SAS7BDAT Database Binary Format

by:

Matthew S. Shotwell, PhD
Assistant Professor
Department of Biostatistics
Vanderbilt University
matt.shotwell@vanderbilt.edu

1/9/2013 update (**u64** format extensions, Row Size fields, and RLE compression) by:

Clint Cummins, PhD
clint@stanford.edu

Contents

Introduction

The SAS7BDAT file is a binary database storage file. At the time of this writing, no description of the SAS7BDAT file format was publicly available. Hence, users who wish to read and manipulate these files were required to obtain a license for the SAS software, or third party software with support for SAS7BDAT files. The purpose of this document is to promote interoperability between SAS and other popular statistical software packages, especially R (http://www.r-project.org/).

The information below was deduced by examining the contents of many SAS7BDAT databases downloaded freely from internet resources (see `data/sas7bdat.sources.RData`). No guarantee is made regarding its accuracy. No SAS software, nor any other software requiring the purchase of a license was used.

SAS7BDAT files consist of binary encoded data. Data files encoded in this format often have the extension '.sas7bdat'. The name 'SAS7BDAT' is not official, but is used throughout this document to refer to SAS database files formatted according to the descriptions below.

There are significant differences in the SAS7BDAT format depending on the operating systems and computer hardware platforms (32bit vs. 64bit). See the section on platform differences for more details. The format described below is sufficient to read the entire collection of test files referenced in `data/sas7bdat.sources.RData` (i.e. files associated with 32bit and some 64bit builds of SAS for Microsoft Windows, and **u64** SAS versions). This includes files created with COMPRESS=CHAR. The format described here is probably not sufficient to **write** SAS7BDAT format files, due to lingering uncertainties.

The figure below illustrates the overall structure of the SAS7BDAT database. Each file consists of a header (length := HL bytes), followed by PC pages, each of length PL bytes (PC and PL are shorthand for 'page count' and 'page size' respectively, and are used to denote these quantities throughout this document).:

```
----------
|~~~HL~~~|~~header
----------
|~~~PL~~~|~~page~1
----------
|~~~PL~~~|~~page~2
----------
~~~...
----------
|~~~PL~~~|~~page~PC
----------
```

Throughout this document, hexadecimal digits are denoted with a preceding 'x', binary digits with a preceding 'b', and decimal digits with no preceding character. For example, see the below table of hexadecimal, decimal, and binary values.

SAS7BDAT Header

The SAS7BDAT file header contains a binary file identifier (*i.e.*, a magic number), the dataset name, timestamp, the number pages (PC), their size (PL) and a variety of other values that pertain to the database as a whole. The purpose of many header fields remain unknown, but are likely to include specifications for data compression and encryption, password protection, and dates/times of creation and/or modification. Most files encountered encode multi-byte values little-endian (least significant byte first). However, some files have big-endian values. Hence, it appears that multi-byte values are encoded using endianness of the platform where the file was written. See Platform Differences for a table of key test files which differ in several ways.

The *offset table* below describes the SAS7BDAT file header as a sequence of bytes. Information stored in the table is indexed by its byte offset (first column) in the header and its length (second column) in bytes. For example, the field at offset 0 has length 32 bytes. Hence, bytes 0,1,...,31 comprise the data for this field. Byte lengths having the form '%n' should read: 'the number of bytes remaining up to, but not including byte n'. The fourth column gives a shorthand description of the data contained at the corresponding offset. For example, 'int, page size := PL' indicates that the data stored at the corresponding location is a signed integer representing the page size, which we denote PL. The description *????????????* indicates that the meaning of data stored at the corresponding offset is unknown. The third column represents the author's confidence (low, medium, high) in the corresponding offset, length, and description. Each offset table in this document is formatted in a similar fashion. Variables defined in an offset table are sometimes used in subsequent tables.

Header Offset Table

offset	length	conf.	description
0	32	high	binary, magic number
32	1	high	binary, Alignment: if (byte==x33) a2=4 else a2=0 . **u64** is true if a2=4 (unix 64 bit format).
33	2	low	*????????????*
35	1	high	binary, Alignment if (byte==x33) a1=4 else a1=0
36	1	low	*????????????*
37	1	high	int, endianness (x01-little [Intel] x00-big)
38	1	low	*????????????*
39	1	medium	ascii, OS type (1-UNIX or 2-WIN). Does not affect format except for the OS strings.
40	8	low	*????????????*
48	8	low	*????????????*
56	8	low	repeat of 32:32+8
64	20	low	*????????????*
84	8	high	ascii 'SAS FILE'
92	64	high	ascii, dataset name
156	8	medium	ascii, file type, e.g. `'DATA ~ ~'`
164	a1	medium	zero padding when a1=4 . Aligns the double timestamps below on double word boundaries.
164+a1	8	high	double, timestamp, date created, secs since 1/1/60 (for SAS version 8.x and higher)
172+a1	8	high	double, timestamp, date modified, secs since 1/1/60 (for SAS version 8.x and higher)
180+a1	16	low	*????????????*
196+a1	4	high	int, length of SAS7BDAT header := HL . (1024 or 8192)
200+a1	4	high	int, page size := PL
204+a1	4+a2	high	int, page count := PC . Length 4 or 8 (**u64**), henceforth denoted **4|8**
208+a1+a2	8	low	*????????????*
216+a1+a2	8	high	ascii, SAS release (e.g. 9.0101M3)
224+a1+a2	16	high	ascii, host (SAS server type, longest observed string has 9 bytes)
240+a1+a2	16	high	ascii, OS version number (for UNIX, else null)
256+a1+a2	16	high	ascii, OS maker or version (SUN, IBM, sometimes WIN)
272+a1+a2	16	high	ascii, OS name (for UNIX, else null)
288+a1+a2	32	low	*????????????*
320+a1+a2	4	low	int, page sequence signature? (value is close to the value at start of each Page Offset Table)
324+a1+a2	4	low	*????????????*
328+a1+a2	8	medium	double, 3rd timestamp, sometimes zero
336+a1+a2	%HL	medium	zeros
1024|8192		medium	Total length of header (8192 for **u64**), HL

The 8 bytes beginning at offset 32 hold information which affects the offset of the 'release' and 'host' information. In particular:

1. The byte at offset 32 defines the **u64** (unix 64 bit) file format, which affects many field and header lengths (usually via 4 vs. 8 byte integers).
2. The byte at offset 35 controls an offset before the timestamps.

3. The byte at offset 37 defines byte ordering of ints and doubles (most test files were created on Windows and use Intel byte ordering; little endian).

4. The byte at offset 39 appears to distinguish the OS type, where '1' indicates that the file was generated on a UNIX-like system, such as Linux or SunOS, and '2' indicates the file was generated on a Microsoft Windows platform. However, this does not affect any important fields in the file format.

The following table describes some of the possible polymorphisms for the 8 bytes at offset 32. The first field lists the name of the file where the sequence was found (see `data/sas7bdat.sources.RData`), the second lists the eight byte values (hexadecimal), the third field shows bytes 216-239 in ASCII ('.' represents a non-ASCII character or '0'), and the fourth field lists the SAS7BDAT sub-format.

filename	**bytes 32-39**	**bytes 216-239**	**format**
`compress_no.sas7bdat`	`x22 x22 x00 x32 x22 x01 x02 x32`	`9.0101M3NET_ASRV........`	Windows Intel
`compress_yes.sas7bdat`	`x22 x22 x00 x32 x22 x01 x02 x32`	`9.0101M3NET_ASRV........`	Windows Intel
`lowbwt_i386.sas7bdat`	`x22 x22 x00 x32 x22 x01 x02 x32`	`9.0202M0W32_VSPRO.......`	Windows Intel
`missing_values.sas7bdat`	`x22 x22 x00 x32 x22 x01 x02 x32`	`9.0202M0W32_VSPRO.......`	Windows Intel
`obs_all_perf_1.sas7bdat`	`x22 x22 x00 x32 x22 x01 x02 x32`	`9.0101M3XP_PRO..........`	Windows Intel
`adsl.sas7bdat`	`x22 x22 x00 x33 x33 x01 x02 x32`	`....9.0202M3X64_ESRV....`	Windows x64 Intel
`eyecarex.sas7bdat`	`x22 x22 x00 x33 x22 x00 x02 x31`	`....9.0000M0WIN.........`	Unix non-Intel
`lowbwt_x64.sas7bdat`	`x22 x22 x00 x33 x33 x01 x02 x32`	`....9.0202M2X64_VSPRO...`	Windows x64 Intel
`natlterr1994.sas7bdat`	`x33 x22 x00 x33 x33 x00 x02 x31`	`........9.0101M3SunOS...`	u64 Unix non-Intel
`natlterr2006.sas7bdat`	`x33 x22 x00 x33 x33 x00 x02 x31`	`........9.0101M3SunOS...`	u64 Unix non-Intel
`txzips.sas7bdat`	`x33 x22 x00 x33 x33 x01 x02 x31`	`........9.0201M0Linux...`	u64 Unix Intel

The binary representation for the hexadecimal values present in the table above are given below.

hexadecimal	**decimal**	**binary**
`x01`	`001`	`b00000001`
`x02`	`002`	`b00000010`
`x22`	`034`	`b00010010`
`x31`	`049`	`b00011001`
`x32`	`050`	`b00011010`
`x33`	`051`	`b00011011`

Alignment

In files generated by 64 bit builds of SAS, 'alignment' means that all data field offsets containing doubles or 8 byte ints should be a factor of 8 bytes. For files generated by 32 bit builds of SAS, the alignment is 4 bytes. Because SAS7BDAT Packed Binary Data may contain double precision values, it appears that all data rows are 64 bit aligned, regardless of whether the file was written with a 32 bit or 64 bit build of SAS. Alignment of data structures according to the platform word length (4 bytes for 32 bit, and 8 bytes for 64 bit architectures) facilitates efficient operations on data stored in memory. It also suggests that parts of SAS7BDAT data file format are platform dependent. One theory is that the SAS implementation utilizes a common C or C++ structure or class to reference data stored in memory. When compiled, these structures are aligned according to the word length of the target platform. Of course, when SAS was originally written, platform differences may not have been forseeable. Hence, these inconsistencies may not have been intentional.

Magic Number

The SAS7BDAT magic number is the following 32 byte (hex) sequence.:

```
x00~x00~x00~x00~~~x00~x00~x00~x00
x00~x00~x00~x00~~~xc2~xea~x81~x60
xb3~x14~x11~xcf~~~xbd~x92~x08~x00
x09~xc7~x31~x8c~~~x18~x1f~x10~x11
```

In all test files except one (not listed in `data/sas7bdat.sources.RData`), the magic number above holds. The one anomalous file has the following magic number:

```
x00~x00~x00~x00~~~x00~x00~x00~x00
x00~x00~x00~x00~~~x00~x00~x00~x00
x00~x00~x00~x00~~~x00~x00~x00~x00
x00~x00~x00~x00~~~x18~x1f~x10~x11
```

In addition, the anomalous file is associated with the SAS release “3.2TK”. Indeed, this file may not have been written by SAS. Otherwise, the anomalous file appears to be formatted similarly to other test files.

SAS7BDAT Pages

Following the SAS7BDAT header are pages of data. Each page can be one of (at least) four types. The first three are those that contain meta-information (e.g. field/column attributes), packed binary data, or a combination of both. These types are denoted 'meta', 'data', and 'mix' respectively. Meta-information is required to correctly interpret the packed binary information. Hence, this information must be parsed first. In test files, 'meta' and 'mix' pages always precede 'data' pages. In some test data files, there is a fourth page type, denoted 'amd' which appears to encode additional meta information. This page usually occurs last, and appears to contain amended meta information.

The page offset table below describes each page type. Byte offsets appended with one of '(meta/mix)', '(mix)', or '(data)' indicate that the corresponding length and description apply only to pages of the listed type. Provisionally, the internal structure of the 'amd' page type is considered identical to the 'meta' page type.

Page Offset Table

offset	**length**	**conf.**	**description**
0	4	low	int, page sequence signature?
4	12|28	low	*????????????* length 12 or 28 (**u64**)
B	2	medium	int, bit field page type := _PGTYPE; B = 16|32
B+2	2	medium	int, data block count := BC
B+4	2	medium	int, subheader pointers count := SC <= BC
B+6	2	low	*????????????*
B+8	SC*SL	medium	SC subheader pointers, SL = 12|24

... continued on next page

offset	length	conf.	description
B+8+SC*SL	DL	medium	if NRD>0, 8 byte alignment; DL = (B+8+SC*SL+7) % 8 * 8
B+8+SC*SL+DL	RC*'RL'_	medium	SAS7BDAT packed binary data data row count := RC = (BC-SC)
C	%'PL'_	medium	subheader data and/or filler; C = (B+8+SC*SL+DL+RC*RL)

Page Type

PGTYPE	name	subheaders	uncompressed row data (after subheaders)	compressed row data (in subheaders)
0	meta	yes (SC>0)	no (BC=SC)	yes
256	data	no (SC=0)	yes (RC=BC)	no
512	mix	yes (SC>0)	yes (RC=BC-SC)	no
1024	amd	yes?	yes?	no?
16384	meta	yes (SC>0)	no (BC=SC)	yes
-28672	comp	no	no	no

There are at least four page types 'meta', 'data', 'mix', and 'amd'. These types are encoded in the most significant byte of a two byte bit field at page offset 16|32. If no bit is set, the following page is of type 'meta'. If the first, second, or third bits are set, then the page is of type 'data', 'mix', or 'amd', respectively. Hence, if the two bytes are interpreted as an unsigned integer, then the 'meta', 'data', 'mix', and 'amd' types correspond to 0, 256, 512, and 1024, respectively. In compressed files, other bits (and sometimes multiple bits) have been set (e.g., `1 << 16 | 1 << 13`, which is `-28672` signed, or `36864` unsigned). However, the pattern is unclear.

If a page is of type 'meta', 'mix', or 'amd', data beginning at offset byte 24|40 are a sequence of SC SL-byte subheader pointers, which point to an offset farther down the page. SAS7BDAT Subheaders stored at these offsets hold meta information about the database, including the column names, labels, and types. If a page is of type 'mix', then **packed binary data begin at the next 8 byte boundary following the last subheader pointer**. In this case, the data begin at offset B+8+SC*SL+DL, where DL = (B+8+SC*SL+PL+7) % 8 * 8, and '%' is the modulo operator.

If a page is of type 'data', then packed binary data begin at offset 24|40.

The 'comp' page was observed as page 2 of the compress_yes.sas7bdat test file (not distributed with the `sas7bdat` package). It has BC and SC fields, but no subheader pointers. It contains some initial data and 2 tables. The first table has many rows of length 24; its purpose is unknown. The second table has one entry per data page with the page number and the number of data rows on the page for SC pages. It could be used to access a particular row without reading all preceding data pages.

Subheader Pointers

The subheader pointers encode information about the offset and length of subheaders relative to the beginning of the page where the subheader pointer is located. The purpose of the last four bytes of the subheader pointer are uncertain, but may indicate that additional subheader pointers are to be found on the next page, or that the corresponding subheader is not crucial.

offset	length	conf.	description
0	4|8	high	int, offset from page start to subheader

... continued on next page

offset	length	conf.	description
4|8	4|8	high	int, length of subheader := QL
8|16	1	medium	int, compression := COMP
9|17	1	low	int, subheader type := ST
10|18	2|6	low	zeroes
12|24		high	Total length of subheader pointer 12|24 (**u64**), SL

QL is sometimes zero, which indicates that no data is referenced by the corresponding subheader pointer. When this occurs, the subheader pointer may be ignored.

COMP	description
0	uncompressed
1	truncated (ignore data)
4	RLE compressed row data with control byte

ST	subheaders
0	Row Size, Column Size, Subheader Counts, Column Format and Label, in Uncompressed file
1	Column Text, Column Names, Column Attributes, Column List
1	all subheaders (including row data), in Compressed file.

SAS7BDAT Subheaders

Subheaders contain meta information regarding the SAS7BDAT database, including row and column counts, column names, labels, and types. Each subheader is associated with a four- or eight-byte 'signature' (**u64**) that identifies the subheader type, and hence, how it should be parsed.

Row Size Subheader

The row size subheader holds information about row length (in bytes), their total count, and their count on a page of type 'mix'. Fields at offset 28|56 and higher are not needed to read the file, but are documented here for completeness. The four test files used for example data in the higher fields are `eyecarex.sas7bdat`, `acadindx.sas7bdat`, `natlterr1994.sas7bdat`, `txzips.sas7bdat` (non-Intel/Intel x regular/u64).

offset	length	conf.	description
0	4|8	high	binary, signature xF7F7F7F7|xF7F7F7F700000000
4|8	16|32	low	*????????????*
20|40	4|8	high	int, row length (in bytes) := RL
24|48	4|8	high	int, total row count := TRC
28|56	8|16	low	*????????????*
36|72	4|8	medium	int, number of Column Format and Label Subheader on first page where they appear := NCFL1
40|80	4|8	medium	int, number of Column Format and Label Subheader on second page where they appear (or 0) := NCFL2
44|88	8|16	low	*????????????*
52|104	4|8	medium	int, page size, equals PL
56|112	4|8	low	*????????????*
60|120	4|8	medium	int, max row count on "mix" page := MRC

... continued on next page

offset	length	conf.	description
64|128	8|16	medium	sequence of 8|16 FF, end of initial header
72|144	148|296	medium	zeroes
220|440	4	low	int, page sequence signature (equals current page sequence signature)
224|444	40|68	low	zeroes
264|512	4|8	low	int, value 1 observed in 4 test files
268|520	2	low	int, value 2 observed
270|522	2|6	low	zeroes (pads length of 3 fields to 8|16)
272|528	4|8	medium	int, number of pages with subheader data := NPSHD
276|536	2	medium	int, number of subheaders with positive length on last page with subheader data := NSHPL
278|538	2|6	low	zeroes
280|544	4|8	low	int, values equal to NPSHD observed
284|552	2	low	int, values equal to NSHPL+2 observed
286|554	2|6	low	zeroes
288|560	4|8	medium	int, number of pages in file, equals PC
292|568	2	low	int, values 22,26,9,56 observed
294|570	2|6	low	zeroes
296|576	4|8	low	int, value 1 observed
300|584	2	low	int, values 7|8 observed
302|586	2|6	low	zeroes
304|592	40|80	low	zeroes
344|672	2	low	int, value 0
346|674	2	low	int, values 0|8
348|676	2	low	int, value 4
350|678	2	low	int, value 0
352|680	2	low	int, values 12,32|0
354|682	2	low	int, length of Creator Software string := LCS
356|684	2	low	int, value 0
358|686	2	low	int, value 20
360|688	2	low	int, value of 8 indicates MXNAM and MXLAB valid := IMAXN
362|690	8	low	zeroes
370|698	2	low	int, value 12
372|700	2	low	int, value 8
374|702	2	low	int, value 0
376|704	2	low	int, value 28
378|706	2	low	int, length of Creator PROC step name := LCP
380|708	36	low	zeroes
416|744	2	low	int, value 4
418|746	2	low	int, value 1
420|748	2	low	int, number of Column Text subheaders in file := NCT
422|750	2	low	int, max length of column names := MXNAM (see IMAXN)
424|752	2	low	int, max length of column labels := MXLAB (see IMAXN)
426|754	12	low	zeroes
438|766	2	medium	int, number of data rows on a full page INT[(PL - 24 / 40)/RL]; 0 for compressed file

... continued on next page

offset	length	conf.	description
440|768	27	low	zeroes
467|795	1	low	int, bit field, values 1,5
468|796	12	low	zeroes
480|808		medium	Total length of subheader, QL

Column Size Subheader

The column size subheader holds the number of columns (variables).

offset	length	conf.	description
0	4|8	high	binary, signature xF6F6F6F6|xF6F6F6F600000000
4|8	4|8	high	int, number of columns := NCOL
8|16	4|8	low	*????????????* usually zeroes
12|24		medium	Total length of subheader, QL

Subheader Counts Subheader

This subheader contains information on the first and last appearances of at least 7 common subheader types. Any of these subheaders may appear once or more. Multiple instances of a subheader provide information for an exclusive subset of columns. The order in which data is read from multiple subheaders corresponds to the reading order (left to right) of columns. The structure of this subheader was deduced and reported by Clint Cummins.

offset	length	conf.	description
0	4|8	high	int, signature -1024 (x00FCFFFF|x00FCFFFFFFFFFFFF)
4|8	4|8	low	int, length or offset, usually >= 48
8|16	4|8	low	int, usually 4
12|24	2	low	int, usually 7 (number of nonzero SCVs?)
14|26	50|94	low	*????????????*
64|120	12*LSCV	medium	12 subheader count vectors, length := LSCV = 20|40 bytes each
304|600		medium	Total length of subheader, QL

Subheader Count Vectors

The subheader count vectors encode information for each of 4 common subheader types, and potentially 12 total subheader types.

offset	length	conf.	description
0	4|8	high	int, signature (see list below)
4|8	4|8	medium	int, page where this subheader first appears := PAGE1
8|16	2	medium	int, position of subheader pointer in PAGE1 := LOC1
10|18	2|6	low	*????????????* zero padding
12|24	4|8	medium	int, page where this subheader last appears := PAGEL
16|32	2	medium	int, position of subheader pointer in PAGEL := LOCL
18|34	2|6	low	*????????????* zero padding
20|40		medium	Total length of subheader count vector, LSCV

The LOC1 and LOCL give the positions of the corresponding subheader pointer in PAGE1 and PAGEL, respectively. That is, if there are SC subheader pointers on page PAGE1, then the corresponding subheader pointer first occurs at the LOC1'th position in this array, enumerating from 1. If PAGE1=0, the subheader is not present. If PAGE1=PAGEL and LOC1=LOCL, the subheader appears exactly once. If PAGE1!=PAGEL or LOC1!=LOCL, the subheader appears 2 or more times. In all test files, PAGE1 <= PAGEL, and the corresponding subheaders appear only once per page. The variable NCT in the Row Size Subheader should be used to ensure that all Column Text subheaders are located (and to avoid scanning through all pages in the file when all subheaders are already located).

The first 7 binary signatures in the Subheader Count Vectors array are always:

signature	description
-4	Column Attributes
-3	Column Text
-1	Column Names
-2	Column List
-5	unknown signature #1
-6	unknown signature #2
-7	unknown signature #3

The remaining 5 out of 12 signatures are zeros in the observed source files. Presumably, these are for subheaders not yet defined, or not present in the collection of test files.

A Column Format and Label Subheader may appear on multiple pages, but are not indexed in Subheader Counts. The variables NCFL1 and NCFL2 in the Row Size subheader may be helpful if you want to know in advance if these appear across multiple pages.

Column Text Subheader

The column text subheader contains a block of text associated with columns, including the column names, labels, and formats. However, this subheader is not sufficient to parse this information. Other subheaders (e.g. the column name subheader), which point to specific elements within this subheader are also needed.

offset	length	conf.	description
0	4|8	high	int, signature -3 (xFDFFFFFF|xFDFFFFFFFFFFFFFF)
4|8	2	medium	int, size of text block (QL - 16|20)
6|10	2	low	*????????????*
8|12	2	low	*????????????*
10|14	2	low	*????????????*
12|16	2	low	*????????????*
14|18	2	low	*????????????*
16|20	varies	medium	ascii, compression & Creator PROC step name that generated data
varies	%QL	high	ascii, combined column names, labels, formats

This subheader sometimes appears more than once; each is a separate array. If so, the "column name index" field in column name pointers selects a particular text array - 0 for the first array, 1 for the second, etc. Similarly, "column format index" and "column label index" fields also select a text array. Offsets to strings within the text array are multiples of 4, so the column names and labels section of the array often contains many nulls for padding.

The variables LCS and LCP from the Row Size subheader refer to a text field at the start of the text array (at offset 16|20) in the first Column Text subheader (before the column name strings). This text field also contains compression information. The following logic decodes this initial field:

1. If the first 8 bytes of the field are blank, file is not compressed, and set LCS=0. The Creator PROC step name is the LCP bytes starting at offset 16.

2. If LCS > 0 (still), the file is not compressed, the first LCS bytes are the Creator Software string (padded with nulls). Set LCP=0. Stat/Transfer files use this pattern.

3. If the first 8 bytes of the field are `SASYZCRL`, the file is compressed with Run Length Encoding. The Creator PROC step name is the LCP bytes starting at offset 24.

4. If the first 8 bytes are nonblank and options 2 or 3 above are not used, this probably indicates COMPRESS=BINARY. We need test files to confirm this, though.

Column Name Subheader

Column name subheaders contain a sequence of column name pointers to the offset of each column name **relative to a** column text subheader. There may be multiple column name subheaders, indexing into multiple column text subheaders.

offset	length	conf.	description
0	4|8	high	int, signature -1 (xFFFFFFFF|xFFFFFFFFFFFFFFFF)
4|8	2	medium	int, length of remaining subheader (QL - 16|20)
6|10	2	low	*????????????*
8|12	2	low	*????????????*
10|14	2	low	*????????????*
12|16	8*CMAX	medium	column name pointers (see below), CMAX=(QL-20|28)/8
MCN	8|12	low	zeros, 12|16 + 8*CMAX := MCN

Each column name subheader holds CMAX column name pointers. When there are multiple column name subheaders, CMAX will be less than NCOL.

Column Name Pointers

offset	length	conf.	description
0	2	high	int, column name index to select Column Text Subheader
2	2	high	int, column name offset w.r.t. end of selected Column Text signature. Always a multiple of 4.
4	2	high	int, column name length
6	2	low	zeros
8		high	Total length of column name pointer

Column Attributes Subheader

The column attribute subheader holds information regarding the column offsets within a data row, the column widths, and the column types (either numeric or character). The column attribute subheader sometimes occurs more than once (in test data). In these cases, column attributes are applied in the order they are parsed.

offset	length	conf.	description
0	4|8	high	int, signature -4 (hex xFCFFFFFF|FCFFFFFFFFFFFFFF)
4|8	2	medium	int, length of remaining subheader
6|10	2	low	*????????????*
8|12	2	low	*????????????*
10|14	2	low	*????????????*

... continued on next page

offset	length	conf.	description
12|16	LCAV*CMAX	high	column attribute vectors (see below), CMAX=(QL-20|28)/LCAV, LCAV=12|16
MCA	8|12	low	MCA = 12|16 + LCAV*CMAX

Column Attribute Vectors

offset	length	conf.	description
0	4|8	high	int, column offset in data row (in bytes)
4|8	4	high	int, column width
8|12	2	low	name length flag
10|14	1	high	int, column type (1 = numeric, 2 = character)
11|15	1	low	*????????????*
12|16		high	Total length of column attribute vector, LCAV

Observed values of name length flag in the source files:

name length flag	description
4	name length <= 8
1024	usually means name length <= 8 , but sometimes the length is 9-12
2048	name length > 8
2560	name length > 8

Column Format and Label Subheader

The column format and label subheader contains pointers to a column format and label **relative to a** column text subheader. Since the column label subheader only contains information regarding a single column, there are typically as many of these subheaders as columns. The structure of column format pointers was contributed by Clint Cummins.

offset	length	conf.	description
0	4|8	high	int, signature -1026 (hex FEFB & 2 or 6 FFs)
4|8	30|38	low	*????????????*
34|46	2	high	int, column format index to select Column Text Subheader
36|48	2	high	int, column format offset w.r.t. end of selected Column Text signature. A multiple of 4.
38|50	2	high	int, column format length
40|52	2	high	int, column label index to select Column Text Subheader
42|54	2	high	int, column label offset w.r.t. end of selected Column Text signature. A multiple of 4.
44|56	2	high	int, column label length
46|58	6	low	*????????????*
52|64		medium	Total length of subheader, QL

Column List Subheader

The purpose of this subheader is not clear. But the structure is partly identified. Information related to this subheader was contributed by Clint Cummins. eyecarex (created by Stat/Transfer) does not have this subheader.

<table>
<tr><th>offset</th><th>length</th><th>conf.</th><th>description</th></tr>
<tr><td>0</td><td>4|8</td><td>high</td><td>int, signature -2 (hex FE & 3 or 7 FFs)</td></tr>
<tr><td>4|8</td><td>2</td><td>low</td><td>int, value close to offset in subheader pointer</td></tr>
<tr><td>6|10</td><td>6</td><td>low</td><td>????????????</td></tr>
<tr><td>12|16</td><td>4|8</td><td>medium</td><td>int, length of remaining subheader</td></tr>
<tr><td>16|24</td><td>2</td><td>low</td><td>int, usually equals NCOL</td></tr>
<tr><td>18|26</td><td>2</td><td>medium</td><td>int, length of column list := CL, usually CL > NCOL</td></tr>
<tr><td>20|28</td><td>2</td><td>low</td><td>int, usually 1</td></tr>
<tr><td>22|30</td><td>2</td><td>low</td><td>int, usually equals NCOL</td></tr>
<tr><td>24|32</td><td>2</td><td>low</td><td>int, usually 3 equal values</td></tr>
<tr><td>26|34</td><td>2</td><td>low</td><td>int, usually 3 equal values</td></tr>
<tr><td>28|36</td><td>2</td><td>low</td><td>int, usually 3 equal values</td></tr>
<tr><td>30|38</td><td>2*CL</td><td>medium</td><td>column list values (see below)</td></tr>
<tr><td>MCL</td><td>8</td><td>low</td><td>usually zeros, 30|38 + 2*CL := MCL</td></tr>
</table>

Column List Values

These values are 2 byte integers, with (CL-NCOL) zero values. Each nonzero value is unique, between -NCOL and NCOL. The significance of signedness and ordering is unknown. The values do not correspond to a sorting order of columns.

Compressed Binary Data Subheader

When a SAS7BDAT file is created by SAS with the option COMPRESS=CHAR or COMPRESS=YES, each row of data is compressed independently with a Run Length Encoding (RLE) structure. This yields a variable length compressed row. Each such row is stored in a single subheader in sequential order, indexed by the subheader pointers. A RLE compressed data row is identified by COMP=4 in the subheader pointer, and does not have a subheader signature. If a particular row had highly variable data and yielded no compression, it is still stored in a subheader, but uncompressed with COMP=0 instead of COMP=4. The test file `compress_yes.sas7bdat` has such highly variable (random) data and all its rows are in this COMP=0 form of subheaders. It takes up more space than the uncompressed version `compress_no.sas7bdat`, due to the extra length of the subheader pointers. The final subheader on a page is usually COMP=1, which indicates a truncated row to be ignored; the complete data row appears on the next page.

The SAS option COMPRESS=BINARY apparently uses a RDC (Ross Data Compression) structure instead of RLE. We need more test files to investigate this structure, and only document RLE at present.

Run Length Encoding

In RLE, the compressed row data is a series of control bytes, each optionally followed by data bytes. The control byte specifies how the data bytes are interpreted, or is self contained. The control byte has 2 parts - the upper 4 bits are the Command, and the lower 4 bits are the Length. Each is an uint in the range 0-15. For example, control byte 82 (hex) is Command 8 and Length 2, and control byte F4 (hex) is command 15 (F hex) and Length 4. We have identified the functions of the 11 different Command values which are observed in the test files. The RLE structure was contributed by Clint Cummins.

Command	Length	Name	Function
0	0	Copy64	using the first byte as a uint length L (0-255), Copy the next N=64+L bytes from the input to the output (copies 64 to 319 bytes)
1	?	?	*????????????* (not observed in test files)
2	?	?	*????????????* (not observed in test files)

... continued on next page

Command	Length	Name	Function
3	?	?	???????????? (not observed in test files)
4	?	?	???????????? (not observed in test files)
5	?	?	???????????? (not observed in test files)
6	0	InsertBlank17	using the first byte as a uint length L, Insert N=17+L blanks (decimal 32, hex 20) in the output (inserts 17 to 273 blanks)
7	0	InsertZero17	using the first byte as a uint length L, Insert N=17+L zero bytes in the output
8	L	Copy1	using the Length bits as a uint length L (0-15), Copy the next N=1+L bytes from the input to the output (copies 1 to 16 bytes)
9	L	Copy17	Copy the next N=17+L bytes from the input to the output (copies 17 to 32 bytes)
10 (A)	L	Copy33	Copy the next N=33+L bytes from the input to the output (copies 33 to 48 bytes)
11 (B)	L	Copy49	Copy the next N=49+L bytes from the input to the output (copies 49 to 64 bytes)
12 (C)	L	InsertByte3	Insert N=3+L copies of the next byte in the output (inserts 3 to 18 bytes)
13 (D)	L	Insert@2	Insert N=2+L @ (decimal 64, hex 40) bytes in the output (inserts 2 to 17 @ bytes)
14 (E)	L	InsertBlank2	Insert N=2+L blanks in the output
15 (F)	L	InsertZero2	Insert N=2+L zero bytes in the output

The most common Commands in `obs_all_perf_1.sas7bdat` are F and 8 (alternating). This file is entirely 8 byte doubles, so the F commands often handle consecutive zero bytes in zero value doubles.

RLE Example 1

Compressed data row:

```
87 A B C D E F G H F2 8A 1 2 3 4 5 6 7 8 9 A B D0 A1 a b c d e f g ...  z
CB -8-data-bytes-- CB CB --11-data-bytes------ CB CB --34-data-bytes--
Copy1 ~ ~ ~ ~ ~ ~ ~InsertZero2 ~ ~ ~ ~ ~ ~ ~ ~ Ins Copy33 next 34 bytes
Next 8 bytes ~ ~ ~ 4 00h bytes ~ ~ ~ ~ ~ ~ ~ ~ 2 40h
```

There are 5 Control Bytes (CB) in the above sequence.

1. 87: Copy1 next 8 bytes
2. F2: InsertZero2 4 00h bytes
3. 8A: Copy1 next 11 bytes
4. D0: Insert@2 2 40h bytes
5. A1: Copy33 next 34 bytes

Output uncompressed row:

```
A B C D E F G H 00 00 00 00 1 2 3 4 5 6 7 8 9 A B 40 40 a b c ...  z
```

RLE Example 2

Compressed data row:

```
87 A B C D E F G H C1 99 A5 a b c ...  z
CB -8-data-bytes-- CB ar CB -last-bytes
Copy1 8 ~ ~ ~ ~ ~ ~InsBy Copy33 38 bytes
```

Control Bytes in Example 2:

1. 87: Copy1 next 8 bytes
2. C1,99: InsertByte3 4 99h bytes
3. A5: Copy33 next 38 bytes

Output uncompressed row:
`A B C D E F G H 99 99 99 99 a b c ...  z`
Once a data row is uncompressed, use the SAS7BDAT Packed Binary Data description below to read the variables.

SAS7BDAT Packed Binary Data

SAS7BDAT packed binary are uncompressed, and appear after any subheaders on the page; see the Page Offset Table. These data are stored by rows, where the size of a row (in bytes) is defined by the row size subheader. When multiple rows occur on a single page, they are immediately adjacent. When a database contains many rows, it is typical that the collection of rows (i.e. their data) is evenly distributed to a number of 'data' pages. However, in test files, no single row's data is broken across two or more pages. A single data row is parsed by interpreting the binary data according to the collection of column attributes contained in the column attributes subheader. Binary data can be interpreted in two ways, as ASCII characters, or as floating point numbers. The column width attribute specifies the number of bytes associated with a column. For character data, this interpretation is straight-forward. For numeric data, interpretation of the column width is more complex.

The common binary representation of floating point numbers has three parts; the sign (`s`), exponent (`e`), and mantissa (`m`). The corresponding floating point number is `s * m * b ^ e`, where `b` is the base (2 for binary, 10 for decimal). Under the IEEE 754 floating point standard, the sign, exponent, and mantissa are encoded by 1, 11, and 52 bits respectively, totaling 8 bytes. In SAS7BDAT file, numeric quantities can be 3, 4, 5, 6, 7, or 8 bytes in length. For numeric quantities of less than 8 bytes, the remaining number of bytes are truncated from the least significant part of the mantissa. Hence, the minimum and maximum numeric values are identical for all byte lengths, but shorter numeric values have reduced precision.

Reduction in precision is characterized by the largest integer such that itself and all smaller integers have an exact representation, denoted `M`. At best, all integers greater than `M` are approximated to the nearest multiple of `b`. The table of numeric binary formats below lists `M` values and describes how bits are distributed among the six possible column widths in SAS7BDAT files, and lists.

Numeric Binary Formats

size	bytes	sign	exponent	mantissa	`M`
24bit	3	1	11	12	8192
32bit	4	1	11	20	2097152
40bit	5	1	11	28	536870912
48bit	6	1	11	36	137438953472
56bit	7	1	11	44	35184372088832
64bit	8	1	11	52	9007199254740990

Dates, Currency, and Formatting

Column formatting infomation is encoded within the Column Text Subheader and Column Format and Label Subheader. Columns with formatting information have special meaning and interpretation. For example, numeric values may represent dates, encoded as the number of seconds since midnight, January 1, 1960. The format string for fields encoded this way is "DATETIME". Using R, these values may be converted using the as.POSIXct or as.POSIXlt functions with argument `origin="1960-01-01"`. The most common date format strings correspond to numeric fields, and are interpreted as follows:

Format	Interpretation	R Function
DATE	Number of days since January 1, 1960	chron::chron
TIME	Number of seconds since midnight	as.POSIXct
DATETIME	Number of seconds since January 1, 1960	as.POSIXct

There are many additional format strings for numeric and character fields.

Platform Differences

The test files referenced in `data/sas7bdat.sources.RData` were examined over a period of time. Files with non-Microsoft Windows markings were only observed late into the writing of this document. Consequently (but not intentionally), the SAS7BDAT description above was first deduced for SAS datasets generated on the most commonly observed platform: Microsoft Windows. The extensions to SAS7BDAT files for **u64** and non-Intel formats was contributed a little later by Clint Cummins.

In particular, the files `natlerr1944.sas7bdat`, `natlerr2006.sas7bdat` appear to be generated on the 'SunOS' platform (**u64**, non-Intel). `txzips.sas7bdat` was created on Linux 64-bit SAS server (**u64**, Intel). `eyecarex.sas7bdat` is non-Intel, possibly 32-bit PowerPC.

The files `cfrance2.sas7bdat`, `cfrance.sas7bdat`, `coutline.sas7bdat`, `gfrance2.sas7bdat`, `gfrance.sas7bdat`, `goutline.sas7bdat`, `xfrance2.sas7bdat`, `xfrance.sas7bdat`, `xoutline.sas7bdat` appear to be generated on a 32-bit 'Linux' Intel system. They have the same format as Windows files, except for the (ignorable) OS strings in the first header.

Text may appear in non-ASCII compatible, partially ASCII compatible, or multi-byte encodings. In particular, Kasper Sorenson discovered some text that appears to be encoded using the Windows-1252 'code page'.

Key Test Files

filename	format features
`acadindx.sas7bdat`	non-u64, Intel (most files are like this one)
`br.sas7bdat`	truncated doubles (widths 3,4,6; compare with br2 widths all 8)
`eyecarex.sas7bdat`	non-u64, non-Intel, written by Stat/Transfer
`txzips.sas7bdat`	u64, Intel
`natlterr1994.sas7bdat`	u64, non-Intel
`hltheds2006.sas7bdat`	2 Column Attributes subheaders
`moshim.sas7bdat`	3 Column Attributes subheaders
`flightdelays.sas7bdat`	2 Column Text subheaders
`ymcls_p2_long_040506.sas7bdat`	5 Column Text subheaders, first Column Attributes subheader is on page 6
`flightschedule.sas7bdat`	2+ Column Text subheaders
`internationalflight.sas7bdat`	2+ Column Text subheaders
`marchflights.sas7bdat`	2+ Column Text subheaders
`mechanicslevel1.sas7bdat`	2+ Column Text subheaders
`compress_yes.sas7bdat`	COMPRESS=CHAR, one PGTYPE=-28672, no RLE compression (COMP=0)
`obs_all_perf_1.sas7bdat`	COMPRESS=CHAR, many PGTYPE=16384, much RLE compression (COMP=4)

Compression Data

The table below presents the results of compression tests on a collection of 142 SAS7BDAT data files (sources in `data/`). The 'type' field represents the type of compression, 'ctime' is the compression time (in seconds), 'dtime' is the decompression time, and the 'compression ratio' field holds the cumulative disk usage (in megabytes) before and after compression. Although the `xz` algorithm requires significantly more time to compress these data, the decompression time is on par with gzip.

type	**ctime**	**dtime**	**compression ratio**
gzip -9	76.7s	2.6s	541M / 30.3M = 17.9
bzip2 -9	92.7s	11.2s	541M / 19.0M = 28.5
xz -9	434.2s	2.7s	541M / 12.8M = 42.3

Software Prototype

The prototype program for reading SAS7BDAT formatted files is implemented entirely in R (see file `src/sas7bdat.R`). Files not recognized as having been generated under a Microsoft Windows platform are rejected (for now). Implementation of the `read.sas7bdat` function should be considered a 'reference implementation', and not one designed with performance in mind.

There are certain advantages and disadvantages to developing a prototype of this nature in R.

Advantages:

1. R is an interpreted language with built-in debugger. Hence, experimental routines may be implemented and debugged quickly and interactively, without the need of external compiler or debugger tools (e.g. gcc, gdb).

2. R programs are portable across a variety of computing platforms. This is especially important in the present context, because manipulating files stored on disk is a platform-specific task. Platform-specific operations are abstracted from the R user.

Disadvantages:

1. Manipulating binary (raw) data in R is a relatively new capability. The best tools and practices for binary data operations are not as developed as those for other data types.

2. Interpreted code is often much less efficient than compiled code. This is not major disadvantage for prototype implementations because human code development is far less efficient than the R interpreter. Gains made in efficient code development using an interpreted language far outweigh benefit of compiled languages.

Another software implementation was made by Clint Cummins, in the TSP econometrics package (mainly as an independent platform for exploring the format).

ToDo

- obtain test files which use COMPRESS=BINARY, and develop identification and uncompression procedures

- look for data which will reliably distinguish between structural subheaders (which have one of the known signatures) and uncompressed row data, which may have row data in the signature position that matches one of the known signatures. Both use COMP=0. Are NPSHD and NSHPL sufficient to do this?

- obtain test files with more than 2.1 billion (and more than 4.2 billion) data rows, i.e. where 8 byte integer TRC in **u64** is apparently needed. Do the non-u64 files handle this, with additional fields beyond the 4 byte TRC used for segmentation? Is TRC a (signed) int or (unsigned) uint?

- identify any SAS7BDAT encryption flag (this is not the same as 'cracking', or breaking encryption); we just identify if a file is encrypted and not readable without a key

- experiment further with 'amendment page' concept

- consider header bytes -by- SAS_host
- check that only one page of type “mix” is observed. If so insert “In all test cases (`data/sources.csv`), there are exactly zero or one pages of type 'mix'.” under the Page Offset Table header. [May not be needed, because the BC and SC fields in each Page Offset Table make the MRC field in the initial header unnecessary.]
- identify all missing value representations: missing numeric values appear to be represented as '0000000000D1FFFF' (nan) for numeric 'double' quantities.
- identify purpose of various unknown header quantities
- determine purpose of Column List subheader
- determine purpose and pattern of 'page sequence signature' fields. Are they useful?
- identify how non-ASCII encoding is specified
- implement R options to read just header (and subheader) information without data, and an option to read just some data fields, and not all fields. [The TSP implemenation already does this, and can also read a subset of the data rows.]