How to Retrieve Text from a Binary .doc File

DIaLOGIKa/makz/math/wk/divo 4 March 2008

Contents

Introduction

This document explains how the text content in a binary .doc file can be identified and extracted.

The following descriptions and algorithms are based on the *Microsoft Word 2007 Binary File Format Documentation* and our own findings.

How Text is Stored in a .doc File

The binary .doc files use the structured storage format (aka *compound binary file format*) to save several streams in one file. The text and its attributes are stored in two of these streams.

Note: For information about the structured storage format please have a look at the "Windows Compound Binary File Format Specification"

Streams in a .doc File

A .doc file contains a stream called *WordDocument* consisting of a header part and a text part. The header, called *File Information Block* or *FIB*, contains information about the document and pointers into the text part and into other streams. The text part contains all text of the document (including footnotes, header and footer lines, etc.), but, not necessarily consecutively, i.e. it might be fragmented.

Another stream, called *0Table* or *1Table* (a flag in the *FIB* determines which name is actually used for this stream, see below) contains information about the fragmentation of the text part. This information is called *piece table*.

The *piece table*

The *piece table* is a data structure that describes the logical sequence of characters in the document: The text part in the *WordDocument* stream can be divided into several subparts or pieces. Each piece contains information about the encoding and the logical place in the text.

Word splits the text into several pieces if different encodings are used for different paragraphs or text runs, example:

```
Hello World
…some more CP1252 text…
αβγ – Greek is nice!
```

The text in this example would be divided into two pieces:

- The first piece contains the all the characters up to the Greek characters and is encoded using the codepage 1252.
- The second piece contains the Greek and remaining characters and has Unicode encoding

Note: We assume that Word does this switch between CP1252 and Unicode encoding due to optimization reasons. If the CP1252 text part is shorter than a certain (unknown) threshold all characters are Unicode-encoded.

Algorithm for Retrieving Text

To retrieve text from a document, the *piece table* has to be loaded and parsed and the pieces assembled in the right sequence.

The following algorithm assembles the text of the document and saves it in a string variable:

```
string text = "";
```

Loading the *FIB*

The *FIB* has a fixed length of 1472 bytes (Word 2003 and higher; earlier versions might have a smaller *FIB*) and starts at the first byte in the *WordDocument* stream of the .doc file. For the following examples, we assume that the streams of a .doc file can be treated as C# streams (we actually use our own assembly – called StructuredStorageReader – to access the structured storage file format. This assembly provides for a stream-like object).

```
Stream wordDocumentStream = new Stream("WordDocument");
byte[] fib = new byte[1472];
wordDocumentStream.Read(fib, 0, 1472);
```

Loading and Parsing the *piece table*

The *FIB* contains two variables (*FC/LCB* pair) that specify the beginning of the data structure holding the *piece table* (*FC* = file character position) and the length (*LCB* = long count of bytes). This information is placed in the table stream, i.e. the *FC* is a pointer to an address in that stream. Both values are 32 bit integer values and are stored at offset 0x01A2 and offset 0x01A6, respectively.

Note: The Word file format specification describes this FC/LCB pair as "Offset in table stream of beginning of information for complex files." It should be borne in mind that this is not correct since the piece table information is also used for non-complex files.

```
UInt32 fcClx = System.BitConverter.ToUInt32(fib, 0x01A2);
UInt32 lcbClx = System.BitConverter.ToUInt32(fib, 0x01A6);
```

To load the *piece table* we must determine how the table stream was named by Word. The *FIB* contains a flag which decides if the stream was saved as *0Table* or as *1Table*. Bit 0x0200 in word 0x000A of the *FIB* determines how the table stream is named:

```
bool flag1Table = ( (fib[0x000A] & 0x0200) == 0x0200);
string tableStreamName = "0Table";
if(flag1Table)
    tableStreamName = "1Table";
```

Excerpt from a hex dump of the *FIB*:

0x12F0 AND 0x0200: *1Table* is used

```
(0000)  EC A5 C1 00 7D 80 09 04 00 00 F0 12 BF 00 00 00   ....}...........
(0010)  00 00 00 10 00 00 00 00 00 08 00 00 DE 4C 00 00   .............L..
…
(0190)  00 00 A7 19 00 00 A2 02 00 00 49 1C 00 00 74 00   ..........I...t.
(01A0)  00 00 96 17 00 00 2D 00 00 00 00 00 00 00 00 00   ......-.........
(01B0)  00 00 00 00 00 00 00 00 00 00 55 15 00 00 00 00   ..........U.....
…
```

0x0000002D: complex information length in *1Table*

0x00001796: complex information position in *1Table*

Note: All the number values shown in this and the following hex dumps are in swapped byte order (“little endian”).

After that we load the data structure holding the *piece table* into the byte array *clx*:

```
Stream tableStream = new Stream(tableStreamName);
byte[] clx = new byte[lcbClx];
tableStream.Read(clx, fcClx, lcbClx);
```

The *clx* byte array can contain multiple substructures and only one of these substructures is the *piece table*. Each substructure starts with a byte which denotes the type of the substructure, followed by a value indicating the length of the substructure.

If the substructure describes a *piece table* the value of this byte is 2, otherwise 1. The length of the entry is a 32 bit value for a *piece table* and an 8 bit value for all other entries.

In order to identify the *piece table* in the *clx* byte array, the following algorithm can be used:

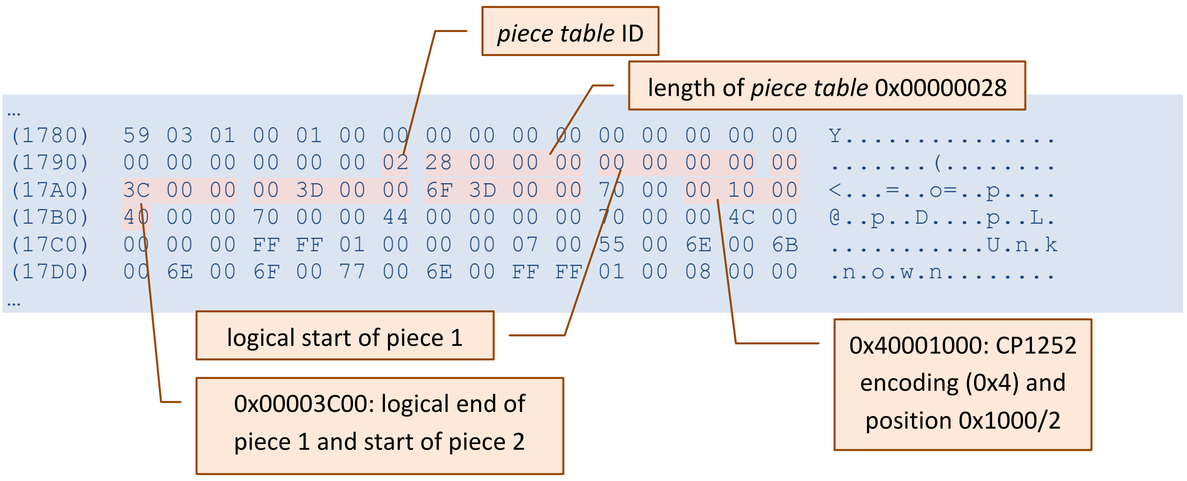
```
int pos = 0;
bool goOn = true;
while (goOn)
{
    byte typeEntry = clx[pos];

    if (typeEntry == 2)
    {
        //this entry is the piece table
        goOn = false;
        Int32 lcbPieceTable = System.BitConverter.ToInt32(clx, pos + 1);
        byte[] pieceTable = new byte[lcbPieceTable];
        Array.Copy(bytes, pos + 5, pieceTable, 0, pieceTable.Length);
    }
    else if (typeEntry == 1)
    {
        //skip this entry
        pos = pos + 1 + 1 + clx[pos + 1];
    }
    else
    {
        goOn = false;
    }
}
```

The *piece table* itself contains two arrays:

- The first array contains *n+1* logical character positions (*n* is the number of pieces). The entries are the logical start and end positions of the pieces in the text sequence, i.e. the first piece starts at logical position 1 and extends to position 2, the second starts at position 2, etc. Logical position *x* means that this is the *x-th* character in the document, i.e. this is not the file character position in the *WordDocument* stream. The positions are 32 bit values.
- The second array contains *n piece descriptor* structures. Each structure has a length of 8 bytes. The physical location of the piece inside of the *WordDocument* stream and the encoding of the text can be found in these 8 bytes from byte 3 to byte 6. This file character (*FC*) position is a 32 bit integer value. The second most significant bit is a flag that specifies the encoding of the piece: if the bit is set, the piece is CP1252-encoded and the *FC* is a word pointer; otherwise, the piece is Unicode-encoded and the FC is a byte pointer.

Hex dump from a *1Table* stream containing a *piece table* with 3 pieces:



```
…
(1780)  59 03 01 00 01 00 00 00 00 00 00 00 00 00 00 00  Y...............
(1790)  00 00 00 00 00 00 02 28 00 00 00 00 00 00 00 00  .......(........
(17A0)  3C 00 00 00 3D 00 00 6F 3D 00 00 70 00 00 10 00  <...=..o=..p....
(17B0)  40 00 00 70 00 00 44 00 00 00 00 70 00 00 4C 00  @..p..D....p..L.
(17C0)  00 00 00 FF FF 01 00 00 00 07 00 55 00 6E 00 6B  ...........U.n.k
(17D0)  00 6E 00 6F 00 77 00 6E 00 FF FF 01 00 08 00 00  .n.o.w.n........
…
```

piece table ID
length of *piece table* 0x00000028
logical start of piece 1
0x00003C00: logical end of piece 1 and start of piece 2
0x40001000: CP1252 encoding (0x4) and position 0x1000/2

```
int pieceCount = (lcbPieceTable - 4) / 12;

for (int i = 0; i < pieceCount; i++)
{
    //get the position
    Int32 cpStart = System.BitConverter.ToInt32(pieceTable, i * 4);
    Int32 cpEnd = System.BitConverter.ToInt32(pieceTable, (i+1) * 4);

    //get the descriptor
    byte[] pieceDescriptor = new byte[8];
    int offsetPieceDescriptor = ((pieceCount +1)*4) + (i*8);
    Array.Copy(pieceTable, offsetPieceDescriptor, pieceDescriptor, 0, 8);
}
```

The interpretation of the encoding flag and the calculation of the *FC* pointer are as follows:

```
UInt32 fcValue = System.BitConverter.ToUInt32(pieceDescriptor, 2);
bool isANSI = ( (fcValue & 0x40000000) == 0x40000000);
Int32 fc = fcValue & 0xBFFFFFFF;

Encoding encoding = Encoding.GetEncoding(1252);
Int32 cb = cpEnd - cpStart;
if (!isANSI)
{
    encoding = Encoding.Unicode;
    cb *= 2;
}
```

Now, the text which is described by that piece can be appended to our text string as follows:

```
byte[] bytesOfText = new byte[cb];
wordDocumentStream.Read(bytesOfText, bytesOfText.Length, fc);
text += encoding.GetString(bytesOfText);
```

Iterating over all the pieces in the *piece table* will finally append all the text in the document to our text string.

Note: It should be borne in mind that the text of header and footer lines, footnotes, endnotes, etc. is also stored inside the text part of the WordDocument stream; consequently, our text string will also contain the text of these elements.