CSC Format Specifications

By Cascade

(preliminary version)

2004

Ver. 0.9

Copyright and Disclaimer

* Work in progress. Expect this to appear/improve in later revisions.

1. Introduction

This document attempts do describe the file formats used by Deus Ex: Invisible War and Thief: Deadly Shadows for its audio.

2. Data structures

2.1 Files

In the "Deus Ex: Invisible War\Content\Dx2\Sounds\" folder you will find the following files:
- **SchemaMetafile_Harddrive.csc** - Contains sounds and information such as filename, where to find the data, subtitles.
- **SchemaMetafile_DVD1.csc** - Contains sounds.
- **SchemaMetafile_DVD2.csc** - Contains sounds.
- **SchemaMetafile_DVD3.csc** - Contains sounds.
- **SchemaMetafile.csc** - Unknown. Contains information about sounds but not where to find them.
- **SchemaMetafile_Memory.csc** - Empty file. This might be XBox related (XBox version of Harddrive?).

In the "Thief: Deadly Shadows\Content\T3\Sounds\" folder you will find the following files:
- **SchemaMetafile_Harddrive.csc** - Contains sounds and information such as filename, where to find the data, subtitles.
- **SchemaMetafile_DVD1.csc** - Contains sounds.
- **SchemaMetafile_DVD2.csc** - Contains sounds.
- **SchemaMetafile_DVD3.csc** - Contains sounds.
- **SchemaMetafile_Memory.csc** - Empty file. This might be XBox related (XBox version of Harddrive?).

2.2 Basic Types And Conventions

- The following basic types are used to describe the data structures:

```
char          8 bit signed integer
short         8 bit signed integer
int           16 bit signed integer
long          32 bit signed integer
NIBBLE        4 bit unsigned integer
BYTE          8 bit unsigned integer
WORD          16 bit unsigned integer
DWORD         32 bit unsigned integer
```

- All structures and members are little-endian and byte aligned unless stated otherwise.
- Hexadecimal numbers are shown using the '0x' prefix (example: 0x2F).
- Binary numbers use the 'b' suffix (example: 0101b).
- Data structures and code are presented in (pseudo) C.

2.3 String Type

A string is stored starting with its length:

```
struct SM_String
{
   DWORD dwLen;               // Length of the string, 0 is allowed.
   char  acData[dwLen];       // The string itself. The string is NOT zero terminated,
                              // and only exists when dwLen > 0.
};
```

2.4 Offset Type

The offset indicates both an identification for the file the offset is for and the actual file offset into that file.

```
struct SM_Offset
{
   DWORD dwFileAndOffset;
};
```

- (dwFileAndOffset & 0x1FFFFFFF) or the 29 least significant bits gives the offset into the file.
- (dwFileAndOffset & 0xE0000000) or the 3 most significant bits gives the identification of the file:

```
000b = SchemaMetafile.csc
001b = ??? (not used, possibly SchemaMetafile_Memory.csc)
010b = SchemaMetafile_Hardrive.csc
011b = SchemaMetafile_DVD1.csc
100b = SchemaMetafile_DVD2.csc
101b = SchemaMetafile_DVD3.csc
110b = ??? (not used)
111b = ??? (not used)
```

At this point it is not clear whether a 4:28 or a 3:29 scheme was used for file id and offset. The 3:29 is more intuitive because it makes the file ids consecutive. However, both scheme's will work since the current audio resource files of Deus Ex 2 are 'only' 150 MB in size. Once the files approach 256 MB, the 4:28 scheme will cause some trouble and 3:29 will be ok. Not an issue at the moment, but keep it in mind.

2.5 SchemaMetafile_Harddrive.csc File Format

2.5.1 File Structure

General structure of the SchemaMetafile_Harddrive.csc file:

```
struct SM_HarddriveFile
{
   BYTE              aUnknown[13];   // Unknown data.
   SM_TagTable       Tags;           // Table of all sound tags.
   SM_TaggedSounds   TSounds;        // Table of all sounds with tags, the 'tagged sounds'.
   SM_MissionSounds  MSounds;        // Table of all sounds without tags, the 'mission sounds'.
   SM_SoundInfoTable SoundInfo;      // Information about all sounds; filename, where to find it,
                                     // subtitles.
   BYTE              aData[];        // The Harddrive file contains a number of sounds itself.
};
```

So, how does this work? In short: the TaggedSounds and MissionSounds contain an identifier and a reference into the SoundInfo table for every sound. The SoundInfo table tells where to find the audio data for a sound, including its size, its filename and its subtitle for different languages.

2.5.2 Sound Tags

What is a Sound Tag? What can be derived from the text in the sound tags and the combination of tags of the tagged sounds section, is that a sound tag is used as a property for a sound. It describes what or who uses the sound, in what situation the sound should be used etc.

The table:

```
struct SM_TagTable
{
   BYTE     nNumTags;
   SM_Tag   aTags[nNumTags];
};
```

A sound tag and its subtags:

```
struct SM_Tag
{
   SM_String strTagName;                     // All Tag names begin with '+'.
   BYTE      ucData;                         // Unknown data.
   BYTE      nNumSubTags;                    // The number of SubTags for this SuperTag, can be 0.
   SM_String astrSubTagNames[nNumSubTags]; // All SubTags (only exists when nNumSubTags>0).
};
```

2.5.3 Tagged Sounds

Tagged sounds are sounds with additional properties. They contain a combination of the various sound tags.

An example, consider the first tagged sound in the table.
- The subtitle for the sound is: "This is your last chance!".
- It has two tags: "+reprimandplayer.final" and "+voice.orderguardfemale1".
 So, what we have here is:
- WHEN the sound should be used; reprimand the player one last time.
- WHO should use the sound; a female order guard type 1.

This system looks like it allows for an incredible powerful and efficient sound design. Very cool stuff!

The table:

```
struct SM_TaggedSounds
{
   WORD           nNumSounds;              // total number of tagged sounds.
   SM_TaggedSound aSounds[nNumSounds];     // the tagged sounds.
};
```

A tagged sound:

```
struct SM_TaggedSound
{
   BYTE      nNumTags;               // The number of tags for this sound.
   SM_TagRef aTagRefs[nNumTags];     // A table of the tags.
   SM_Offset Offset;                 // An offset into the SM_SoundInfoTable section of the file.
};
```

Reference to a sound tag:

```
struct SM_TagRef
{
   BYTE  nTag;                       // Index of the Tag in the SM_TagTable.
   DWORD nSubTag;                    // Index of the SubTag of the corresponding Tag.*
};
```

* An nSubTag member with value 0xFFFFFFFF indicates NO SUBTAG, it references the tag itself, not one of the subtags. A problem with this member is that a few tagged sounds have an nSubTag which is clearly out of range for the corresponding tag, so the current information regarding this is preliminary. Out of range tends to occur on sounds concerning material and movement properties.

2.5.4 Mission Sounds

Mission sounds are straightforward sounds with only a single string as an identifier. No sound tags.

The table:

```
struct SM_MissionSounds
{
   WORD nNumSounds;                        // total number of mission sounds.
   SM_MissionSound aSounds[nNumSounds];    // the mission sounds.
};
```

A mission sound:

```
struct SM_MissionSound
{
   SM_String strID;             // sound identifier string.
   SM_Offset Offset;            // An offset into the SM_SoundInfoTable section of the file.
};
```

2.5.5 Sound Information

The Sound Information tells Deus Ex 2 and T3 where to find the audio data. It is references by both the Tagged and the Mission Sounds. Note that a single Tagged/Mission Sound always refers to a single SM_SoundInfo. But, a SoundInfo can contain more than one sound!

The table:

```
struct SM_SoundInfoTable
{
   SM_SoundInfo aSounds[];
};
```

The actual size of this table is unknown. Use the offset provided by the Tagged- and Mission Sounds to find the correct SM_SoundInfo.

Sound Information:

```
struct SM_SoundInfo
{
   BYTE              aUnknown[45];                // Unknown data.
   BYTE              nNumFiles;                   // the number of sound files for this sound.
   SM_SoundFileInfo  aFileInfo[nNumFiles];        // sound file specific information.
};
```

Sound File Information:

```
struct SM_SoundFileInfo
{
   SM_String  strFilename;                 // Original filename.
   SM_Offset  Offset;                      // resourcefile id and offset into resourcefile. 1)
   DWORD      dwSizeWAV;                   // Size (bytes) of WAV file in resourcefile. 2)
   DWORD      dwSizeOGG;                   // Size (bytes) of OGG file in resourcefile. 2)
   DWORD      dwUnknown;                   // Unknown data.
   SM_String  strSubTitle_English;         // SubTitles in different languages.
   SM_String  strSubTitle_French;
   SM_String  strSubTitle_Italian;
   SM_String  strSubTitle_German;
   SM_String  strSubTitle_Spanish;
};
```

Notes:
 1) The offset can be 0xFFFFFFFF to indicate there is NO sound. In that situation, both dwSizeWAV and dwSizeOGG are 0 and the strFilename is "none".
 2) A sound is either a WAV or an OGG, so dwSizeWAV will be zero when it's an OGG and vice versa.

2.6 SchemaMetafile.csc File Format

Later. Since it is not important for extracting/inserting sounds, this will be added in a later revision of the document.

3. Extracting/Inserting Sounds

A. Microsoft WAV File Format

A RIFF/WAV file is build out of the same basic element every time. It consists of a collection of 'chunks'.

The basic chunk structure:

```
struct SChunk
{
   DWORD dwID;                  // four character chunk identification.
   DWORD dwSize;                // size (in bytes) of the data in the chunk.

   BYTE  aData[dwSize];         // the data of the chunk.
};
```

At the start of a WAV file there is a chunk which identifies the file, its size and its content. This chunk encapsulates a number of other chunks.

The top-level encapsulating structure:

```
struct SRIFFChunk
{
   DWORD dwID;               // identification "RIFF".
   DWORD dwSize;             // size of data = 4 + size of all encapsulated chunks.

   DWORD dwFormat;           // data format identification "WAVE"
};
```

The encapsulated chunks:

The format chunk specifies the audio format used in the audio data.

```
struct SFormatChunk
{
   // standard chunk data.
   DWORD dwID;               // identification "fmt ".
   DWORD dwSize;             // size of data = 16 for PCM (WAVEFORMAT) or
                             // 18 + wExtraParamsSize for non-PCM (WAVEFORMATEX).

   // wave format data.
   WORD  wAudioFormat;       // 0x0001 = PCM, 0x0069 = Deus Ex 2 and Thief 3 specific format.
   WORD  nNumChannels;       // number of audio channels (1 = mono, 2 = stereo, etc).
   DWORD nSampleRate;        // sample rate (samples per second).
   DWORD nAvgBytesPerSec;    // (average) byte rate (bytes per second).
   WORD  nBlockAlign;        // minimum size of an audio data block (bytes).
   WORD  nBitsPerSample;     // bits per sample.

   // optional data (for non-PCM).
   WORD  wExtraParamsSize;   // size (in bytes) of the extra parameters.
   BYTE  aExtraParams[wExtraParamsSize];
};
```

The fact chunk only exists for non-PCM formats and provides some additional information about the sound.
Precise format for the fact chunk data is unclear at the moment, but also not important.

```
struct SFactChunk
{
   DWORD dwID;               // identification "fact".
   DWORD dwSize;             // size of data.

   BYTE  aData[dwSize];      // the fact data.
};
```

The data chunk contains the actual audio data.

```
struct SDataChunk
{
   DWORD dwID;               // identification "data".
   DWORD dwSize;             // size of data.

   BYTE  aData[dwSize];      // the audio data.
};
```

Note that other types of chunks are possible, but are not common. Always check the id when reading a chunk...

B. Appendix B. DVI4/IMA ADPCM Audio Format

B.1 Deus Ex 2 and Thief 3 Audio Format

Deus Ex 2 and Thief 3 uses a mixture of different file types for audio. Two basic file types are used: Ogg Vorbis (.OGG files) and Wave (.WAV files).

Ogg Vorbis is an open source audio compression standard (actually: Ogg = the container format, Vorbis = the compression) which gives better sound quality at equal bitrate when compared to mp3. The format is getting increasingly popular by both consumers and (game) developers because of the quality and the free nature of the standard (no fees!). A lot of DX2 and T3 sounds are encoded in Ogg Vorbis. SDKs for encoding/decoding Ogg Vorbis files are freely available at the website of the developers; please see http://www.xiph.org/ for more information.
The wave format, or rather RIFF wave format is a standard developed by Microsoft/IBM. It supports not only the bog-standard PCM audio (uncompressed), but also several non-PCM audio (compressed) types. Microsoft Windows already has codecs for a number of these standards built in. Developers are allowed to use their own compression standard (and are in fact provided with the means to do so using the 'audio compression manager').

Guess what? Deus Ex 2 and Thief 3 seems to do just that. Besides some normal PCM wave files, it also uses wave file with audio format 0x0069, which is not one of the standard codecs provided by Microsoft. The format used is either the now obsolete 'VoxWare Byte Aligned' standard or Ion Storm custom audio codec. Considering the obsolete status of the first, a custom format is expected (note: specification of the VoxWare format were not found on the net, so it cannot be verified at this time).

What can be found out about this custom format?
- nBitsPerSample is always set to 4. This already indicates some sort of ADPCM compression method, since the existing methods (Microsoft and IMA) always compress from 16 bits to 4 (or 3) bits.
- nAvgBytesPerSec is always the same, depends only on samplerate, this suggests a fixed compression rate. Again: possibly ADPCM.
- wExtraParamsSize is always 2, extra parameters are always 0x0040 (64). This fits with IMA ADPCM specs which say the extra parameter is a WORD with the number of samples per block per channel. However, the value does NOT fit, the specs say they are supposed to fit the rule 'Samples%8==1' and they don't.

The primary suspect for the custom audio codec is now IMA ADPCM with some (slight) adjustments. Now, take a look inside the file DX2.exe version 1.0 at offset 0x005525F0. Guess what that is...this is the 'stepsizeTable' followed by the 'indexTable' exactly as used in the IMA ADPCM specifications. This confirms this audio codec is used.

There is a variant of IMA ADCPM called DVI4 or DVI ADPCM. It is basically the same but has one significant difference. IMA ADPCM uses this header in a block of audio data:

```
struct SIMABlockHeader
{
   int   nSample;            // first sample of the block.
   short nStep;              // index into 'stepsize table'.
   short nReserved;          // reserved, assumed 0.
};
```

Followed by groups of 8 samples (4 bytes) until the end of the block. As a consequence, an IMA block always contains a multiple of 8 samples per block PLUS ONE (the extra sample in the block header). DVI4 does not use this value in the header as an actual sample, but as a prediction. The block header becomes:

```
struct SDVI4BlockHeader
{
   int   nPredict;            // prediction used to calculate first sample.
   short nStep;               // index into 'stepsize table'.
   short nReserved;           // reserved, 0 when writing, ignore when reading.
};
```

Again followed by groups of 8 samples (4 bytes) until the end of the block. So here are always a multiple of 8 samples per block which fits nicely with the value provided by the extra parameter 0x0040.

DX2 and T3 Audio Format 0x0069 Specifications:
 - use adaptation of IMA ADPCM -> DVI4.
 - nBitsPerSample set to 4 bits.
 - wExtraParamsSize set to 2 bytes.
 - extra parameters are the a single word; the number of samples per block per channel, set to 64 samples.
 - nBlockAlign set to 36 (mono) or 72 (stereo) bytes.
 - 64 samples x 4 bits per sample = 32 bytes sample data per block (per channel).
 - for stereo, a block contains data for both channels interleaved on a 4 byte boundary like this:

```
 0          4           8              12             16             20             24
[ch0 header][ch1 header][ch0 8 samples][ch1 8 samples][ch0 8 samples][ch1 8 samples]…
```

This (de-)compression method gives good results on the sounds of Deus Ex 2 and Thief 3. All 0x0069 sounds can be decoded this way and they seem to be correct for the most part.

However...
Small problem: few sounds show some clipping and need to be investigated.

Big problem: the results for STEREO 0x0069 sounds are NOT satisfactory. Although the method appears to be correct, the resulting sounds exhibit extreme clipping (too often). Since there are only a few stereo 0x0069 sounds in the Deus Ex 2 and Thief 3 files, first thing to research is whether these sounds are actually used by the game or not. Expect an update on this in the coming weeks.

B.2 Decoding DVI4/IMA ADPCM

Variables/constants:

```
// quantizer lookup table.
const int stepsizeTable[89] =
{
   7     , 8     , 9     , 10    , 11    , 12    , 13    , 14    , 16    ,
   17    , 19    , 21    , 23    , 25    , 28    , 31    , 34    , 37    ,
   41    , 45    , 50    , 55    , 60    , 66    , 73    , 80    , 88    ,
   97    , 107   , 118   , 130   , 143   , 157   , 173   , 190   , 209   ,
   230   , 253   , 279   , 307   , 337   , 371   , 408   , 449   , 494   ,
   544   , 598   , 658   , 724   , 796   , 876   , 963   , 1060  , 1166  ,
   1282  , 1411  , 1552  , 1707  , 1878  , 2066  , 2272  , 2499  , 2749  ,
   3024  , 3327  , 3660  , 4026  , 4428  , 4871  , 5358  , 5894  , 6484  ,
   7132  , 7845  , 8630  , 9493  , 10442 , 11487 , 12635 , 13899 , 15289 ,
   16818 , 18500 , 20350 , 22385 , 24623 , 27086 , 29794 , 32767
};
```

```
// table of index changes.
const int indexTable[16] =
{
   -1, -1, -1, -1, +2, +4, +6, +8,
   -1, -1, -1, -1, +2, +4, +6, +8
};

Header           // block header, see SIMABlockHeader or SDVI4BlockHeader structs.
int newSample    // sample for output.
short index      // index into stepsizeTable.
int stepsize     // stepsize value.
NIBBLE sample    // current sample (NOTE: two samples of 4 bits are packed into a BYTE of
                 // the data block, process the low nibble first, then the high nibble).
int delta        // difference between consecutive (output) samples.
```

Pseudo-code for the algorithm:

```
// initial state.
get block header -> header
if(method==IMA)
{
   newSample = header.nSample
   output newSample
}
else if(method==DVI4)
   newSample = header.nPredict
index = header.nStep
stepsize = stepsizeTable[index]

// process the samples.
for every sample in the block
{
   // get sample.
   get sample -> sample

   // calculate delta value.
   delta = 0
   if(sample & 4)    delta += stepsize
   if(sample & 2)    delta += stepsize>>1
   if(sample & 1)    delta += stepsize>>2
   delta += stepsize>>3;
   if(sample & 8)    delta = -delta

   // calculate and clip output sample.
   newSample += delta
   if( newSample > 32767 )          newSample = 32767
   else if( newSample < -32768 )    newSample = -32768
   // NOTE: watch those 16 bit limits here -> better use temporary long to add the delta...

   // output the new sample.
   output new Sample

   // new stepsize.
   index += indexTable[sample]
   if( index < 0 )         index = 0
   else if( index > 88 )   index = 88
   stepsize = stepsizeTable[index]
}
```

B.3 Encoding DVI4/IMA ADPCM

Encoding will be explained in a later revision of this document.

Acknowledgements

- Ion Storm for creating the enjoyable game "Deus Ex - Invisible War" and "Thief: Deadly Shadows". And, of course all their other games, including Daikatana.

- The Deus Ex 2 community for their enthusiasm and efforts made for this game.

- The Xiph.org Foundation for their free multimedia standards (Ogg Vorbis) at www.xiph.org.

- Microsoft/IBM for the RIFF Wave file format.
- Microsoft/Intel/IMA for the DVI/IMA ADPCM algorithms.

- The Stanford University for providing a lecture on RIFF/WAVE at https://ccrma.stanford.edu/courses/422/projects/WaveFormat/.

- The University of North Carolina for providing a lecture on IMA ADPCM at http://www.cs.unc.edu/Courses/comp249-s03/lectures/comp249_s03_5/sld013.htm.

- The SoX Sound Exchange open source project at http://sox.sourceforge.net/ for providing a good implementation of IMA ADPCM as a reference.

- The "Connected: An Internet Encyclopedia" at http://www.freesoft.org/CIE/index.htm for providing some insight in the DVI4 ADPCM algorithm described in RFC 1890.

- Everyone I forgot.