the data stream.
        if (0 == opcode) {
            fwrite(reader.CurrentAddress(), 1, count, pFile);
            reader.Skip(count);
        }

        // Copy 'count' bytes from the 'pNewFile' array.
        else if (1 == opcode) {
            fwrite(pNewFile + offset, 1, count, pFile);
        }

        else {
            // The only other value is 2, which indicates
            // the end of the difference data.
        }

    } while (opcode < 2);
}
```

Branch File: "BF"

When a file is branched, there are several records of it. By way of example:

- Create file "foo.c" is in folder "bar\".
- Share "foo.c" into folder "blitz\".
- This adds a new parent chunk in "foo.c" that references "blitz\".
- A "shared file" log entry is added to "blitz\".
- A new reference to "foo.c" is added to the child list in "blitz\".
- Now do a branch operation on the "foo.c" in "blitz\".
- The parent chunk in "foo.c" that references "blitz\" will be reset to an empty string.
- A new file is created that contains current version of "foo.c".
- A "branch file" chunk is added to the original "foo.c" in "bar\" that references the new copy of "foo.c" in "blitz\".
- A "branched file" log entry is added to "blitz\" that references the new copy of "foo.c", along with a reference to the original "foo.c" that still exists in "bar\".

Note that the newly branched file does not contain the history from the original file. It records only those operations that are performed on the branched copy after the branch is performed. To locate the pre-branch history of the file, you need to look at the parent folder of the branched file, which contains a "branched file" log entry. This will allow you to reference back to the pre-branch file, which contains the rest of the history of the file.

Name Header: "HN"

This header appears at the beginning of the names.dat file. It is always 80 bytes long, and filled with zeroes. The only non-zero value is 16 bytes from the start of the data, where you file a 32-bit word that contains the current size of the file (probably to compute where to append any new Short Name to the file).

Since I had no need for the 8.3 short names, I did not pay much attention to the contents of names.dat (at least not after realizing it only contains the short names of files).

Short Name: "SN"

Short name entries are stored in names.dat. Each entry is between 19 and 21 bytes in length -- files with a 3-byte file extension are 21 bytes long, files with a 1-byte extension are only 19 bytes long.

If the original file name already fits within the 8.3 format, it will not have an entry in names.dat.

There are 8 extra bytes at the start of the chunk. It's not clear what these bytes contain, but may warrant investigation if you need to retain the short names for files.