Liberating data from Clarion TPS files

Migrating a customer from a legacy system to his shiny new one never goes smoothly. In this case the customer had a huge amount of data in a Clarion based system with the tables stored in proprietary TPS (TopSpeed) files. In this blog post I'll discuss the various ways of liberating data from their TPS container and dive deep into the internals of the TPS file itself. Finally I present a TPS to CSV converter.

Existing Clarion Tooling

A quick browse on the internet revealed that I was not the only one looking for a way to read data from TPS files. There were no libraries or file format specifications that I could find. However Clarions vendor, Soft Velocity, provides some tooling to work with TPS files.

The most obvious you'll encounter is the TPS ODBC driver. You'll have to buy it of course. However, as I could not find any documentation on how to use it, I skipped this option. Then I found out that together with the Clarion product comes a tool called TOPSCAN which you can use to view, edit and export a TPS file as a delimited file. Great! Now where to get this utility. As said it's part of Clarion, but buying the whole product for a conversion utility is a bit expensive. Fortunately our customer had an existing Clarion development setup so we could use his.

Getting the data out using TOPSCAN involves a lot of clicking, first you have to make sure you've got all the columns visible, then you need to do the export. You're best off with the defaults as the exporter dialog keeps its settings but doesn't apply them across exports so the first export may be correct, but the 2nd is always in default. Also the the export doesn't properly escape any quotes and comma's, so importing it into another tool is .. annoying.. requiring manual fixes.

Strangely enough, for some files, TOPSCAN would display only the column information and not the data itself, while the file was clearly not empty. After a bit of experimentation we found out that those files where from an older version of Clarion, and could only be opened with that version of the software. Oh dear.. All in all, while it is possible to get the data out using existing tooling, its not exactly a smooth ride. So I decided to take it a step further, and see if I could write a tool to parse TPS files and do the conversion automatically.

binary mode enabled

Reverse engineering TPS files

If you open a TPS file in your favorite hex-editor you'll notice the string 'tOpS' at position 0x0E which identifies the file as a TopSpeed one.The first 0x0200 bytes form the header of the file as indicated by the length value at position 0x04. You'll also notice that the format uses the so called little endian notation with the least significant byte first (so 0x0200 would be written as 0x00 and 0x02, whereas big endian is 0x02 and 0x00, which is directly readable for humans). Which leaves us the four mysterious 0x00's at the start of the file.

```
00000000h: 00 00 00 00 00 02 00 DB 05 00 00 DB 05 00 74 4F ; .......Û...Û..tO
00000010h: 70 53 00 00 00 00 13 B8 23 3D 00 00 07 00 00 00 ; pS…..¸#=......
```

As it turns out, those four zero's indicate the position in the file. If you scan through the whole file you'll find plenty of other locations where the value matches the file position (all at 0x0100 boundaries), followed by a two byte length value. This is the main mechanism how TopSpeed organizes its pages. However, one would expect a

kind of index to these pages so that you don't have to scan the whole file (or accidentally misinterpret a value). Well, there is some index, although its not complete. Its in the header as well and pretty obscure.

```
00000020h: 00 00 00 00 00 00 00 00 0D 00 00 00 1B 02 00 00 ; ................
00000030h: E7 03 00 00 E7 04 00 00 72 05 00 00 A1 05 00 00 ; ç...ç...r…¡...

00000110h: 00 00 00 00 0D 00 00 00 1B 02 00 00 E6 03 00 00 ; ............æ...
00000120h: E7 04 00 00 6E 05 00 00 9C 05 00 00 BD 05 00 00 ; ç...n…œ...½...
```

The header contains two arrays of 4 byte values containing references to the start and end of a block of pages. Some arithmetic needs to be applied though. First, the value needs to be shifted 8 bits to the left and the length of the header must be added. This will get you a start and end address for each combination (in this case, at position 1 in the array : start 0x00000000 -> 0x00000200 and end 0x000000D -> 0x00000F00) so the first block runs from 0x0200 to 0x0F00. Let's have a look at at that:

```
00000200h: 00 02 00 00 FD 00 FD 00 1D 07 71 00 00 C0 0B 00 ; ....ý.ý...q..À..

00000300h: 00 03 00 00 FD 00 FD 00 1D 07 71 00 00 C0 0B 00 ; ....ý.ý...q..À..

.. etc ..

00000900h: 00 09 00 00 36 00 36 00 4C 00 04 00 02 17 04 00 ; ....6.6.L…....
```

You'll notice that the block starts with a page at address 0x00000200 of length 0x00FD. The next page in the block is at address 0x00000300 and so on. There appears to be no index or other management information about the pages inside a block. I wonder how it is kept track of.

Pages in a TPS File

So far we've examined the Header, Blocks and we know how to find Pages in a TPS file. Lets examine the pages further, because that's where the data is. In the previous examples, the length was always followed by an identical value. However for block filled with record data their values differ. Could there be some compression algorithm present and does the second value indicate the length of the uncompressed page? It appears it does. For example:

```
00001b00h: 00 1B 00 00 BE 01 25 12 91 12 0A 00 00 05 C0 D5 ; ....¾.%.‘.....ÀÕ
00001b10h: 01 09 00 03 07 01 F3 00 00 02 C3 00 04 01 20 31 ; ......ó...Ã... 1
00001b20h: 01 00 03 01 20 31 03 00 00 20 60 03 00 00 20 22 ; .... 1… `... "
00001b30h: 01 00 03 01 20 A9 01 02 00 20 27 0B 08 C4 3B 02 ; .... ©... '..Ä;.
```

Here, the page length (0x01BE) is different from the uncompressed length (0x1225) indicating that the page is compressed. The actual algorithm used is a form of Run Length Encoding (RLE). This particular implementation alternates between copying bytes and repeating them. The data starts at 0x001b0E with 0x05 bytes to copy, so 0xC0 0xD5 0x01 0x09 0x00 are copied as is. The next value is 0x03 which indicates that the previous byte should be repeated 0x03 times, so 0x00 0x00 0x00, then the next value is 0x07 indicating that a further 0x07 bytes should be copied as-is. And so on and so on.

This is a pretty effective high performance algorithm on data with a lot of blanks.

Sometimes it happens that much more bytes need to be repeated or copied that fits in the single byte used to encode it. In that case a special extension mechanism is used marked by the 0x80 bit of the byte. If set, another byte follows which holds the upper 8 bits of a 15 (!) bit value. (the marker bit is skipped).

So this is a page, lets have a look what is inside.

TPS Records

Pages are populated by records (finally!) which use an interesting mechanism to save some (more) bytes. As the header of a record is often identical to the next record, its possible for records to 'borrow' each others header information. This is indicated by the first byte of the record. If its value 0xC0 this is a complete record. The highest two bytes indicate if length information is present, otherwise its needs to be copied from the previous record. The lower bits indicate how many other bytes need to be copied from the previous record as well.

```
C0 D5 01 09 00 00 00 00 01 F3 00 00 02 C3 00 ..
```

The first length byte is the whole record (0x01D5) , the second byte is the length of the header (0x0009). The header indicates the kind of record. In this case type 0xF3 is a data record with the following bytes indicating the record number (0x000002C3) in big-endian notation! The rest of the data inside the record is actually the row data of the record. Now we only need to be able to understand the values.

There are several other record types, such as memo (0xFC) and table definition (0xFA) records. The table definition records are interesting because they describe the format of the data inside the data records. Of course, this requires some more byte guessing. And then I found this page. It describes most of the TPS format (and specifically how to read the table definition records) in Russian. Mmm. ¡google translate al rescate!

It took a bit of effort to understand but its quite good. Table definitions consist of column, index and memo definitions. Column definitions map directly on the record data using an offset and a length and a data type (how to read the bytes). Indexes are built up using the column values by referencing them and memo's are stored separately.

There are some data types that were present in my data files, but not explained in the Russian page. So I had figure them out myself. These are the types 0x04 (date) and 0x05 (time). Both use a mask encoding to store individual values in a 4 byte integer:

```
Time : 0xHHmm????
Date : 0xyyyyMMdd
```

Strangely enough there is another date encoding possible as well, but those are stored as normal integer values and not as (some form of) dates. After some searching I found out that they hold the number of days since 1800-12-28.

A TPS to CSV converter

After all this reverse engineering I've built a tool that extracts record and memo information out of an given TPS file and stores this as a (properly escaped) CSV file. I've licensed it under the Apache 2 license, so feel free to make code adjustments yourself. You can also use it as a library for writing your own tools. If you come across any record types that are not understood, please send me a copy of the TPS file if possible. If you decide to use this tool for migrating data, do double check the results, as its all based on reverse engineering and may not be accurate or complete.

Checkout the Sourcecode on Github.

TAGS: Java clarion tps

*Jan '13*21 Erik Hooijmeijer's foto Author Erik Hooijmeijer
(Show all articles by Erik Hooijmeijer)